

LINUX DEVICE DRIVERS

2nd Edition
covers Linux 2.4



O'REILLY®

ALESSANDRO RUBINI & JONATHAN CORBET

Linux 设备驱动程序

(第二版)

内容提要

本书面向的读者，是那些想在 Linux 操作系统下支持各种计算机外设，或者想开发新的硬件并在 Linux 下运行的人们。Linux 是 Unix 市场中增长最为快速的部分，并且在许多应用领域获得了广泛而热情的支持。现在，人们越来越清楚地认识到 Linux 是嵌入式系统的一个极好平台。《Linux 设备驱动程序》已经成为该领域的一流著作，此书将以往那些口述式的经验和知识，或者隐晦的源代码注释变成了系统地讲述各种设备驱动程序编写方法的著作。

Linux 内核的 2.4 版在设备驱动程序方面发生了重大变化，它简化了许多工作，但同时提供了许多新的功能，可让驱动程序更加有效而且灵活。本书第二版彻底讲述了这些变化，并介绍了许多新的处理器和总线结构。

要阅读此书，并不要求读者成为一名内核黑客；我们仅仅希望读者理解 C 语言并熟悉 Unix 系统调用。我们循序渐进地讲述了字符设备、块设备和网络设备的驱动程序，并且给出了功能完善的示例驱动程序。这些示例驱动程序说明了驱动程序设计中的许多问题以及解决方法，并且不需要任何特定的硬件就可以运行。本书第二版的重要修订包括：对对称多处理器（SMP）系统和锁机制的讨论、对新 CPU 的支持以及新近支持的总线的讨论等等。

如果读者对操作系统完成其任务的方式感兴趣，本书则提供了对地址空间、异步事件和 I/O 的深入讨论。

可移植性是本书的一个主要关注点。尽管本书主要讲述 2.4 版本，但只要可能，我们也会讲述向后直到 2.0 版本的相关内容。《Linux 设备驱动程序》也讲述了如何在各种硬件平台上实现最大的可移植性；示例驱动程序已经在 IA32（PC）和 IA64、PowerPC、SPARC 和 SPARC64、Alpha、ARM 以及 MIPS 等平台上经过了测试。

作者简介

Alessandro Rubini 在他获得电子工程师职称后不久，就安装了 Linux 0.99.14 版本。后来，他在 Pavia 大学获得了计算机科学博士学位。但很快他就离开了大学，因为他实在不想写很多的论文。现在，他是一名自由撰稿人，编写和设备驱动程序相关的文章和论文（很有讽刺意味）。在他的小孩出世之前，他曾是一名年轻的黑客；而现在则是一名年老的、偏爱非 PC 计算机平台开发的自由软件鼓吹者。

Jonathan Corbet 早在 1981 年就接触了 BSD Unix 的源代码。那时，科罗拉多大学的一名教员让他“修正”其中的分页算法。从那时起直到现在，他深入研究了其所遇到的每一个系统，其中包括 VAX、Sun、Ardent 以及 x86 系统的驱动程序。他在 1993 年第一次接触 Linux 系统，从此以后一直从事 Linux 的开发。Corbet 先生是 Linux Weekly News (<http://LWN.net>) 的奠基人和执行主编；他和妻子及两个孩子生活在科罗拉多州巨石市。

目 录

前 言	i
Alessandro 的介绍	i
Jon 的介绍	ii
本书面向的读者	ii
内容的组织	iii
背景信息	iii
其它信息来源	iv
在线版本和条款	v
本书使用的约定	v
我们希望得到来自读者的反馈	vi
致谢	vi
 第 1 章 设备驱动程序简介	 1
1.1 设备驱动程序的作用	2
1.2 内核功能划分	3
进程管理	3
内存管理	4
文件系统	4
设备控制	4
网络功能	4
1.3 设备和模块分类	5
字符设备	5
块设备	5
网络接口	5
1.4 安全问题	6
1.5 版本编号	7
1.6 许可证条款	8
1.7 加入内核开发社团	9
1.8 本书概要	9
 第 2 章 构造和运行模块	 11
2.1 核心模块与应用程序的对比	12
2.1.1 用户空间和内核空间	14
2.1.2 内核中的并发	15
2.1.3 当前进程	15
2.2 编译和装载	16
2.2.1 版本依赖	18
2.2.2 平台依赖	19
2.3 内核符号表	20
2.4 初始化和关闭	22
2.4.1 init_module 中的出错处理	22

2.4.2	使用计数.....	24
2.4.3	卸载	25
2.4.4	显式的初始化和清除函数.....	25
2.5	使用资源	26
2.5.1	I/O 端口和 I/O 内存	27
	端口	27
	内存	29
2.5.2	Linux 2.4 中的资源分配	30
2.6	自动和手动配置	31
2.7	在用户空间编写驱动程序	33
2.8	向后兼容性	35
2.8.1	资源管理的改变.....	35
2.8.2	多处理器系统上的编译.....	36
2.8.3	在 Linux 2.0 中导出符号	36
2.8.4	模块配置参数.....	37
2.9	快速参考	37
第 3 章	字符设备驱动程序	40
3.1	scull 的设计	40
3.2	主设备号和次设备号	41
3.2.1	动态分配主设备号.....	43
3.2.2	从系统中删除设备驱动程序.....	45
3.2.3	dev_t 和 kdev_t.....	46
3.3	文件操作	47
3.3.1	file 结构	50
3.3.2	open 和 release.....	51
	open 方法	51
	release 方法	54
3.4	scull 的内存使用.....	54
3.5	竞态的简介	57
3.6	read 和 write	58
3.6.1	read 方法.....	60
3.6.2	write 方法	62
3.6.3	readv 和 writev	63
3.7	试试新设备	63
3.8	设备文件系统	64
3.8.1	实际使用 devfs.....	66
3.8.2	可移植性问题和 devfs	68
3.9	向后兼容性	68
3.9.1	文件操作数据结构的变化.....	69
3.9.2	模块使用计数.....	71
3.9.3	信号量支持的变化.....	71
3.9.4	用户空间访问的变化.....	71
3.10	快速索引	72

第 4 章 调试技术	74
4.1 通过打印调试	74
4.1.1 printk	74
4.1.2 消息如何被记录	76
4.1.3 开启及关闭消息	77
4.2 通过查询调试	79
4.2.1 使用 /proc 文件系统	79
4.2.2 ioctl 方法	83
4.3 通过监视调试	83
4.4 调试系统故障	85
4.4.1 oops 消息	85
使用 klogd	87
使用 ksymoops	88
4.4.2 系统挂起	91
4.5 调试器和相关工具	93
4.5.1 使用 gdb	93
4.5.2 kdb 内核调试器	94
4.5.3 集成的内核调试器补丁	96
4.5.4 kgdb 补丁	97
4.5.5 内核崩溃转储分析器	97
4.5.6 用户模式的 Linux 虚拟机	97
4.5.7 Linux 跟踪工具包	98
4.5.8 Dynamic Probes	98
第 5 章 增强的字符驱动程序操作	99
5.1 ioctl	99
5.1.1 选择 ioctl 命令	100
类型 (type)	101
号码 (number)	101
方向 (direction)	101
尺寸 (size)	101
5.1.2 返回值	103
5.1.3 预定义命令	103
5.1.4 使用 ioctl 参数	104
5.1.5 权能与受限操作	106
5.1.6 ioctl 命令的实现	107
5.1.7 非 ioctl 的设备控制	108
5.2 阻塞型 I/O	109
5.2.1 睡眠和唤醒	109
5.2.2 等待队列的深入分析	111
5.2.3 编写可重入代码	113
5.2.4 阻塞和非阻塞型操作	114
5.2.5 一个样例实现: sculpipe	115

5.3 poll 和 select.....	118
5.3.1 与 read 和 write 的交互.....	121
从设备读取数据	121
向设备写数据	121
刷新待处理输出	121
5.3.2 底层的数据结构.....	122
5.4 异步通知	123
5.4.1 从驱动程序的角度看.....	124
5.5 定位设备	125
5.5.1 lseek 实现	125
5.6 设备文件的访问控制	126
5.6.1 独享设备.....	127
5.6.2 关于竞态的问题.....	128
5.6.3 限制每次只由一个用户访问.....	129
5.6.4 替代 EBUSY 的阻塞型 open	129
5.6.5 在打开时复制设备.....	130
5.7 向后兼容性	132
5.7.1 Linux 2.2 和 2.0 中的等待队列	132
5.7.2 异步通知.....	133
5.7.3 fsync 方法.....	133
5.7.4 在 Linux 2.0 中访问用户空间	134
5.7.5 2.0 中的权能.....	135
5.7.6 Linux 2.0 的 select 方法.....	135
5.7.7 Linux 2.0 的设备定位	136
5.7.8 2.0 和 SMP.....	136
5.8 快速参考	136
第 6 章 时间流	140
6.1 内核中的时间间隔	140
6.1.1 处理器特有的寄存器.....	141
6.2 获取当前时间	142
6.3 延迟执行	143
6.3.1 长延迟.....	144
6.3.2 短延迟.....	145
6.4 任务队列	146
6.4.1 任务队列的本质.....	147
6.4.2 任务队列的运行.....	148
6.4.3 预定义的任务队列.....	149
示例程序是如何工作的	150
调度器队列	151
定时器队列	152
立即队列	153
6.4.4 运行自己的工作队列.....	153
6.4.5 Tasklets	154

6.5	内核定时器	155
6.6	向后兼容性	158
6.7	快速参考	159
第 7 章	获取内存	162
7.1	kmalloc 函数的内幕	162
7.1.1	flags 参数	162
	内存区段	163
7.1.2	size 参数	164
7.2	后备式高速缓存	164
7.2.1	基于高速缓存的 scull: sculld	166
7.3	get_free_page 和相关函数	167
7.3.1	使用一整页的 scull: sculld	168
7.4	vmalloc 与相关函数	169
7.4.1	使用虚拟地址的 scull: sculld	171
7.5	引导时的内存分配	172
7.5.1	在引导时获得专用缓冲区	172
7.5.2	bigphysarea 补丁	173
7.5.3	保留高端 RAM 地址	173
7.6	向后兼容性	173
7.7	快速参考	174
第 8 章	硬件管理	176
8.1	I/O 端口和 I/O 内存	176
8.1.1	I/O 寄存器和常规内存	177
8.2	使用 I/O 端口	178
8.2.1	串操作	180
8.2.2	暂停式 I/O	180
8.2.3	平台相关性	181
8.3	使用数字 I/O 端口	182
8.3.1	并口简介	182
8.3.2	示例驱动程序	183
8.4	使用 I/O 内存	185
8.4.1	直接映射的内存	186
8.4.2	在 short 中使用 I/O 内存	187
8.4.3	通过软件映射的 I/O 内存	187
8.4.4	1M 地址空间之下的 ISA 内存	189
8.4.5	isa_readb 及相关函数	190
8.4.6	探测 ISA 内存	190
8.5	向后兼容性	192
8.6	快速参考	193
第 9 章	中断处理	195
9.1	中断的整体控制	195

9.2	准备并口	196
9.3	安装中断处理程序	197
9.3.1	/proc 接口	199
9.3.2	自动检测 IRQ 号	200
	内核帮助下的探测	201
	DIY 探测	202
9.3.3	快速和慢速处理程序	203
	x86 平台上中断处理的内幕	204
9.4	实现中断处理程序	204
9.4.1	使用参数	207
9.4.2	打开和禁止中断	207
9.5	tasklet 和底半部处理	208
9.5.1	tasklet	209
9.5.2	BH 机制	210
9.5.3	编写 BH 底半部	211
9.6	中断共享	212
9.6.1	安装共享的处理程序	213
9.6.2	运行处理程序	214
9.6.3	/proc 接口	215
9.7	中断驱动的 I/O	215
9.8	竞态	216
9.8.1	使用循环缓冲区	217
9.8.2	使用自旋锁	218
9.8.3	使用锁变量	220
	位操作	220
	原子性的整数操作	221
9.8.4	无竞争地进入睡眠	222
9.9	向后兼容性	223
9.9.1	与 2.2 内核的区别	223
9.9.2	与 2.0 内核的更多区别	224
9.10	快速参考	224
第 10 章	合理使用数据类型	227
10.1	使用标准 C 语言类型	227
10.2	为数据项分配确定的空间大小	228
10.3	接口特有的类型	229
10.4	其它有关移植性的问题	230
10.4.1	时间间隔	230
10.4.2	页大小	230
10.4.3	字节序	231
10.4.4	数据对齐	232
10.5	链表	233
10.6	快速索引	235
第 11 章	kmod 和高级模块化	237

11.1	按需加载模块	237
11.1.1	在内核中请求模块.....	238
11.1.2	用户空间方面.....	238
11.1.3	模块加载和安全性.....	240
11.1.4	模块加载实例.....	240
11.1.5	运行用户态辅助程序.....	241
11.2	模块间通讯	242
11.3	模块中的版本控制	244
11.3.1	在模块中使用版本支持.....	245
11.3.2	导出版本化符号.....	246
11.4	向后兼容性	247
11.5	快速索引	248
第 12 章	装载块设备驱动程序	250
12.1	注册驱动程序	250
12.2	头文件 blk.h	256
12.3	请求处理简介	257
12.3.1	请求队列.....	257
12.3.2	执行实际的数据传输.....	258
12.4	请求处理详解	261
12.4.1	I/O 请求队列	261
	request 结构和缓冲区缓存.....	261
	操作请求队列	263
	I/O 请求锁	263
	blk.h 中的宏和函数是如何工作的	264
12.4.2	集群请求.....	265
	活动的队列头	266
12.4.2	多队列的块驱动程序.....	266
12.4.4	没有请求队列的情况.....	269
12.5	挂装和卸载是如何工作的	271
12.6	ioctl 方法	272
12.7	可移动设备	274
12.7.1	revalidation	275
12.7.2	需要特别注意的事项.....	275
12.8	可分区设备	276
12.8.1	一般性硬盘.....	277
12.8.2	分区检测.....	278
12.8.3	使用 initrd 完成分区检测	280
12.8.4	spull 的设备方法.....	280
12.9	中断驱动的块驱动程序	282
12.10	向后兼容性	283
12.11	快速参考	285
第 13 章	mmap 和 DMA	288

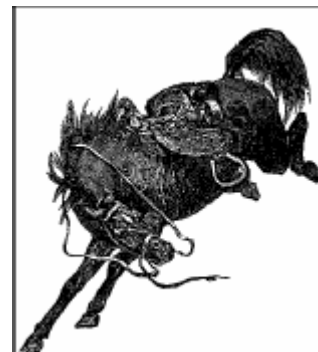
13.1 Linux 的内存管理.....	288
13.1.1 地址类型.....	288
用户虚拟地址.....	289
物理地址.....	289
总线地址.....	289
内核逻辑地址.....	289
内核虚拟地址.....	289
13.1.2 高端与低端内存.....	290
低端内存.....	290
高端内存.....	290
13.1.3 内存映射和页结构.....	290
13.1.4 页表.....	292
页目录 (PGD)	293
中级页目录 (PMD)	293
页表.....	293
13.1.5 虚拟内存区域.....	294
13.2 mmap 设备操作	297
13.2.1 使用 remap_page_range	299
13.2.2 一个简单的实现.....	299
13.2.3 增加 VMA 操作	300
13.2.4 使用 nopage 映射内存	301
13.2.5 重映射特定的 I/O 区域	303
13.2.6 重映射 RAM	303
使用 nopage 方法重映射 RAM.....	304
13.2.7 重映射虚拟地址.....	307
13.3 kiobuf 接口.....	308
13.3.1 kiobuf 结构.....	308
13.3.2 映射用户空间缓冲区以及裸 I/O	309
13.4 直接内存访问和总线控制.....	312
13.4.1 DMA 数据传输概览.....	312
13.4.2 分配 DMA 缓冲区.....	313
DIY 分配.....	313
13.4.3 总线地址.....	314
13.4.4 PCI 总线上的 DMA	315
处理不同硬件.....	315
DMA 映射.....	315
建立一致 DMA 映射.....	316
建立流式 DMA 映射.....	317
分散/集中映射.....	318
支持 PCI DMA 的不同体系结构.....	319
一个简单的 PCI DMA 例子.....	320
简单看看 Sbus 上的情况.....	321
13.4.5 ISA 设备的 DMA.....	321
注册 DMA 的方法.....	322

与 DMA 控制器通讯	323
13.5 向后兼容性	325
13.5.1 内存管理部分的改变	325
13.5.2 DMA 的变化	327
13.6 快速参考	328
第 14 章 网络驱动程序	331
14.1 snull 的设计	332
14.1.1 赋予 IP 号	332
14.1.2 数据包的物理传输	334
14.2 连接到内核	335
14.2.1 模块的装载	335
14.2.2 初始化每个设备	336
14.2.3 模块的卸载	337
14.2.4 模块化和非模块化的驱动程序	338
14.3 net_device 结构的细节	338
14.3.1 可见的成员	339
14.3.2 隐藏的成员	339
接口信息	340
设备方法	342
工具成员	344
14.4 打开和关闭	344
14.5 数据包传输	346
14.5.1 控制并发传输	347
14.5.2 传输超时	347
14.6 数据包的接收	348
14.7 中断处理程序	350
14.8 链路状态的改变	351
14.9 套接字缓冲区	351
14.9.1 重要成员	352
14.9.2 操作套接字缓冲区的函数	353
14.10 MAC 地址解析	354
14.10.1 在以太网中使用 ARP	354
14.10.2 重载 ARP	354
14.10.3 非以太网头	355
14.11 定制 ioctl 命令	356
14.12 统计信息	357
14.13 组播	358
14.13.1 对组播的内核支持	358
14.13.2 一个典型实现	359
14.14 向后兼容性	360
14.14.1 Linux 2.2 中的不同	360
14.14.2 Linux 2.0 中其它不同	361
14.14.3 探测和 HAVE_DEVLIST	362

14.15 快速参考	362
第 15 章 外设总线综述	365
15.1 PCI 接口	365
15.1.1 PCI 寻址	366
15.1.2 引导阶段	368
15.1.3 配置寄存器和初始化	368
15.1.4 访问配置空间	372
配置空间示例	373
15.1.5 访问 I/O 和内存空间	374
Linux 2.4 中的 PCI I/O 资源	375
基地址寄存器	376
15.1.6 PCI 中断	378
15.1.7 处理热插拔设备	379
pci_driver 结构	380
15.1.8 硬件抽象	382
15.2 回顾 ISA	382
15.2.1 硬件资源	383
15.2.2 ISA 编程	383
15.2.3 即插即用规范	384
15.3 PC/104 和 PC/104+	384
15.4 其它 PC 总线	384
15.4.1 MCA	385
15.4.2 EISA	385
15.4.3 VLB	385
15.5 SBus	386
15.6 NuBus	386
15.7 外部总线	386
15.7.1 USB	387
15.7.2 编写 USB 驱动程序	387
15.8 向后兼容性	389
15.9 快速参考	389
第 16 章 内核源代码的物理布局	391
16.1 引导内核	391
16.2 引导之前	393
16.3 init 进程	395
16.4 kernel 目录	395
16.5 fs 目录	396
16.6 mm 目录	397
16.7 net 目录	398
16.8 ipc 和 lib	399
16.9 include 和 arch 目录	399
16.10 drivers 目录	400

16.10.1	drivers/char	400
16.10.2	drivers/block	400
16.10.3	drivers/ide	401
16.10.4	drivers/md	401
16.10.5	drivers/cdrom	401
16.10.6	drivers/scsi	402
16.10.7	drivers/net	402
16.10.8	drivers/sound.....	403
16.10.9	drivers/video	403
16.10.10	drivers/input	404
16.10.11	drivers/media.....	404
16.10.12	总线相关目录.....	404
16.10.13	平台相关目录.....	405
16.10.14	其它子目录.....	405
附录 A 参考书目		407
A.1	Linux 内核书籍	407
A.2	Unix 设计和内幕.....	407
附录 B 封面故事		409

前 言



顾名思义，本书是讲述 Linux 设备驱动程序编写的。面对层出不穷的新硬件产品，必须有人不断编写新的驱动程序，以便让这些设备能够在 Linux 下正常工作，从这个意义上讲，讲述驱动程序的编写，本身就是一件非常有意义的工作。但本书也涉及到 Linux 内核的工作原理，同时还讲述如何根据自己的需要和兴趣来定制 Linux 内核。Linux 是一个开放的系统，我们希望借助本书，它能够更加开放，从而能够吸引更多的开发人员。

自本书第一版问世以来，Linux 本身的变化非常巨大。现在的 Linux 能够在更多的处理器上运行，并且支持更加广泛的硬件，许多内部的编程接口也相应发生了重大变化，因此，我们决定编写本书的第二个版本。这一版本以 Linux 2.4 版本的内核为主，讲述了新内核提供的所有新特色，同时，仍然兼顾了早期的内核版本。

我们希望读者能够从本书的学习当中获得乐趣，就像我们自己从编写本书的过程中获得乐趣一样。

Alessandro 的介绍

作为一个喜欢 DIY 的电子工程师，我一直乐于使用计算机来控制一些外部的硬件设备。从我小时候使用父亲的 Apple IIe 计算机开始，我就开始寻找另外一个平台，以便能够将我自制的电路板连接其上，并能够编写自己的驱动程序。不幸的是，不管是从硬件级别，还是从软件级别看，80 年代 PC 的功能都不是非常强大：PC 的内部设计比起 Apple II 来简直是差远了，而且可供利用的文档也远远不能令人满意。但在 Linux 出现之后，我决定尝试利用这个新的操作系统，为此，我购买了一个昂贵的 386 主板，但没有购买任何受到所有权保护的软件。

那时，我在大学里使用 Unix 系统，这个设计精巧的系统令我激动不已，尤其在有了由 GNU 项目提供给用户使用的更加精巧的工具之后，这个系统更加令我着迷。对我来讲，在我自己的 PC 主板上使用 Linux 内核，一直是最为难忘的经历，我不仅可以编写自己的设备驱动程序，而且还有了机会再次拿起电烙铁。我不断地告诉别人，“我长大之后，一定要成为一名黑客，”而 GNU/Linux 则是实现这一梦想的最佳平台。可是，我不知道我是否真正长大。

随着 Linux 的成熟，越来越多的人开始乐于为自制的电子设备或者商用设备编写驱动程序。就像 Linus Torvalds 所说的那样，“我们又回到了为自己编写设备驱动程序的‘远古’时代。”

1996 年的时候，我经常为那些从别人那里借来的，或者别人给我的，或者是自己在家里制作的硬件设备编写自己的设备驱动程序，并且乐此不疲。那时，我已经为 Michael Johnson 所著的《内核黑客指南》贡献了一些内容，并开始为《Linux 杂志》编写内核相关的文章。有了 Michael 的帮助，我认识了在 O'Reilly 工作的 Andy Oram，他希望我能就设备驱动程序编写一本书，我非常乐意地接受了这一工作，有很长一段时间我一直忙于编写这本书。

到了 1999 年，我发现，我已经没有足够的精力来独自完成本书的更新了：我的家庭在长大，而更多的时间要花费在编写 GPL 软件的工作上。除此之外，内核也变得更大，而且可以支持比已往更多的平台，而 API 也变得更加复杂和成熟。这时，Jonathan 开始帮助我更新本书。他拥有足够的技巧、能力和热情，而我则继续负责已经拉下很多的进度跟踪。他利用自己良好的技能和热情，已经成为推进这个进程的最有力助手，这些却是我无法达到的。我非常高兴能够和他共事，不管在技术上还是在私人方面。

Jon 的介绍

我从 1994 初开始接触 Linux，那时，我正在说服自己的老伴为我购买一台 Fintronic Systems 公司生产的笔记本电脑。作为 80 年代初期（那时起我就在和源代码打交道）的一名 Unix 用户，我立即被 Linux 所吸引。恰好在 1994 年，Linux 已经成为一个非常实用的系统，而且也是我所遇到的第一个真正自由的系统。那时，我几乎完全丧失了对所有权系统的兴趣。

但我并没有一个完整的计划想为 Linux 编写什么著作。当 O'Reilly 和我讨论有关帮助编写本书第二版事宜的时候，我刚刚从我工作了 18 年的公司辞职，并成立了一个 Linux 顾问咨询公司。为了吸引别人的注意力，我们建立了一个 Linux 新闻站点，即 Linux Weekly News (<http://lwn.net>)，该站点的内容主要集中于内核开发。随着 Linux 的大众化，该 web 站点也变得非常知名，而我们的咨询业务却最终被人遗忘。

然而，我的第一兴趣却始终还是系统编程。早些时候，我“修正”最初 BSD Unix 系统当中的分页代码（这是一个可怕的黑客工作），或者在 VAX/VMS 系统上编写磁带驱动器的驱动程序（这些源代码是可获得的，如果你不在意这些由汇编和 BLISS 语言编写的代码的话）。随着时间的推移，我又为 Alliant、Ardent 和 Sun 等系统编写驱动程序。后来，我开始利用 Linux 开发雷达数据收集系统，这个时候，也就是编写本书的时候，也正是修正 Linux 软盘驱动程序中 I/O 请求队列锁的实现的时候。

我为能参与本书的编写而感到高兴。首先，通过本书的编写，我能够更加深入地研究内核代码，同时能够帮助别人达到同样的目的。Linux 是个实用的系统，而同时也是一个带给人乐趣的系统，而围绕内核工作，则是所有事情当中最令人兴奋和激动的事情之一。和 Alessandro 一起工作也令人高兴，我必须感谢他信任我对他优秀的文字所作的修修改改，也感谢他在我出现错误或者不能赶上进度时的耐心，当然也得感谢那次到 Pavia 的破单车旅行。编写本书的那些时光的确难忘！

本书面向的读者

在技术方面，本书提供了一条理解内核内幕以及理解一些 Linux 开发者所做出的设计决策的行家途径。尽管本书的主要目的是教读者如何编写设备驱动程序，但同时也给出了内核实现方面的概览。

尽管真正的黑客能够从正式的内核源代码中找到所有必要的信息，但通常来讲，编写好的书籍能够更好地帮助读者提高编程技巧。读者将要看到的文字，来自对内核源代码的仔细分析，我们希望我们所付出的努力是值得的。

本书对那些希望编写计算机设备驱动程序的人员，或者那些要解决 Linux 机器内部问题的程序员来讲，将是非常有帮助的。请注意，“Linux 机器”是一个比“运行 Linux 的 PC”更为宽泛的概念，因为 Linux 现在能够支持许多不同的硬件平台，而内核编程不再绑定到某个特定的平台。我们希望本书能够成为那些想成为内核黑客，但却不知如何下手的人们的良好起点。

Linux 狂热者将从本书找到深入内核代码的足够精神食粮；通过本书的学习，将有能力加入到为某个新功能或性能增强不停工作的开发小组当中。本书并没有涵盖 Linux 内核的全部，但是，作为 Linux 设备驱动程序开发人员，你需要了解如何和许多的内核子系统一起工作。因此，本书对内核编程作了一个一般性的介绍。Linux 仍然在不断改进和发展，因此，新程序员始终有机会加入到这一 Linux 的开发大军中。

另一方面，如果你只是为了为自己的设备编写一个驱动程序，而不想过多了解内核的内幕信息的话，本书内容则足够模块化以满足你的需求。如果你不想深入到细节当中，则可以简单跳过大部分的技术章节，而直接阅读可由设备驱动程序使用的、能够和内核的其它部分无缝结合的标准 API。

本书的主要讲述对象是如何为 Linux 内核 2.4 版本编写内核模块。模块是能够在运行时装载到内核的目标代码，它能够为正在运行的内核添加新的功能。我们尽其可能地让示例代码也能够在内核的 2.2 和 2.0 版本中运行，如果需要有所改动则会指出。

内容的组织

本书内容由简到难，并划分为两大部分。第一部分（第 1 章到第 10 章）首先讲述了如何编写内核模块，然后讲述了编写功能完备的字符设备驱动程序所涉及的各个编程主题。每一章分别讲述某一个特定问题，并在每章结尾包含一个“符号表”，该符号表可在实际开发中作为参考使用。

在本书第一部分中，内容从软件相关的概念过渡到硬件相关的概念。这种组织方法意味着，你能够尽可能不在机器中插入任何外部硬件而测试示例代码。每章都包含有源代码，并给出了能够在任意一台 Linux 计算机上运行的示例驱动程序。但是，在第 8 章和第 9 章中，我们需要读者在并口上连接一些电线，以便测试硬件处理代码，当然，这一要求对任何人来讲都是可以做到的。

本书的第二部分讲述了块设备驱动程序和网络接口，并深入讨论了一些更加高级的内容。许多驱动程序作者可能不需要这些内容，但我们鼓励你阅读这些章节。尽管对某个特定的项目来说，你并不需要了解这些知识，但第二部分的许多内容和了解 Linux 内核的工作原理一样重要。

背景信息

为了更好地利用本书，我们希望读者熟悉 C 语言编程。因为我们经常会提到 Unix 命令和管道，因此，也需要读者拥有 Unix 的使用经验。

在硬件级，不需要读者有任何预先的经验就可以理解本书内容，当然，一些一般性的概念是必须清晰的。本书内容并不基于某个特定的 PC 硬件，我们在提到某个特定的硬件时，会提供给读者所有必要的信息。

建立内核需要一些自由软件工具，而且经常要求使用这些工具的特定版本。太老的工具可能缺少一些必要的特性，而太新的工具又可能会偶尔生成不能正常工作的内核。通常而言，当前流行的 Linux 发行版所提供的工具能够很好地工作。不同的内核版本对工具的版本需求不同，这时，你可以参考内核源代码树中的 `Documentation/Changes` 文件。

其它信息来源

本书提供的大部分信息直接来自内核源代码以及相关文档。我们要特别注意内核源代码树中的 `Documentation` 目录，其中包含有大量有用的信息，比如内核 API 中新增的部分（在 `DocBook` 子目录）。

还有一些有用的书籍包含了更为广泛的内容，这些书籍列在本书的“参考书目”中。

Internet 上有大量可用的信息，下面将列出部分站点。当然，Internet 站点的信息在以爆炸式的方式增加，而印刷书籍却难以及时更新。这样，下面的清单可在本书过时的情况下发挥作用。

```
http://www.kernel.org
ftp://ftp.kernel.org
```

本站点是 Linux 内核开发的主站点，其中包含了最新的内核发行版本以及相关信息。注意该 FTP 站点的镜像遍布全球，因此，应该选择最近的镜像站点下载 Linux 源代码。

```
http://www.linuxdoc.org
```

“Linux Documentation Project”拥有大量称作“HOWTO”的文档，其中一些是技术性的，并涉及到一些内核主题。

```
http://www.linux-mag.com/depts/gear.html
```

“Gearheads only”中经常发布一些转载自《Linux Magazine》的、由知名开发人员编写的关于内核的文章。

```
http://www.linux.it/kerneldocs
```

其中包含有许多 Aleesandro 所著的有关内核的杂志文章。

```
http://lwn.net
```

该新闻站点由本书的作者之一编辑维护，提供了定期的内核开发相关报道。

```
http://kt.zork.net
```

“Kernel Traffic”是一个大众性的站点，它提供了每周 Linux 内核开发邮件列表中的讨论总结。

```
http://www.atnf.csiro.au/~rgooch/linux/docs/kernel-newsflash.html
```

“Kernel Newsflash”站点是一个内核新闻的集散地。该站点尤其专注于当前内核版本中的兼容性问题，人们可以非常容易地看到为什么自己的驱动程序不能在最新的内核当中正常工作。

<http://www.kernelnotes.org>

“Kernel Notes” 是一个关于内核版本信息、非正式补丁等的经典站点。

<http://www.kernelnewbies.org>

该站点面向新的内核开发人员。其中包含有针对初学者的内容和 FAQ，而且还有一个 IRC 频道，可获得即时的帮助。

<http://lksr.org>

“Linux Kernel Source Referenct” 是几乎所有内核历史版本的 CVS 归档的 web 接口。如果你想知道某个特定主题的历史变迁，这个站点再合适不过了。

<http://www.linux-mm.org>

该网页是面向 Linux 内存管理开发的，其中包含有大量有用信息，并且还包含有许多内核相关的 Web 站点链接。

<http://www.conecta.it/linux>

这个意大利站点包含了几乎所有正在开发的 Linux 相关项目的信息，并且更新及时。也许读者已经知道了包含有大量 Linux 开发的 HTTP 链接的站点，如果没有，这个站点将是一个非常好的选择。

在线版本和条款

本书作者已经选择让本书在 GNU Free Documentation License (GNU 自由文档许可证) 版本 1.1 的保护下免费获取。

该许可证全文可见：

<http://www.oreilly.com/catalog/linuxdrive2/chapter/licenseinfo.html>;

HTML

<http://www.oreilly.com/catalog/linuxdrive2/chapter/book>;

DocBook

<http://www.oreilly.com/catalog/linuxdrive2/chapter/bookindex.xml>;

PDF

<http://www.oreilly.com/catalog/linuxdrive2/chapter/bookindexpdf.html>.

本书使用的约定

下面是本书中用到的一些印刷约定：

斜体	用于文件、目录的名称，程序和命令的名称，命令行选项，URL 以及新的术语
等宽字体	用于文件内容或者命令的输出，还用于正文中出现的 C 代码或者其它字符串
等宽斜体	用于可变选项、关键词，或者需要用户用实际值替换的文字
等宽黑体	用在示例中，表示需要用户照原文键入的命令或者其它文字

读者还要注意文中带有如下图标的特殊段落：



表示技巧。其中包含了相关主体的有用辅助信息。



表示警告。它可以帮助你解决或者避免一些问题。

我们希望得到来自读者的反馈

我们已经尽我们所能验证了本书内容，但是读者可能会发现某些功能已经改变（或者甚至是我们所犯的错误！）。请将你发现的所有错误，或者对于将来版本的建议告诉我们，来信请寄：

```
O'Reilly & Associates, Inc.  
101 Morris Street  
Sebastopol, CA 95472  
(800) 998-9938 (in the United States or Canada)  
(707) 829-0515 (international/local)  
(707) 829-0104 (fax)
```

我们还为本书建立了一个网页，其中列出了勘误、示例等内容。该网页地址如下：

```
http://www.oreilly.com/catalog/linuxdrive2
```

如果你希望对本书进行评论，或者遇到有关本书的技术问题，可发电子邮件到：

```
bookquestions@oreilly.com
```

有关 O'Reilly 图书的更多信息，包括会议、软件、资源中心以及 O'Reilly Network，可访问我们的 Web 站点：

```
http://www.oreilly.com
```

致谢

本书的编写得到了许多人的帮助，我们向他们致以诚挚的谢意。

我（Alessandro）要感谢促成本书的那些人。首先要感谢的是 **Federica**，在我们的蜜月期间，我在帐篷的笔记本电脑上审校本书第一版时，她给予充分的理解和支持。**Giorgio** 和 **Giulia** 只牵涉到本书的第二版，她们在我集中精力编写的时候吃纸、拉线、哭泣，不时将我带回现实。我还要感谢四位祖母，他们在我临近最后期限时，整日代我履行父亲的职责，帮助我集中精力于代码和咖啡。我仍然非常感激 **Michael Johnson**，在他的帮助下，我开始了本书的编写。尽管这已经是几年前的事情了（那时，我离开了学校，以便专心编写程序而不是撰写论文），但他仍然是促使本书问世的第一人。作为一名独立的技术顾问，没有哪个老板阻止我在工作时间编写本书，但另一方面，我仍然要感谢 **Francesco Magenta** 和 **Rodolfo Giometti**，他们帮助我成了一名“有靠山的顾问。”

最后，我还要感谢许多自由软件的作者，是他们真正教会了我如何忘我编程，这包括内核作者以及我所读过的用户级应用程序的作者。限于篇幅，我不能在这里列出他们的名字。

我(Jon)要感谢许多帮助过我的人。首先要感谢的是我的夫人 **Lanra**，她在我试图建立一个“.com”公司的同时编写书籍而花费了大量时间。我的两个孩子，**Michela** 和 **Guilia**，始终是我快乐和灵感的源泉。我在 **LWN.net** 的同事们对我因编写本书而分心表现出了极大的容忍；我还要感谢 **LWN** 内核网页的读者对我的支持。如果没有 **Boulder** 的本地社区广播电台（可称为“**KGNU**”），这一版本可能就无法问世。这个广播电台播放吸引人的音乐，以及 **Lake Eldora** 的滑雪广告。有了它，我才得以在孩子们滑雪的时候，整日带着笔记本电脑露营，并享受咖啡。我还要特别感谢 **Evi Nemth**，她让我在她的 **VAX** 机器上研究早期的 **BSD** 源代码；还要感谢 **William Waite**，是他真正教会了我如何编程；最后要感谢 **National Center for Atmospheric Research (NCAR)** 的 **Rit Carbone**，是他给了我一个长期的职位，我在那里学到了所有其它的东西。

我们两个作者还要感谢本书的编辑，**Andy Oram**，在他的努力下，本书成为一本非常好的图书产品。我们要感激许多推进自由软件思想并让这些软件发挥作用的伟大的人（这主要归功于 **Richard Stallman**，但他绝不是唯一的）。

许多人还帮助我们建立了硬件环境，没有这些来自外部的帮助，我们不可能研究这么多的平台。我们要感谢 **Intel** 借给我们一台早期的 **IA-64** 系统，还有 **Rebel.com** 捐赠了一台 **Netwinder**（基于 **ARM** 的微型计算机）。**Prosa Labs**，即 **Linuxcare-Italia** 的前身，借给我们一台配置非常好的 **PowerPC** 系统；**NEC Electronics** 捐赠了他们最有趣的 **VR4181** 处理器开发系统，这个系统是一个掌上型电脑，我们可以将 **GNU/Linux** 系统烧到 **Flash** 存储器中，并在这个系统上运行。**Sun-Italia** 借给我们一台 **SPARC** 系统和一台 **SPARC64** 系统。所有这些公司和他们的系统让 **Alessandro** 忙于解决移植问题，而且还不得不多用一间屋子来建立他的“硅片兽动物园”。

本书第一版由 **Alan Cox**、**Greg Hankins**、**Hans Lermen**、**Heiko Eissfeldt**，以及 **Miguel de Icaza**（依照名字字母排序）进行了技术审校。第二版的技术审校人是 **Allan B. Cruse**、**Christian Morgner**、**Jake Edge**、**Jeff Garzik**、**Jens Axboe**、**Jerry Cooperstein**、**Jerome Peter Lynch**、**Michael Kerrisk**、**Paul Kinzelman** 和 **Raph Levien**。他们花费了大量时间寻找本书的错误或者问题，并且指出了文中可以提高的地方。

最后，让我们感谢 **Linux** 开发人员所做出的艰苦工作。这包括内核程序员以及经常会被遗忘的应用软件开发人员。本书中，我们一直没有提到他们的名字，以避免因为遗忘其他人的名字而显得不公平。有时也有例外，我们会提到 **Linus** 的名字，当然，我们希望他不会介意。

第 1 章 设备驱动程序简介



随着 Linux 系统变得越来越流行，人们编写 Linux 驱动程序的兴趣也在稳步增长。Linux 的大部分内容独立于底层硬件运行，许多用户也无需关心硬件问题。但是，Linux 所支持的每一款硬件，一定有人曾为它编写过驱动程序，否则就无法在 Linux 系统下发挥功能。也就是说，没有设备驱动程序，就不会有功能完整的运行系统。

设备驱动程序在 Linux 内核中扮演着特殊角色。它们是一个个独立的“黑盒子”，使某个特定硬件响应一个定义良好的内部编程接口，这些接口完全隐藏了设备的工作细节。用户操作通过一组标准化的调用执行，而这些调用是和特定的驱动程序无关的。将这些调用映射到作用于实际硬件的设备特有操作上，则是设备驱动程序的任务。这个编程接口能够使得驱动程序独立于内核的其它部分而建立，必要的情况下，可在运行时“插入”内核。这种模块化的特点，使得 Linux 驱动程序的编写非常简单，因此内核驱动程序的数目也增长迅速，目前已有成百上千的驱动程序可用。

促使我们对编写 Linux 驱动程序感兴趣的原因有很多。首先，仅新硬件问世（或过时）的速度就会使驱动程序编写人员面临很多任务；其次，个人用户需要了解一些驱动程序知识才能访问设备；另外，硬件厂商通过提供 Linux 驱动程序，能为自己的产品带来数目庞大且日益增长的潜在用户群；最后，Linux 系统是开放源码的，如果驱动程序作者愿意，驱动程序源码就可以在用户中间迅速流传。

本书将讲述有关驱动程序编程以及内核的相关知识。我们采取独立于硬件的方法，所讲述的编程技巧和接口尽可能不依赖于任何具体设备。每个驱动程序都不尽相同，作为驱动程序开发者也应该理解自己的具体设备。然而，所有驱动程序的基本原理和技巧都是相同的，本书不准备讲述具体的设备，而主要集中在让设备工作的背景知识上。

刚开始学习编写驱动程序时，我们经常会碰到许多关于 Linux 内核的知识。它将帮助我们理解机器如何工作，工作为什么不如预期的那样快，或者为什么没有产生预期的结果等等。我们将逐渐介绍新知识，先从简单的驱动程序开始，然后逐渐构造复杂的驱动程序。每个新概念都带有示例代码，它们不需要特别的硬件支持就可以运行。

本章不涉及实际的编程。然而，我们会介绍一些有关 Linux 内核的背景知识，这些知识在后来进

行实际编程时将非常有用。

1.1 设备驱动程序的作用

作为驱动程序编写者，我们需要在所需的编程时间以及驱动程序的灵活性之间选择一个可接受的折中。读者可能奇怪于说驱动程序“灵活”，我们用这个词实际上是强调设备驱动程序的作用在于提供机制，而不是提供策略。

区分机制和策略是 Unix 设计背后隐含的最好思想之一。大多数编程问题实际上都可以分成两部分：“需要提供什么功能”（机制）和“如何使用这些功能”（策略）。如果这两个问题由程序的不同部分来处理，或者甚至由不同的程序来处理，则这个软件包更易开发，也更容易根据需要进行调整。

例如，Unix 中图形显示器的管理就分成 X 服务器以及窗口和会话管理器两部分。前者操作硬件，给用户程序提供统一接口；后者实现特定策略，不用知道任何与硬件相关的知识。我们可以在不同硬件上运行同样的窗口管理器，不同的用户也可以在相同的工作站上使用不同的配置。即使完全不同的桌面环境，比如 KDE 和 GNOME，也能在同一个系统中共存。另外一个例子是具有分层结构的 TCP/IP 网络。位于下层的操作系统负责提供套接字抽象层，但在所传输的数据上则没有附加任何策略；上面各层的服务器则分别提供不同的服务（以及相关策略）。另外，一个类似 ftpd 这样的服务器提供文件传输机制，用户可以使用任何自己喜欢的客户端传输文件，例如命令行和图形客户端；而任何人也可以写一个新的用户界面来传输文件。

驱动程序同样存在机制和策略的分离。例如，软驱驱动程序不带策略，它的作用是将磁盘表示为一个连续的数据块序列，而系统高层负责提供策略，比如谁有权访问软盘驱动器，是直接访问驱动器还是通过文件系统，以及用户是否可以在驱动器上挂装文件系统等等。既然不同的环境通常需要不同的方式来使用硬件，我们应当尽可能做到让驱动程序不带策略。

程序员编写驱动程序时应该特别注意下面这个基本概念：编写访问硬件的内核代码时不要给用户强加任何策略。因为不同用户有不同需求，驱动程序应该处理如何使硬件可用的问题，而将怎样使用硬件的问题留给上层应用。因此，当驱动程序只提供了访问硬件的功能而没有附加任何限制时，这个驱动程序就比较灵活。然而，有时候我们也需要在驱动程序中实现一些策略。例如，某个数字 I/O 驱动程序只提供了以字节为单位访问硬件的方法，这样就省去了编写额外的代码处理单个数据位的麻烦。

如果从另外一个角度来看驱动程序，它可以看作是应用和设备之间的一个软件层。这种定位使驱动程序编写者可以选择如何展现设备特性：即使对于相同设备，不同的驱动程序也可以提供不同的功能。实际的驱动程序设计应该在许多考虑因素之间作出平衡。例如，某个驱动程序可能同时被多个进程使用，我们就应当考虑如何处理并发问题：可以在设备上实现独立于硬件功能的内存映射；也可以提供一个用户函数库，以帮助应用程序开发者在原语基础上实现新的策略。总的来说，驱动程序设计主要还是综合考虑下面三方面的因素：提供给用户尽量多的选项、驱动程序编写占用较少时间以及尽量保持程序简单而不至于错误丛生。

不带策略的驱动程序包括一些典型的特征：同步和异步操作都支持、驱动程序能够多次打开、能够充分利用硬件特性以及不具备“简化任务”功能或提供与策略相关的软件层等。这种类型的驱动程序不仅能很好地服务最终用户，而且易于编写和维护。实际上，不带策略是软件设计者的一个共同

目标。

然而，许多驱动程序是同用户程序一起发行的。这些用户程序主要用来帮助配置和访问目标设备。它们可能是简单的工具，也可能是完整的图形应用程序。例如，用来调整并口打印机驱动程序工作方式的 `tunelp` 程序；作为 PCMCIA 驱动程序包一部分的图形化的 `cardctl` 工具等等。和驱动程序一起提供的还会有一个客户程序库，它提供了那些不必在驱动程序本身实现的功能。

本书的讨论范围局限于内核，因此我们将尽量避免讨论策略、应用程序和支持库的问题。有时可能会确实会涉及到有关策略以及如何支持策略的内容，但我们不会深入讨论使用设备的用户程序及它们所实现的策略。另外，我们应该清楚，用户程序是软件包的有机组成部分，即使不带策略的软件包，也会同时发布配置文件为下层机制提供缺省配置。

1.2 内核功能划分

Unix 系统支持多个进程并发运行，每个进程都请求系统资源，比如处理能力、内存、网络连接和其它一些资源等。内核负责处理所有这些请求，根据内核完成任务的不同（这些任务之间的区别可能不总是那么清楚），如图 1-1 所示，可将内核功能分成如下几部分：

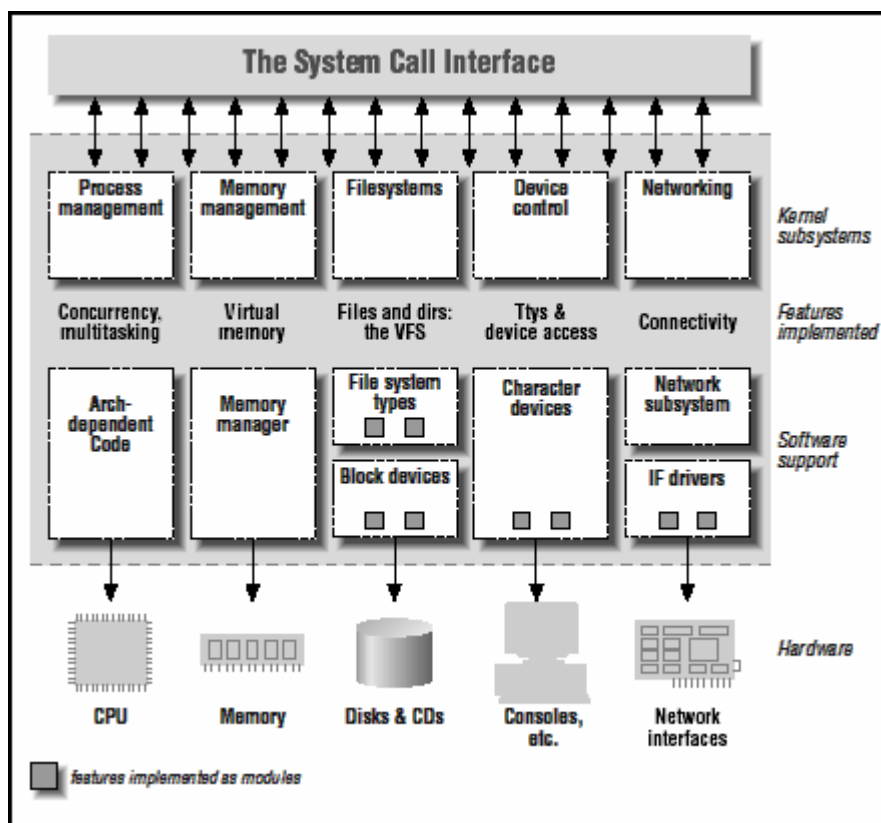


图 1-1：内核功能的划分

进程管理

进程管理功能负责创建和撤销进程以及处理它们和外部世界的连接（输入输出）。不同进程之间的通讯（通过信号、管道或进程间通讯原语）是整个系统的基本功能，因此也由内核处理。除此之外，

控制进程如何共享 CPU 的调度器也是进程管理的一部分。概括来说，内核进程管理活动就是在单个或多个 CPU 上实现了多个进程的抽象。

内存管理

内存是计算机的主要资源之一，用来管理内存的策略是决定系统性能的一个关键因素。内核在有限的可用资源之上为每个进程都创建了一个虚拟地址空间。内核的不同部分在和内存管理子系统交互时使用同一套系统调用，这包括从简单的 `malloc/free` 到其它一些不常用的系统调用。

文件系统

Unix 在很大程度上依赖于文件系统的概念，Unix 中的每个对象几乎都可以当作文件来看待。内核在没有结构的硬件上构造结构化的文件系统，构造的文件系统抽象在整个系统中广泛使用。另外，Linux 支持多个文件系统类型，即在物理介质上不同组织数据的方式。例如，磁盘可以格式化为符合 Linux 标准的 `ext2` 文件系统，也可格式化为常用的 `FAT` 文件系统。

设备控制

几乎每条系统操作最终都会映射到物理设备上。除了处理器、内存以及其它很有限的几个对象外，所有设备控制操作都由与被控制设备相关的代码来完成。这段代码就叫做驱动程序，内核必须为系统中的每件外设嵌入相应的驱动程序，包括硬盘驱动器、键盘和磁带条（`streamer`）等。这方面的内核功能将是本书讨论的主题。

网络功能

网络功能也必须由操作系统来管理，因为大部分网络操作和具体进程无关——数据包的传入是异步事件。在某个进程处理这些数据包之前必须已经被收集、标识和分发。系统负责在应用程序和网络接口之间传递数据包，并根据网络活动控制程序的执行。另外，所有的路由和地址解析问题都有内核处理。

在本书末尾的第 16 章，我们会看到 Linux 内核的导引图（`road map`），目前先暂时介绍到这儿。

Linux 的优良特性之一是能够在运行时动态扩展内核特性，当系统正在运行时我们就可以给内核增添新的功能。

运行时向内核中添加的代码称之为 `模块`，Linux 内核支持几种不同类型（或分类）的模块，这当中包括（但不仅仅局限于）驱动程序。每个模块由 `目标文件`（没有连接为完整的可执行文件）构成，可以由 `insmod` 程序 `加载` 到运行内核，也可以由 `rmmod` 程序 `卸载`。

图 1-1 中列出负责特定任务的几个不同类型的模块。根据模块所提供的功能我们确定它属于哪个类型，图 1-1 中列出了最重要的几个类型，但远远不是全部，因为越来越多的 Linux 功能正在被模块化。

1.3 设备和模块分类

Unix 系统将设备分成三种类型：字符设备、块设备和网络设备。每个模块通常实现其中一种类型的设备，相应地，模块可分为字符模块、块设备模块和网络模块三种。然而这种分类方式并非非常严格，程序员可以构造一个大的模块，在其中实现不同类型的设备驱动程序。然而，优秀程序员通常还是为每个功能创建一个不同的模块，从而实现良好的伸缩性和扩展性。

三种类型的设备如下：

字符设备

字符设备是个能够象字节流（比如文件）一样访问的设备，由字符设备驱动程序来实现这种特性。字符设备驱动程序通常至少需要实现 `open`、`close`、`read` 和 `write` 系统调用。字符终端（`/dev/console`）和串口（`/dev/ttys0` 以及类似设备）就是两个字符设备，它们能够很好地表示成流抽象。字符设备可以通过文件系统节点（比如 `/dev/tty1` 和 `/dev/lp0` 等）来访问，它和普通文件之间的唯一差别在于对普通文件的访问可以前后移动访问指针，而大多数字符设备是个只能顺序访问的数据通道。然而，也存在和数据区特性类似的字符设备，访问它们时可前后移动访问指针。例如帧抓取器就是这样一个设备，应用程序可以用 `mmap` 或 `lseek` 访问抓取的整个图象。

块设备

和字符设备一样，块设备也是通过 `/dev` 目录下的文件系统节点来访问。块设备（例如磁盘）上能够容纳文件系统。在大多数 Unix 系统中，块设备包含整数个块，而每块包含 1K 或 2 的其它次幂字节的数据。Linux 可以让应用程序象字符设备一样地读写块设备，允许一次传递任意多字节的数据。因而，块设备和字符设备的区别仅仅在于内核内部管理数据的方式，也就是内核和驱动程序的接口不同。象字符设备一样，块设备也是通过文件系统节点来访问，它们之间的差异对用户是透明的。块设备除了给内核提供和字符设备一样的接口外，另外还提供了专门面向块设备的接口，不过这些接口对于那些从 `/dev` 目录下某个目录项打开块设备的用户和应用程序都是不可见的。另外，块设备的接口必须支持挂装文件系统。

网络接口

任何网络事务都要经过一个网络接口，即一个能够和其它主机交换数据的设备。通常接口是个硬件设备，但也可能是个纯软件设备，比如回环接口。网络接口由内核中的网络子系统驱动，负责发送和接收数据包，它不用了解每项事务如何映射到实际传送的数据包。尽管 Telnet 和 FTP 连接都是面向流的，它们都使用了同一个设备，而这个设备看到的只是数据包，而不是一个个流。

由于不是面向流的设备，因此将网络接口映射到文件系统节点（比如 `/dev/tty1`）比较困难。Unix 式的访问网络接口的方法是给它们分配一个唯一的名字（比如 `eth0`），这个名字在文件系统中不存在对应的节点项。内核和网络驱动程序间的通讯完全不同于内核和字符设备以及块设备驱动程序之间的通信，内核调用一套和数据包传输相关的函数而不是 `open`、`write` 等。

Linux 中还存在其它类型的驱动程序模块，这些模块利用内核提供的公共服务来处理特定类型的设备。因此我们能够和通用串行总线（USB）模块、串口模块等通信。最常见的非标准类型的设备是

SCSI 设备*。

尽管和 SCSI 总线连接的每一款外部设备都在 `/dev` 目录下作为块设备或字符设备出现，软件的内部组织却不一样。

就像网卡给网络子系统提供与硬件相关的功能一样，SCSI 控制器给 SCSI 子系统提供访问实际接口电缆的能力。SCSI 是计算机和外部设备之间的一个通信协议，不管计算机上插入什么类型的控制板，每个 SCSI 设备都响应同样的协议。Linux 内核中因此实现了 SCSI 模块（即文件操作到 SCSI 通信协议之间的映射）。驱动程序开发者必须实现 SCSI 抽象和物理数据线之间的映射，这种映射依赖于 SCSI 控制器，而与连接到 SCSI 数据线上的设备无关。

最近还有其它类型的设备驱动程序加入内核，例如 USB 驱动程序、FireWire 驱动程序和 I2O 驱动程序等。和处理 SCSI 驱动程序的方法一样，内核开发者实现整个设备类型的共有特性，然后提供给驱动程序实现者，从而避免了重复工作以及 bug，简化并增强了编写这些驱动程序的过程。

除了驱动程序外，内核中其它一些功能，不管是硬件还是软件功能，都模块化了。文件系统可能是除驱动程序外 Linux 系统中最重要模块类型，它决定了信息如何在块设备上组织，以表示目录和文件树。文件系统并不是设备驱动程序，因为没有任何实际物理设备同这种信息组织方式相关联。相反，文件系统类型是个软件驱动程序，它将低层数据结构映射到高层数据结构，决定文件名可以有多长以及在目录项中存储文件的哪些信息等等。文件系统模块必须实现访问目录和文件的底层系统调用，方法是将文件名和路径（以及其它一些信息，比如访问模式等）映射到位于数据块上的数据结构中。这种接口完全独立于磁盘（或其它介质）上的数据读写操作，这种操作由块设备驱动程序负责完成。

由于 Unix 系统严重依赖于底层的文件系统，因此文件系统概念对系统操作具有重要意义。访问文件系统的功能位于内核层次结构的最底层，具有非常重要的作用。如果我们想为一款新的 CD-ROM 编写块驱动程序，则必须提供对 CD-ROM 上包含的数据进行 `ls` 或 `cp` 等操作的功能，否则驱动程序毫无用处。Linux 支持文件系统模块的概念，它的软件接口声明了可以在文件系统节点、目录、文件以及超级块上执行的不同操作。不过，程序员需要自己编写文件系统模块的情况比较少见，因为正式发行的内核版本中已经包含了最重要文件系统类型的代码。

1.4 安全问题

安全问题日益引起人们的关注，本书在适当的时候都会讨论这一问题。然而，有必要现在就弄清楚几个原则性的概念。

安全问题分为偶然性的和故意的两类。前者是由于用户不正确使用现有程序或者不小心使用了程序中的 bug 而造成的破坏；后者则是由于程序员故意实现的某些带有恶意功能的程序而带来的安全问题，这种程序员通常比一般用户拥有更多的特权。因此，如果我们运行的程序是从具有 root 帐户的第三方获得的，它的危险性等同于直接给第三方一个 root 命令解释器。另外，尽管拥有访问编译器的权限本身并不是一个安全漏洞，然而当执行由这个编译器编译的代码时就可能出现安全漏洞。由于内核模块可以执行任何操作，它和超级用户命令解释器一样强大，所以编写模块时我们应

* SCSI 是“Small Computer Systems Interface”的缩写，它是工作站和高端服务器领域事实上的标准。

当倍加小心。

系统中的所有安全检查都是由内核代码进行的，如果内核有安全漏洞，则整个系统就会有安全漏洞。在正式发行的内核版本中，只有授权用户才能装载模块，系统调用 `create_module` 检查调用进程是否具有装载模块的权利。因此，运行正式发行的内核版本时，只有超级用户^{*}，或者成功成为超级用户的入侵者，才能使用特权代码。

驱动程序编写者应当尽量避免在代码中实现安全策略，它最好在系统管理员控制之下，由内核的高层来实现。当然也会有例外，作为驱动程序编写者，我们应当清楚有时候某些设备访问操作能够影响整个系统，因此应该严格控制。例如，能够影响全局资源（比如设置中断线）的设备操作，或者影响其它用户（比如给磁带驱动器设置块尺寸的缺省值）的设备操作，通常只能由特权用户执行，并且只能由驱动程序本身才能检查用户的权限。

当然，驱动程序编写者应当避免由于自身原因引入安全方面的 bug。C 语言很容易产生几种类型的错误，比如缓冲区溢出就会导致许多安全问题。缓冲区溢出通常是由于程序员忘记检查缓冲区中已写了多少数据，导致数据写到了缓冲区边界之外，覆盖了系统中其它数据。这种错误可能破坏整个系统，因此必须尽量避免。幸运的是，在驱动程序环境中避免这种错误通常相对容易，因为此时用户接口比较有限而且严格控制。

还有其它一些原则性的安全概念值得注意。任何从用户进程得到的输入只有经过内核严格验证后才能使用，内核内存在分配给用户进程和设备之前必须清零或者以其它方式初始化，否则就会发生信息泄漏。如果设备能够解释它从内核获得的数据，则确保它不能输出任何可能损害系统的内容。最后，我们还应当考虑设备操作可能造成的影响，如果某些操作（比如重新装载适配卡上的固件或者格式化磁盘）能够影响整个系统，则它应当仅限于特权用户使用。

应当小心使用从第三方获得的软件，特别是与内核相关时更是如此。这是因为源码是开放的，每个人都可以修改和重新编译它。通常我们可以信任发行版本中预先编译的内核，但当使用由一个我们不是非常熟悉的朋友编译的内核时就得当心。如果我们不想以 `root` 身份运行一个预先编译过的二进制文件，则也不应当运行一个预先编译的内核。一个恶意修改过的内核允许任何人装载模块，因此通过 `create_module` 开了一扇后门。

Linux 内核也可编译为不支持模块方式，因而关闭了任何相关的安全漏洞，但这种情况下驱动程序需要直接嵌入内核。2.2 及以后的内核版本还可以通过权能机制禁止内核在系统启动后装载模块。

1.5 版本编号

在深入探讨编程之前，我们希望探讨一下 Linux 使用的版本编号机制以及本书中所讲到的内核版本。

首先，Linux 的每个软件包都有自己的发行版本号，并且它们之间存在相互依赖关系，例如只有存在某个软件包的某个特定版本时才能运行另一个软件包的某个特定版本。通常 Linux 正式发行版本已经解决了复杂的包匹配问题，此时安装系统不需要我们自己处理版本号。然而如果我们需要自

^{*} 内核 2.0 版本只允许超级用户运行特权代码，而在内核 2.2 版本中具有更复杂的权能检查方法。在第 5 章“权能

已替换或者更新系统中某个软件包则另当别论。幸运的是，现在几乎所有的发行版本中都带有包管理器，它在验证包之间的依赖关系满足后才允许升级包。

本书中的某些示例代码需要特定的某些内核版本才能运行，除此之外，它对其它工具则没有版本要求。任何最近发行的 Linux 版本都可以运行我们的例子。对内核版本要求的具体细节本文没有描述，读者遇到任何问题时可参考内核源文件 `Documentation/Changes`。

偶数编号的内核版本(如 `2.2.x` 和 `2.4.x`)是用于正式发行的稳定版本，而奇数编号的版本(如 `2.3.x`)则是开发过程中的一个快照，它将很快被下一开发版本更新。最新的开发版本只是代表了内核开发目前的状态，几天后可能会过时。

本书中讲到了从 `2.0` 到 `2.4` 的各个版本。不过我们的注意力主要集中在内核 `2.4`（写本书时的最新稳定版本）给设备驱动程序编写者所提供的各种特性上，但也将尽可能地全面介绍 `2.2` 内核区别于 `2.4` 内核的不同之处。我们还会讲到内核 `2.0` 版本同 `2.2` 和 `2.4` 内核相比所缺乏的一些特性以及一些替代办法。总之，本书中的代码可以在内核的一系列版本中运行，在 `2.4.4` 内核版本中全部测试通过，在 `2.2.18` 和 `2.0.38` 内核版本中也测试过部分示例程序。

本书很少讨论到奇数编号的内核版本，普通用户也很少会有使用这种版本的需求。然而，如果我们希望了解、跟踪开发版本的新特性，就需要运行最近发行的开发版本。而且还得随着开发版本的更新，不断获取 `bug` 的补丁以及新实现的特性。对于开发版本，我们必须记住它没有任何责任担保*。而且如果我们碰到的问题是由于老版本奇数编号的内核引起的，则没有人可以求助。那些运行奇数编号内核版本的程序员通常具有足够的知识，无需求助教科书就可以自己钻研内核代码。这也是我们为什么不在这儿讨论内核开发版本的另外一个原因。

Linux 的另外一个特性是它是一个平台独立的操作系统，不再仅仅是“PC 克隆上的 Unix 克隆”。它现在已经成功地应用在 `Alpha` 和 `SPARC` 处理器上，以及 `68000`、`PowerPC` 和其它几个平台上。本书尽可能做到与平台无关，所有示例代码都在几个平台上测试过，包括 `PC`、`Alpha`、`ARM`、`IA-64`、`M68k`、`PowerPC`、`SPARC`、`APARC64` 以及 `VR41xx(MIPS)` 等。本书所有示例代码在 `32` 位和 `64` 位处理器上都测试过，在其它平台上基本都能编译运行。然而，如果示例代码依赖于特定硬件，则不能在所有支持的平台上都正常工作，此时在源码中我们会特别声明这一点。

1.6 许可证条款

Linux 受 GNU 通用公共许可证 (GPL) 保护。GPL 由自由软件基金会为 GNU 项目而设计，它允许任何人重新发行甚至销售由 GPL 条款限制的产品，前提是产品接收者能够从源码中重新构建一个同样的二进制副本。另外，任何从 GPL 保护的产品中派生出来的软件产品，也必须随 GPL 条款一起才能发行。

这样一个许可证的主要目的是通过允许每个人自由修改程序来实现知识增长；同时，向公众卖软件的人仍旧可以获利。但就是这样目的简单的条款，关于 GPL 及其使用仍然存在着无休止的争论。如果读者想去读一下这个许可证原文，可以在系统中几个地方找到它，包括目录 `/usr/src/linux` 中的 `COPYING` 文件。

和受限操作”中我们将具体讨论这种方法。

第三方模块和定制的模块不是 Linux 内核的一部分，因此它们可以不受 GPL 条款的限制。模块通过一个良好的接口使用内核，但不是内核的一部分，这同用户程序通过系统调用使用内核的方式类似。记住免受 GPL 条款限制仅适用于那些只使用了公开发布的模块接口的模块。深深嵌入内核的那些模块应当遵守 GPL 中关于“派生工作”的条款。

总之，如果我们的代码嵌入内核内部，则发布代码时就得立即应用 GPL 条款。如果出于个人使用目的，则不必强制执行 GPL，但如果正式发布代码就必须包含源码，使获得我们软件包的用户能够自由的重建二进制代码。另一方面，如果我们写了一个模块，我们可以只发布它的二进制形式。然而，现实中这样做并不总是行的通。因为通常对于每个与之连接的内核版本（第 2 章“版本依赖”和第 11 章“模块中的版本控制”中解释），模块都需要重新编译。新的内核发布版本，即使是很小的稳定发布，也经常会破坏编译的模块，因此需要重新编译模块。Linus Torvalds 曾公开表示他认为这种方式没有什么问题，因为二进制模块应该只工作在它编译时所使用的内核版本上。然而作为模块编写者，提供源代码通常能够更好的服务用户。

就本书而言，不管是源代码还是二进制形式，大部分代码都可以免费重新发布，并且不管是作者还是 O'Reilly & Associates 出版社，都不会对任何派生的产品保留任何权利。所有程序都可以从 <ftp://ftp.ora.com/pub/examples/linux/drivers/> 中得到，许可证条款在同一目录下的 LICENSE 文件中表述。

当示例程序包含了部分内核代码时，此时适用 GPL 条款，源文件中的注释已经非常明确地阐明了这一点。这仅仅发生在与本书主题关系不大的几个源文件中。

1.7 加入内核开发社团

当我们为 Linux 内核编写模块的时候，我们就成为一个巨大开发人员社团中的一员。在这个社团中，我们不仅发现很多人从事类似的工作，而且发现一帮具有高度使命感的工程师正朝着将 Linux 发展成为一个更好系统的目标前进。这些人是我们获得帮助、思路以及严格评价的源泉。当我们为新的驱动程序寻找测试者的时候，他们将是乐于提交的第一批人。

Linux-kernel 邮件列表是 Linux 内核开发者的聚集中心。从 Linus Torvalds 往下，所有主要内核开发者都订阅这个邮件列表。请注意这个列表不适合那些心脏比较脆弱的人：该邮件列表每天都有 200 条消息或者更多。然而，对于那些对内核开发感兴趣的人来说，跟踪这个列表是必要的，对于那些需要内核开发帮助的人来讲，它更是一个顶级质量的资源。

要加入 Linux-kernel 列表，遵照 linux-kernel 邮件列表 FAQ：<http://www.tux.org/lkml> 中的指示即可。如果已经打开了这个 FAQ，我们还应当看看它的其它内容，它上面有大量的有用信息。Linux 内核开发者都比较忙，他们更愿意帮助那些首先了解了基本知识的人。

1.8 本书概要

从第 2 章起，我们将进入内核编程领域。第 2 章介绍了模块化技术，解释了其实现技巧，并讲解

* 注意即使对于偶数编号的内核版本，同样没有任何责任担保。只有商业版本发行公司才会对他们的产品提供担保。

了运行模块的代码。第 3 章讨论字符驱动程序，给出了一个基于内存的设备驱动程序的完整代码。将内存作为设备硬件，可允许任何人在无需特殊硬件的情况下，运行我们的示例代码。

对程序员来讲，调试技术是很重要的工具，我们将在第 4 章介绍内核调试技术。随后，带着新的调试技巧，我们转到字符驱动程序的高级特性，比如阻塞操作、`select` 使用以及重要的 `ioctl` 调用等，这些都是第 5 章的内容。

在讨论硬件管理之前，我们先剖析内核的几个软件接口：第 6 章讨论内核的时间管理，第 7 章讨论内存分配。

接下来我们集中于硬件问题。第 8 章描述 I/O 端口管理和设备上内存缓冲区的管理。之后，我们在第 9 章讨论中断处理。不幸的是，并不是所有人都愿意运行这些章节中的示例代码，因为需要一些硬件支持才能测试中断的软件接口。我们尽可能使必需的硬件支持减到最小，但仍然需要自己手工建造一些硬件“设备”。这个设备就是一个插到并口上的简单跳线，因此我们希望这不是一个问题。

第 10 章提供了一些有关编写内核软件以及可移植性问题的建议。

在本书的第二部分，我们将探讨更深层次的内容。因此，第 11 章将再次讨论模块化问题，只不过这次更加深入一些。

接着第 12 章介绍块设备驱动程序的实现方法，概述了它们区别于字符设备的不同之处。在那之后，第 13 章讲述了我们在前面讨论内存管理时遗留的内容：`mmap` 和直接内存访问（DMA）。这个时候，关于字符和块设备驱动程序的所有要点都已讲述清楚。

接下来介绍驱动程序的第三个主要类型。第 14 章介绍了关于网络接口的一些细节并且解剖了一个网络驱动程序例子的代码。

驱动程序的有几个特性直接依赖于外围设备所连接的接口总线。因此第 15 章中介绍了现在流行的几款总线的主要特性，特别重点讨论了内核提供的 `PCI` 和 `USB` 支持。

最后，第 16 章是对内核源码的一个概览，它希望给下面的一种人提供一个学习的起点：希望了解内核整体设计，但又惧怕庞大的内核源码。

第 2 章 构造和运行模块



非常高兴现在终于可以开始编程了。本章将介绍所有关于模块编程和内核编程的必要概念，并用有限的篇幅构建和运行一个完整的模块。掌握这种技能是编写任何驱动程序模块的基础。为了避免一次引入太多概念，本章将只讨论模块，而避免涉及任何特定类型的设备。

本章中引入的所有内核条目（函数、变量、头文件和宏）在本章末尾的“快速参考”一节会集中描述。

我们要讨论的第一个模块，其实是一个完整的“Hello, World”模块（这个模块实际上并没有任何特别的功能），它可以在 Linux 2.0 到 2.4 的各个内核版本中编译、运行。^{*}

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

int init_module(void) { printk("<1>Hello, world\n"); return 0; }
void cleanup_module(void) { printk("<1>Goodbye cruel world\n"); }
```

函数 `printk` 在 Linux 内核中定义，功能和标准 C 库中的函数 `printf` 类似。内核需要自己单独的打印输出函数是因为它在运行时不能依赖于 C 库。模块能够调用 `printk` 是因为在 `insmod` 函数装入模块后，模块就连接到内核，因而可以访问内核的公用符号（包括函数和变量，下一节详述）。代码中的字符串 `<1>` 定义了这条消息的优先级。我们需要在模块代码中显式指定高优先级（小级别编号）的原因在于：具有默认优先级的消息可能不会输出在控制台上，这依赖于内核版本、`klogd` 守护进程的版本以及具体的配置。读者可以暂时忽略这个问题，我们将在第 4 章“`printk`”一节中仔细阐述。

如下面的命令行以及屏幕输出所示，读者可以通过调用函数 `insmod` 和 `rmmod` 来测试模块。值得注意的是只有超级用户才有权装入和卸载模块。

要想成功装入和卸载上面的模块，就必须禁止内核的模块版本控制功能。然而，大多数 Linux 发行版本所预先安装的内核都具有版本控制功能（在第 11 章“模块中的版本控制”中将会讲到版本控制功能）。虽然老版本的 `modutils` 允许将不支持版本控制功能的模块装入支持版本控制功能的内核，新版本的 `modutils` 却不再支持这一功能。为了解决上面的 `hello.c` 遇到的这个问题，我们在示例程序 `misc-modules` 目录中的源文件中增加几行代码，使它在支持和不支持版本控制功能

的内核中都可以运行。尽管这样，我们仍然强烈建议读者在运行示例代码前将自己的内核编译为不支持版本控制功能。^{*}

```
root# gcc -c hello.c
root# insmod ./hello.o
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

根据系统传递消息行机制的不同，我们自己得到的输出结果可能不一样。需要特别指出的是，上面的屏幕输出是在字符终端上得到的，如果在 **xterm** 上运行 **insmod** 和 **rmmod**，则不会在 **xterm** 的 TTY 上看到任何输出。实际上，它可能输出到某个系统日志文件里，比如 **/var/log/messages**（实际的名称随 **Linux** 发行版的不同可能会有所变化）。内核消息的传递机制将在第 4 章“消息是如何记录的”中详细讨论。

我们已经看到，编写一个模块并没有想象的那么困难，困难在于理解设备并优化其性能。本章我们将深入讨论模块问题，而把设备相关的问题留到以后的章节。

2.1 核心模块与应用程序的对比

在进一步讨论之前，有必要搞清楚内核模块和应用程序之间的种种不同之处。

尽管应用程序是从头到尾执行单个任务，而模块却只是预先注册自己以便服务于将来的某个请求，然后，它的“**main**”函数就立即结束。换句话说，函数 **init_module**（模块的入口）的任务是为以后调用模块函数预先做准备；这就像模块在说，“我在这儿，并且我能做这些工作。”模块的第二个入口点，**cleanup_module**，在模块即将卸载之前调用。它告诉内核，“我要离开啦，不要再让我做任何事情。”能够卸载模块可能是模块化驱动程序编程当中，读者最为喜欢的一个特色，因为它帮助缩短模块的开发周期：我们可以测试新驱动的一序列版本却不需要每次都经过冗长的关机/重启过程。

作为程序员，我们知道应用程序可以调用它并未定义的函数，这是因为连接过程能够解析外部引用从而使用适当的函数库。例如，定义在 **libc** 中的 **printf** 函数，就是这种可被调用的函数之一。然而，模块仅仅被连接到内核，因此它能调用的函数仅仅是由内核导出的那些函数，而没有任何可连接的库。例如，前面 **hello.c** 中使用的 **printk** 函数，就是由内核定义并导出给模块使用的一个 **printf** 的内核版本。除了几个细小差别外，它和 **printf** 函数功能类似，最大的不同在于它缺乏对浮点数的支持^{*}。

图 2-1 展示了如何在模块中使用函数调用和函数指针为运行中的内核增加新功能。

^{*} 这个例子以及本书中的其它例子都可以从第 1 章中提到的 O'Reilly FTP 网站获得。

^{*} 如果读者还不知道怎样构造内核，我们建议你首先阅读 Alessandro 在 <http://www.linux.it/kerneldocs/kconf> 上发表的一篇文章，它对初学者很有帮助。

^{*} 在 **Linux 2.0** 和 **Linux 2.2** 中 **printk** 函数不支持 **L** 和 **Z** 限定符，**Linux 2.4** 中才首次增加支持。

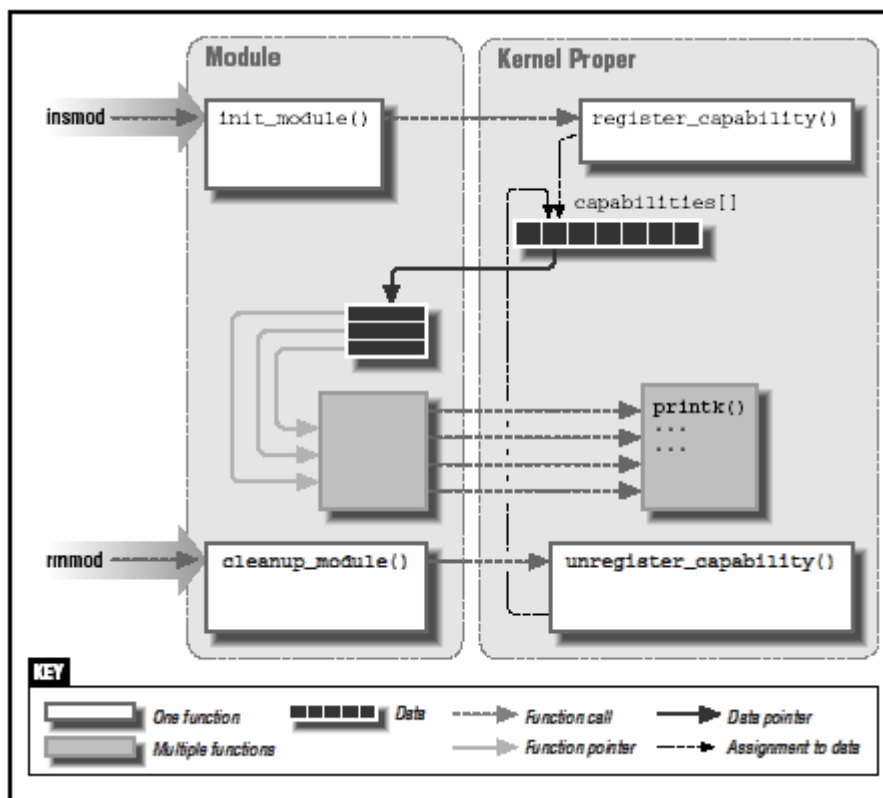


图 2-1: 将模块连接到内核

因为模块不和函数库连接，因此在源文件中不能包含通常的头文件。内核模块只能使用作为内核一部分的函数。和内核相关的任何内容都声明在内核源码(通常位于`/usr/src/linux`)的 `include/linux` 和 `include/asm` 目录下的头文件里。老的发行版(基于 `libc` 版本 5 或者更早)中，通常利用符号链接将 `/usr/include/linux` 和 `/usr/include/asm` 指向实际的内核源代码，因而 `libc` 的头文件树总是能够指向实际安装的内核源代码的头文件。通过这些符号链接，用户空间的应用程序能够很方便地引用内核头文件。它们偶尔确实需要这样做。

尽管现在用户空间头文件和核心空间头文件是分离的，有时应用程序仍然需要包含内核头文件。比如说在使用一个老版本库时，或者需要一些无法从用户空间头文件中获得的新信息时等等。然而，内核头文件中的许多声明仅仅与内核本身相关，不应暴露给用户空间的应用程序。因此，我们用 `#ifdef __KERNEL__` 块来保护这些声明，这就是为什么驱动程序要象其它内核代码一样，必须在定义了预处理符号 `__KERNEL__` 的情况下编译。

每个内核头文件的作用将在本书中需要用到它们的时候加以介绍。

开发大型软件系统(比如内核)的程序员必须注意到并且尽力避免“名字空间污染”。当存在大量的函数和全局变量，并且它们的名字没有明确的含义以至于很难区别时，就会发生所谓的名字空间污染。当不得不面对这样的一个系统时，程序员需要花费更多的精力去记住这些已经“保留”的名字并且为新符号寻找一个不重复的名字。名字空间冲突可能造成很多问题，例如模块装载失败，或者一些古怪的问题，比如它可能只发生在使用代码的远程用户身上，而这个用户仅仅使用了一个不同的配置选项集来编译内核。

在编写内核代码时，这种错误将是开发人员的恶梦，这是因为即使最小的模块也将要连接到整个内核。防止名字空间污染的最好办法是将所有符号定义为静态变量，并且在表示全局变量的符号前加上一个内核中唯一的前缀。还应该注意的是，作为一个模块编写者，我们可以控制是否导出符号被外部使用，这一点将在本章后面“内核符号表”中讲到*。

为模块中的私有符号选择前缀也将是一个不错的习惯，因为这样可以简化调试。在测试驱动程序的时候，就可以导出所有符号而不会污染名字空间。根据惯例，内核中使用的前缀都是小写的，我们将遵循这个惯例。

内核编程和应用程序编程的最后一点不同之处在于各环境下处理错误的方式不同：应用程序开发过程中段错误是无害的，并且总是可以使用调试器跟踪到源代码中的问题所在，而一个内核错误即使不对整个系统是致命的，也至少会对当前进程造成致命错误。在第 4 章“调试系统错误”一节中，我们将看到如何跟踪内核错误。

2.1.1 用户空间和内核空间

模块运行在所谓的内核空间里，而应用程序运行在所谓的用户空间中。这个概念是操作系统理论的基础之一。

实际上，操作系统的作用是为应用程序提供一个对计算机硬件的一致视图。除此之外，操作系统必须负责程序的独立操作以及保护资源不受非法访问。这个重要任务只有在 CPU 能够保护系统软件不受应用程序破坏时才能完成。

所有的现代处理器都具备这个功能。人们选择的方法是在 CPU 中实现不同的操作模式（或者级别）。不同的级具有不同功能，在较低的级别中将禁止某些操作。程序代码只能通过有限数目的“门”来从一级切换到另一级。Unix 系统设计时利用了这些硬件特性，使用了两个这样的级。当前所有的处理器都至少具有两个保护级，而象其它的一些处理器，比如 x86 系列，则有更多的级。当处理器存在多个级时，Unix 使用了最高级和最低级。在 Unix 当中，内核运行在最高级（也称作管理员态），在这级中可以进行所有操作。而应用运行在最低级（即所谓的用户态），在这级当中处理器控制着对硬件的直接访问以及对内存的非授权访问。

我们通常将运行模式称做内核空间和用户空间。这两个术语不仅说明两种模式具有不同的优先权等级，而且还说明每个模式都有自己的内存映射，也即自己的地址空间。

每当应用程序执行系统调用或者被硬件中断挂起时，Unix 将执行模式从用户空间切换到内核空间。执行系统调用的内核代码运行在进程上下文中，它代表调用进程执行操作，因此能够访问进程地址空间的所有数据；而处理硬件中断的内核代码和进程是异步的，与任何一个特定进程无关。

模块运行在内核空间中，扩展内核的功能。通常一个驱动程序模块既要执行系统调用，也要进行中断处理。

* 如果模块中没有特定的指令，则大多数版本的 `insmod`（但不是所有）导出所有非静态符号。因此，如果我们不想导出某些符号的话，则应该将其声明为静态变量。

2.1.2 内核中的并发

驱动程序编程区别于大部分应用程序编程的一个重要方面是并发问题的处理。应用程序通常从头到尾顺序执行，不必担心发生其它情况而改变它的运行环境。而内核代码却不会运行在这样一个简单环境中，它必须考虑到可能会同时发生很多事情。

有几方面的原因促使内核编程必须考虑并发问题。首先，Linux 系统中通常正在运行多个并发进程，并且可能有多于一个的进程同时使用我们的驱动程序。其次，大多数设备能够中断处理器，而中断处理程序异步运行，而且可能在我们的驱动程序正试图处理其他任务时被调用。另外，还有几个软件抽象（比如第 6 章中谈到的内核定时器）也异步运行。最后，Linux 还可以运行在对称多处理器（SMP）系统上，因此可能同时有不止一个 CPU 运行我们的驱动程序。

结果，Linux 内核代码，包括驱动程序代码，必须是可重入的，它必须能够同时运行在多个上下文中。因此，内核数据结构需要精心设计才能保证多个线程分开执行，代码访问共享数据时也必须避免破坏共享数据。要编写能够处理并发问题同时可避免竞态（不同的执行顺序导致不同的，非预期行为发生的情况）的代码，需要一些技巧和细致的思考。本书中的示例驱动程序在编写时都考虑到了并发问题，在讲到时我们将具体介绍所使用的技术。

驱动程序编写人员所犯的一个常见错误是认为只要某段代码没有进入睡眠状态（或者阻塞），就不会产生并发问题。的确，在过去大多数时间里，Linux 内核是非抢占式的（一个重要的例外是对中断的服务，它不会从不愿放弃处理器的内核代码中抢占处理器）。过去，这种非抢占式行为可避免大部分意想不到的并发问题，然而，在 SMP 系统中，不需要抢占性就可以导致并发执行。

如果我们编写的代码基于系统不是抢占式的这一假设，它就不能在 SMP 系统中正确运行。即使我们自己没有 SMP 系统，其他运行我们程序的人也可能拥有 SMP 系统。而且，将来内核或许会转变为抢占式运行，那么即使是单处理器系统也必须随时随地处理并发问题（一些内核变种已经实现了抢占式）。因此，一个谨慎的程序员应该总是假设他的程序运行在 SMP 系统中。

2.1.3 当前进程

虽然内核模块不象应用程序那样顺序执行，然而内核当前执行的大多数操作还是和某个特定的进程相关，这个特定进程就是当前进程。内核通过访问全局变量 `current` 获得当前进程。`current` 是一个指向结构 `task_struct` 的指针，在内核 2.4 中 `task_struct` 在 `<asm/current.h>` 中定义，并包含在 `<linux/sched.h>` 头文件中。`current` 指针指向当前正在运行的用户进程，在 `open`、`read` 等系统调用的执行过程中，当前进程指的是调用这些系统调用的进程。如果需要，内核代码可以通过 `current` 获得与当前进程相关的信息，在第 5 章“设备文件的访问控制”中将会介绍这样一个例子。

实际上，与早期 Linux 内核版本不同，最近版本的内核中 `current` 不再是一个全局变量。内核开发者将描述当前进程的结构隐藏在栈页（stack page）中，从而优化了对该结构的访问。读者可以查阅 `<asm/current.h>` 获得 `current` 的详细细节，尽管这些代码可能看起来有些凌乱，我们必须记住 Linux 系统是一个 SMP 兼容的系统，全局变量在处理多 CPU 系统时并不能正常工作。然而 `current` 的具体实现细节对内核其它子系统是透明的，驱动程序可以包含头文件 `<linux/sched.h>` 来引用当前进程。

从模块的角度来看，`current` 和 `printk` 一样，都是外部引用。模块在需要时总是可以引用 `current`。例如，下面的语句通过访问 `task_struct` 结构的某些成员，打印当前进程的进程 ID 和命令名：

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

存储在 `current->comm` 成员中的命令名是当前进程所执行的程序文件的基本名称（base name）。

2.2 编译和装载

本章余下内容将讨论编写一个完整的，然而没有类型的驱动程序模块。也就是说，它不属于第 1 章“设备和模块类型”中列出的任何一个模块类型。我们把这个示例模块称作 `skull`，即“Simple Kernel Utility for Loading Localities”的缩写。在去掉它的示例功能后，我们可以重用这个模块，装载自己的本地代码到内核*。

在讨论 `init_module` 和 `cleanup_module` 作用之前，我们先编写一个 `makefile` 文件，用来构造可以装入内核的目标代码。

首先，在包含任何头文件之前，我们必须在预处理器中定义 `__KERNEL__` 符号。如前所述，没有这个符号，模块不能使用内核头文件中针对内核的特殊内容。

另外一个重要的符号是 `MODULE`，它必须定义在包含 `<linux/module.h>` 之前（除非驱动程序直接嵌入内核）。本书中将不介绍直接连接入的模块，因而我们的示例程序中总是定义 `MODULE`。

如果是为一个 `SMP` 系统编译模块，我们还需要在包含内核头文件之前定义符号 `__SMP__`。在版本 2.2 中，“多处理器或单处理器”是一个内核配置选项，因此在模块起始部分加上下面几行预处理代码，就可以为多处理器系统定义符号 `__SMP__`：

```
#include <linux/config.h>
#ifdef CONFIG_SMP
# define __SMP__
#endif
```

因为许多函数在头文件中定义为内嵌函数，所以模块编写者还需要对编译器指定 `-O` 选项。如果不使用此优化选项，`gcc` 就不会展开内嵌函数。`gcc` 可以同时使用 `-g` 和 `-O` 编译选项，允许我们对调试代码使用内嵌函数。*

因为内核中广泛使用内嵌函数，因此正确展开这些内嵌函数非常重要。

我们还需要参阅内核源码树中的 `Documentation/Changes` 文件，检查编译所模块使用的编译器和内核使用的编译器是否匹配。尽管由不同的小组开发，但内核和编译器的开发工作却是同时进行的，因此，它们其中之一改变了，另一种也得作出相应的调整，否则可能会引出问题。有些 Linux 发

* 我们在这里使用“本地（local）”一词，表示是个人对系统作出的修改。这种用法借鉴了 Unix 中 `/usr/local` 的用法。

* 然而，注意不要使用比 `-O2` 更高的优化级别。如果使用了这个选项，编译器将会把那些没有声明为内嵌函数的函数当作内嵌函数来编译，这对内核代码将是一个问题，因为一些函数在被调用时需要使用一个标准的栈布局。

行版本所带的编译工具同内核相比比较新，因此不能用来编译内核模块。在这种情况下，Linux 发行版本中通常带有一个专门用来编译内核的编译器软件包（通常叫做 **kgcc**）。

最后，为了避免一些令人讨厌的错误，我们建议使用 **-Wall**（显示所有警告信息）编译选项，并且修改代码中所有能引起编译警告的编程习惯，即使这样做会改变我们一贯的编程风格。当编写内核代码时，我们应该借鉴学习 **Linus** 本人的编程风格；如果我们还有志于研究内核代码，则必须首先阅读 **Documentation/CodingStyle**，以了解内核的编程风格。

到目前为止，我们所介绍的符号定义和编译选项都包含在 **make** 命令使用的变量 **CFLAGS** 中。

除了需要合适的 **CFLAGS** 外，当模块源代码由几个源文件组成时，**makefile** 文件还需要一条规则来连接多个目标文件。现实情况下模块大多由多个源文件组成，连接多个目标文件的命令是 **ld -r**，虽然它使用了连接器，但它并没有执行真正的连接操作。它集中所有输入的目标文件的目标代码，输出另一个目标文件。选项 **-r** 的含义是“可重定位”：输出的目标文件中没有包含任何绝对地址，因而是可重定位的。

下面的 **makefile** 文件展示了如何构造一个由两个源文件组成的模块。如果读者的模块仅由一个源文件组成，则只需删去含有 **ld -r** 的那条规则即可。

```
# Change it here or specify it on the "make" command line
KERNELDIR = /usr/src/linux

include $(KERNELDIR)/.config

CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include \
        -O -Wall

ifdef CONFIG_SMP
    CFLAGS += -D__SMP__ -DSMP
endif

all: skull.o

skull.o: skull_init.o skull_clean.o
    $(LD) -r $^ -o $@

clean:
```

若读者不熟悉 **make** 命令，可能疑惑为什么上面的 **makefile** 文件中没有 **.c** 文件和编译规则。其实，这些声明是不必要的，**make** 命令能够自动将 **.c** 文件编译成 **.o** 文件，并且使用当前（或默认）的编译器 **\$(CC)** 以及编译标记 **\$(CFLAGS)**。

在构造模块之后，下一步就是装入内核。如前所述，**insmod** 为我们完成这项工作。**insmod** 程序和 **ld** 有些类似，它使用运行内核的符号表解析模块中任何未解析的符号。然而，它和连接器还是有些区别，因为它没有修改磁盘文件，而仅仅修改了内存中的副本。**insmod** 可以接受一些命令行选项（参见它的手册页），并且可以在模块连接到内核之前给模块中的整型和字符串型变量赋值。因此，一个良好设计的模块可以在装载时进行配置，这比编译时的配置为用户提供了更多的灵活性，然而，有些情况下仍然要使用编译时的配置。本章后面的“自动和手动配置”一节中将会介绍装载时配置。

感兴趣的读者可能想知道内核是如何支持 **insmod** 工作的，实际上它依赖于定义在

`kernel/module.c` 中的几个系统调用。函数 `sys_create_module` 给模块分配内核内存（函数 `vmalloc` 负责内核内存分配，详见第 7 章的“`vmalloc` 及其相关函数”）。系统调用 `get_kernel_syms` 返回内核符号表，用来解析模块中的内核引用。`sys_init_module` 拷贝可重定位的模块目标代码到内核空间并且调用模块的初始化函数。

如果仔细阅读内核源码，我们会发现有且只有系统调用的名字前带有 `sys_` 前缀，而其它任何函数都没有这个前缀。这种命名上的区别使我们在源码中 `grep` 系统调用时非常方便。

2.2.1 版本依赖

当模块需要和不同版本的内核连接时，模块代码就需要重新编译。每个模块都定义了符号 `__module_kernel_version`，`insmod` 使用它检查模块和当前的内核版本是否匹配。这个符号位于 ELF 的 `.modinfo` 段，第 11 章中将会详细讲到。值得注意的是，这种检查版本匹配的机制仅仅适用内核 2.2 和 2.4，内核 2.0 使用不同的方式实现了同样的目标。

只要包含头文件 `<linux/module.h>`，编译器就会自动为我们定义这个符号。（这就是为什么前面的 `hello.c` 没有显式定义此符号的原因。）这也意味着如果模块由多个源文件组成，则我们只需在一个源文件中包含 `<linux/module.h>` 即可。（除非使用了 `__NO_VERSION__`，待会儿我们将会讲到这一点。）

当模块和内核版本不匹配时，我们仍然可以通过指定 `insmod` 的 `-f`（“force，强制”）选项来强行装入模块。然而这种操作是不安全的，可能失败，并且很难预测将会发生什么样的后果。例如，装载过程可能出现符号不匹配现象，此时我们将得到一条出错消息；也可能由于内核内部发生了变化，从而导致出现严重错误甚至系统 `panic`，这也是我们力图避免版本不匹配的一个重要原因。版本不匹配问题可以通过使用内核的版本控制功能来更好地解决（版本控制功能是个更深层次的话题，将在第 11 章“模块中的版本控制”中介绍）。

如果读者想为一个特定版本的内核编译模块，则必须在上面的 `makefile` 文件中包含该版本内核特有的头文件（例如，可以声明一个不同的 `KERNELDIR`）。这种情况在和内核源码打交道时将会很常见，因为很多情况下我们可能具有多个版本的源码树。本书中的所有示例模块都使用 `KERNELDIR` 变量指向实际的内核源码，它可以在环境变量中设置或者通过 `make` 命令行指定。

在装入模块时，`insmod` 按照自己的搜索路径寻找模块的目标代码，它在 `/lib/modules` 目录下与内核版本相关的子目录中寻找。老版本的 `insmod` 将首先在当前目录下寻找目标代码，而由于安全方面的原因这一功能现在不再支持（`PATH` 环境变量中也存在类似的情况）。因此，如果需要从当前目录中装载模块，则应该使用 `./module.o`，它在 `insmod` 的所有版本中都能正常工作。

我们有时可能碰到在内核 2.0.x 和 2.4.x 之间表现不一致的内核接口。这种情况下，我们要求助于定义当前内核版本号的宏，它们在头文件 `<linux/version.h>` 中定义。为了简化 2.4 内核版本的讨论，我们将在本章里，或者在关于版本依赖的的本章末尾一节里，专门指出接口改变的情况。

自动包含在 `linux/module.h` 中的头文件定义了下面一些宏：

```
UTS_RELEASE
```

宏 `UTS_RELEASE` 扩展为一个描述内核版本的字符串，例如“2.3.48”。

LINUX_VERSION_CODE

宏 `LINUX_VERSION_CODE` 扩展为内核版本的二进制表示，版本发行号中的每一部分对应一个字节。例如，2.3.48 对应的 `LINUX_VERSION_CODE` 是 131888（即 0x020330）。*
因此，使用这个宏我们很容易确定正在使用的内核版本。

KERNEL_VERSION(major,minor,release)

宏 `KERNEL_VERSION` 以组成版本号三部分（三个整数）为参数，创建“`kernel_version_code`”。例如，`KERNEL_VERSION(2,3,48)` 扩展为 131888。这个宏在我们需要将当前版本和一个已知的检查点比较时非常有用，本书中将多次用到这个宏。

文件 `version.h` 包含在 `module.h` 中，因此我们通常不用显式包含 `version.h`。另一方面，如果我们预先定义 `__NO_VERSION__`，则 `module.h` 就不再包含 `version.h`。如果我们希望在组成单个模块的多个源文件中包含 `<linux/module.h>`，比如希望使用 `module.h` 中定义的预处理宏，这时，就需要在包含 `module.h` 之前定义 `__NO_VERSION__`。在包含 `module.h` 之前声明 `__NO_VERSION__`，可以避免在源文件中不需要的地方自动声明字符串 `__module_kernel_version` 或其他等价的符号（`ld -r` 将会给出符号多重定义的错误信息）。本书中的示例模块就是出于这个目的使用 `__NO_VERSION__`。

预处理条件语句使用 `KERNEL_VERSION` 和 `LINUX_VERSION_CODE` 能够解决大部分基于内核版本的依赖问题。然而，我们不应该胡乱使用 `#ifdef` 条件语句将整个驱动程序代码弄得杂乱无章。最好的一个解决方法就是将所有相关的预处理条件语句集中存放在一个特定的头文件里。我们的示例代码中就包含了这样一个头文件 `sysdep.h`，它用适当的宏定义隐藏了不兼容性。

我们碰到的第一个版本依赖问题是为驱动程序定义“`make install`”规则。我们能够猜到，驱动程序的安装目录将根据内核版本的不同而不同，因此需要查找 `version.h` 来确定合适的安装目录。下面的文件片段摘自 `Rules.make`，`Rules.make` 将包含在所有的 `makefile` 里面。

```
VERSIONFILE = $(INCLUDEDIR)/linux/version.h
VERSION = $(shell awk -F\" '/REL/ {print $$2}' $(VERSIONFILE))
INSTALLDIR = /lib/modules/$(VERSION)/misc
```

我们选择将驱动程序安装在 `misc` 目录中，这是添加杂项驱动程序的一个不错的地方，同时还能够避免 `/lib/modules` 下目录结构的改变带来的问题——这种变化在 2.4 版本内核发布之前刚刚引入。尽管新目录结构变得更加复杂，但新老版本的 `modutils` 包都会使用 `misc` 目录。

如上所示定义好 `INSTALLDIR` 后，每个 `makefile` 的安装规则如下所示：

```
install:
    install -d $(INSTALLDIR)
    install -c $(OBSJ) $(INSTALLDIR)
```

2.2.2 平台依赖

每种计算机体系结构都有自己的独特特性，内核设计者可以充分利用这些特性来达到目标平台上目标文件的最优性能。

* 因此，在两个稳定版本之间，最多可以存在 256 个开发版本。

对于应用程序开发人员，他们必须将程序代码和预编译过的库连接并且遵循参数传递规则。而内核开发人员则不同，他们可以根据不同需求，将某些寄存器指定为特定用途——实际上他们也确实这么做了。而且内核代码可以针对某个 CPU 家族的某种特定处理器进行优化，从而充分利用目标平台的特性。和应用程序以二进制的形式发布不同，内核需要发布源码，针对目标平台定制编译后才能达到对某个特定计算机集合的优化。

为了能够和内核互操作，模块代码在编译时需要使用和内核编译时一样的编译选项（例如，为特定用途保留同样的寄存器并且进行同样的优化）。因此，顶层的 `Rules.make` 包含一个与平台相关的文件，对 `makefile` 补充额外的定义。所有这些文件都叫做 `Makefile.platform`，并且根据当前的内核配置给 `make` 变量赋值。

`makefile` 文件的这种布局使得它能够支持所有示例文件的交叉编译。当我们为目标平台进行交叉编译的时候，需要另外一套编译工具（如 `m68k-linux-gcc`、`m68k-linux-ld` 等等）来代替现在的编译工具（比如 `gcc`、`ld` 等等）。交叉编译工具名字前使用的前缀由 `$(CROSS_COMPILE)` 表示，在 `make` 命令行或者环境变量中指定。

SPARC 结构需要 `makefile` 的特殊处理。运行在 SPARC64（SPARC V9）平台上的用户空间程序和运行在 SPARC32（SPARC V8）平台上的用户空间进程具有同样的二进制代码，因此，SPARC64 上运行的默认编译器（如 `gcc`）产生 SPARC32 平台的目标代码。然而，内核必须运行 SPARC V9 目标代码，因此需要一个交叉编译器。所有为 SPARC64 平台发布的 GNU/Linux 版本都包含有一个适当的交叉编译器，`makefile` 会选择这个交叉编译器。

版本和平台依赖性的问题清单可能比上述情况稍微复杂一些，然而上面的介绍以及所提供的 `makefile` 文件已经足以让我们继续进行后面的讨论。如果读者想了解更多的详细信息，可以参看 `makefile` 文件以及内核源码。

2.3 内核符号表

在上面的讨论中，我们了解了 `insmod` 使用公用内核符号表解析模块中未定义符号的原理。内核公用符号表中包含了所有的全局内核符号（即函数和变量）的地址，实现驱动程序模块时，在很多情况下都需要使用这些全局符号。公用符号表能够从文件 `/proc/ksyms` 中以文本格式读取（前提是内核支持 `/proc` 文件系统）。

当模块被装入内核后，它所导出的任何符号都变成公用符号表的一部分，在 `/proc/ksyms` 或者 `ksyms` 命令的输出中我们能够看到这些新增加的符号。

新模块可以使用我们模块导出的符号，而且我们还可以在其它模块上层叠新模块。模块层叠技术也使用在很多主流的内核代码中。例如 `msdos` 文件系统依赖于 `fat` 模块导出的符号；而每个 USB 输入设备模块层叠在 `usbcore` 和 `input` 模块之上。

模块层叠技术在复杂的项目中非常有用。如果以设备驱动程序的形式实现一个新的软件抽象，它可以为硬件相关的实现提供一个插接口（`plug`）。例如，视频驱动程序可以分出一个通用模块，它导出符号供下层与具体硬件相关的驱动程序使用。根据安装硬件的不同，我们加载通用视频模块以及

与具体硬件相关的模块。另外，并口支持，以及大量可挂接设备的处理，比如 USB 内核子系统，都使用了类似的层叠方法。图 2-2 中给出了并口子系统的层叠方式，箭头显示了模块之间（带有一些示例函数和数据结构）以及和内核编程接口之间的消息传输。

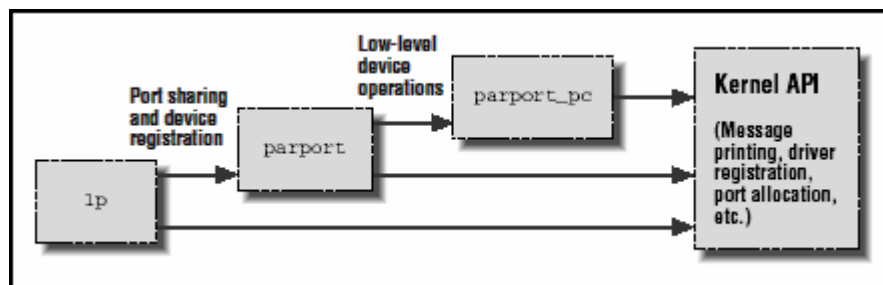


图 2-2: 并行子系统的层叠方式

`modprobe` 是处理层叠模块的一个实用工具。它的功能在很大程度上和 `insmod` 类似，但是它除了装入指定模块外，还同时装入指定模块所依赖的其它模块。因此，一个 `modprobe` 命令有时候相当于调用几次 `insmod` 命令（然而，在从当前目录装入模块时，仍然需要使用 `insmod`，因为 `modprobe` 只能从已安装的模块树中搜索需要装入的模块）。

通过对每一层进行简化，分层的模块化编程缩短了开发时间。这种方法和我们在第 1 章中提到的机制和策略的分离有点类似。

通常情况下，模块只需实现自己的功能，不必导出任何符号。然而，如果有其它模块需要使用我们模块导出的符号时，我们就需要导出这些符号。另外，我们可能还得特别引用某些指令来避免导出所有的非静态符号，因为在默认情况下，大部分版本的 `modutils`（但并非所有）将导出所有非静态符号。

Linux 内核头文件提供了一个方便的方法来管理符号对模块外部的可见性，从而减少了可能造成的名字空间污染并且适当隐藏信息。本节中描述的这种方法适用于 2.1.18 及其后的内核版本，2.0 版本的内核拥有一套完全不同的机制，我们将在本章末尾进行描述。

如果不希望模块导出任何符号，则可以在源文件中添加如下一行宏调用来显式说明：

```
EXPORT_NO_SYMBOLS;
```

这个宏将扩展为一条汇编指令，并且可以出现在模块的任何地方。然而，可移植代码应该将它放在模块的初始化函数 `init_module` 中，这是因为文件 `sysdep.h` 中为老内核定义的这个宏只能在模块的初始化函数中起作用。

如果我们准备导出模块中符号的一个子集，则首先需要定义预处理宏 `EXPORT_SYMTAB`，而且这个宏必须在包含 `module.h` 之前定义。一种很常见的定义方法是在模块编译时在 `Makefile` 中使用 `-D` 编译选项指定。

在定义了 `EXPORT_SYMTAB` 之后，可以通过下面两个宏之一来导出模块的一个符号：

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_NOVERS(name);
```

这两个宏都可以用来导出符号，不过第二个宏（`EXPORT_SYMBOL_NOVERS`）导出的符号不带版本控制信息（第 11 章将讨论版本控制）。符号导出必须位于任何函数的外部，因为这些宏扩展为一个变量声明。（感兴趣的读者可以查阅 `<linux/module.h>` 获得更详细的信息。）

2.4 初始化和关闭

前面已经提到，函数 `init_module` 负责注册模块所提供的任何设施。这里的设施指的是一个可以被应用程序访问的新功能，它可能是一个完整的驱动程序或者仅仅是一个新的软件抽象。

模块可以注册许多不同类型的设施。对于每个设施都会有相应的内核函数完成注册工作。传给内核注册函数的参数通常包括：一个指向描述这个新设施的数据结构的指针以及要注册的设施的名称。描述设施的数据结构中通常包含有指向模块函数的指针，因此我们可以调用模块中的函数。

能够注册的设施类型超出了在第 1 章中给出的设备类型列表，它们包括串口、杂项设备、`/proc` 文件、可执行域以及线路规程（`line discipline`）等。很多可注册的设施所支持的功能属于“软件抽象”范畴，而不与任何硬件直接相关。这种类型的设施能够被注册，是因为它们能够以某种方式集成到驱动程序功能当中（如 `/proc` 文件系统以及线路规程）。

还有其它一些设施可以注册为特定设备的附加功能，但是它们的用途有限因而不在这里具体讨论。它们使用在前面“内核符号表”中提到的层叠技术。如果读者想进一步了解，可以在内核源文件中 `grep EXPORT_SYMBOL`，并找出由不同驱动程序提供的入口点。另外，大部分注册函数名字带有 `register_` 前缀，因此找到它们的另一种方法是在 `/proc/ksyms` 中 `grep register_`。

2.4.1 `init_module` 中的出错处理

设施的注册过程中如果出现任何错误，都要将出错之前的注册工作撤销。下面几种情况下都可能发生错误，例如，系统中没有足够的内存分配给一个数据结构，或者，请求的资源正在被其它驱动程序使用等。虽然错误不是经常发生，但它仍然有可能发生，因此我们必须做好处理这些错误的准备。

Linux 中没有记录每个模块都注册了哪些设施，因此模块必须自己备份每步操作，防止 `init_module` 在某步出错。如果由于某种原因我们未能撤销已经注册的设施，则内核会处于一种不稳定状态：一方面，这些设施处于忙的状态，我们不能重新装入模块来再次注册设施；另一方面，我们也不能撤销对它们的注册，因为我们失去了注册这些设施时使用的指向描述设施的数据结构的指针。想从这种处境中恢复比较困难，通常需要重启机器并装入修改后的新模块。

错误恢复的处理有时使用 `goto` 语句比较有效。通常情况下我们很少使用 `goto`，但在处理错误时（可能是唯一的情况）它却非常有用。下面的例子给出了内核中使用 `goto` 语句处理错误的方式。

不管初始化过程在什么时刻失败，下面的例子（使用了虚构的注册和撤销注册函数）都能正确工作。

```
int init_module(void)
```

```

{
int err;

/* registration takes a pointer and a name */
err = register_this(ptr1, "skull");
if (err) goto fail_this;
err = register_that(ptr2, "skull");
if (err) goto fail_that;
err = register_those(ptr3, "skull");
if (err) goto fail_those;

return 0; /* success */

fail_those: unregister_that(ptr2, "skull");
fail_that: unregister_this(ptr1, "skull");
fail_this: return err; /* propagate the error */
}

```

这段代码准备注册三个（虚构的）设施。在出错的时候调用 `goto` 语句，它将只撤销出错时刻以前所成功注册的那些设施。

错误处理的另一种方法不需要使用杂乱的 `goto` 语句。它记录任何成功注册的设施，然后在出错的时候调用 `cleanup_module`。这个函数将仅仅回滚成功完成的步骤。然而，这种替代方法需要更多的代码和 CPU 时间，因此在追求效率的代码中仍然使用 `goto` 语句作为最好的错误恢复机制。`init_module` 的返回值 `err` 是一个错误编码。在 Linux 内核中，错误编码是定义在 `<linux/errno.h>` 中的一个负整数集合。如果我们不想使用从其它函数返回的错误编码而是自己产生的错误码，则应该包含 `<linux/errno.h>`，使用诸如 `-ENODEV`、`-ENOMEM` 之类的符号值。每次返回合适的错误编码将是一个好习惯，因为用户程序可以通过 `perror` 或类似程序将它们转换为有意义的字符串。（然而，有趣的是，有几个版本的 `modutils` 对 `init_module` 返回的任何错误码都转换为“设备忙”，不过这个问题在最近发布的版本中得以纠正。）

显然，`cleanup_module` 需要撤销 `init_module` 注册的所有设施，并且习惯上（但不是必须的）以相反于注册的顺序撤销设施。

```

void cleanup_module(void)
{
    unregister_those(ptr3, "skull");
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    return;
}

```

如果初始化和清除工作涉及很多设施，则 `goto` 方法可能变得难以管理。因为所有用于清除设施的代码在 `init_module` 中重复，同时标号交织在一起。因此，有时候需要考虑重新构思代码结构。

每当发生一个错误时，`init_module` 就调用 `cleanup_module`，这种方法将减少代码重复并且使代码有条理，而清除函数必须在撤销每项设施的注册之前检查它的状态。下面是这种方法的简单示例：

```

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

void cleanup_module(void)
{
    if (item1)

```

```

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

int init_module(void)
{
    int err = -ENOMEM;

    item1 = allocate_thing(arguments);
    item2 = allocate_thing2(arguments2);
    if (!item1 || !item2)
        goto fail;
    err = register_stuff(item1, item2);
    if (!err)
        stuff_ok = 1;
    else
        goto fail;
    return 0; /* success */

fail:
    cleanup_module();
    return err;
}

```

如这段代码所示，根据调用的注册/分配函数语义，我们可以使用或不使用外部标号来标记每个初始化步骤的成功。不管是否使用标记，这种方式的初始化能够很好地扩展到对大量设施的支持，因此比前面介绍的技术更具优越性。

2.4.2 使用计数

为了确定模块是否能够安全卸载，系统为每个模块保留一个使用计数。因为在模块忙时不能被卸载，所以系统需要这个信息确定模块是否忙。例如，当一个文件系统已被安装时，我们就不能卸载它；当进程正在使用一个字符设备时我们也不能卸载这个字符设备，否则，当我们使用无效的指针时，可能会遇到某种类型的段错误甚至是系统的崩溃。

在比较新的内核版本中，系统能够自动跟踪使用计数，下一章中我们会描述这种机制。然而，有些时候仍然需要自己手动去调整使用计数。需要向老版本内核中移植的代码也必须手动维持使用计数。手动进行使用计数时需要使用下面三个宏：

MOD_INC_USE_COUNT

当前模块计数加 1。

MOD_DEC_USE_COUNT

当前模块计数减 1。

MOD_IN_USE

计数非 0 时返回真。

这些宏定义在 `<linux/module.h>` 中，操作那些不能由程序员直接访问的内部数据结构。虽然模块管理的内部机制在 2.1 开发过程中改变很大，并且在 2.1.18 版本中完全重写，然而这三个宏的使用方式并没有改变。

注意在 `cleanup_module` 函数内部不必检查 `MOD_IN_USE`，因为系统调用 `sys_delete_module`（定义在 `kernel/module.c` 中）时预先执行了这种检查。

模块计数管理对系统的稳定性非常重要。因为内核能在任何时刻卸载模块，一个常见的模块编程错误就是在启动了一系列操作后（例如，响应一个 `open` 请求），在模块的末尾才增加使用计数。这时，如果内核在那些操作的中途卸载这个模块，则就会产生混乱。为了避免这种类型错误的发生，我们应该在模块几乎还没有做任何事情之前就调用 `MOD_INC_USE_COUNT`。

如果失去了对某个模块使用计数的跟踪，就无法卸载这个模块。这种情况在模块开发过程中经常发生，我们应当十分小心。例如，如果驱动程序使用了一个空指针指向的内容，就可能导致调用进程的崩溃，此时驱动程序不能关闭对应的设备，而引用计数无法减小为 0。一个可能的解决办法是在调试阶段重定义 `MOD_INC_USE_COUNT` 和 `MOD_DEC_USE_COUNT` 为空，从而取消引用计数；另外一种解决方案是使用某种方法将引用计数器强行置 0（在第 5 章“使用 `ioctl` 参数”一节中我们将会看到这种方法）。虽然在一个正式发布的模块产品中不能省略正确的错误检查机制，然而，在调试阶段使用“蛮力”，却能够缩短开发时间，因而不失为一种可取的方法。

`/proc/module` 文件中每项的第三个域给出了引用计数的当前值。这个文件列出了系统中当前加载的所有模块，它包含若干个项，每一项对应一个模块；每项包含若干个域，分别表示模块名、模块使用内存的字节数、模块的当前使用计数等。下面是一个典型的 `/proc/modules` 文件：

```
parport_pc 7604      1 (autoclean)
lp         4800      0 (unused)
parport    8084      1 [parport_probe parport_pc lp]
lockd     33256     1 (autoclean)
sunrpc    56612     1 (autoclean) [lockd]
ds         6252      1
i82365    22304     1
pcmcia_core 1280     0 [ds i82365]
```

这儿我们看到了系统中装入的几个模块。如图 2-2 所示，并口模块以层叠方式装入系统。`autoclean` 标记表示模块由 `kmod` 或 `kernel`（见第 12 章）管理；而 `unused` 标记的意思和字面意思完全一样，即模块尚未被使用，除这两个标记外，还存在其它一些标记。在 `Linux 2.0` 当中，第二个域（`size`）是以页（大多数平台上每页大小为 4 KB）而不是以字节为单位表示的大小。

2.4.3 卸载

使用 `rmmod` 可以卸载一个模块。卸载模块不像装入模块那样需要进行连接操作，因而任务比较简单。`rmmod` 命令调用系统调用 `delete_module`，这个系统调用随后检查模块引用计数，为 0 时调用模块本身的 `cleanup_module` 函数，否则返回出错信息。

模块所注册的每一项都需要函数 `cleanup_module` 进行注销，只有模块导出的符号可以自动从内核符号表中删除。

2.4.4 显式的初始化和清除函数

如上所述，内核调用 `init_module` 来初始化一个刚刚加载的模块，并在模块即将卸载之前调用 `cleanup_module` 函数。然而，较新的内核中通常给这两个函数重新命名。从 2.3.13 内核版本开始，

增加了一项设施来显式命名初始化和清除函数，使用这项设施是一种更好的编程风格。

下面举个例子。如果我们将模块初始化函数命名为 `my_init`(而不是 `init_module`)，将清除函数命名为 `my_cleanup`，则可以使用下面两行来进行标记（通常在源文件末尾）：

```
module_init(my_init);
module_exit(my_cleanup);
```

注意，要使用 `module_init` 和 `module_exit`，代码必须包含头文件 `<linux/init.h>`。

这样做的好处是内核中每个初始化和清除函数都有一个唯一的名字，因而给调试带来了方便；同时也使那些既可以作为一个模块也可以直接嵌入内核的驱动程序更加容易编写。然而，如果我们仍然使用老的初始化和清除函数的名字，则不需要使用函数 `module_init` 和 `module_exit`。实际上，这两个函数对模块所做的唯一事情就是将 `init_module` 和 `cleanup_module` 定义为给定函数的名字。

如果深入研究一下内核源码（版本 2.2 及其以后），我们会发现模块初始化函数和清除函数的原型略有不同，如下所示：

```
static int __init my_init(void)
{
    ....
}

static void __exit my_cleanup(void)
{
    ....
}
```

象这样使用属性 `__init`，它会在初始化工作完成后，丢弃初始化函数并且回收它所占内存。然而，它仅仅对直接嵌入内核的驱动程序有效，对驱动程序模块则没有作用。相反，当 `__exit` 用在直接嵌入内核的驱动程序中时，它将忽略所标记的函数；而用在模块时则没有任何作用。

使用 `__init`（如果是数据条目，就是 `__initdata`）能够减少内核使用的内存。在模块初始化函数前标记 `__init` 虽然在现在没有什么好处，但也没有任何坏处。而且尽管内核当前对模块初始化阶段没有进行管理，但以后可能会在这方面加强。

2.5 使用资源

如果模块不使用内存、I/O 断口、I/O 内存以及中断线等系统资源，就不能完成自己的任务。如果我们用的是一个老式的 DMA 控制器（例如 ISA 总线的 DMA 控制器），则还需要 DMA 通道。

作为程序员，我们可能已经习惯于管理内存分配；在这方面，编写内核代码并没有任何区别。我们的程序需要使用 `kmalloc` 获得一块内存区，然后使用 `kfree` 释放。这两个函数和 `malloc` 以及 `free` 功能基本类似，不过 `kmalloc` 带有一个额外的参数 `priority`。通常使用 `GFP_KERNEL` 或者 `GFP_USER` 作为 `priority` 参数值，缩写 GFP 代表“get free page，获得空闲页面。”（第 7 章将详述有关内存分配的问题。）

驱动程序新手刚开始可能对需要自己来分配 I/O 断口、I/O 内存^{*}以及中断线等等感到奇怪。毕竟，内核模块可以只访问这些资源而不用告诉操作系统。然而，尽管系统内存是匿名的并且可以从任何地方开始分配，但 I/O 内存、端口以及中断等却都有各自特定的作用。例如，驱动程序可能需要请求分配某个特定端口，而不是随便一个端口。然而驱动程序不能仅仅只是请求这些系统资源，它应该首先弄清楚这些资源是否已在其它地方使用。

2.5.1 I/O 端口和 I/O 内存

多数情况下，一个典型驱动程序的任务是读写 I/O 端口和 I/O 内存。在初始化阶段和正常运行时，驱动程序都可能访问 I/O 端口和 I/O 内存（统称 I/O 区域）。

然而，不幸地是，并不是所有的总线结构都能够清楚地区分每种设备所属的 I/O 区域。有些时候，驱动程序需要猜测它的 I/O 区域在哪儿，甚至需要通过读写可能的地址范围来探测设备。ISA 总线中就存在这个问题，并且它仍然使用在充满简单设备的个人计算机中，在工业领域它的 PC/104 实现也依然非常流行（参看第 15 章的 PC/104 和 PC/104+）。

尽管一些总线中存在这些特性（或者缺少这些特性），设备驱动程序仍然必须保证它对 I/O 区域的独占式访问，以防止其它驱动程序干扰。例如，如果一个正在探测硬件的驱动程序碰巧将数据写到属于另一个设备的 I/O 端口，则肯定会发生问题。

为了避免这种不同设备之间的冲突，Linux 开发者实现了 I/O 区域的请求/释放机制。这种机制在 I/O 端口中早已使用，最近才推广到整个资源分配的管理。注意这种机制仅仅是一个帮助系统管理资源的软件抽象，并不要求硬件支持。例如，如果非授权访问 I/O 端口，并不会产生类似于“段错误”的错误状态，因为硬件并不强制要求对端口进行注册。

文件 `/proc/ioports` 和 `/proc/iomem` 以文本形式列出了已注册的资源。`/proc/iomem` 是在 2.3 版本开发过程中才刚刚引入的。我们先讨论 2.4 版本，将可移植性问题留在本章末尾讨论。

端口

一台运行 2.4 版本内核的 PC 机上一个典型的 `/proc/ioports` 如下所示：

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : idel
01f0-01f7 : ide0
02f8-02ff : serial(set)
0300-031f : NE2000
0376-0376 : idel
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(set)
```

^{*} 位于外围设备上的内存区域通常称作 I/O 内存，以和系统 RAM 相区别。后者一般就叫做内存。

```

1000-103f : Intel Corporation 82371AB PIIX4 ACPI
  1000-1003 : acpi
  1004-1005 : acpi
  1008-100b : acpi
  100c-100f : acpi
1100-110f : Intel Corporation 82371AB PIIX4 IDE
1300-131f : pcnet_cs
1400-141f : Intel Corporation 82371AB PIIX4 ACPI
1800-18ff : PCI CardBus #02
1c00-1cff : PCI CardBus #04
5800-581f : Intel Corporation 82371AB PIIX4 USB
d000-dfff : PCI Bus #01
d000-d0ff : ATI Technologies Inc 3D Rage LT Pro AGP-133

```

文件中的每一项表示（以 16 进制形式）一个被某个驱动程序锁定或者属于某个硬件设备的端口范围。早期版本的内核中这个文件具有同样的格式，但是没有上面的由缩进表示的“分层”结构。

当系统增添一个新设备并且需要通过跳线来选择一个 I/O 范围时，这个文件可以用来避免端口冲突：用户检查已经使用的端口，然后设置新设备使用一个空闲的 I/O 范围。尽管大部分现在的硬件不再使用跳线，这种方法在处理定制设备或者工业部件时仍然非常有用。

隐藏在文件 `ioports` 后面的数据结构比文件本身更重要。当设备驱动程序初始化的时候，它能够知道哪些端口范围已经使用。如果需要检测 I/O 端口来探测新设备，它能够避开检测那些被其它驱动程序使用的端口。

ISA 探测实际上不是很安全。Linux 正式发行版本中的几个驱动程序在以模块的方式加载后，不再执行探测任务，也不再检测那些可能被某个未知设备使用的端口，从而避免破坏系统运行。幸运的是，新式总线结构（以及那些经过改良的老式总线结构）中不存在这些问题。

访问 I/O 注册表的编程接口由下面三个函数组成：

```

int check_region(unsigned long start, unsigned long len);
struct resource *request_region(unsigned long start,
unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);

```

`check_region` 用来检查是否可以分配某个端口范围，如果不可以则返回一个负的错误编码（例如 `-EBUSY` 或者 `-EINVAL`）。`request_region` 完成真正的端口范围分配，分配成功则返回一个非空指针。驱动程序并不需要使用或者保存这个返回的指针，我们只是检查它是否非空即可。^{*}

如果代码仅仅在 2.4 内核中运行，则根本不需要调用 `check_region`。而且实际上这种情况下也最好不要调用它，因为在调用 `check_region` 和 `request_region` 之间，情况可能发生改变。然而，如果我们的代码需要向老的内核移植，则必须使用 `check_region`，因为在 2.4 版本之前 `request_region` 返回 `void`。最后，驱动程序在完成任务之后还需要调用 `release_region` 释放分配的端口。

这三个函数实际上都是宏，定义在 `<linux/ioport.h>` 中。

下面的代码以及我们的驱动程序例子中都给出了注册端口的典型顺序。（因为包含与具体设备相关

^{*} 只有一种情况下会使用这个实际指针，这就是当这个函数被内核资源管理子系统内部调用时。

的代码，这里没有给出函数 `skull_probe_hw` 的定义。)

```
#include <linux/ioport.h>
#include <linux/errno.h>
static int skull_detect(unsigned int port, unsigned int range)
{
    int err;

    if ((err = check_region(port,range)) < 0) return err; /* busy */
    if (skull_probe_hw(port,range) != 0) return -ENODEV; /* not found */
    request_region(port,range,"skull"); /* "Can't fail" */
    return 0;
}
```

这段代码首先检查请求的端口范围是否可用，如果不能被分配，就没有必要继续探测硬件设备。实际的端口分配是在检测到硬件之后发生的。对 `request_region` 的调用从来不会失败；因为内核一次只装载一个模块，因此不可能出现其它的模块插进来取走已请求端口的情况。如果怀疑这一点，我们可以编写代码进行测试，但是应当记住 2.4 内核之前 `request_region` 返回 `void`。

驱动程序分配的任何 I/O 端口最终都必须释放，`skull` 在函数 `cleanup_module` 内部完成这个任务：

```
static void skull_release(unsigned int port, unsigned int range)
{
    release_region(port,range);
}
```

资源的请求/释放方式同前面提到的设施的注册/注销的顺序非常相似，并且同样可以使用前面提到的基于 `goto` 语句的出错处理机制。

内存

同 I/O 端口的情况类似，I/O 内存的信息可从文件 `/proc/iomem` 中获得。下面是一台个人微机上的这个文件的部分内容：

```
00000000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000c7fff : Video ROM
000f0000-000fffff : System ROM
00100000-03feffff : System RAM
00100000-0022c557 : Kernel code
0022c558-0024455f : Kernel data
20000000-2fffffff : Intel Corporation 440BX/ZX - 82443BX/ZX Host bridge
68000000-68000fff : Texas Instruments PCI1225
68001000-68001fff : Texas Instruments PCI1225 (#2)
e0000000-e3ffffff : PCI Bus #01
e4000000-e7ffffff : PCI Bus #01
e4000000-e4ffffff : ATI Technologies Inc 3D Rage LT Pro AGP-133
e6000000-e6000fff : ATI Technologies Inc 3D Rage LT Pro AGP-133
fffc0000-ffffffff : reserved
```

同样，这里列出的值是十六进制表示的范围，在冒号后的字符串是该 I/O 区域“所有者”的名字。

就驱动程序编程而言，I/O 内存的访问方式同 I/O 端口的访问方式相同，因为实际上它们都基于

同样的内部机制。

驱动程序使用下列函数调用来获得和释放对某个 I/O 内存区域的访问。

```
int check_mem_region(unsigned long start, unsigned long len);
int request_mem_region(unsigned long start, unsigned long len,
    char *name);
int release_mem_region(unsigned long start, unsigned long len);
```

典型的驱动程序通常事先知道它的 I/O 内存范围，因此相对于前面 I/O 端口的请求方法，对 I/O 内存的请求缩减为如下几行代码：

```
if (check_mem_region(mem_addr, mem_size)) { printk("drivername:
    memory already in use\n"); return -EBUSY; }
request_mem_region(mem_addr, mem_size, "drivername");
```

2.5.2 Linux 2.4 中的资源分配

目前的资源分配机制是在 Linux 2.3.11 中引入的，它提供了一种灵活的控制系统资源的方法，本节将简述这种机制。然而，基本的资源分配函数（`request_region` 及其它）在系统中仍然保留它们的实现（以宏的方式）并且广泛使用，因为利用它们可保持和以前版本的向后兼容性。多数模块开发人员无需知道幕后究竟发生了什么，但是需要开发复杂驱动程序的程序员还是需要了解一点。

Linux 的资源管理以一种分层结构管理任意的资源。全局性的资源（例如 I/O 端口范围）可以分割为小一些的子集，比如是和某个具体总线插槽相关联的资源。驱动程序还可以根据需要进一步细分属于自己的资源子集。

资源范围由在 `<linux/ioport.h>` 中定义的资源结构来描述：

```
struct resource {
    const char *name;
    unsigned long start, end;
    unsigned long flags;
    struct resource *parent, *sibling, *child;
};
```

顶层的（根）资源在系统启动时创建。例如下面的代码给出了一个描述 I/O 范围的资源结构的创建过程：

```
struct resource ioport_resource =
    { "PCI IO", 0x0000, IO_SPACE_LIMIT, IORESOURCE_IO };
```

因此，我们知道资源名称是 PCI IO，包含从 0 到 `IO_SPACE_LIMIT` 的范围。根据底层硬件平台的不同，`IO_SPACE_LIMIT` 可能是 `0xffff`（16 位的地址空间，例如在 x86、IA-64、M68k 和 MIPS 平台上）、`0xffffffff`（32 位：SPARC、PPC、SH）或者 `0xffffffffffffffff`（64 位：SPARC64）。

函数 `allocate_resource` 可以创建一个给定资源的子集。例如，在 PCI 初始化期间可以为实际分配给物理设备的区域创建一个新的资源。当 PCI 代码读取这些端口或内存时，它就为那些区域创建一个新资源，然后在 `ioport_resource` 或者 `iomem_resource` 中分配。

驱动程序因此可以请求某个资源的一个子集（实际上是一个全局资源的子集）并且调用 `_request_region` 将它标记为忙，`_request_region` 返回的指针指向描述所请求资源的资源数据结构（出错情况下则返回空指针）。这个数据结构是全局资源树的一部分，因此驱动程序需要小心使用。

感兴趣的读者可以浏览 `kernel/resources.c` 的源码或者查阅内核其它部分使用资源管理的机制来获得细节信息。然而，对于大多数驱动程序编程人员来说，前面一节中讲到的 `request_region` 和其它函数已经足够使用了。

这种分层机制带来了两个好处。其中之一 I/O 结构隐藏在内核的数据结构内部。例如，`/proc/ioproports` 中显示的结果：

```
e800-e8ff : Adaptec AHA-2940U2/W / 7890
e800-e8be : aic7xxx
```

`e800-e8ff` 分配给了适配卡，它在 PCI 总线驱动程序中标识自己。接着 `aic7xxx` 驱动程序请求获得那个 I/O 范围的大部分，即适配卡中端口实际对应的部分。

这种资源管理方式的另一个好处是，它将端口空间分割成与底层硬件对应的端口子集。因为资源分配器不允许跨子集分配端口，因此它可以防止带有 bug 的驱动程序（或者一个正在探测系统中根本不存在的硬件的驱动程序）获得属于不同范围的端口，即使这些端口范围此时尚未全部分配。

2.6 自动和手动配置

驱动程序需要了解的几个参数随着系统的不同而不同。例如，驱动程序必须知道硬件实际的 I/O 地址或者内存范围（只有 ISA 设备是这样，对于设计良好的总线接口这可能不是问题）。有时候我们还需要传递参数给驱动程序，帮助它找到对应的硬件或者激活/禁止某些特性。

根据设备的不同，除了 I/O 地址外可能还有其它参数影响驱动程序特性，例如设备牌号和发行版本号等。驱动程序要想正确运行必须知道这些参数值。然而，给驱动程序正确设置参数值（即配置驱动程序）是个比较困难的工作，需要在驱动程序初始化阶段完成。

总的说来，驱动程序有两种获得正确参数值的方法：一种方法是由用户显式指定，另一种方法是驱动程序自动检测。自动检测毫无疑问是最好的驱动配置方法，然而用户手动配置却更容易实现。对驱动程序开发人员来说，一种折中的方法是尽可能使用自动配置，然而允许用户手动配置并覆盖自动检测的参数值。这样做的另外一个好处是我们在驱动程序开发初始阶段不必使用自动检测，而在装入驱动程序模块的时候手动指定参数值；而在开发后期实现自动检测。

许多驱动程序还有一些配置选项来控制其操作的其它特征。例如，SCSI 适配器驱动程序通常会带有选项来控制标签命令排队的使用，而 IDE 驱动程序允许用户控制 DMA 操作。因此，即使驱动程序完全依靠自动检测来定位硬件，我们仍然需要给用户提供其它配置选项。

参数值可由 `insmod` 或者 `modprobe` 在装载模块时设置，后者还可以从配置文件（通常是 `/etc/modules.conf`）中获得参数赋值。这些命令能够在命令行中接受整型和字符串型赋值。因此，如果模块需要获得一个叫做 `skull_ival` 的整型参数和一个叫做 `skull_sval` 的字符串型参数，我们

可以在模块装载时以下面的方式使用 `insmod` 命令设置参数：

```
insmod skull skull_ival=666 skull_sval="the beast"
```

然而，在 `insmod` 能够改变模块参数之前，模块必须能够访问这些参数。参数由定义在 `module.h` 中的宏 `MODULE_PARM` 声明。`MODULE_PARM` 必须带两个参数：变量名和描述变量类型的字符串。这个宏应位于任何函数之外，通常放在源文件的起始部分。上面提到的两个参数可用下面几行来声明：

```
int skull_ival=0;
char *skull_sval;

MODULE_PARM (skull_ival, "i");
MODULE_PARM (skull_sval, "s");
```

目前模块参数只支持五种类型：**b**，字节 (**byte**)；**h**，短整型 (**short**，两字节)；**i**，整型 (**integer**)；**l**，长整型 (**long**)；**s**，字符串 (**string**)。如果是字符串值，则需要声明一个指针变量。`insmod` 负责为用户提供的参数分配内存并设置相应变量。类型前面的整数表明这是一个指定长度的数组，被间隔符分开的两个数字指明了数组元素数目的最大最小值。如果我们想了解设计者对这个特征的详细描述，可以参考头文件 `<linux/module.h>`。

作为一个例子，至少有两个元素、至多不超过 4 个元素的数组可定义为：

```
int skull_array[4];
MODULE_PARM (skull_array, "2-4i");
```

还有一个名为 `MODULE_PARM_DESC` 的宏，它让模块开发者为模块参数提供描述性文字。这段描述存储在目标文件中，能够用类似 `objdump` 的工具查看，也可用自动的系统管理工具来显示。例如：

```
int base_port = 0x300;
MODULE_PARM (base_port, "i");
MODULE_PARM_DESC (base_port, "The base I/O port (default 0x300)");
```

所有模块参数都应该赋予一个默认值，用户可以使用 `insmod` 来显式改变。模块通过和默认值比较来确定显式参数值。因此，可以这样设计模块自动配置：如果配置变量具有默认值则执行自动检测，否则保持当前值。要想让这种方法生效，默认值应该是用户在装载模块时从来不会使用的参数值。

下面的代码演示了 `skull` 如何自动检测设备的端口范围。这个例子中，使用自动检测探测多个设备，而手动配置只限于单个设备。首先使用函数 `skull_detect` 探测“端口”，然后使用 `skull_init_board` 进行设备特有的初始化工作，这两个函数在这里没有给出。

```
/*
 * port ranges: the device can reside between
 * 0x280 and 0x300, in steps of 0x10. It uses 0x10 ports.
 */
#define SKULL_PORT_FLOOR 0x280
#define SKULL_PORT_CEIL 0x300
#define SKULL_PORT_RANGE 0x010
```



```

/*
 * the following function performs autodetection, unless a specific
 * value was assigned by insmod to "skull_port_base"
 */

static int skull_port_base=0; /* 0 forces autodetection */
MODULE_PARM (skull_port_base, "i");
MODULE_PARM_DESC (skull_port_base, "Base I/O port for skull");

static int skull_find_hw(void) /* returns the # of devices */
{
    /* base is either the load-time value or the first trial */
    int base = skull_port_base ? skull_port_base
        : SKULL_PORT_FLOOR;
    int result = 0;

    /* loop one time if value assigned; try them all if autodetecting */
    do {
        if (skull_detect(base, SKULL_PORT_RANGE) == 0) {
            skull_init_board(base);
            result++;
        }
        base += SKULL_PORT_RANGE; /* prepare for next trial */
    }
    while (skull_port_base == 0 && base < SKULL_PORT_CEIL);

    return result;
}

```

如果仅仅在驱动程序中使用配置变量（没有导出到内核符号表中），驱动编写者可以省略变量名前缀（本例中是 `skull_`），因而方便模块用户的使用。除了要多敲几个字母外，前缀对用户通常没有其它影响。

为了完整性起见，我们继续介绍其它三个在目标文件中增加说明文档的宏：

MODULE_AUTHOR(name)

将模块作者名加入目标文件

MODULE_DESCRIPTION(desc)

在目标文件中增加模块描述文字

MODULE_SUPPORTED_DEVICE(dev)

描述模块所支持的设备。内核中的注释指明这个参数可能最终用来帮助模块自动装载，然而目前还没有起到这种作用。

2.7 在用户空间编写驱动程序

首次接触内核的 Unix 程序员可能对编写模块比较紧张，然而编写用户空间程序来直接对设备端口进行读写就容易多了。

相对于内核空间编程，用户空间编程具有自己的一些优点。有时候编写一个所谓的用户空间驱动程序是替代内核空间驱动程序的一个不错的方法。

用户空间驱动程序的优点可以归纳如下：

- 可以和整个 C 库连接。驱动程序不用借助外部程序（即前面提到的和驱动程序一起发行的用于提供策略的用户程序）就可以完成许多非常规任务。
- 可以使用通常的调试器调试驱动程序代码，而不用费力地调试运行内核。
- 如果用户空间驱动程序挂起，则简单地杀掉它就行了。驱动程序带来的问题不会挂起整个系统，除非驱动硬件已经发生严重故障。
- 和内核内存不同地是，用户内存可以换出。如果驱动程序很大但是不经常使用，则除了正在使用的时候之外，不会占用内存。
- 良好设计的驱动程序仍然支持对设备的并发访问。

X 服务器是用户空间驱动程序的一个例子。它十分清楚硬件可以做什么，不可以做什么，并且为所有的 X 客户提供图形资源。然而，值得注意的是目前基于帧缓冲区（frame-buffer）的图形环境正在慢慢成为发展趋势。这种环境下对于实际的图形操作，X 服务器仅仅是一个基于真正内核空间驱动程序的服务器。

另外一个用户空间驱动程序的例子是 `gpm` 鼠标服务器。它在多个客户端之间分发鼠标事件，因此和鼠标相关的应用可以同时跑在不同的虚拟控制台上。

然而，有些时候用户空间驱动程序将设备的访问权只授与一个程序，这就是 `libsvga` 库工作的原理。它和应用程序连接，将一个 TTY 转变为一个图形显示设备，因此在不借助于中央控制服务（如服务器）的情况下扩展了应用的功能。这种方法由于省去了通信开销，因而提供了较好的性能，然而它要求应用必须以特权用户的身份运行（不过运行在内核空间的帧缓冲区驱动程序现在已经解决了这个问题）。

除了具备上述优点外，用户空间驱动程序也有很多缺点，下面列出其中最重要的几点：

- 中断在用户空间中不可用。解决这一问题的唯一办法是使用 `vm86` 系统调用（在 `x86` 平台上），然而它却带来了性能损失。^{*}
- 只有通过 `mmap` 映射 `/dev/mem` 才能直接访问内存，但只有特权用户才可以执行这个操作。
- 只有在调用 `ioperm` 或 `iopl` 后才可以访问 I/O 端口。然而并不是所有平台都支持这两个系统调用，并且访问 `/dev/port` 可能非常慢，因而并非十分有效。同样只有特权用户才能引用这些系统调用和访问设备文件。
- 响应时间很慢。这是因为在客户端和硬件之间传递信息和动作需要上下文切换。
- 更严重的是，如果驱动程序被换出到磁盘，响应时间将难以忍受。引用 `mlock` 系统调用或许可以减轻这一问题，但由于用户空间程序一般需要连接多个库，因此通常需要占用多个内存页。同样，`mlock` 也只有特权用户才能引用。
- 用户空间中不能处理一些非常重要的设备，包括网络接口和块设备等。

如上所述，我们看到用户空间驱动程序毕竟做不了太多的工作。然而依然存在一些有意义的应用，例如，对 `SCSI` 扫描设备（由包 `SANE` 实现）和 `CD` 刻录设备（由 `cdrecord` 和其它工具实现）的支持。这两种情况下，用户空间驱动程序都依赖内核空间驱动程序“`SCSI generic`”，它导出底层通用的 `SCSI` 功能到用户空间程序，然后再由用户空间驱动程序驱动自己的硬件。

^{*} 本书的主题将局限于内核驱动程序，因此不准备讨论 `vm86`。而且，这个系统调用太依赖于平台，可能也很难引起一般读者的兴趣。

要想编写用户空间驱动程序，了解硬件知识就足够了，没有必要了解内核的细微之处。本书中我们将不再进一步讨论用户空间驱动，而将主要精力集中于内核代码。

有一种情况适合在用户空间处理，这就是当我们准备处理一种新的不常见的硬件时。在用户空间中我们可以研究如何管理这个硬件而不用担心挂起整个系统。一旦完成，很容易就能将用户空间驱动程序封装到内核模块中。

2.8 向后兼容性

Linux 内核处于不断发展之中，许多内容逐渐改变，新的特色也在逐渐发展。本章中描述的接口都属于 2.4 版本，我们还要做很多工作才能使基于这些接口的代码工作在老版本内核中。

这一节是本书中第一个关于“向后兼容性”的小节，本书后面还有很多关于这个主题的小节。每章末尾我们将讨论从 2.0 版本以来内核发生的变化，以及为了移植代码我们需要做的工作。

作为开始，首先讲到的一个变化是在内核 2.1.90 中才第一次引入宏 `KERNEL_VERSION`。头文件 `sysdep.h` 中为那些需要使用这个宏的内核定义了一个替代实现。

2.8.1 资源管理的改变

如果想编写一个能够运行在 2.4 版本以前内核上的驱动程序，则新版本内核中新的资源管理方式可能会带来移植方面的问题。本节我们讨论可能遇到的这些问题以及 `sysdep.h` 如何替用户隐藏这一问题。

新的资源管理方式带来的最明显的改变是增加了 `request_mem_region` 和相关的一些函数。它们的作用仅局限于访问 I/O 内存数据库，不会执行任何与硬件相关的操作。因此，在以前的内核上，只要不去调用这些函数就不会产生麻烦。头文件 `sysdep.h` 为我们做到这一点，它将 `request_mem_region` 定义为宏，如果内核版本是 2.4 以前，这个宏就返回 0。

2.4 和以前内核版本的另一个区别是 `request_region` 以及相关函数的原型不同。

2.4 版本以前的内核将 `request_region` 和 `release_region` 的返回值定义为 `void`（因此必须事先使用 `check_region`）。而新版本的函数实现更加精确，它返回一个指针，因此可以报告错误状态（所以此时的 `check_region` 没有什么用处）。实际返回的指针除了用来测试是否非空外，几乎没有其它用处。指针为空时表示请求失败。

如果想在驱动程序代码中少写几行，并且不考虑向后兼容性，则可以在代码中使用这些函数的新版本并避免使用 `check_region`。实际上，现在的 `check_region` 函数是基于 `request_region` 实现的，它释放 I/O 区域并且在请求满足后返回成功信息，开销可以忽略不计，因为没有人会在一个时间关键的代码段中使用这些函数。

如果我们希望代码具有可移植性，则需要遵守前面提到的函数调用顺序并且忽略 `request_region` 和 `release_region` 的返回值。不管怎样，`sysdep.h` 将两个函数都定义为宏，并且调用成功后返回

0。因此我们的代码在具备可移植性的同时，仍然可以检查每个调用函数的返回值。

2.4 内核和以前内核在 I/O 注册方面的最后一点不同是参数 `start` 和 `len` 的数据类型。新版本内核中总是使用 `unsigned long`，而老版本内核中使用短一些的整型。不过，这个变化对于驱动程序移植性并没有太大的影响。

2.8.2 多处理器系统上的编译

内核 2.0 并没有使用 `CONFIG_SMP` 配置选项来构造 SMP 系统。代替地，我们在内核的主 `makefile` 里使用全局变量来指示构造 SMP 系统。值得注意的是，为 SMP 机器编译的内核在单一处理器的机器上不能工作，反之亦然，因此我们必须做出正确的配置。

本书所带的示例代码在 `makefile` 中自动处理有关 SMP 的问题，因此前面提到的代码并不用拷贝到每个模块中。然而，我们并不支持 2.0 版本内核以下的 SMP。这应该不是一个大问题，因为 2.0 中多处理器的支持并不可靠，我们通常使用 2.2 或者 2.4 版本内核运行 SMP 系统。本书之所以还要讲 2.0 是因为它仍然是小型嵌入式系统选择的平台之一（特别是它没有 MMU 的实现），但是这些系统都不带多处理器。

2.8.3 在 Linux 2.0 中导出符号

2.0 中的符号导出机制建立在函数 `register_symtab` 之上。2.0 中的模块需要建立一张表描述所有需要导出的符号，接着在它的初始化函数中调用 `register_symtab`。只有那些在这张显式符号表中列出的符号才会导出到内核。相反，如果没有调用这个函数，则导出所有的全局变量。

如果模块不需要导出任何符号，并且我们也不想将所有符号定义为静态，则可以在 `init_module` 中增加下面一行来隐藏全局变量。函数 `register_symtab` 简单地用一张空的符号表来覆盖模块默认的符号表。

```
register_symtab(NULL);
```

这实际上就是在为 2.0 内核编译模块时，`sysdep.h` 对 `EXPORT_NO_SYMBOLS` 所作的定义。这也是为什么 `EXPORT_NO_SYMBOLS` 必须位于 `init_module` 函数内才能在 2.0 内核中正常工作的原因。

如果我们确实需要从模块中导出符号，则需要创建一个描述这些符号的符号表数据结构。给一个 2.0 内核上的符号表数据结构赋值需要非常小心，不过内核开发者提供了头文件来简化操作。下面的几行代码展示了如何利用 2.0 中头文件提供的设施，来定义和导出一个符号表：

```
static struct symbol_table skull_syms = {
#include <linux/symtab_begin.h>
    X(skull_fn1),
    X(skull_fn2),
    X(skull_variable),
#include <linux/symtab_end.h>
};

register_symtab(&skull_syms);
```

驱动程序编写人员需要做很多工作，才能编写出能够控制符号可见性的可移植模块代码。特别是当仅仅定义几个处理兼容性的宏不足以解决问题时，此时，保证可移植性需要大量的条件预处理语句。不过原理比较简单，第一步是识别当前内核版本并定义一些相应的符号，我们在 `sysdep.h` 中就定义了一个宏 `REGISTER_SYMTAB()`，它在 2.2 和以后的内核版本中扩展为空，在 2.0 版本中扩展为 `register_syntab`。另外，如果需要使用老版本内核上的代码，则需要定义 `__USE_OLD_SYMTAB`。

通过使用这些代码，导出符号的模块实现了可移植。示例代码中有一个叫做 `misc-modules/export.c` 的模块，它只有导出一个符号的功能。这个模块在 11 章“模块中的版本控制”中还会详细讲到，它包含下面几行来导出符号并且保持可移植性：

```
#ifdef __USE_OLD_SYMTAB__
static struct symbol_table export_syms = {
    #include <linux/symtab_begin.h>
    X(export_function),
    #include <linux/symtab_end.h>
};
#else
EXPORT_SYMBOL(export_function);
#endif

int export_init(void)
{
    REGISTER_SYMTAB(&export_syms);
    return 0;
}
```

如果设置了 `__USE_OLD_SYMTAB`（即我们在处理 2.0 内核），则根据需要定义 `symbol_table`。另外，`EXPORT_SYMBOL` 用来直接导出符号。因此在 `init_module` 中需要调用 `REGISTER_SYMTAB`，在 2.0 以外的任何内核版本中，它将扩展为空。

2.8.4 模块配置参数

`MODULE_PARM` 在 2.1.18 内核中首次引入。在 2.0 内核中，没有显式的参数定义方法。因此，`insmod` 可以改变模块中的任何变量值。这种方法有一个很大的弊端就是用户可以访问他不应该访问的变量，而且这种方法也没有参数类型检查。`MODULE_PARM` 使模块参数更加清楚安全，然而也使 2.2 的模块和 2.0 内核不相兼容。

如果考虑与 2.0 的兼容性问题，则可以使用一个简单的预处理测试将各种 `MODULE_` 宏定义为空。示例代码中的头文件 `sysdep.h` 在需要的时候就处理这些宏。

2.9 快速参考

本节将总结这一章提到的内核函数、变量、宏以及 `/proc` 文件，可以作为对这些内容的一个参考。从本章开始，以后每一章里都会有类似的一节来总结引入的新符号。

```
__KERNEL__ __MODULE__
```

预处理符号。在编译模块化内核代码时必须定义。

```
__SMP__
```

预处理符号。为多处理器系统编译模块时必须定义。

```
int init_module(void);
void cleanup_module(void);
```

模块入口点。在模块目标文件中必须定义。

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

新版本内核中用来标记模块初始化和清除函数的新机制。

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

必需的头文件。它必须包含在模块源代码中。

```
MOD_INC_USE_COUNT;
MOD_DEC_USE_COUNT;
MOD_IN_USE;
```

操作使用计数的宏。

```
/proc/modules
```

列出装入内核的模块列表。每个列表项包含模块名、模块占用内存大小以及使用计数等域，还有一些附加字符串指明模块当前的活动选项。

```
EXPORT_SYMTAB;
```

预处理宏。在模块需要导出符号时定义。

```
EXPORT_NO_SYMBOLS;
```

指明模块不需要导出任何符号到内核。

```
EXPORT_SYMBOL (symbol);
EXPORT_SYMBOL_NOVERS (symbol);
```

用来导出单个符号到内核的宏。第二个宏导出的符号不带版本控制信息。

```
int register_symtab(struct symbol_table *);
```

用来指定模块中公用符号集合的函数。仅用于 2.0 内核。

```
#include <linux/symtab_begin.h>
X(symbol),
#include <linux/symtab_end.h>
```

2.0 内核中用于声明符号表的头文件和预处理器。

```
MODULE_PARM(variable, type);
MODULE_PARM_DESC (variable, description);
```

将一个模块变量定义为参数的宏，用户随后可在装入模块时调整这个变量值。

```
MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_SUPPORTED_DEVICE(device);
```

在目标文件中添加关于模块的文档信息。

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

必需的头文件。除非 `__NO_VERSION__`（见下面）被定义，否则它被 `<linux/module.h>` 包含。

LINUX_VERSION_CODE

整数宏，用在处理版本依赖的预处理条件语句中。

char kernel_version[] = UTS_RELEASE;

每个都模块必需的变量。除非已经定义 `__NO_VERSION__`（见下一项），否则 `<linux/module.h>` 必须定义它。

__NO_VERSION__

预处理器符号。用来防止在 `<linux/module.h>` 中声明 `kernel_version`。

#include <linux/sched.h>

最重要的头文件之一，包含驱动程序使用的大部分内核 API 的定义，包括睡眠函数以及各种变量声明。

struct task_struct *current;

当前进程。

**current->pid
current->comm**

当前进程的进程 ID 和命令名。

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

函数 `printf` 的内核版。

**#include <linux/malloc.h>
void *kmalloc(unsigned int size, int priority); "void kfree(void *obj);**

函数 `malloc` 和 `free` 的内核版。使用 `GFP_KERNEL` 作为 `priority` 参数值。

**#include <linux/ioport.h>
int check_region(unsigned long from, unsigned long extent);
struct resource *request_region(unsigned long from, unsigned long extent, const char *name);
void release_region(unsigned long from, unsigned long extent);**

注册和释放 I/O 端口的函数。

**int check_mem_region (unsigned long start, unsigned long extent);
struct resource *request_mem_region (unsigned long start, unsigned long extent, const char *name);
void release_mem_region (unsigned long start, unsigned long extent);**

注册和释放 I/O 内存区域的宏。

/proc/ksyms

公用内核符号表。

/proc/ioports

系统中安装的设备所占用的 I/O 端口列表。

/proc/iomem

已分配内存区域的列表。

第 3 章 字符设备驱动程序



本章的目标是编写一个完整的字符设备驱动程序。我们将开发一个字符设备驱动程序，此类驱动程序适合于大多数简单的硬件设备，而且比起块设备或网络等驱动程序，字符设备驱动程序也较容易理解。我们的最终目标是编写一个模块化的字符设备驱动程序，但本章我们不会讨论模块化的相关问题。

贯穿全章，我们将介绍一些代码段，它们取自一个真正的设备驱动程序：**scull**，即“**Simple Character Utility for Loading Localities**”的缩写。**scull** 是一个操作内存区域的字符设备驱动程序，这片内存区域就当作一个设备。这种处理的副作用在于，只要涉及 **scull**，“设备”这个词就可与“**scull** 所使用的内存区域”互换使用。

scull 的优点在于它不和硬件相关，因为每台计算机都有内存。**scull** 只是操作某些内存，通过 **kmalloc** 进行分配。任何人都可以编译和运行 **scull**，而且 **scull** 可以移植到所有 **Linux** 支持的计算机平台上。但另一方面，除了展示内核和字符设备驱动程序之间的接口，并让用户运行某些测试例程外，**scull** 设备做不了任何“有用的”事情。

3.1 **scull** 的设计

编写驱动程序的第一步就是定义驱动程序为用户程序所提供的能力（机制）。由于我们的“设备”是计算机内存的一部分，所以可以利用它随意地做我们想做的事情。它可以是顺序或随机存取设备，也可以是一个或多个设备等。

为了让 **scull** 能够为编写真正的设备驱动程序提供一个样板，我们将讲解怎样在计算机内存之上实现若干设备抽象，而且每个都具有各自的特点。

scull 的源代码实现了下列设备。我们将模块实现的每种设备称作一种“类型”：

```
scull0 — scull3
```

四个设备各由一个全局和持久的内存区域组成。“全局”是指，如果设备被多次打开，则打开它的所有文件描述符可共享该设备所包含的数据。“持久”是指，如果设备关闭后再打开，其中的数据不丢失。可以使用常用命令来访问和测试这个设备，如 **cp**、**cat** 以及 **shell** 的 **I/O** 重定向等。在本章我们将深入探讨它的内部结构。


```
scullpipe0 - scullpipe3
```

四个 FIFO（先入先出）设备，与管道类似。一个进程读取由另一个进程写入的数据。如果多个进程读取同一个设备，它们会为数据发生竞争。**scullpipe** 的内部实现将说明阻塞式和非阻塞式读/写如何实现，而无须借助于中断。虽然实际的驱动程序使用硬件中断与它们的设备保持同步，但阻塞式和非阻塞式操作是一个重要内容，并且区别于中断处理（第 9 章将作介绍）。

```
scullsingle
scullpriv
sculluid
scullwuid
```

这些设备与 **scull0** 相似，但在何时允许 **open** 操作方面受到某些限制。第一个（**scullsingle**）一次只允许一个进程使用该驱动程序，而 **scullpriv** 对每个虚拟控制台（或 X 终端会话）是私有的，因为每个控制台/终端将获取一块与其它控制台上进程不同的内存区。**sculluid** 和 **scullwuid** 可被多次打开，但每次只能由一个用户打开；如果另一用户锁定了设备，**sculluid** 返回“设备忙”的错误，而 **scullwuid** 则实现了阻塞式的 **open**。这些 **scull** 设备的变化类型相对“机制”而言，所增加的更多是“策略。”总之，这类处理值得去了解的，因为某些设备需要不同类型的管理方式，就象 **scull** 各种类型的设备一样，它们就是这些管理机制的一部分。

每个 **scull** 设备都展示了驱动程序的不同功能，也提出了不同的难点。本章主要涉及 **scull0-3** 的内部结构；更为复杂的设备将在第 5 章介绍：“样例实现：**scullpipe**”讲解 **scullpipe**，“设备文件的访问控制”介绍其它设备。

3.2 主设备号和次设备号

访问字符设备是通过文件系统内的设备名称进行的。那些名称被称为特殊文件、设备文件，或者简单称之为文件系统树的节点，它们通常位于 **/dev** 目录。字符设备驱动程序的设备文件可通过 **ls -l** 命令输出的第一列中的“c”来识别。块设备也在 **/dev** 下，但它们是由字符“b”标识的。本章内容主要集中于字符设备，不过下面介绍的许多内容同样也适用于块设备。

如果执行 **ls -l** 命令，就可以在设备文件项的最后修改日期前看到两个数（用逗号分隔），这个位置通常显示的是普通文件的长度，而这时这两个数就是相应设备的主设备号和次设备号。下面的列表给出了典型系统中的一些设备。它们的主设备号是 1，4，7 和 10，而次设备号是 1，3，5，64，65 和 129。

```
crw-rw-rw- 1 root  root   1, 3  Feb 23 1999 null
crw----- 1 root  root  10, 1  Feb 23 1999 psaux
crw----- 1 rubini tty   4, 1   Aug 16 22:22 tty1
crw-rw-rw- 1 root  dialout 4, 64  Jun 30 11:19 ttyS0
crw-rw-rw- 1 root  dialout 4, 65  Aug 16 00:00 ttyS1
crw----- 1 root  sys    7, 1   Feb 23 1999 vcs1
crw----- 1 root  sys    7, 129 Feb 23 1999 vcsa1
crw-rw-rw- 1 root  root   1, 5   Feb 23 1999 zero
```

主设备号标识设备对应的驱动程序。例如，**/dev/null** 和 **/dev/zero** 由驱动程序 1 管理，而虚拟控制台和串口终端由驱动程序 4 管理；类似地，**vcs1** 和 **vcsa1** 设备都由驱动程序 7 管理。内核利用

主设备号在 `open` 操作中将设备与相应的驱动程序对应起来。

次设备号只是由那些主设备号已经确定的驱动程序使用，内核的其它部分不会用到它，而仅是把它传递给驱动程序。一个驱动程序控制多个设备是常有的事情（如上面的例子所示），而次设备号为驱动程序提供了一种区分不同设备的方法。

2.4 内核引入了一种新的（可选的）特征，也就是设备文件系统或 `devfs`，如果使用这种文件系统，设备文件管理将被简化，而且也会大为不同。另一方面，这种新的文件系统带来了一些用户可见的不兼容性，在本书编写阶段，它还没被系统发行商选作一个缺省的特征。因此，前面以及接下来的内容中，关于增加新的驱动程序和设备文件的讲解，都假定 `devfs` 尚未引入。本章后面的“设备文件系统”一节将讲述 `devfs` 的相关内容。

当未采用 `devfs` 时，向系统增加一个新的驱动程序意味着为其分配一个主设备号。这个分配工作在驱动程序（模块）初始化时进行，由定义在 `<linux/fs.h>` 中的如下函数实现：

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

返回值提示操作成功还是失败。负的返回值提示错误；0 或正的返回值表明成功完成。`major` 参数是被请求的主设备号，`name` 是设备的名称，该名称将出现在 `/proc/devices` 中，`fops` 是指向函数指针数组的指针，这些函数是调用驱动程序的入口点，这些将在本章后面的“文件操作”一节进行说明。

主设备号是用来索引字符设备静态数组的一个小整数，本章后面的“动态分配主设备号”一节将介绍怎样选择一个主设备号。2.0 内核支持 128 个设备；2.2 和 2.4 内核则增加到 256 个（保留了 0 和 255 这两个值以备将来之需）。次设备号同样也占 8 位；它们不必传递给函数 `register_chrdev`，因为正如已经提到的，它们仅由驱动程序自己使用。目前有来自开发者群体的极大压力，要求进一步增加内核可支持的设备数目；将设备号增加到至少 16 位将是 2.5 版本内核开发的一个既定目标。

一旦设备注册到内核表中，它的操作就和指定的主设备号对应了起来。当我们在主设备号对应的字符设备文件上进行某个操作时，内核将从 `file_operations` 结构中找到并调用正确的函数。出于这个原因，传递给 `register_chrdev` 的指针应指向驱动程序中的一个全局性数据结构，而不是模块初始化函数中的局部数据结构。

接下来的问题就是，如何给程序一个名字，通过这个名字，程序向设备驱动程序发出请求。这个名字必须插入到 `/dev` 目录中，并与驱动程序的主设备号和次设备号相关联。

在文件系统上创建一个设备节点的命令是 `mknod`，只有超级用户才能创建设备。除了要创建的节点名字外，该命令还带三个参数。例如，命令：

```
mknod /dev/scull0 c 254 0
```

创建一个字符设备（c），主设备号是 254，次设备号是 0。次设备号应该在 0-255 范围内，这是因

为出于某些历史原因，它们存储在单个字节中。有很多原因要求扩展次设备号可用的范围，但就目前来说，仍然限制在 8 位。

请注意，一旦通过 `mknod` 创建了设备文件，该文件将一直保留下来，除非明确地将其删除，这一点与存储在磁盘上的其它信息是类似的。读者可以通过命令 `rm /dev/scull0` 将这个例子中创建的设备删除。

3.2.1 动态分配主设备号

一部分主设备号已经静态地分配给了大部分常见设备。在内核源码树的 `Documentation/device.txt` 文件中可以找到这些设备的清单。由于许多数字已经分配了，为新设备选择一个独一无二的设备号是很困难的——定制的驱动程序远比可用的主设备号多得多。不过可以使用为“实验或本地用途”^{*}所保留的某个主设备号。

但是如果对多个“本地”驱动程序作实验，或者要为第三方提供驱动程序的话，就会再次面临选择合适设备号的问题。

幸运的是（更恰当地说是感谢某些人的才智），我们现在可以动态分配主设备号。如果在调用 `register_chrdev` 时的 `major` 为零的话，这个函数就会选择一个空闲号码并做为返回值返回。返回的主设备号总是正的，负的返回值则是错误码。请注意在两种情形下，行为稍有不同：如果调用者是请求一个动态设备号的话，这个函数返回动态分配的数字，而当成功注册了一个预定义好的主设备号时，它返回的是 0（并非主设备号）。

对于专有的驱动程序，我们强烈推荐读者不要随便选择一个当前未使用的设备号做为主设备号，而应该使用动态分配机制获取主设备号。另一方面，如果驱动程序对大家有使用价值，并被包含进正式的内核源码树的话，就需要申请分配一个独占使用的主设备号。

动态分配的缺点是，由于分配的主设备号不能保证始终一致，所以无法预先创建设备节点。这意味着我们无法使用驱动程序的“按需载入”功能（第 11 章将介绍这一高级功能）。对于驱动程序的一般用法，这倒不是什么问题，因为一旦分配了设备号，就可以从 `/proc/devices` 中读取得到。为了加载一个使用动态主设备号的设备驱动程序，对 `insmod` 的调用可替换为一个简单的脚本，该脚本在调用 `insmod` 之后，读取 `/proc/devices` 获得新分配的主设备号，以便创建对应的设备文件。

典型的 `/proc/devices` 文件一般如下所示：

```
Character devices:
1 mem
2 pty
3 tty
4 ttyS
6 lp
7 vcs
10 misc
13 input
14 sound
21 sg
```

^{*} 位于 60 到 63, 120 到 127, 240 到 254 范围内的主设备号是为本地或实验用途所保留的，也就是说，不会有实际设备采用这些主设备号。

```
180 usb
```

```
Block devices:
```

```
2 fd
8 sd
11 sr
65 sd
66 sd
```

主设备号动态分配时，加载这类驱动程序模块的脚本，可以利用 `awk` 这类工具从 `/proc/devices` 中获取信息，并在 `/dev` 目录中创建文件。

下面名为 `scull_load` 的脚本，是 `scull` 发布内容的一部分。使用以模块形式发行驱动程序的用户，可以在系统的 `rc.local` 文件中调用这个脚本，或是在需要模块时手工调用。

```
#!/bin/sh
module="scull"
device="scull"
mode="664"

# invoke insmod with all arguments we were passed
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod -f ./${module}.o $* || exit 1

# remove stale nodes
rm -f /dev/${device}[0-3]

major=`awk "\$2==\"$module\" {print \$1}" /proc/devices`

mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

# give appropriate group/permissions, and change the group.
# Not all distributions have staff; some have "wheel" instead.
group="staff"
grep '^staff:' /etc/group > /dev/null || group="wheel"

chgrp $group /dev/${device}[0-3]
chmod $mode /dev/${device}[0-3]
```

这个脚本同样可以适用于其它驱动程序，只要重新定义变量并调整 `mknod` 那几行就可以了。该脚本创建了 4 个设备，因为 `scull` 的源码默认即创建 4 个设备。

脚本的最后几行看起来有点奇怪：为什么要改变设备的组和访问模式呢？原因在于这个脚本必须由超级用户运行，所以新创建的设备文件自然属于 `root`。缺省权限位只允许 `root` 对其有写访问权，而其它用户只有读权限。正常情况下，设备节点需要不同的访问策略，因此有时需要修改访问权限。我们的脚本默认地把访问权赋予一个用户组，而读者的需求可能有所不同。第 5 章“设备文件的访问控制”一节中，`sculluid` 的代码将会展示设备驱动程序如何实现自己的设备访问授权。除 `scull_load` 外，还有一个 `scull_unload` 脚本用来清除 `/dev` 目录下的相关设备文件并卸载这个模块。

除了使用这一对装载和卸载模块的脚本外，我们还可以编写一个 `init` 脚本，并将其保存在发行版

使用的 `init` 脚本目录中。^{*}

作为 `scull` 源码的一部分，我们提供了相当详尽和可配置的 `init` 脚本范例，名为 `scull.init`。它接收常用的参数 —— “`start`”、“`stop`”或“`restart`”，而且可完成 `scull_load` 和 `scull_unload` 的双重任务。

如果反复创建和删除 `/dev` 节点显得有些不必要的话，有一个解决的方法。如果只是装载和卸载单个驱动程序，可在第一次创建设备文件之后，仅使用 `rmmod` 和 `insmod` 这两个命令：因为动态设备号不是随机生成的，如果不受其它（动态）模块影响的话，可以预期获得到相同的动态主设备号。在开发过程中避免脚本过长是有益的。但很明显，这个技巧不能适用于同时有多个驱动程序的情况。

在我们看来，分配主设备号的最佳方式是，默认地采用动态分配，同时保留在加载时，甚至是编译时，指定主设备号的余地。我们所建议的代码和用于端口自动探测的代码很相似。`scull` 设计中使用了一个全局变量，`scull_major`，保存所选择的设备号。该变量的初始化值是 `SCULL_MAJOR`，这个宏定义在 `scull.h` 中。在发行的源码中 `SCULL_MAJOR` 默认为 0，即“选择动态分配”。用户可以使用这个默认值或选择某个特定的主设备号，既可以在编译前修改宏定义，也可以在 `insmod` 命令行中指定。最后，通过使用 `scull_load` 脚本，用户可以在 `scull_load` 的命令行中将参数传递给 `insmod`。^{*}

下面是 `scull.c` 中用来获取主设备号的代码：

```
result = register_chrdev(scull_major, "scull", &scull_fops);
if (result < 0) {
    printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
    return result;
}
if (scull_major == 0) scull_major = result; /* dynamic */
```

3.2.2 从系统中删除设备驱动程序

当从系统中卸载一个模块时，应该释放主设备号。这一操作可以在 `cleanup_module` 中调用如下函数完成：

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

参数列表包括要释放的主设备号和相应的设备名。参数中的这个设备名，会被内核用来和设备号参数所对应的已注册设备名进行比较，如果不同，返回 `-EINVAL`。如果主设备号超出了所允许的范围，内核同样返回 `-EINVAL`。

在 `cleanup_module` 中注销资源失败会有很不利的后果。下次读取 `/proc/devices` 时，由于其中一个 `name` 字符串仍然指向模块内存，而这块内存不再被映射，所以系统将出错。这种类型的出错称

^{*} 不同的发行版把 `init` 脚本放在不同的位置，最常见的目录是 `/etc/init.d`、`/etc/rc.d/init.d` 和 `/sbin/init.d` 等等。另外，如果脚本需要在引导阶段执行，则需要在对应该运行级别的目录（比如，`.../rc3.d`）中建立一个指向该脚本的符号链接。

^{*} `init` 脚本 `scull.init` 不能在命令行中接收驱动程序选项，但它支持一个配置文件，这是因为这个脚本是为启动和关机时自动运行设计的。

为 `oops`^{*}，因为这是内核访问无效地址时打印的消息。

如果在卸载驱动程序时没有注销主设备号，情况将是很难恢复的，因为 `unregister_chrdev` 中调用的 `strcmp` 函数会使用指向最初模块的指针（即 `name`）。如果注销主设备号失败，就必须重新载入相同的模块，以及另一个用于注销这个主设备号的模块。如果没有修改代码，运气好的话，这个出问题的模块将获得相同的地址，而 `name` 字符串处于相同的位置。当然，更为安全的选择就是重新启动系统。

除了卸载模块，还经常需要在卸载驱动程序时删除设备节点。这样的任务可以由与装载时使用的脚本相配对的脚本来完成。对于我们的样例设备，脚本 `scull_unload` 完成了这个工作，另外通过调用 `scull.init stop` 也可做到这一点。如果动态节点没有从 `/dev` 中删除，就有可能造成不可预期的错误：如果两个驱动程序所使用的动态主设备号相同，开发者计算机上的某个空闲 `/dev/framegrabber` 就有可能在一个月之后才访问火警设备。试图打开 `/dev/framegrabber` 时，如果能得到 “No such file or directory（没有这个文件或目录）” 的回应，总比这个新设备可能导致的后果要好得多。

3.2.3 `dev_t` 和 `kdev_t`

到目前为止，我们已经谈论了主设备号，接下来讨论次设备号，以及驱动程序是如何使用次设备号来区分设备的。

每次在内核调用一个设备驱动程序时，它都会告诉驱动程序它正在操作哪个设备。主设备号和次设备号合在一起构成单个数据类型并用来标识特定的设备。组合后的设备号（主设备号和次设备号合在一起）保存于在稍后介绍的索引节点（`inode`）结构的 `i_rdev` 成员中。每个驱动程序接收一个指向 `inode` 结构的指针作为第一个参数。假定该指针称作 `inode`（和大多数驱动程序开发人员），则可以通过 `inode->i_rdev` 得出设备号。

历史上，Unix 使用 `dev_t`（设备类型）保存设备号。`dev_t` 通常是 `<sys/types.h>` 中定义的一个 16 位整数。而现在有时需要超过 256 个次设备号，但是由于有许多应用程序都已“了解” `dev_t` 的内部结构，一旦改变 `dev_t` 的结构就会造成这些应用程序无法运行，所以很难改变 `dev_t` 的定义。因此，虽然已有的很多基础性工作是为了能够处理更大的设备号而准备的，但它们如今仍被视为 16 位整数。

不过，现在的 Linux 内核使用了一个新的类型，即 `kdev_t`。对于每一个内核函数来说，这个新类型被设计为一个黑箱。用户程序完全不了解 `kdev_t`，而且内核函数也不知道 `kdev_t` 中究竟有些什么。如果 `kdev_t` 一直保持隐藏，它就可以在内核的不同版本间任意变化，而不必修改每个人的设备驱动程序。

`kdev_t` 的相关信息在 `<linux/kdev_t.h>` 中定义，其中大部分是注释。如果读者对代码背后的推理感兴趣的话，这个头文件是一段很有指导性的代码。因为 `<linux/fs.h>` 已经包含了这个头文件，所以没有必要显式地包含这个文件。下列的这些宏和函数是可以对 `kdev_t` 执行的操作：

```
MAJOR(kdev_t dev);
```

^{*} `oops` 的英文原意是“表示惊讶时所发出的喊声”，这个词被热衷于 Linux 的爱好者们既作名词用也作动词用。

从 `kdev_t` 结构中得出主设备号。

```
MINOR(kdev_t dev);
```

得出次设备号。

```
MKDEV(int ma, int mi);
```

通过主设备号和次设备号创建 `kdev_t`。

```
kdev_t_to_nr(kdev_t dev);
```

将 `kdev_t` 转换为一个整数（一个 `dev_t`）。

```
to_kdev_t(int dev);
```

将一个整数转换为 `kdev_t`。注意，内核模式中没有定义 `dev_t`，因此使用了 `int`。

只要在程序中采用这些操作去处理设备号，即便内部数据结构发生了变化，代码依然能正常工作。

3.3 文件操作

在接下来的几节中，我们将着眼于对所管理的设备，驱动程序能完成哪些不同的操作。打开的设备在内核内部由 `file` 结构标识，内核使用 `file_operations` 结构访问驱动程序的函数。`file_operations` 结构是一个定义在 `<linux/fs.h>` 中的函数指针数组。每个文件都与它自己的函数集相关联（通过指向 `file_operations` 结构的一个名为 `f_op` 的指针成员）。这些操作主要负责系统调用的实现，并因此被命名为 `open`，`read` 等。我们可以认为文件是一个“对象”，操作它的函数是“方法”。如果采用面向对象编程的术语来表达就是，对象声明的动作，将作用于其本身。这是我们在 Linux 内核中看到的面向对象化编程的第一个例证，在后面的章节中还会看到更多。

按照惯例，`file_operations` 结构或指向这类结构的指针称为 `fops`（或者是与此相关的其它称法）；我们已经看到 `register_chrdev` 调用中有一个指针参数就是 `fops`。这个结构中的每一个成员都必须指向驱动程序中实现特定操作的函数。对于不支持的操作，对应的成员可置为 `NULL` 值。对各个函数而言，如果对应成员被赋为 `NULL` 指针，那么内核的具体处理行为是不尽相同的，本节后面的列表会列出这些差异。

随着内核不断增加新的功能，`file_operations` 结构已逐渐变得越来越大。新增加的操作自然会对设备驱动程序带来移植性的问题。每个驱动程序中该结构的实例都是用标准的 C 语法声明的，新的操作一般添加在该结构的末尾，这样，对驱动程序简单地进行一次重新编译，这些操作都会被赋予 `NULL`，因此也就选择为缺省的行为，一般来说这正是期望的状态。

后来，内核开发人员又转而采用一种“标记化”的初始化格式，这种格式允许用名字对这类结构的成员进行初始化，也就避免了因数据结构发生变化而带来的麻烦。这种标记化的初始化处理，并不是标准 C 的规范，而是对 GUN 编译器的一种（有用的）特殊扩展。很快我们会看到一个标记化的结构初始化范例。

下面列出了应用程序可在某个设备上调用的所有操作。为便于查询，我们尽量使之简洁，仅仅总结了每个操作，以及使用 `NULL` 时的内核缺省行为。读者可以在初次阅读时跳过这张表，需要的时候再来查阅。

介绍完另一个重要数据结构（也就是 **file**，它实际上包含了指向它拥有的 **file_operations** 结构的指针）后，本章其余部分将讲解最重要的一些操作并给出一些技巧、警告和实际的代码样例。由于我们尚未深入探讨内存管理、块操作和异步通知机制，其它更为复杂的操作将在以后的章节中介绍。

下面的表给出了 2.4 系列内核中 **file_operations** 结构所包括的操作。在这方面，2.4 内核和早期版本之间的差别较小，本章后面将谈到这些差异，因此，我们首先专注 2.4 这个版本的情况。各操作的返回值为 0 表示成功，为负的话则说明发生错误，除非另有所指。

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

方法 **llseek** 用来修改文件的当前读写位置，并将新位置做为（正的）返回值返回。参数 **loff_t** 一个“长偏移量”，即使在 32 位平台上也至少占用 64 位的数据宽度。出错时返回一个负的返回值。如果驱动程序没有设置这个函数，相对文件尾（EOF）的定位操作会失败，而其它的定位操作将修改 **file** 结构（在本章后面的“**file** 结构”一节介绍）中的位置计数器并成功返回。

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

用来从设备中读取数据。该函数指针被赋为 **NULL** 值时，将导致 **read** 系统调用出错并返回 **-EINVAL**（“Invalid argument，非法参数”）。函数返回非负值表示成功读取的字节数（返回值为“signed size”数据类型，通常就是目标平台上的固有整数类型）。

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

向设备发送数据。如果没有这个函数，**write** 系统调用向调用程序返回一个 **-EINVAL**。如果返回值非负，则表示成功写入的字节数。

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

对于设备节点来说，这个成员应该为 **NULL**，它仅用于读取目录，只文件系统有用。

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

poll 方法是 **poll** 和 **select** 这两个系统调用的后端实现。这两个系统调用可用来查询设备是否可读或可写，或是否处于某种特殊状态。这两个系统调用是可阻塞的，直至设备变为可读或可写状态为止。如果驱动程序没有定义它的 **poll** 方法，它驱动的设备就会被认为既可读也可写，并且不会处于其它的特殊状态。返回值是一个描述设备状态的位掩码。

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

系统调用 **ioctl** 提供一种执行设备相关命令的方法（如格式化软盘的某个磁道，这既不是读操作也不是写操作）。另外，内核还能识别一部分 **ioctl** 命令，而不必调用 **fops** 表中的 **ioctl**。如果设备不提供 **ioctl** 入口点，则对于任何内核未预先定义的请求，**ioctl** 系统调用将返回错误（**-ENOTTY**，“No such ioctl for device，该设备无此 **ioctl** 命令”）。如果该设备方法返回一个非负值，那相同的值会返回给调用程序提示调用成功。

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

mmap 用于请求将设备内存映射到进程地址空间。如果设备没有实现这个方法，**mmap** 系统调用将返回 **-ENODEV**。

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

尽管这始终是对设备文件执行的第一个操作，然而却并不要求驱动程序一定要声明这个方法。如果这个入口为 **NULL**，设备的打开操作永远成功，但系统不会通知驱动程序。


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

对 **flush** 操作的调用发生在进程关闭设备文件描述符复本的时候，它应该完成（并等待）设备上尚为完结的操作。请不要将它同用户程序使用的 **fsync** 操作相混淆。目前，**flush** 仅仅用于网络文件系统（NFS）代码。如果 **flush** 被置为 **NULL**，它只是简单地不被调用。

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

当 **file** 结构被释放时，将调用这个操作。与 **open** 相仿，也可以没有 **release**^{*}。

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

该方法是 **fsync** 系统调用的后端实现，用户调用它来刷新待处理的数据。如果驱动程序没有实现这一方法，**fsync** 系统调用返回 **-EINVAL**。

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

这个操作用来通知设备，它的 **FASYNC** 标志发生了变化。异步通知是比较高级的话题，将在第 5 章介绍。如果设备不支持异步通知，该成员可以是 **NULL**。

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

lock 方法用于实现文件锁定，锁定是常规文件不可缺少的特性，但设备驱动程序几乎从来不会实现这个方法。

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

这些方法是在 2.3 版本周期的后期新增的，用来实现“分散/聚集”型的读写操作。应用程序有时需要进行涉及多个内存区域的单次读或写操作，利用上面这些系统调用可完成这类工作，而不必强加额外的数据拷贝操作。

```
struct module *owner;
```

这个成员并不象 **file_operations** 结构中的其它部分那样是个方法。取而代之的是，它是指向“拥有”该结构的模块的指针，内核使用该指针维护模块的使用计数。**scull** 设备驱动程序所实现的只是最重要的设备方法，并且采用标记化格式声明它的 **file_operations** 结构：

```
struct file_operations scull_fops = {
    llseek: scull_llseek,
    read: scull_read,
    write: scull_write,
    ioctl: scull_ioctl,
    open: scull_open,
    release: scull_release,
};
```

这个声明采用了前面提到的标记化的结构初始化语法。这种语法是值得采用的，因为它使驱动程序在结构的定义发生变化时更具可移植性，并且使得代码更加紧凑且易读。标记化的初始化方法允许对结构成员进行重新排列。在某些场合下，将频繁访问的成员放在相同的硬件缓存线上，将大大提高性能。

设置 **file_operations** 结构中的 **owner** 成员也是必要的。在有些内核代码中，常可以看到以下面这样的标记语法初始化 **owner** 成员：

^{*} 注意，**release** 并不会在进程每次调用 **close** 时都会被调用。只要 **file** 结构被共享（如在 **fork** 或 **dup** 调用之后），**release** 会等到所有的复本都关闭之后才会得到调用。如果希望在任意一个复本被关闭时，刷新那些待处理的数据，则应实现 **flush** 方法。

```
owner: THIS_MODULE,
```

不过这种方法仅在 2.4 内核上才会起作用。更具可移植性的方法是使用宏 `SET_MODULE_OWNER`，它定义在 `<linux/module.h>` 中。`scull` 如下执行这种初始化工作：

```
SET_MODULE_OWNER(&scull_fops);
```

这个宏对于任何含 `owner` 成员的结构均有效。我们将在本书其余部分多次遇到这个成员。

3.3.1 file 结构

`<linux/fs.h>` 中定义的 `file` 结构是设备驱动程序所使用的另一个最重要的数据结构。注意，`file` 结构与用户程序中的 `FILE` 没有任何关联。`FILE` 定义在 C 库且不会出现在内核代码中；而 `struct file` 是一个内核结构，它不会出现在用户程序中。

`file` 结构代表一个打开的文件（它并不仅仅限定于设备驱动程序；系统中每个打开的文件在内核空间都有一个对应的 `file` 结构）。它由内核在 `open` 时创建，并传递给在该文件上进行操作的所有函数，直到最后的 `close` 函数。在文件的所有实例都被关闭之后，内核释放这个数据结构。注意，一个打开的文件和磁盘文件不同，后者由 `struct inode` 表示。

在内核源码中，指向 `struct file` 的指针通常称为 `file` 或 `filp`（“文件指针”）。为了不致于和这个结构本身相混淆，我们一致称该指针为 `filp`。这样的话，`file` 指的是结构本身，`filp` 则是指向该结构的指针。

`struct file` 中最重要的成员罗列如下。与上节相似，这张表在首次阅读时可以略过。在下一节中，将看到一些真正的 C 代码，我们会讨论其中的某些成员，到时读者可以反过来查阅这张表。

```
mode_t f_mode;
```

文件模式通过 `FMODE_READ` 和 `FMODE_WRITE` 位来标识文件是否可读或可写。读者可能会认为要在自己的 `ioctl` 函数中查看这个成员来检查读/写许可，但由于内核在调用驱动程序的 `read` 和 `write` 前已经检查了许可，所以并不必为这两个方法检查许可。例如，一个未得到允许的写操作在驱动程序还不知道的情况下就已经被内核拒绝了。

```
loff_t f_pos;
```

当前读/写位置。`loff_t` 是一个 64 位的数（用 `gcc` 的术语就是 `long long`）。如果驱动程序需要知道文件中的当前位置，可以读取这个值，但不要去修改它（`read/write` 会使用它们接收到的最后那个指针参数来更新这一位置，而不是直接对 `filp->f_pos` 操作）。

```
unsigned int f_flags;
```

文件标志，如 `O_RDONLY`、`O_NONBLOCK` 和 `O_SYNC`。驱动程序为了支持非阻塞型操作需要检查这个标志，而其它标志很少用到。注意，检查读/写许可应该查看 `f_mode` 而不是 `f_flags`。所有这些标志都定义在 `<linux/fcntl.h>` 中。

```
struct file_operations *f_op;
```

与文件相关的操作。内核在执行 `open` 操作时对这个指针赋值，以后需要处理这些操作时就读取这个指针。`filp->f_op` 中的值决不会为方便引用而保存起来，也就是说，我们可以在任何需要的时

候修改文件的关联操作，在返回给调用者之后，新的操作方法就会立即生效。例如，对应于主设备号 1（/dev/null、/dev/zero 等等）的 `open` 代码根据要打开的次设备号替换 `filp->f_op` 中的操作。这种技巧允许相同主设备号下的设备实现多种操作行为，而不会增加系统调用的负担。这种替换文件操作的能力在面向对象编程技术中称为“方法重载”。

```
void *private_data;
```

`open` 系统调用在调用驱动程序的 `open` 方法前将这个指针置为 `NULL`。驱动程序可以将这个成员用于任何目的或者忽略这个成员。驱动程序可以用这个成员指向已分配的数据，但是一定要在内核销毁 `file` 结构前在 `release` 方法中释放内存。`private_data` 是跨系统调用时保存状态信息的非常有用的资源，我们的大部分示例都使用了它。

```
struct dentry *f_dentry;
```

文件对应的目录项（`dentry`）结构。目录项是一种优化设计，在 2.1 系列版本内核中就已经引入了。除了用 `filp->f_dentry->d_inode` 的方式来访问索引节点结构之外，设备驱动程序的开发者们一般无需关心 `dentry` 结构。

实际的结构里还有其它一些成员，但它们对于设备驱动程序并没有多大用处。由于驱动程序从不填写 `file` 结构，而只是对别处创建的这些结构进行访问，所以可以安全地忽略这些成员。

3.3.2 `open` 和 `release`

现在我们已经简单浏览了这些成员，下面将在实际的 `scull` 函数中使用这些成员。

`open` 方法

`open` 方法是为以后的操作完成初始化准备工作而提供给驱动程序的。此外，`open` 一般还会增加设备的使用计数，防止在文件关闭前模块被卸载出内核。这个计数值在 `release` 方法中被递减，第 2 章的“使用计数”一节已经作了讲解。

在大部分驱动程序中，`open` 应完成如下工作：

- 增加使用计数。
 - 检查设备相关错误（诸如设备未就绪或类似的硬件问题）。
 - 如果设备是首次打开，则对其初始化。
 - 识别次设备号，如有必要更新 `f_op` 指针。
- 分配并填写置于 `filp->private_data` 里的数据结构。

在 `scull` 中，上面大部分预先的操作都取决于被打开设备的次设备号。因此，首先要做的就是识别要操作的是哪个设备。可以通过查看 `inode->i_rdev` 做到这一点。

我们已经谈过内核是不使用次设备号的，因此驱动程序可以随意使用它们。事实上，不同的次设备号用来访问不同的设备，或以不同的方式打开同一个设备。例如，`/dev/st0`（次设备号 0）和 `/dev/st1`（次设备号 1）涉及两个不同的 `SCSI` 磁带驱动，而 `/dev/nst0`（次设备号 128）与 `/dev/st0` 是相同的物理设备，但它的操作行为不同（该设备在关闭时并不重绕磁带）。所有的磁带设备都有不同的

次设备号，这样驱动程序就能区分它们。

驱动程序实际上完全不知道被打开设备的名字，它仅仅知道设备号——用户可以根据这一点，为设备取一个别名。如果创建了两个主/次设备号完全相同的设备文件，却只有一个相同设备的话，那就没有方法区分它们。同样的效果可以采用符号链接或硬链接来获得，实现别名的推荐方法是创建符号链接。

scull 驱动程序是这样使用次设备号的：字节的高 4 位标识设备的类型（personality，个性），低 4 位可以在某个类型的设备支持多个设备实例时，用于区分各个设备。因此，scull0 的高 4 位与 scullpipe0 不同，而 scull0 的低 4 位与 scull1 不同*。

源码中定义了两个宏（TYPE 和 NUM）从设备号中分解出这些位，如下所示：

```
#define TYPE(dev) (MINOR(dev) >> 4) /* high nibble */
#define NUM(dev) (MINOR(dev) & 0xf) /* low nibble */
```

对于每一设备类型，scull 定义了一个特定的 file_operations 结构，它在 open 操作时赋给 filp->f_op。下面的代码显示了多个 fops 是如何实现的：

```
struct file_operations *scull_fop_array[]={
    &scull_fops, /* type 0 */
    &scull_priv_fops, /* type 1 */
    &scull_pipe_fops, /* type 2 */
    &scull_sngl_fops, /* type 3 */
    &scull_user_fops, /* type 4 */
    &scull_wusr_fops /* type 5 */
};
#define SCULL_MAX_TYPE 5

/* In scull_open, the fop_array is used according to TYPE(dev) */
int type = TYPE(inode->i_rdev);

    if (type > SCULL_MAX_TYPE) return -ENODEV;
    filp->f_op = scull_fop_array[type];
```

内核根据主设备号调用 open，scull 则使用上述宏分解出的次设备号。TYPE 用以索引 scull_fop_array 数组，从中取出正被打开的设备类型所对应的方法集。

在 scull 中，根据次设备号判断设备的类型，并赋予 filp->f_op 由设备类型所决定的对应 file_operations 结构。然后新的 fops 中声明的 open 方法得到调用。通常，驱动程序并不调用自己的 fops，这些 fops 是内核分发对应的驱动程序方法时调用的。不过，当 open 方法不得不处理不同的设备类型时，则在修改了 fops 指针后，根据被打开设备的次设备号，或许就需要调用 fops->open 了。

scull_open 的实际代码如下。它使用了前面那段代码中定义的 TYPE 和 NUM 两个宏来得出次设备号：

```
int scull_open(struct inode *inode, struct file *filp)
{
    Scull_Dev *dev; /* device information */
```

* 位切分是使用次设备号的一种典型方式。例如，IDE 驱动程序使用高 2 位表示磁盘号，低 6 位表示分区号。

```

int num = NUM(inode->i_rdev);
int type = TYPE(inode->i_rdev);

/*
 * If private data is not valid, we are not using devfs
 * so use the type (from minor nr.) to select a new f_op
 */
if (!filp->private_data && type) {
    if (type > SCULL_MAX_TYPE) return -ENODEV;
    filp->f_op = scull_fop_array[type];
    return filp->f_op->open(inode, filp); /* dispatch to specific open */
}

/* type 0, check the device number (unless private_data valid) */
dev = (Scull_Dev *)filp->private_data;
if (!dev) {
    if (num >= scull_nr_devs) return -ENODEV;
    dev = &scull_devices[num];
    filp->private_data = dev; /* for other methods */
}

MOD_INC_USE_COUNT; /* Before we maybe sleep */
/* now trim to 0 the length of the device if open was write-only */
if ((filp->f_flags & O_ACCMODE) == O_WRONLY) {
    if (down_interruptible(&dev->sem)) {
        MOD_DEC_USE_COUNT;
        return -ERESTARTSYS;
    }
    scull_trim(dev); /* ignore errors */
    up(&dev->sem);
}

return 0; /* success */
}

```

这里需要作一些解释。用于记录内存区域的数据结构是 **Scull_Dev**，这里简要介绍一下。全局变量 **scull_nr_devs** 和 **scull_devices[]**（全部小写）分别是可用设备数和指向 **Scull_Dev** 的指针数组。

对 **down_interruptible** 和 **up** 的调用现在可以先忽略，我们很快会谈它们。

这段代码看起来相当短小，是因为在调用 **open** 时它并没有做针对某个设备的任何处理。由于 **scull0-3** 设备被设计为全局和持久性的，这段代码无须做什么工作。特别是，由于我们并不维护 **scull** 的打开计数，也就是模块的使用计数，因此也就没有类似于“首次打开时初始化设备”的这类动作。

既然说内核能通过 **owner** 成员维护模块的使用计数的话，那读者可能会奇怪为什么在这里却手工地来增加计数。答案在于早期版本的内核要求模块自己去处理有关维护其使用计数的所有工作，那时 **owner** 的机制还并不存在。为在老版本的内核上具有可移植性，**scull** 自行增加其使用计数。这样的处理在 2.4 版本的系统中会引起使用计数值过高的情况，但那并不算是个问题，因为当模块不再被使用时，计数值会降至零。

对设备唯一的实际操作是，当设备以写方式打开时，它的长度将被截为 0。出现这种特性的原因在于，设计上，用更短的文件覆盖一个 **scull** 设备时，设备数据区应相应缩小。这与普通文件写打开时长度截短为 0 的方式很相似。如果设备以读方式打开，则什么也不做。

稍后在浏览其它 **scull** 设备类型（**personality**）的代码时，将会看到真正的初始化工作是如何完成

的。

release 方法

`release` 方法的作用正好与 `open` 相反。有时读者会发现这个方法的实现被称为 `device_close`，而不是 `device_release`。无论是哪种形式，这个设备方法应该完成下面的任务：

- 释放由 `open` 分配的，保存在 `filp->private_data` 中的所有内容
- 在最后一次关闭操作时关闭设备
- 使用计数减 1

`scull` 的基本模型无需执行关闭设备的动作，因此所需的代码量最少*：

```
int scull_release(struct inode *inode, struct file *filp)
{
    MOD_DEC_USE_COUNT;
    return 0;
}
```

如果在 `open` 期间递增使用计数的话，则不应忘记对其递减，因为如果使用计数不归 0，内核无法卸载模块。

如果某个时刻一个尚未被打开的文件被关闭了，计数将如何保证一致呢？要知道，`dup` 和 `fork` 都会在不调用 `open` 的情况下，创建已打开文件的复本，但每一个都会在程序终止时关闭。例如，大多数程序从来不打开它们的 `stdin` 文件（或设备），但它们都会在终止时关闭。

答案很简单：并不是每个 `close` 系统调用都会引起对 `release` 方法的调用。仅仅是那些实际释放设备数据结构的那些 `close` 调用才会调用这个方法。内核维护一个 `file` 结构被使用多少次的计数器。无论是 `fork` 还是 `dup` 都不创建新的数据结构（仅由 `open` 创建），它们只是增加已有结构中的计数。

只有在 `file` 结构的计数归 0 时，`close` 系统调用才会执行 `release` 方法，这只在删除这个结构时才会发生。`release` 方法与 `close` 系统调用间的关系保证了模块使用计数总是一致的。

注意：`flush` 方法在应用程序每次调用 `close` 时都会被调用。不过，很少有驱动程序去实现 `flush`，因为在 `close` 时并没有什么事情需要去做，除非 `release` 被调用。正如读者猜想的那样，甚至在应用程序未明确关闭它所打开的文件就中止时，以上的讨论同样也是适用的：内核在进程退出的时候，通过内部使用 `close` 系统调用自动关闭相关的文件。

3.4 `scull` 的内存使用

在介绍读写操作以前，我们最好先看看 `scull` 如何并且为什么进行内存分配。为了全面理解代码，我们需要知道“如何分配”，而“为什么分配”则表明了驱动程序编写者所需做出的选择，尽管 `scull`

* 因为 `scull_open` 为每种设备都替换了不同的 `filp->f_op`，所以不同的设备由不同的函数关闭。我们会随后看到这些内容。

作为设备来说肯定还不够代表性。

本节只讲解 `scull` 中的内存分配策略，而不会涉及编写实际驱动程序时需要的硬件管理技巧。这些技巧将在第 8 章和第 9 章中介绍。因此，如果读者对针对内存操作的 `scull` 驱动程序的内部工作原理不感兴趣的话，可以跳过这一节。

`scull` 使用的内存区域，这里也称为设备，其长度是可变的。写的越多，它就变得越长；用更短的文件以覆盖方式写设备时则会变短。

`scull` 的实现方法并不是很巧妙。更为巧妙的实现，其代码读起来会较困难，而本节的目的是讲解 `read` 和 `write`，并非内存管理。这也就是为什么虽然分配整个页面会更有效，但代码只使用了 `kmalloc` 和 `kfree`，而没有采取分配整个页面的操作。

另一方面，从理论和实际角度考虑，我们不想限制“设备”的尺寸。理论角度上，对所管理的数据项任意增加限制总是很糟糕的想法。实际角度上，为了在内存短缺的情况下进行测试，可利用 `scull` 暂时将系统的内存吃光，进行这样的测试有助于了解系统内部。我们可以使用命令 `cp /dev/zero /dev/scull0` 用光所有的系统 RAM，也可以用 `dd` 工具选择复制多少数据到 `scull` 设备中。

在 `scull` 中，每个设备都是一个指针链表，其中每个指针都指向一个 `Scull_Dev` 结构。默认情况下，每一个这样的结构通过一个中间指针数组最多可引用 4,000,000 个字节。我们发布的源码使用了一个有 1000 个指针的数组，每个指针指向一个 4000 字节的区域。我们把每一个内存区称为一个量子，这个指针数组（或它的长度）称为量子集。`scull` 设备和它的内存区如图 3-1 所示。

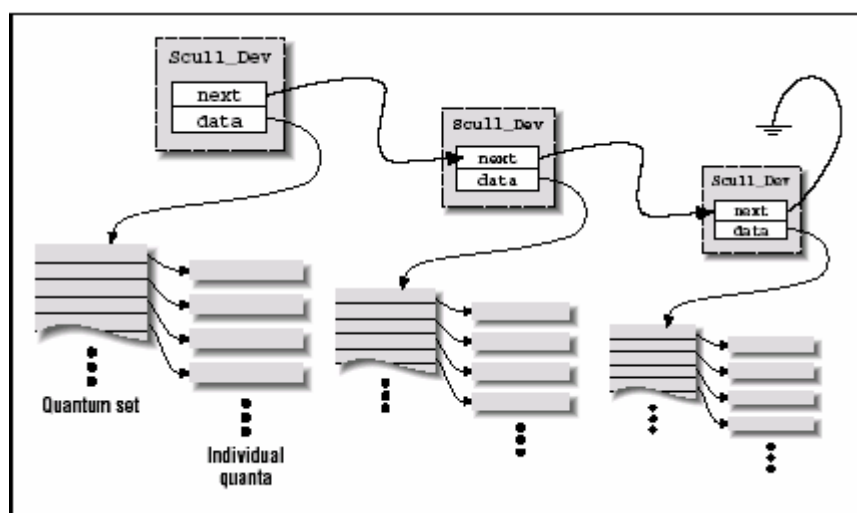


图 3-1: `scull` 设备的布局

这样，向 `scull` 写一个字节就会消耗内存 8000 或 12000 个字节：每个量子占用 4000 个，量子集占用 4000 或 8000 个字节（取决于目标平台上指针本身占用 32 位还是 64 位）。然而，如果向 `scull` 写大量的数据，链表的开支并不会太大。每 4MB 数据只对应一个表项，而设备的最大尺寸被计算机内存的大小所限制。

为量子 and 量子集选择合适的数值是一个策略问题，而非机制问题，而且最优数值依赖于如何使用设备。因此 `scull` 设备的驱动程序不应对量子 and 量子集的尺寸强制使用某个特定的数值。在 `scull` 设备中，用户可以采用几种方式来修改这些值：在编译时，可以修改 `scull.h` 中的 `SCULL_QUANTUM` 和 `SCULL_QSET`；而在模块加载时，可以设置 `scull_quantum` 和 `scull_qset` 的整数值；或者在运行时，使用 `ioctl` 修改当前值和默认值。

使用宏和整数值同时允许在编译期间和加载阶段进行配置，这种方法和前面选择主设备号的方法类似。对于驱动程序中任何不确定的，或与策略相关的数值，我们都可以使用这种技巧。

余下的唯一问题是如何选择缺省数值。在这个例子里，量子 and 量子集未充分填满会导致内存浪费，而量子 and 量子集过小则会在进行内存分配、释放和指针链接等操作时增加系统开销，缺省数值的选择问题就在于寻找这两者之间的最佳平衡点。

此外，还必须考虑 `kmalloc` 的内部设计，然而目前我们还无法涉及这一点。`kmalloc` 的内部结构将在第 7 章 “The Real Story of `kmalloc`” 一节中探讨。

缺省数值的选择基于这样的假设，在测试 `scull` 时，可能会有大块的数据写到其中，但大多数情况下，对该设备的正常使用可能只传递几 K 的数据量。

用来保存设备信息的数据结构如下：

```
typedef struct Scull_Dev {
    void **data;
    struct Scull_Dev *next; /* next list item */
    int quantum; /* the current quantum size */
    int qset; /* the current array size */
    unsigned long size;
    devfs_handle_t handle; /* only used if devfs is there */
    unsigned int access_key; /* used by sculluid and scullpriv */
    struct semaphore sem; /* mutual exclusion semaphore */
} Scull_Dev;
```

下面的代码说明如何利用 `Scull_Dev` 保存数据。`scull_trim` 函数负责释放整个数据区，并且在文件以写方式打开时由 `scull_open` 调用。它简单地遍历链表，释放所有找到的量子 and 量子集。

```
int scull_trim(Scull_Dev *dev)
{
    Scull_Dev *next, *dptr;
    int qset = dev->qset; /* "dev" is not null */
    int i;

    for (dptr = dev; dptr; dptr = next) { /* all the list items */
        if (dptr->data) {
            for (i = 0; i < qset; i++)
                if (dptr->data[i])
                    kfree(dptr->data[i]);
            kfree(dptr->data);
            dptr->data=NULL;
        }
        next=dptr->next;
        if (dptr != dev) kfree(dptr); /* all of them but the first */
    }
    dev->size = 0;
    dev->quantum = scull_quantum;
    dev->qset = scull_qset;
```



```
dev->next = NULL;
return 0;
}
```

3.5 竞态的简介

读者已经了解到 `scull` 管理内存的方法，现在，我们考虑这样一种情形：两个进程，**A** 和 **B**，它们都打开了一个相同的 `scull` 设备用于写操作。这两个进程试图同时添加数据到设备中。这种操作的成功需要一个新的量子，所以每个进程都分配了所需的内存，并在量子集中保存指向这些新分配内存的指针。

结果麻烦就来了。因为两个进程所见的是同一个 `scull` 设备，每个都会把各自新分配的内存指针存放在量子集的相同位置。如果 **A** 先存储了它的指针，**B** 则会在随后同样的存储操作中覆盖原有的指针。因此 **A** 分配的内存，以及已经写入这个区域的数据，都会丢失。

这种状况就是一种典型的竞态——结果取决于是 **A** 还是 **B** 首先执行存储操作，而且一般地说总会有一些无法预期的事情发生。在单处理器的 Linux 系统中，`scull` 的代码不会遇到这种问题，因为运行内核代码的进程是非抢占式的。但在 **SMP** 系统中，情形则要复杂的多。进程 **A** 和 **B** 可能运行在不同的处理器上，从而互相干扰。

Linux 内核提供了几种机制避免并处理竞态。这些机制的完整描述将放在第 9 章，不过在这里有必要先作一个初期讨论。

信号量是用于资源访问控制的一般机制。在最为简单的形式里，信号量可用于互斥操作——使用互斥模式的信号量，可以防止多个进程同时运行相同的代码或存取相同的数据。这种信号量通常被成为 `mutex`，源于互斥（*mutual exclusion*）这个词。

Linux 信号量定义在 `<asm/semaphore.h>` 中，具有 `struct semaphore` 类型，驱动程序只能使用给定的接口操作信号量。在 `scull` 中，每个设备都在 `Scull_Dev` 结构中分配了一个信号量。因为设备是彼此独立的，所以没有必要在多个设备间进行互斥操作。

使用信号量之前，必须传递一个数值参数给 `sema_init` 函数进行初始化。对于互斥型的应用（比如，避免多个线程同时存取相同的数据），信号量须初始化为 `1`，即信号量是可用的。作为设备初始化设置的一部分，`scull` 模块初始化函数（`scull_init`）中接下来的代码说明了信号量是如何被初始化的。

```
for (i=0; i < scull_nr_devs; i++) {
    scull_devices[i].quantum = scull_quantum;
    scull_devices[i].qset = scull_qset;
    sema_init(&scull_devices[i].sem, 1);
}
```

如果一个进程想进入一段被信号量保护的代码的话，必须首先确保没有其它的进程对此进行访问。经典计算机科学把获取信号量的函数功能称为 **P**，而 Linux 中称其为 `down` 或 `down_interruptible`。这些函数检查信号量的值，看它是否大于 `0`；如果是的话，它们将递减信号量的值并返回。如果信号量的值为 `0`，那函数将进入睡眠，直到其它进程释放该信号量并将其唤醒之后，再次进行尝试。`down_interruptible` 可以用一个信号来打断，但 `down` 则不允许有信号抵送

到进程。大多数情况都是希望信号起作用的；否则，就有可能建立一个无法杀掉的进程，以及其它不可预期的结果。但是，允许信号中断将使得信号量的处理复杂化，因为我们总要去检查函数（这里的 `down_interruptible`）是否已被中断。一般地，该函数返回 0 时表示成功，非 0 则出错。如果这个处理过程被中途打断，它并不会获得信号量，因此，也就不能调用 `up` 函数了。因此，援引信号量的典型调用通常是下面的这种形式：

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

返回值 `-ERESTARTSYS` 通知系统操作被信号打断。调用这个设备方法的内核函数或者重新尝试，或者返回 `-EINTR` 至应用程序，这取决于应用程序是如何设置信号处理函数的。当然，如果是以此种方式中断操作的话，代码应在返回前完成清理工作。

获取信号量的进程必须在随后的某个时候释放它。计算机科学称释放操作为 `V`，而 Linux 使用的表达方式是 `up`。一个简单的调用形式就象下面这样：

```
up (&sem);
```

它递增信号量的值，并唤醒正在等待信号量转为可用状态的进程。

信号量必须小心使用。被信号量保护的数据必须是定义清晰的，并且存取这些数据的所有代码都必须首先获得信号量。使用 `down_interruptible` 来获取信号量的代码不应调用其它也试图获得信号量的函数，否则就会陷入死锁。如果驱动程序中的某段程序对其持有的信号量释放失败的话（可能就是一次出错返回的结果），那么其它任何获取该信号量的尝试都将陷在那里。互斥操作一般来说是很需要技巧的，一个定义良好和条理化的方法会带来很多好处。

在 `scull` 中，各设备的信号量用以保护对存储数据的存取。任何访问 `Scull_Dev` 结构之 `data` 成员的代码都必须首先获得信号量。为避免死锁，仅仅是那些实现设备方法的函数才会去获取信号量。而先前介绍过的 `scull_trim` 等内部程序，则假定信号量已经取得。只要保持这几个方面不变，访问 `Scull_Dev` 结构时就可以避免竞态的发生。

3.6 read 和 write

读/写方法完成的任务是相似的，也就是，拷贝数据到应用程序空间，或反过来从应用程序空间拷贝数据。因此，它们的原型相当相似，不妨同时介绍它们：

```
ssize_t read(struct file *filp, char *buff,
             size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char *buff,
             size_t count, loff_t *offp);
```

对于这两个方法，参数 `filp` 是文件指针，参数 `count` 是请求传输的数据长度。参数 `buff` 是指向用户空间的缓冲区，这个缓冲区或者保存将写入的数据，或者是一个存放新读入数据的空缓冲区。最后的 `offp` 是一个指向“long offset type（长偏移量类型）”对象的指针，这个对象指明用户在文件中存取操作的位置。返回值是“signed size type（有符号的尺寸类型）”，后面会谈到它的使用。

谈到数据传输，和两个设备方法的相关的主要问题是，需要在内核地址空间和用户地址空间之间传输数据，不能用通常的办法利用指针或 `memcpy` 来完成这样的操作。出于许多原因，不能在内核空间中直接使用用户空间地址。

内核空间地址与用户空间地址之间很大的一个差异，就是用户空间的内存是可被换出的。当内核访问用户空间指针时，相对应的页面可能已不在内存中了，这样的话就会产生一个页面错。在本节和第 5 章将介绍的一些函数使用了一些隐式技巧来正确地处理页面错，即使 CPU 正在内核空间执行时。

这里，值得注意的是，Linux 2.0 在 x86 平台上，在用户空间和内核空间之间采用了一种完全不同的内存映射方式。这样，用户空间的指针根本不能在内核空间中使用。如果目标设备不是 RAM 而是扩展卡，也有同样的问题，因为驱动程序必须在用户空间缓冲区和内核空间之间拷贝数据（也可能是在内核空间和 I/O 内存之间）。

Linux 中此类跨空间的拷贝是由一些特定的函数完成的，它们定义在 `<asm/uaccess.h>` 中。这样的拷贝或者是通过一般的（如 `memcpy`）函数完成，或者是通过为特定的数据大小（`char`, `short`, `int`, `long`）作了优化的函数来完成，它们大多数将在第 5 章的“使用 `ioctl` 参数”一节介绍。

`scull` 的 `read` 和 `write` 代码要做的工作，就是在用户地址空间和内核地址空间之间进行整段数据的拷贝。这种能力是由下面的内核函数提供的，它们用于拷贝任意的一段字节序列，这也是每个 `read` 和 `write` 方法实现的核心部分。

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

虽然这些函数的行为很象通常的 `memcpy` 函数，但当内核空间内运行的代码访问用户空间时，要多加小心。被寻址的用户空间的页面可能当前并不在内存，于是页面错的对应处理程序会使访问进程转入睡眠，直到该页面被传送至期望的位置。例如，当页面必须从交换空间取回时，这样的情况就会发生。对于驱动程序编写人员，这带来的结果就是访问用户空间的任何函数都必须是可重入的，并且必须能和其它驱动程序函数并发执行（也可参阅第 5 章的“编写重入代码”）。这就是我们使用信号量来控制并发访问的原因。

这两个函数的作用并不限于在内核空间和用户空间之间拷贝数据，它们还检查用户空间的指针是否有效。如果指针无效，就不会进行拷贝；另一方面，如果在拷贝过程中遇到无效地址，则仅仅会复制部分数据。在这两种情况中，返回值是还未拷贝完的内存的数量值。`scull` 代码如果发现这样的错误返回，就会在返回值不为 0 时，返回 `-EFAULT` 给用户。

关于用户空间访问和无效用户空间指针的内容是相对高级的话题，第 5 章的“使用 `ioctl` 参数”一节会对它们进行进一步讨论。如果并不需要检查用户空间指针，那么建议你转而调用 `__copy_to_user` 和 `__copy_from_user`。例如，在知道这些参数已经过检查时，这还是很有用的。

至于谈到实际的设备方法，`read` 方法的任务是从设备拷贝数据到用户空间（使用 `copy_to_user`），而 `write` 方法则是从用户空间拷贝数据到设备上（使用 `copy_from_user`）。每次 `read` 或 `write` 系统调用都会请求一定数目的字节传输，不过驱动程序也并没限制小数据量的传输——读/写之间的

确切规则还是有些细微差异的，本章后面的内容会提到。

无论这些方法传输了多少数据，一般而言都应更新 `*offp` 所表示的文件位置，以便反映在新系统调用成功完成之后当前的文件位置。大多数时候，`offp` 参数就是指向 `filp->f_pos` 的指针，但在对 `pread` 和 `pwrite` 系统调用的支持中，使用了一个不同的指针，`pread/pwrite` 是在单个原子操作种完成 `lseek` 和 `read/write` 等价操作的两个系统调用。图 3-2 表明了一个典型的 `read` 实现是如何使用其参数的。

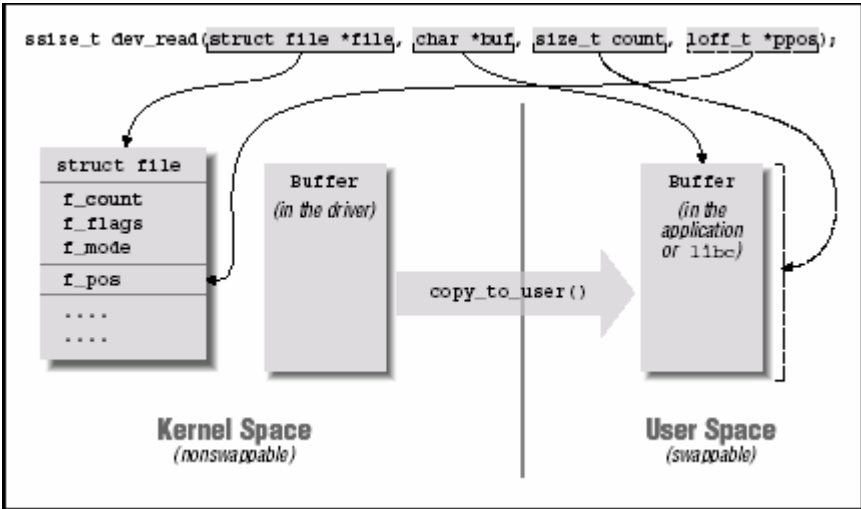


图 3-2: read 的参数

出错时，`read` 和 `write` 方法都返回一个负值。大于等于 0 的返回值告诉调用程序成功传输了多少字节。如果在正确传输部分数据之后发生错误，则返回值必须是成功传输的字节数，这个错误只能在下一次函数调用时才会得到报告。

尽管内核函数通过返回负值来表示错误，而且该返回值表明了错误的类型（见第 2 章“`init_module` 中的错误处理”一节），但运行在用户空间的程序看到的始终是作为返回值的 -1。为了找到出错原因，用户空间的程序必须访问 `errno` 变量。这种行为上的不同源于 POSIX 的系统调用标准，而且还有一个好处，就是内核无须处理 `errno`。

3.6.1 read 方法

调用程序对 `read` 的返回值解释如下：如果返回值等于作为 `count` 参数传递给 `read` 系统调用的值，所请求的字节数传输就成功完成了。这是最理想的情况。如果返回值是正的，但是比 `count` 小，则只有部分数据成功传送。这种情况因设备的不同可能有许多原因。大部分情况下，程序会重新读数据。例如，如果用 `fread` 函数读数据，这个库函数会不断调用系统调用，直至所请求的数据传输完成。

如果返回值为 0，它表示已经到达了文件尾；负值意味着发生了错误，该值指明了发生了什么错误，错误码在 `<linux/errno.h>` 中定义。比如这样的一些错误：-EINTR（系统调用被打断）或 -EFAULT（无效地址）。

上面的表格中遗漏了一种情况，就是“现在还没有数据，但以后会有”。在这种情况下，`read` 系统调用应该阻塞。我们将在第 5 章的“阻塞型 I/O”一节中处理阻塞读入。

`scull` 代码利用了这些规则，特别地，它利用了部分读取的规则。每一次调用 `scull_read` 只处理一个数据量子，而不必通过循环收集所有数据；这样一来代码就更短更易读了。如果读取数据的程序确实需要更多的数据，它可以重新调用这个调用。如果用标准 I/O 库（如 `fread` 等）读取设备，应用程序将不会注意到数据传送的量子化过程。

如果当前的读位置超出了设备大小，`scull` 的 `read` 方法就返回 0 告知程序这里已经没有数据了（换句话说就是，我们已经到文件尾了）。如果进程 A 正在读设备，而此时进程 B 以写入模式打开这个设备，于是设备被截断为长度 0，这种情况是有可能发生的。进程 A 突然发现自己超过了文件尾，并且在下次调用 `read` 时返回 0。

下面是 `read` 的代码：

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

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    if (*f_pos >= dev->size)
        goto out;
    if (*f_pos + count > dev->size)
        count = dev->size - *f_pos;
    /* find list item, qset index, and offset in the quantum */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* follow the list up to the right position (defined elsewhere) */
    dptr = scull_follow(dev, item);

    if (!dptr->data)
        goto out; /* don't fill holes */
    if (!dptr->data[s_pos])
        goto out;
    /* read only up to the end of this quantum */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

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

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

3.6.2 write 方法

与 `read` 类似，根据如下返回值规则，`write` 也能传输少于请求的数据量：

如果返回值等于 `count`，则完成了请求数目的字节传送。如果返回值是正的，但小于 `count`，只传输了部分数据。程序很可能再次试图写入余下的数据。如果值为 0，意味着什么也没写入。这个结果不是错误，而且也没有理由返回一个错误码。再次说明，标准库会重复调用 `write`。在第 5 章介绍阻塞型 `write` 时，我们将详细说明这种情形。负值意味发生了错误，与 `read` 相同，有效的错误码定义在 `<linux/errno.h>` 中。

很不幸，有些错误程序只进行了部分传输就报错并异常退出。这种情况的发生是由于程序员习惯于认定 `write` 调用要么失败要么就完全成功，在大多数时候的确是这样的，设备驱动也应对此进行支持。这种局限性在 `scull` 的实现中可以弥补，但我们不想把代码搞得太复杂，能说明问题就行了，所以，与 `read` 方法一样，`scull` 的 `write` 代码每次只处理一个量子：

```
ssize_t scull_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos)
{
    Scull_Dev *dev = filp->private_data;
    Scull_Dev *dptr;
    int quantum = dev->quantum;
    int qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t ret = -ENOMEM; /* value used in "goto out" statements */

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

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

    /* follow the list up to the right position */
    dptr = scull_follow(dev, item);
    if (!dptr->data) {
        dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
        if (!dptr->data)
            goto out;
        memset(dptr->data, 0, qset * sizeof(char *));
    }
    if (!dptr->data[s_pos]) {
        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
        if (!dptr->data[s_pos])
            goto out;
    }
    /* write only up to the end of this quantum */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

    if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
        ret = -EFAULT;
        goto out;
    }
    *f_pos += count;
    ret = count;

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

```

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

```

3.6.3 readv 和 writev

Unix 系统很久以来就已支持两个可选的系统调用：**readv** 和 **writev**。这些“向量”型的函数具有一个结构数组，每个结构包含一个指向缓冲区的指针和一个长度值。**readv** 调用可用于将指定数量的数据依次读入每个缓冲。 **writev** 则是把各个缓冲区的内容收集起来，并将它们在一次 **write** 操作中进行输出。

然而直到 2.3.44 内核，Linux 始终是通过多次调用 **read/write** 来模拟 **readv** 和 **writev** 调用的。如果驱动程序没有提供用于处理向量操作的方法，这类操作也就只好仍采用模拟方法来实现了。不过，在很多情况下，直接在驱动程序中实现 **readv** 和 **writev** 可以获得更高的效率。

向量操作的函数原型如下：

```

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

```

在这里，**filp** 和 **ppos** 参数与在 **read/write** 中的用法是相同的。**iovec** 结构定义在 `<linux/uio.h>` 中，形式如下：

```

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

```

每个 **iovec** 结构都描述了一个用于传输的数据块——这个数据块的起始位置在 **iov_base**（在用户空间中），长度为 **iov_len** 个字节。函数中的 **count** 参数指明要操作多少个 **iovec** 结构。这些结构由应用程序创建，而内核在调用驱动程序之前会把它们拷贝到内核空间。

向量化操作最简单的实现，可能就是只传递每个 **iovec** 结构的地址和长度给驱动程序的 **read** 或 **write** 函数。不过，正确而有效率的操作经常需要驱动程序做一些更为巧妙的事情。例如，磁带驱动程序的 **writev** 就应在单个磁带录制过程里写入所有 **iovec** 结构的内容。

但是很多驱动程序并不期望通过自己实现这些方法来获益。所以，**scull** 忽略了它们。内核将会通过 **read/write** 来模拟它们，而最终结果一样。

3.7 试试新设备

一旦准备好了刚才讲述的四个方法，驱动程序就可以编译和测试了，它保留写入的数据，直至用新数据覆盖它们。这个设备有点象长度只受物理 **RAM** 容量限制的数据缓冲区。可以试试用 **cp**、**dd**，或者输入/输出重定向等命令来测试这个驱动程序。

依据写入 `scull` 的数据量，用 `free` 命令可以看到空闲内存的缩减和扩增。

为了进一步证实每次读写一个量子，可以在驱动程序的适当位置加入 `printk`，通过可了解到程序读/写大数据块时会发生什么事情。此外，还可以用工具 `strace` 来监视应用程序调用的系统调用以及它们的返回值。跟踪 `cp` 或 `ls -l > /dev/scull0` 会显示出量子化的读写过程。下一章将会详细介绍监视（或跟踪）技术。

3.8 设备文件系统

正如本章开始所提到的，Linux 内核的新近版本为设备入口点提供了一种特殊的文件系统。这个文件系统曾一度以一个非正式的补丁形式提供给大家；而在 2.3.46 版本中，它已成为正式源代码树的一部分。该文件系统对 2.2 版本也是支持的，虽然它并未被正式的 2.2 内核所包括。

尽管在本书编写的时候，这个特殊文件系统还没有被广泛使用，但它的新特点对设备驱动程序的编写人员会有一定帮助。因此当 `devfs` 在目标系统上得到应用的时候，我们这个版本的 `scull` 也将利用这一文件系统。模块在编译时通过内核配置信息来获知特定的特征是否被支持，在这个例子里，我们依赖宏 `CONFIG_DEVFS_FS` 的定义与否来决定是否对这个文件系统进行支持。

`devfs` 的主要优势如下：

- 设备初始化时在 `/dev` 目录下创建设备入口点，移除设备时将被删除。
- 设备驱动程序可以指定设备名、所有者和权限位，而用户空间程序仍可以修改所有者和权限位（文件名则不能修改）。
- 不再需要为设备驱动程序分配主设备号以及处理次设备号。

结果是，当模块装载和卸载时，不再需要运行一个脚本来创建设备文件，因为驱动程序会自主地管理它自己的设备文件。

驱动程序应调用下面这些函数来处理设备的创建和删除工作。

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

devfs_handle_t devfs_mk_dir (devfs_handle_t dir,
    const char *name, void *info);

devfs_handle_t devfs_register (devfs_handle_t dir,
    const char *name, unsigned int flags,
    unsigned int major, unsigned int minor,
    umode_t mode, void *ops, void *info);

void devfs_unregister (devfs_handle_t de);
```

`devfs` 还为内核编码提供了其它的一些函数，它们可以创建符号链接、访问内部数据结构并从索引节点中获取 `devfs_handle_t` 项等等任务。这里并不涉及这部分函数，因为它们不太重要并且也不太容易理解。感兴趣的读者可以参阅头文件以得到更多信息。

注册/注销函数中的各类参数如下：

dir

新创建的设备文件所在目录。大多数驱动程序使用 `NULL` 值，把设备文件创建在目录 `/dev` 下。如果需要创建一个自己的目录，驱动程序应该调用 `devfs_mk_dir`。

name

设备的名称，前面无须加上 `/dev/`。如果想让设备处于子目录下的话，名字中可以包含斜线符，子目录会在注册过程中被创建。此外，还可以指定一个指向目标子目录的 `dir` 指针。

flags

`devfs` 标志的位掩码。`DEVFS_FL_DEFAULT` 就是个好的选择，`DEVFS_FL_AUTO_DEVNUM` 是在需要自动进行主、次设备号分配时使用的标志。随后会说明这些标志。

major minor

设备的主/次设备号。如果在 `flags` 参数项指定了 `DEVFS_FL_AUTO_DEVNUM`，这个参数就不再有用。

mode

新设备的访问模式。

ops

指向设备的文件操作数据结构的指针。

info

用于设置 `filp->private_data` 的缺省值。当设备打开时，文件系统将把 `filp->private_data` 指针初始化为该值。传递给 `devfs_mk_dir` 的 `info` 指针并不是给 `devfs` 使用的，它作为“客户私有数据”指针来使用。

de

`devfs_register` 调用中获得的“`devfs` 入口”。

`flags` 参数用于为所创建的设备文件选择特定的功能特点。虽然在 `<linux/devfs_fs_kernel.h>` 中已经对 `flags` 进行了简要而清楚的文档描述，但在这里仍有必要介绍其中的一部分。

DEVFS_FL_NONE DEVFS_FL_DEFAULT

前者只是简单地为 0 值，它是为提高代码的可读性而提出的。后面这个宏目前即定义为 `DEVFS_FL_NONE`，不过它对与这种文件系统将来的实现保持向前兼容性是个好的选择。

DEVFS_FL_AUTO_OWNER

这个标记使得设备似乎由最后打开它的 `uid/gid` 所拥有，而且在没有进程打开它的时候，可以被任何人读取/写入。这个特点对 `tty` 设备文件非常有用，而且设备驱动程序也可以利用这个特点避免对某个非共享设备的并发访问。我们将在第 5 章中讨论访问策略问题。

DEVFS_FL_SHOW_UNREG DEVFS_FL_HIDE

`DEVFS_FL_SHOW_UNREG` 标志请求在注销时不要删除 `/dev` 目录下的设备文件。

DEVFS_FL_HIDE 标志则请求将设备文件隐藏在 `/dev` 目录下。通常的设备一般并不需要这两个标记。

DEVFS_FL_AUTO_DEVNUM

自动为设备分配设备号。即使 `devfs` 中对应入口文件可能已经被注销，这个设备号仍将保持与设备名的关联，所以，如果在系统关机之前驱动程序被再次载入，则将获取相同的主/次设备号。

DEVFS_FL_NO_PERSISTENCE

当设备文件删除后，不再保留相关的信息。使用这个标志可以在模块卸载后节省一些系统内存，代价是丢失了模块卸载/重新装载之间的设备特征的持久性记录。这些持久性特征包括访问模式、文件所有者和主/次设备号等。

在运行时可以查询设备关联的标志并进行修改。下面的这两个函数完成这个任务：

```
int devfs_get_flags (devfs_handle_t de, unsigned int *flags);
int devfs_set_flags (devfs_handle_t de, unsigned int flags);
```

3.8.1 实际使用 devfs

因为涉及到设备名称，`devfs` 会带来严重的用户空间的不兼容性，所以并不是所有的系统都会使用它。读者不太可能在近期编写一个仅支持 `devfs` 的驱动程序，这与新的特性如何被 Linux 使用者们所接受并没有关联。因此，需要增加对“老”方法的支持，主要在用户空间中的文件创建和权限处理，以及在内核空间使用主/次设备号等方面。

实现一个仅支持 `devfs` 的驱动所需要的代码是支持这两个环境所需代码的子集，所以我们只讲解双模式下的初始化工作。我们并不去编写一个专门的驱动程序样例去试验 `devfs`，取而代之的是，为现有的 `scull` 驱动程序增加对 `devfs` 的支持。如果向内核载入使用 `devfs` 的 `scull` 模块的话，需要直接调用 `insmod`，而不是运行 `scull_load` 脚本。

我们选择创建一个目录来存放所有的 `scull` 设备文件，因为 `devfs` 的结构是高度层次化的，没有理由不去遵循这个协定。而且，这样的话，还可以演示如何创建一个目录以及删除它。

在 `scull_init` 中，下面的代码用于设备的创建，其中 `Scull_Dev` 结构的 `handle` 成员用来保存被注册了的设备：

```
/* If we have devfs, create /dev/scull to put files in there */
scull_devfs_dir = devfs_mk_dir(NULL, "scull", NULL);
if (!scull_devfs_dir) return -EBUSY; /* problem */

for (i=0; i < scull_nr_devs; i++) {
    sprintf(devname, "%i", i);
    devfs_register(scull_devfs_dir, devname,
        DEVFS_FL_AUTO_DEVNUM,
        0, 0, S_IFCHR | S_IRUGO | S_IWUGO,
        &scull_fops,
        scull_devices+i);
}
```

上面的代码与下面这段从 `scull_cleanup` 摘录代码的其中两行是相呼应的。

```

if (scull_devices) {
    for (i=0; i<scull_nr_devs; i++) {
        scull_trim(scull_devices+i);
        /* the following line is only used for devfs */
        devfs_unregister(scull_devices[i].handle);
    }
    kfree(scull_devices);
}

/* once again, only for devfs */
devfs_unregister(scull_devfs_dir);

```

上面的这部分代码段是被编译条件 `#ifdef CONFIG_DEVFS_FS` 所保护的。如果在当前内核中 `devfs` 特性没有被支持，`scull` 将转而使用 `register_chrdev`。

为了支持这两种环境，仅有的额外工作，即在 `open` 这个设备方法中处理 `filp->f_ops` 和 `filp->private_data` 的初始化工作。前一个指针只是简单地不作修改，因为 `devfs_register` 已经指定了合适的文件操作。后一个指针也仅当值为 `NULL` 时通过 `open` 方法进行初始化，因为只有在不使用 `devfs` 的情况下，才会是 `NULL` 值。

```

/*
 * If private data is not valid, we are not using devfs
 * so use the type (from minor nr.) to select a new f_op
 */
if (!filp->private_data && type) {
    if (type > SCULL_MAX_TYPE) return -ENODEV;
    filp->f_op = scull_fop_array[type];
    return filp->f_op->open(inode, filp); /* dispatch to specific open */
}

/* type 0, check the device number (unless private_data valid) */
dev = (Scull_Dev *)filp->private_data;
if (!dev) {
    if (num >= scull_nr_devs) return -ENODEV;
    dev = &scull_devices[num];
    filp->private_data = dev; /* for other methods */
}

```

一旦采用了上面的这些代码，`scull` 就能被载入到运行 `devfs` 的系统中。执行命令 `ls -l /dev/scull`，我们可以看到下面的这些输出：

```

crw-rw-rw- 1 root root 144, 1 Jan 1 1970 0
crw-rw-rw- 1 root root 144, 2 Jan 1 1970 1
crw-rw-rw- 1 root root 144, 3 Jan 1 1970 2
crw-rw-rw- 1 root root 144, 4 Jan 1 1970 3
crw-rw-rw- 1 root root 144, 5 Jan 1 1970 pipe0
crw-rw-rw- 1 root root 144, 6 Jan 1 1970 pipe1
crw-rw-rw- 1 root root 144, 7 Jan 1 1970 pipe2
crw-rw-rw- 1 root root 144, 8 Jan 1 1970 pipe3
crw-rw-rw- 1 root root 144, 12 Jan 1 1970 priv
crw-rw-rw- 1 root root 144, 9 Jan 1 1970 single
crw-rw-rw- 1 root root 144, 10 Jan 1 1970 user
crw-rw-rw- 1 root root 144, 11 Jan 1 1970 wuser

```

上列各类文件的功能与“普通的” `scull` 模块是相同的，仅有的差别在设备路径上：原来是 `/dev/scull0`，而现在则是 `/dev/scull/0`。

3.8.2 可移植性问题和 devfs

出于能够在 2.0、2.2 和 2.4 这几个版本的 Linux 系统上编译并正常运行的需要，scull 的源码文件显得有些复杂。这种可移植性的需求是借助基于宏 CONFIG_DEVFS_FS 的条件编译来实现的。

幸运的是，大多数开发人员在这一点上是有共识的，即 `#ifdef` 结构出现在函数定义部分时基本上是个很糟糕的情况（如果是用于头文件的话则是可行的）。因此，增加对 `devfs` 的支持需要引入必要的机制以便在代码中完全避免 `#ifdef`。在 `scull` 中，我们仍使用条件编译，因为更老一点内核版本的头文件还不能对上述方式提供支持。

如果代码只是用于 2.4 内核的话，就可以通过调用内核函数，用这两种方法初始化驱动程序，而避免条件编译。因为事情都已安排好了，所以其中的一个初始化工作不必做任何事情，而只是成功返回就行。下面给了一个例子，说明初始化是怎样进行的：

```
#include <devfs_fs_kernel.h>

int init_module()
{
    /* request a major: does nothing if devfs is used */
    result = devfs_register_chrdev(major, "name", &fops);
    if (result < 0) return result;

    /* register using devfs: does nothing if not in use */
    devfs_register(NULL, "name", /* .... */ );
    return 0;
}
```

只要小心不致于对已经在内核头文件中作过定义的函数进行重定义的话，就可以在自己的头文件中采取一些类似的技巧。去除条件编译是件好事情，因为这样可以提高代码的可读性，并借助编译器分析整个输入文件，来减少可能的 `bug` 数目。只要是采用了条件编译，就有这样的风险，比如打字或其它的错误可能被忽视——如果 `C` 的预处理过程因 `#ifdef` 正好把这些错误所处的部分丢弃的话。

例如，下面就是 `scull.h` 如何在程序的 `cleanup` 部分避免条件编译的例子。这段代码对于所有的内核版本都是可移植的，因为它并不依赖于头文件中所知的 `devfs` 结构。

```
#ifdef CONFIG_DEVFS_FS /* only if enabled, to avoid errors in 2.0 */
#include <linux/devfs_fs_kernel.h>
#else
typedef void * devfs_handle_t; /* avoid #ifdef inside the structure */
#endif
```

`sysdep.h` 中没有定义任何东西，因为实现这样的一般性代码是非常困难的。每个驱动程序都应根据自己的需要合理进行安排，以避免在函数代码中出现过多的 `#ifdef` 语句。而且作为 `scull` 的一个例外，我们选择不对 `devfs` 进行支持。我们希望这里的讨论足以帮助读者去利用 `devfs`，如果他们想这样做的话；为了保持代码的简洁，对 `devfs` 的支持将从余下的示例程序中省略。

3.9 向后兼容性

到此为止，本章已经讲述了 2.4 版本 Linux 内核的编程接口。不幸的是，这些接口在内核发展过程

中已经发生了重大的变化。这些变化在实现方式上表现了不断的进步，但同样也对那些期望编写能在多个内核版本间具有兼容性的驱动程序开发人员带来了挑战。

在本章涉及的范围内，2.4 和 2.2 版本之间的差异为数不多。但 2.2 版本修改了 2.0 当中许多 `file_operations` 方法的函数原型，而且对用户空间的访问也作了重大修改（更为简化了）。信号量机制在 2.0 版本中也并不完善。最后要说明的是，2.1 开发系列引入了目录项（`dentry`）高速缓冲。

3.9.1 文件操作数据结构的变化

众多因素促使 `file_operations` 方法发生变化。长久以来的 2GB 文件尺寸的限制甚至在 Linux 的 2.0 版本中就带来了问题。其结果是，2.1 开发系列开始使用 `loff_t` 型的 64 位数据来表示文件位置和长度。大尺寸文件的支持直到 2.4 版本的内核才被完全整合，但很多基础工作很早以前就已经做了，并且也已被驱动开发人员所适应。

2.1 开发版本引入的另一个变化是对 `read/write` 方法增加了 `f_pos` 指针参数。这个变化用于支持 POSIX 标准的 `pread` 和 `pwrite` 系统调用，该参数显式地设置数据读/写的文件偏移量。没有这些系统调用的话，以线程方式工作的程序在对文件的频繁处理中会产生竞态问题。

几乎所有 2.0 版本的 Linux 方法都显式地接收一个索引节点指针参数。2.1 系列将这个参数从部分方法中删去，因为它很少被用到。如果需要索引节点指针的话，仍可以通过 `filp` 参数来获取它。

最终结果是，在 2.0 版本中，最常用的 `file_operations` 方法的原型就象下面列出的这样：

```
int (*lseek) (struct inode *, struct file *, off_t, int);
```

注意，Linux 2.0 中调用的是 `lseek`，并非 `llseek`。名字的变化是用来区分现在的 `seek` 可以进行 64 位偏移量的操作。

```
int (*read) (struct inode *, struct file *, char *, int); "int (*write) (struct inode
*, struct file *, const char *, int);"
```

正如已提到的，Linux 2.0 中这些函数具有索引节点指针参数，而没有位置参数。

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

在 2.0 版本的内核中，`release` 方法是不能失败的，因此返回的是 `void`。

`file_operations` 结构还有很多其它的变化；我们会在后面遇到它们的章节中谈到。同时，对于我们所看到的这些变化，很值得花一点时间看看怎样编写能解决这些变化的可移植性代码。这些方法中的变化非常大，还没有哪种简单精妙的办法可以将它们完全覆盖。

样例代码处理这些变化的办法是定义一组小的封装函数，将旧的 API “翻译”成新的。这些封装程序只能在 2.0 版本的头文件下编译时才可使用，并且必须在 `file_operations` 结构中被替换为“真正的”设备方法。下面这些是为 `scull` 驱动所设计的封装程序：

```
/*
 * The following wrappers are meant to make things work with 2.0 kernels
```

```

*/
#ifdef LINUX_20
int scull_llseek_20(struct inode *ino, struct file *f,
    off_t offset, int whence)
{
    return (int)scull_llseek(f, offset, whence);
}

int scull_read_20(struct inode *ino, struct file *f, char *buf,
    int count)
{
    return (int)scull_read(f, buf, count, &f->f_pos);
}

int scull_write_20(struct inode *ino, struct file *f, const char *b,
    int c)
{
    return (int)scull_write(f, b, c, &f->f_pos);
}

void scull_release_20(struct inode *ino, struct file *f)
{
    scull_release(ino, f);
}

/* Redefine "real" names to the 2.0 ones */
#define scull_llseek scull_llseek_20
#define scull_read scull_read_20
#define scull_write scull_write_20
#define scull_release scull_release_20
#define llseek lseek
#endif /* LINUX_20 */

```

用这种方式重新定义名字，也可以为结构成员解决因时间变迁而发生名称改变的问题（比如从 `lseek` 到 `llseek` 的变化）。

不必说，这种重定义名称的方法应该小心使用；这些代码行必须出现在 `file_operations` 结构定义之前，但在这些名称使用之后????。

还有另外两个不兼容因素与 `file_operations` 结构相关。一个是 `flush` 方法在 2.1 版本的开发中被加入。驱动开发人员几乎毫无必要去担心这个方法，但它在结构中的存在仍然可能带来问题。避免处理 `flush` 方法的最好办法是采用标记化的初始化语法，正如我们在所有的样例源码文件中所做的那样。

另外一个碍事的不同是，索引节点指针需要从 `file` 指针中获得。现代内核使用 `dentry`（目录项）数据结构，但版本 2.0 中却没有这个结构。因此，`sysdep.h` 定义了一个宏，利用这个宏，可从 `filp` 中访问索引节点——它隐藏了版本间的差异。

```

#ifdef LINUX_20
# define INODE_FROM_F(filp) ((filp)->f_inode)
#else
# define INODE_FROM_F(filp) ((filp)->f_dentry->d_inode)
#endif

```

3.9.2 模块使用计数

在 2.2 及更早期的内核中，Linux 内核对维护模块的使用计数并不提供任何帮助。模块只能自行完成这样的工作。这种方法容易出错并且需要进行许多重复工作，而且也会造成竞态的发生，新的方法因而明显改进了。

然而，为编写可移植代码，必须能处理早期版本的工作方式。也就是说，使用计数在新增对模块的一个引用时仍必须递增，反之递减。可移植的代码也必须处理早期版本内核的 `file_operations` 结构并不存在 `owner` 成员这一问题。解决这个问题最简单的办法是使用 `SET_MODULE_OWNER`，而不是直接使用 `owner` 成员。在 `sysdep.h` 中，我们为不具有这一功能的内核提供了一个无效的 `SET_FILE_OWNER` 宏。

3.9.3 信号量支持的变化

信号量的支持在 2.0 版本内核的开发中考虑得很少，对 SMP 系统的支持在那时也比较原始。仅为这个版本编写的驱动根本没有必要使用信号量，因为那时内核代码是运行在单 CPU 上的。尽管如此，还是有对信号量的需求，这对后期内核版本所需要的完整保护并无损害。

本章涉及的大多数信号量函数在 2.0 内核中就已经存在了。所例外的是 `sema_init`，在 2.0 版本中，程序员须手工初始信号量。`sysdep.h` 头文件通过定义另一个版本的 `sema_init` 来解决这个问题，它必须在 2.0 内核下编译。

```
#ifdef LINUX_20
# ifdef MUTEX_LOCKED /* Only if semaphore.h included */
extern inline void sema_init (struct semaphore *sem, int val)
{
    sem->count = val;
    sem->waking = sem->lock = 0;
    sem->wait = NULL;
}
# endif
#endif /* LINUX_20 */
```

3.9.4 用户空间访问的变化

最后，对用户空间的访问在 2.1 系列版本开发的一开始就完全改变了。新的接口有了更好的设计，并能更好地利用硬件以保证对用户空间内存的安全访问。但，接口当然已不同了。2.0 版本的内存访问函数如下：

```
void memcpy_fromfs(void *to, const void *from, unsigned long count);
void memcpy_tofs(void *to, const void *from, unsigned long count);
```

这些函数的名称来源于历史上对 i386 FS 段寄存器的使用。注意，这些函数没有返回值；如果用户提供的是无效地址，数据拷贝会没有任何提示地失败。`sysdep.h` 隐藏了重命名的处理，并允许可移植性地调用 `copy_to_user` 和 `copy_from_user`。

3.10 快速索引

本章介绍了下列符号和头文件。`file_operations` 结构和 `file` 结构的成员清单并没有在这里给出。

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

“文件系统”头文件，它是编写设备驱动程序比许的头文件。所有重要的函数都在这里声明。

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

注册字符设备驱动程序。如果主设备不为 0，则不加修改；如果主设备号为 0，系统动态给这个设备分配一个新设备号。

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

在卸载时注销驱动程序。`major` 和 `name` 字符串都必须存放与注册驱动程序时相同的值。

```
kdev_t inode->i_rdev;
```

当前设备的设备“号”，可从索引节点结构中获取。

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

这两个宏从设备项中分解出主/次设备号。

```
kdev_t MKDEV(int major, int minor);
```

这个宏由主/次设备号构造 `kdev_t` 数据项。

```
SET_MODULE_OWNER(struct file_operations *fops)
```

这个宏用来设置指定 `file_operations` 结构中的 `owner` 成员。

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

定义信号量相关的函数和数据类型。

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

将信号量初始化为一个给定值。互斥信号量通常初始化为 1。

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

分别用于获取信号量（必要时转入睡眠）和释放信号量。

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

`segment.h` 在 2.0 及以上内核中定义跨地址空间拷贝的相关函数。2.1 版本系列将其名称改为 `uaccess.h`。

```
unsigned long __copy_from_user (void *to, const void *from, unsigned long count);
unsigned long __copy_to_user (void *to, const void *from, unsigned long count);
```

在用户空间和内核空间之间拷贝数据。

```
void memcpy_fromfs(void *to, const void *from, unsigned long count);
void memcpy_tofs(void *to, const void *from, unsigned long count);
```


这些函数在 2.0 版本内核上，用来从用户空间到内核空间拷贝字节数组，以及相反方向的拷贝。

```
#include <linux/devfs_fs_kernel.h>
devfs_handle_t devfs_mk_dir (devfs_handle_t dir, const char *name, void *info);
devfs_handle_t devfs_register (devfs_handle_t dir, const char *name, unsigned int
    flags, unsigned int major, unsigned int minor, umode_t mode, void *ops, void *info);
void devfs_unregister (devfs_handle_t de);
```

这些是用于在设备文件系统（**devfs**）中注册设备的基本函数。

第 4 章 调试技术



对于任何一位内核代码的编写者来说，最急迫的问题之一就是如何完成调试。由于内核是一个不与特定进程相关的功能集合，所以内核代码无法轻易地放在调试器中执行，而且也很难跟踪。同样，要想复现内核代码中的错误也是相当困难的，因为这种错误可能导致整个系统崩溃，这样也就破坏了可以用来跟踪它们的现场。

本章将介绍在这种令人痛苦的环境下监视内核代码并跟踪错误的技术。

4.1 通过打印调试

最普通的调试技术就是监视，即在应用程序编程中，在一些适当的地点调用 `printf` 显示监视信息。调试内核代码的时候，则可以用 `printk` 来完成相同的工作。

4.1.1 `printk`

在前面的章节中，我们只是简单假设 `printk` 工作起来和 `printf` 很类似。现在则是介绍它们之间一些不同点的时候了。

其中一个差别就是，通过附加不同日志级别（`loglevel`），或者说消息优先级，可让 `printk` 根据这些级别所标示的严重程度，对消息进行分类。一般采用宏来指示日志级别，例如，`KERN_INFO`，我们在前面已经看到它被添加在一些打印语句的前面，它就是一个可以使用的消息日志级别。日志级别宏展开为一个字符串，在编译时由预处理器将它和消息文本拼接在一起；这也就是为什么下面的例子中优先级和格式字符串间没有逗号的原因。下面有两个 `printk` 的例子，一个是调试信息，一个是临界信息：

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

在头文件 `<linux/kernel.h>` 中定义了 8 种可用的日志级别字符串。

```
KERN_EMERG
```

用于紧急事件消息，它们一般是系统崩溃之前提示的消息。

```
KERN_ALERT
```

用于需要立即采取行动的情况。

KERN_CRIT

临界状态，通常涉及严重的硬件或软件操作失败。

KERN_ERR

用于报告错误状态；设备驱动程序会经常使用 `KERN_ERR` 来报告来自硬件的问题。

KERN_WARNING

对可能出现问题的情况进行警告，这类情况通常不会对系统造成严重问题。

KERN_NOTICE

有必要进行提示的正常情形。许多与安全相关的状况用这个级别进行汇报。

KERN_INFO

提示性信息。很多驱动程序在启动的时候，以这个级别打印出它们找到的硬件信息。

KERN_DEBUG

用于调试信息。

每个字符串（以宏的形式展开）代表一个尖括号中的整数。整数值的范围从 0 到 7，数值越小，优先级就越高。

没有指定优先级的 `printk` 语句默认采用的级别是 `DEFAULT_MESSAGE_LOGLEVEL`，这个宏在文件 `kernel/printk.c` 中指定为一个整数值。在 Linux 的开发过程中，这个默认的级别值已经有过好几次变化，所以我们建议读者始终指定一个明确的级别。

根据日志级别，内核可能会把消息打印到当前控制台上，这个控制台可以是一个字符模式的终端、一个串口打印机或是一个并口打印机。如果优先级小于 `console_loglevel` 这个整数值的话，消息才能显示出来。如果系统同时运行了 `klogd` 和 `syslogd`，则无论 `console_loglevel` 为何值，内核消息都将追加到 `/var/log/messages` 中（否则的话，除此之外的处理方式就依赖于对 `syslogd` 的设置）。如果 `klogd` 没有运行，这些消息就不会传递到用户空间，这种情况下，就只好查看 `/proc/kmsg` 了。

变量 `console_loglevel` 的初始值是 `DEFAULT_CONSOLE_LOGLEVEL`，而且还可以通过 `sys_syslog` 系统调用进行修改。调用 `klogd` 时可以指定 `-c` 开关选项来修改这个变量，`klogd` 的 `man` 手册页对此有详细说明。注意，要修改它的当前值，必须先杀掉 `klogd`，再加 `-c` 选项重新启动它。此外，还可以编写程序来改变控制台日志级别。读者可以在 O'Reilly 的 FTP 站点提供的源文件 `miscprogs/setlevel.c` 里找到这样的一段程序。新优先级被指定为一个 1 到 8 之间的整数值。如果值被设为 1，则只有级别为 0（`KERN_EMERG`）的消息才能到达控制台；如果设为 8，则包括调试信息在内的所有消息都能显示出来。

如果在控制台上工作，而且常常遇到内核错误（参见本章后面的“调试系统故障”一节）的话，就有必要降低日志级别，因为出错处理代码会把 `console_loglevel` 增为它的最大数值，导致随后的所有消息都显示在控制台上。如果需要查看调试信息，就有必要提高日志级别；这在远程调试内核，并且在交互会话未使用文本控制台的情况下，是很有帮助的。

从 2.1.31 这个版本起，可以通过文本文件 `/proc/sys/kernel/printk` 来读取和修改控制台的日志级别。这个文件容纳了 4 个整数值。读者可能会对前面两个感兴趣：控制台的当前日志级别和默认日志级别。例如，在最近的这些内核版本中，可以通过简单地输入下面的命令使所有的内核消息得到显示：

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

不过，如果仍在 2.0 版本下的话，就需要使用 `setlevel` 这样的工具了。

现在大家应该清楚为什么在 `hello.c` 范例中使用 `<1>` 这些标记了，它们用来确保这些消息能在控制台上显示出来。

对于控制台日志策略，Linux 考虑到了某些灵活性，也就是说，可以发送消息到一个指定的虚拟控制台（假如控制台是文本屏幕的话）。默认情况下，“控制台”就是当前地虚拟终端。可以在任何一个控制台设备上调用 `ioctl` (`TIOCLINUX`)，来指定接收消息的虚拟终端。下面的 `setconsole` 程序，可选择专门用来接收内核消息的控制台；这个程序必须由超级用户运行，在 `misc-progs` 目录里可以找到它。下面是程序的代码：

```
int main(int argc, char **argv)
{
    char bytes[2] = {11,0}; /* 11 is the TIOCLINUX cmd number */

    if (argc==2) bytes[1] = atoi(argv[1]); /* the chosen console */
    else {
        fprintf(stderr, "%s: need a single arg\n",argv[0]); exit(1);
    }
    if (ioctl(STDIN_FILENO, TIOCLINUX, bytes)<0) { /* use stdin */
        fprintf(stderr,"%s: ioctl(stdin, TIOCLINUX): %s\n",
            argv[0], strerror(errno));
        exit(1);
    }
    exit(0);
}
```

`setconsole` 使用了特殊的 `ioctl` 命令：`TIOCLINUX`，这个命令可以完成一些特定的 Linux 功能。使用 `TIOCLINUX` 时，需要传给它一个指向字节数组的指针参数。数组的第一个字节指定所请求子命令的数字，接下去的字节所具有的功能则由这个子命令决定。在 `setconsole` 中，使用的子命令是 11，后面那个字节（存于 `bytes[1]` 中）标识虚拟控制台。关于 `TIOCLINUX` 的详尽描述可以在内核源码中的 `drivers/char/tty_io.c` 文件得到。

4.1.2 消息如何被记录

`printk` 函数将消息写到一个长度为 `LOG_BUF_LEN`（定义在 `kernel/printk.c` 中）字节的循环缓冲区中，然后唤醒任何正在等待消息的进程，即那些睡眠在 `syslog` 系统调用上的进程，或者读取 `/proc/kmesg` 的进程。这两个访问日志引擎的接口几乎是等价的，不过请注意，对 `/proc/kmesg` 进行读操作时，日志缓冲区中被读取的数据就不再保留，而 `syslog` 系统调用却能随意地返回日志数据，并保留这些数据以便其它进程也能使用。一般而言，读 `/proc` 文件要容易些，这使它成为 `klogd` 的默认方法。

手工读取内核消息时，在停止 `klogd` 之后，可以发现 `/proc` 文件很象一个 FIFO，读进程会阻塞在里面以等待更多的数据。显然，如果已经有 `klogd` 或其它的进程正在读取相同的数据，就不能采用这种方法进行消息读取，因为会与这些进程发生竞争。

如果循环缓冲区填满了，`printk` 就绕回缓冲区的开始处填写新数据，覆盖最陈旧的数据，于是记录进程就会丢失最早的数据。但与使用循环缓冲区所带来的好处相比，这个问题可以忽略不计。例如，循环缓冲区可以使系统在没有记录进程的情况下照样运行，同时覆盖那些不再会有人去读的旧数据，从而使内存的浪费减到最少。Linux 消息处理方法的另一个特点是，可以在任何地方调用 `printk`，甚至在中断处理函数里也可以调用，而且对数据量的大小没有限制。而这个方法的唯一缺点就是可能丢失某些数据。

`klogd` 运行时，会读取内核消息并将它们分发到 `syslogd`，`syslogd` 随后查看 `/etc/syslog.conf`，找出处理这些数据的方法。`syslogd` 根据设施和优先级对消息进行区分；这两者的允许值均定义在 `<sys/syslog.h>` 中。内核消息由 `LOG_KERN` 设施记录，并以 `printk` 中使用的优先级记录（例如，`printk` 中使用的 `KERN_ERR` 对应于 `syslogd` 中的 `LOG_ERR`）。如果没有运行 `klogd`，数据将保留在循环缓冲区中，直到某个进程读取或缓冲区溢出为止。

如果想避免因为来自驱动程序的大量监视信息而扰乱系统日志，则可以为 `klogd` 指定 `-f (file)` 选项，指示 `klogd` 将消息保存到某个特定的文件，或者修改 `/etc/syslog.conf` 来适应自己的需求。另一种可能的办法是采取强硬措施：杀掉 `klogd`，而将消息详细地打印到空闲的虚拟终端上。^{*}

或者在一个未使用的 `xterm` 上执行 `cat /proc/kmesg` 来显示消息。

4.1.3 开启及关闭消息

在驱动程序开发的初期阶段，`printk` 对于调试和测试新代码是相当有帮助的。不过，当正式发布驱动程序时，就得删除这些打印语句，或至少让它们失效。不幸的是，你可能会发现这样的情况，在删除了那些已被认为不再需要的提示消息后，又需要实现一个新的功能（或是有人发现了一个 bug），这时，又希望至少把一部分消息重新开启。这两个问题可以通过几个办法解决，以便全局地开启或禁止消息，并能对个别消息进行开关控制。

我们在这里给出了一个编写 `printk` 调用的方法，可个别或全局地对它们进行开关；这个技巧是定义一个宏，在需要时，这个宏展开为一个 `printk`（或 `printf`）调用。

可以通过在宏名字中删减或增加一个字母，打开或关闭每一条打印语句。

编译前修改 `CFLAGS` 变量，则可以一次关闭所有消息。

同样的打印语句既可以用在内核态也可以用在用户态，因此，关于这些额外的信息，驱动和测试程序可以用同样的方法来进行管理。

^{*} 例如，使用下面的命令可设置 10 号终端用于消息的显示：

```
setlevel 8
setconsole 10
```

下面这些来自 `scull.h` 的代码，就实现了这些功能。

```
#undef PDEBUG          /* undef it, just in case */
#ifdef SCULL_DEBUG
# ifdef __KERNEL__
    /* This one if debugging is on, and kernel space */
#   define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt,
                                      ## args)
# else
    /* This one for user space */
#   define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
# endif
#else
# define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */
```

符号 `PDEBUG` 依赖于是否定义了 `SCULL_DEBUG`，它能根据代码所运行的环境选择合适的方式显示信息：内核态运行时使用 `printk` 系统调用；用户态下则使用 `libc` 调用 `fprintf`，向标准错误设备进行输出。符号 `PDEBUGG` 则什么也不做；它可以用来将打印语句注释掉，而不必把它们完全删除。

为了进一步简化这个过程，可以在 `Makefile` 加上下面几行：

```
# Comment/uncomment the following line to disable/enable debugging
DEBUG = y

# Add your debugging flag (or not) to CFLAGS
ifeq ($(DEBUG),y)
    DEBFLAGS = -O -g -DSCULL_DEBUG # "-O" is needed to expand inlines
else
    DEBFLAGS = -O2
endif

CFLAGS += $(DEBFLAGS)
```

本节所给出的宏依赖于 `gcc` 对 `ANSI C` 预编译器的扩展，这种扩展支持了带可变数目参数的宏。对 `gcc` 的这种依赖并不是什么问题，因为内核对 `gcc` 特性的依赖更强。此外，`Makefile` 依赖于 `GNU` 的 `make` 版本；基于同样的道理，这也不是什么问题。

如果读者熟悉 `C` 预编译器，可以将上面的定义进行扩展，实现“调试级别”的概念，这需要定义一组不同的级别，并为每个级别赋一个整数（或位掩码），用以决定各个级别消息的详细程度。

但是每一个驱动程序都会有自身的功能和监视需求。良好的编程技术在于选择灵活性和效率的最佳折衷点，对读者来说，我们无法预知最合适的点在哪里。记住，预处理程序的条件语句（以及代码中的常量表达式）只在编译时执行，要再次打开或关闭消息必须重新编译。另一种方法就是使用 `C` 条件语句，它在运行时执行，因此可以在程序运行期间打开或关闭消息。这是个很好的功能，但每次代码执行时系统都要进行额外的处理，甚至在消息关闭后仍然会影响性能。有时这种性能损失是无法接受的。

在很多情况下，本节提到的这些宏都已被证实是很有用的，仅有的缺点是每次开启和关闭消息显示时都要重新编译模块。

4.2 通过查询调试

上一节讲述了 `printk` 是如何工作的以及如何使用它，但还没谈到它的缺点。

由于 `syslogd` 会一直保持对其输出文件的同步刷新，每打印一行都会引起一次磁盘操作，因此大量使用 `printk` 会严重降低系统性能。从 `syslogd` 的角度来看，这样的处理是正确的。它试图把每件事情都记录到磁盘上，以防系统万一崩溃时，最后的记录信息能反应崩溃前的状况；然而，因处理调试信息而使系统性能减慢，是大家所不希望的。这个问题可以通过在 `/etc/syslogd.conf` 中日志文件的名字前面，前缀一个减号符解决。^{*}

修改配置文件带来的问题在于，在完成调试之后改动将依旧保留；即使在一般的系统操作中，当希望尽快把信息刷新到磁盘时，也是如此。如果不愿作这种持久性修改的话，另一个选择是运行一个非 `klogd` 程序（如前面介绍的 `cat /proc/kmesg`），但这样并不能为通常的系统操作提供一个合适的环境。

多数情况中，获取相关信息的最好方法是在需要的时候才去查询系统信息，而不是持续不断地产生数据。实际上，每个 Unix 系统都提供了很多工具，用于获取系统信息，如：`ps`、`netstat`、`vmstat` 等等。

驱动程序开发人员对系统进行查询时，可以采用两种主要的技术：在 `/proc` 文件系统中创建文件，或者使用驱动程序的 `ioctl` 方法。`/proc` 方式的另一个选择是使用 `devfs`，不过用于信息查找时，`/proc` 更为简单一些。

4.2.1 使用 `/proc` 文件系统

`/proc` 文件系统是一种特殊的、由程序创建的文件系统，内核使用它向外界输出信息。`/proc` 下面的每个文件都绑定于一个内核函数，这个函数在文件被读取时，动态地生成文件的“内容”。我们已经见到过这类文件的一些输出情况，例如，`/proc/modules` 列出的是当前载入模块的列表。

Linux 系统对 `/proc` 的使用很频繁。现代 Linux 系统中的很多工具都是通过 `/proc` 来获取它们的信息，例如 `ps`、`top` 和 `uptime`。有些设备驱动程序也通过 `/proc` 输出信息，你的驱动程序当然也可以这么做。因为 `/proc` 文件系统是动态的，所以驱动程序模块可以在任何时候添加或删除其中的文件项。

特征完全的 `/proc` 文件项相当复杂；在所有的这些特征当中，有一点要指出的是，这些 `/proc` 文件不仅可以用于读出数据，也可以用于写入数据。不过，大多数时候，`/proc` 文件项是只读文件。本节将只涉及简单的只读情形。如果有兴趣实现更为复杂的事情，读者可以先在这里了解基础知识，然后参考内核源码来建立完整的认识。

所有使用 `/proc` 的模块必须包含 `<linux/proc_fs.h>`，通过这个头文件定义正确的函数。

^{*} 这个减号是个“特殊”标记，避免 `syslogd` 在每次出现新信息时都去刷新磁盘文件，这些内容记述在 `syslog.conf(5)` 中，这个手册页很值得一读。

为创建一个只读 `/proc` 文件，驱动程序必须实现一个函数，用于在文件读取时生成数据。当某个进程读这个文件时（使用 `read` 系统调用），请求会通过两个不同接口的其中之一发送到驱动程序模块，使用哪个接口取决于注册情况。我们先把注册放到本节后面，先直接讲述读接口。

无论采用哪个接口，在这两种情况下，内核都会分配一页内存（也就是 `PAGE_SIZE` 个字节），驱动程序向这片内存写入将返回给用户空间的数据。

推荐的接口是 `read_proc`，不过还有一个名为 `get_info` 的老一点的接口。

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

参数表中的 `page` 指针指向将写入数据的缓冲区；`start` 被函数用来说明有意义的数据写在页面的什么位置（对此后面还将进一步谈到）；`offset` 和 `count` 这两个参数与在 `read` 实现中的用法相同。`eof` 参数指向一个整型数，当没有数据可返回时，驱动程序必须设置这个参数；`data` 参数是一个驱动程序特有的数据指针，可用于内部记录。^{*}

这个函数可以在 2.4 内核中使用，如果使用我们的 `sysdep.h` 头文件，那么在 2.2 内核中也可以用这个函数。

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

`get_info` 是一个用来读取 `/proc` 文件的较老接口。所有的参数与 `read_proc` 中的对应参数用法相同。缺少的是报告到达文件尾的指针和由 `data` 指针带来的面向对象风格。这个函数可以用在所有我们感兴趣的内核版本中（尽管在它 2.0 版本的实现中有一个额外未用的参数）。

这两个函数的返回值都是实际放入页面缓冲区的数据的字节数，这一点与 `read` 函数对其它类型文件的处理相同。另外还有 `*eof` 和 `*start` 这两个输出值。`eof` 只是一个简单的标记，而 `start` 的用法就有点复杂了。

对于 `/proc` 文件系统的用户扩展，其最初实现中的主要问题在于，数据传输只使用单个内存页面。这样就把用户文件的总体尺寸限制在了 4KB 以内（或者是适合于主机平台的其它值）。`start` 参数在这里就是用来实现大数据文件的，不过该参数可以被忽略。

如果 `proc_read` 函数不对 `*start` 指针进行设置（它最初为 `NULL`），内核就会假定 `offset` 参数被忽略，并且数据页包含了返回给用户空间的整个文件。反之，如果需要通过多个片段创建一个更大的文件，则可以把 `*start` 赋值为页面指针，因此调用者也就知道了新数据放在缓冲区的开始位置。当然，应该跳过前 `offset` 个字节的数据，因为这些数据已经在前面的调用中返回。

长久以来，关于 `/proc` 文件还有另一个主要问题，这也是 `start` 意图解决的一个问题。有时，在连续的 `read` 调用之间，内核数据结构的 ASCII 表述会发生变化，以至于读进程发现前后两次调用所获得的数据不一致。如果把 `*start` 设为一个小的整数值，调用程序可以利用它来增加 `filp->f_pos` 的值，而不依赖于返回的数据量，因此也就使 `f_pos` 成为 `read_proc` 或 `get_info` 程序中的一个内部记录值。例如，如果 `read_proc` 函数从一个大的结构数组返回数据，并且这些结构的前 5 个

^{*} 纵览全书，我们还会发现这样的一些指针；它们表示了这类处理中有关的“对象”，与 C++ 中的同类处理有些相似。

已经在第一次调用中返回，那么可将 `*start` 设置为 5。下次调用中这个值将被作为偏移量；驱动程序也就知道应该从数组的第六个结构开始返回数据。这种方法被它的作者称作“hack”，可以在 `/fs/proc/generic.c` 中看到。

现在来看个例子。下面是 `scull` 设备 `read_proc` 函数的简单实现：

```
int scull_read_procmem(char *buf, char **start, off_t offset,
                      int count, int *eof, void *data)
{
    int i, j, len = 0;
    int limit = count - 80; /* Don't print more than this */

    for (i = 0; i < scull_nr_devs && len <= limit; i++) {
        Scull_Dev *d = &scull_devices[i];
        if (down_interruptible(&d->sem))
            return -ERESTARTSYS;
        len += sprintf(buf+len, "\nDevice %i: qset %i, q %i, sz %li\n",
                       i, d->qset, d->qquantum, d->size);
        for (; d && len <= limit; d = d->next) { /* scan the list */
            len += sprintf(buf+len, "  item at %p, qset at %p\n", d,
                           d->data);
            if (d->data && !d->next) /* dump only the last item
                                     - save space */
                for (j = 0; j < d->qset; j++) {
                    if (d->data[j])
                        len += sprintf(buf+len, "    % 4i: %8p\n",
                                       j, d->data[j]);
                }
            up(&scull_devices[i].sem);
        }
        *eof = 1;
        return len;
    }
}
```

这是一个相当典型的 `read_proc` 实现。它假定决不会有这样的需求，即生成多于一页的数据，因此忽略了 `start` 和 `offset` 值。但是，小心不要超出缓冲区，以防万一。

使用 `get_info` 接口的 `/proc` 函数与上面说明的 `read_proc` 非常相似，除了没有最后的那两个参数。既然这样，则通过返回少于调用者预期的数据（也就是少于 `count` 参数），来提示已到达文件尾。

一旦定义好了一个 `read_proc` 函数，就需要把它与一个 `/proc` 文件项连接起来。依赖于将要支持的内核版本，有两种方法可以建立这样的连接。最容易的方法是简单地调用 `create_proc_read_entry`，但这只能用于 2.4 内核（如果使用我们的 `sysdep.h` 头文件，则也可用于 2.2 内核）。下面就是 `scull` 使用的调用，以 `/proc/scullmem` 的形式来提供 `/proc` 功能。

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

这个函数的参数表包括：`/proc` 文件项的名称、应用于该文件项的文件许可权限（0 是个特殊值，会被转换为一个默认的、完全可读模式的掩码）、文件父目录的 `proc_dir_entry` 指针（我们使用 `NULL` 值使该文件项直接定位在 `/proc` 下）、指向 `read_proc` 的函数指针，以及将传递给

`read_proc` 函数的数据指针。

目录项指针 (`proc_dir_entry`) 可用在 `/proc` 下创建完整的目录层次结构。不过请注意, 将文件项置于 `/proc` 的子目录中有更为简单的方法, 即把目录名称作为文件项名称的一部分——只要目录本身已经存在。例如, 有个新的约定, 要求设备驱动程序对应的 `/proc` 文件项应转移到子目录 `driver/` 中; `scull` 可以简单地指定它的文件项名称为 `driver/scullmem`, 从而把它的 `/proc` 文件放到这个子目录中。

当然, 在模块卸载时, `/proc` 中的文件项也应被删除。 `remove_proc_entry` 就是用来撤销 `create_proc_read_entry` 所做工作的函数。

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

另一个创建 `/proc` 文件项的方法是, 创建并初始化一个 `proc_dir_entry` 结构, 并将该结构传递给函数 `proc_register_dynamic` (2.0 版本)或 `proc_register` (2.2 版本, 如果结构中的索引节点号为 0, 该函数即认为是动态文件)。作为一个例子, 当在 2.0 内核的头文件下进行编译时, 考虑下面 `scull` 所使用的这些代码:

```
static int scull_get_info(char *buf, char **start, off_t offset,
                          int len, int unused)
{
    int eof = 0;
    return scull_read_procmem (buf, start, offset, len, &eof, NULL);
}

struct proc_dir_entry scull_proc_entry = {
    namelen:      8,
    name:         "scullmem",
    mode:         S_IFREG | S_IRUGO,
    nlink:        1,
    get_info:     scull_get_info,
};

static void scull_create_proc()
{
    proc_register_dynamic(&proc_root, &scull_proc_entry);
}

static void scull_remove_proc()
{
    proc_unregister(&proc_root, scull_proc_entry.low_ino);
}
```

代码声明了一个使用 `get_info` 接口的函数, 并填写了一个 `proc_dir_entry` 结构, 用于对文件系统进行注册。

这段代码借助 `sysdep.h` 中宏定义的支持, 提供了 2.0 和 2.4 内核之间的兼容性。因为 2.0 内核不支持 `read_proc`, 它使用了 `get_info` 接口。如果对 `#ifdef` 作一些更多的处理, 可以使这段代码在 2.2 内核中使用 `read_proc`, 不过这样收益并不大。

4.2.2 ioctl 方法

`ioctl` 是作用于文件描述符之上的一个系统调用，我们会在下一章介绍它的用法；它接收一个“命令”号，用以标识将执行的命令；以及另一个（可选的）参数，通常是个指针。

做为替代 `/proc` 文件系统的方法，可以为调试设计若干 `ioctl` 命令。这些命令从驱动程序复制相关数据到用户空间，在用户空间中可以查看这些数据。

使用 `ioctl` 获取信息比起 `/proc` 来要困难一些，因为需要另一个程序调用 `ioctl` 并显示结果。这个程序是必须编写并编译的，而且要和测试模块配合一致。从另一方面来说，相对实现 `/proc` 文件所需的工作，驱动程序的编码则更为容易些。

有时 `ioctl` 是获取信息的最好方法，因为它比起读 `/proc` 要快得多。如果在数据写到屏幕之前要完成某些处理工作，以二进制获取数据要比读取文本文件有效得多。此外，`ioctl` 并不要求把数据分割成不超过一个内存页面的片断。

`ioctl` 方法的一个优点是，在结束调试之后，用来取得信息的这些命令仍可以保留在驱动程序中。`/proc` 文件对任何查看这个目录的人都是可见的(很多人可能会纳闷“这些奇怪的文件是用来做什么的”)，然而与 `/proc` 文件不同，未公开的 `ioctl` 命令通常都不会被注意到。此外，万一驱动程序有什么异常，这些命令仍然可以用来调试。唯一的缺点就是模块会稍微大一些。

4.3 通过监视调试

有时，通过监视用户空间中应用程序的运行情况，可以捕捉到一些小问题。监视程序同样也有助于确认驱动程序工作是否正常。例如，看到 `scull` 的 `read` 实现如何响应不同数据量的 `read` 请求后，我们就可以判断它是否工作正常。

有许多方法可监视用户空间程序的工作情况。可以用调试器一步步跟踪它的函数，插入打印语句，或者在 `strace` 状态下运行程序。在检查内核代码时，最后一项技术最值得关注，我们将在此对它进行讨论。

`strace` 命令是一个功能非常强大的工具，它可以显示程序所调用的所有系统调用。它不仅可以显示调用，而且还能显示调用参数，以及用符号方式表示的返回值。当系统调用失败时，错误的符号值（如 `ENOMEM`）和对应的字符串（如 `Out of memory`）都能被显示出来。`strace` 有许多命令行选项；最为有用的是 `-t`，用来显示调用发生的时间；`-T`，显示调用所花费的时间；`-e`，限定被跟踪的调用类型；`-o`，将输出重定向到一个文件中。默认情况下，`strace` 将跟踪信息打印到 `stderr` 上。

`strace` 从内核中接收信息。这意味着一个程序无论是否按调试方式编译（用 `gcc` 的 `-g` 选项）或是被去掉了符号信息都可以被跟踪。与调试器可以连接到一个运行进程并控制它一样，`strace` 也可以跟踪一个正在运行的进程。

跟踪信息通常用于生成错误报告，然后发给应用开发人员，但是它对内核编程人员来说也同样非常有用。我们已经看到驱动程序是如何通过响应系统调用得到执行的；`strace` 允许我们检查每次调用中输入和输出数据的一致性。

例如，下面的屏幕信息显示了 `strace ls /dev > /dev/scull0` 命令的最后几行：

```
[...]
open("/dev", O_RDONLY|O_NONBLOCK)      = 4
fcntl(4, F_SETFD, FD_CLOEXEC)         = 0
brk(0x8055000)                         = 0x8055000
lseek(4, 0, SEEK_CUR)                  = 0
getdents(4, /* 70 entries */, 3933)   = 1260
[...]
getdents(4, /* 0 entries */, 3933)     = 0
close(4)                               = 0
fstat(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(253, 0), ...}) = 0
ioctl(1, TCGETS, 0xbfffa5c)            = -1 ENOTTY (Inappropriate ioctl
                                         for device)
write(1, "MAKEDEV\natibm\naudio\naudiol\na"... , 4096) = 4000
write(1, "d2\nsdd3\nsdd4\nsdd5\nsdd6\nsdd7"... , 96) = 96
write(1, "4\nsde5\nsde6\nsde7\nsde8\nsde9\n"... , 3325) = 3325
close(1)                                = 0
_exit(0)                                = ?
```

很明显，`ls` 完成对目标目录的检索后，在首次对 `write` 的调用中，它试图写入 4KB 数据。很奇怪（对于 `ls` 来说），实际只写了 4000 个字节，接着它重试这一操作。然而，我们知道 `scull` 的 `write` 实现每次最多只写一个量子（`scull` 中设置的量子大小为 4000 个字节），所以我们所预期的就是这样的部分写入。经过几个步骤之后，每件工作都顺利通过，程序正常退出。

另一个例子，让我们来对 `scull` 设备进行读操作（使用 `wc` 命令）：

```
[...]
open("/dev/scull0", O_RDONLY)           = 4
fstat(4, {st_mode=S_IFCHR|0664, st_rdev=makedev(253, 0), ...}) = 0
read(4, "MAKEDEV\natibm\naudio\naudiol\na"... , 16384) = 4000
read(4, "d2\nsdd3\nsdd4\nsdd5\nsdd6\nsdd7"... , 16384) = 3421
read(4, "", 16384)                      = 0
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(3, 7), ...}) = 0
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
write(1, " 7421 /dev/scull0\n", 20)     = 20
close(4)                                = 0
_exit(0)                                = ?
```

正如所料，`read` 每次只能读取 4000 个字节，但数据总量与前面例子中写入的数量是相同的。与上面的写跟踪相对比，请读者注意本例中重试工作是如何组织的。为了快速读取数据，`wc` 已被优化了，因而它绕过了标准库，试图通过一次系统调用读取更多的数据。可以从跟踪的 `read` 行中看到 `wc` 每次均试图读取 16KB 数据。

Linux 行家可以在 `strace` 的输出中发现很多有用信息。如果觉得这些符号过于拖累的话，则可以仅限于监视文件方法（`open`，`read` 等）是如何工作的。

就个人观点而言，我们发现 `strace` 对于查找系统调用运行时的细微错误最为有用。通常应用或演示程序中的 `perorr` 调用在用于调试时信息还不够详细，而 `strace` 能够确切查明系统调用的哪个参数引发了错误，这一点对调试是大有帮助的。

4.4 调试系统故障

即使采用了所有这些监视和调试技术，有时驱动程序中依然会有错误，这样的驱动程序在执行时就会产生系统故障。在出现这种情况时，获取尽可能多的信息对解决问题是至关重要的。

注意，“故障”并不意味着“panic”。Linux 代码非常健壮（用术语讲即为鲁棒，robust），可以很好地响应大部分错误：故障通常会导致当前进程崩溃，而系统仍会继续运行。如果在进程上下文之外发生故障，或是系统的重要组成部分被损害时，系统才有可能 panic。但如果问题出在驱动程序中时，通常只会导致正在使用驱动程序的那个进程突然终止。唯一不可恢复的损失就是进程被终止时，为进程上下文分配的一些内存可能会丢失；例如，由驱动程序通过 kmalloc 分配的动态链表可能丢失。然而，由于内核在进程中止时会对已打开的设备调用 close 操作，驱动程序仍可以释放由 open 方法分配的资源。

我们已经说过，当内核行为异常时，会在控制台上打印出提示信息。下一节将解释如何解码并使用这些消息。尽管它们对于初学者来说相当晦涩，不过处理器在出错时转储出的这些数据包含了许多值得关注的信息，通常足以查明程序错误，而无需额外的测试。

4.4.1 oops 消息

大部分错误都在于 NULL 指针的使用或其他不正确的指针值的使用上。这些错误通常会导致一个 oops 消息。

由处理器使用的地址都是虚拟地址，而且通过一个复杂的称为页表（见第 13 章中的“页表”一节）的结构映射为物理地址。当引用一个非法指针时，页面映射机制就不能将地址映射到物理地址，此时处理器就会向操作系统发出一个“页面失效”的信号。如果地址非法，内核就无法“换页”到并不存在的地址上；如果此时处理器处于超级用户模式，系统就会产生一个“oops”。

值得注意的是，2.0 版本之后引入的第一个增强是，当向用户空间移动数据或者移出时，无效地址错误会被自动处理。Linus 选择了让硬件来捕捉错误的内存引用，所以正常情况（地址都正确时）就可以更有效地得到处理。

oops 显示发生错误时处理器的状态，包括 CPU 寄存器的内容、页描述符表的位置，以及其它看上去无法理解的信息。这些消息由失效处理函数（arch/*/kernel/traps.c）中的 printk 语句产生，就象前面“printk”一节所介绍的那样分发出来。

让我们看看这样一个消息。当我们在运行 2.4 内核的 PC 机上使用一个 NULL 指针时，就会导致下面这些信息显示出来。这里最为相关的信息就是指令指针（EIP），即出错指令的地址。

```
Unable to handle kernel NULL pointer dereference at virtual address \00000000
printing eip:

c48370c3
*pde = 00000000
Oops: 0002
CPU: 0
EIP: 0010:[<c48370c3>]
EFLAGS: 00010286
eax: ffffffff ebx: c2281a20 ecx: c48370c0 edx: c2281a40
esi: 4000c000 edi: 4000c000 ebp: c38adf8c esp: c38adf8c
```

```

ds: 0018  es: 0018  ss: 0018
Process ls (pid: 23171, stackpage=c38ad000)
Stack: 0000010e c01356e6 c2281a20 4000c000 0000010e c2281a40 c38ac000 \
        0000010e
        4000c000 bffffc1c 00000000 00000000 c38adfc4 c010b860 00000001 \
        4000c000
        0000010e 0000010e 4000c000 bffffc1c 00000004 0000002b 0000002b \
        00000004
Call Trace: [<c01356e6>] [<c010b860>]
Code: c7 05 00 00 00 00 00 00 00 00 31 c0 89 ec 5d c3 8d b6 00 00

```

这个消息是通过对 **faulty** 模块的一个设备进行写操作而产生的，**faulty** 这个模块专为演示出错而编写。**faulty.c** 中 **write** 方法的实现很简单：

```

ssize_t faulty_write (struct file *filp, const char *buf, size_t count,
                      loff_t *pos)
{
    /* make a simple fault by dereferencing a NULL pointer */
    *(int *)0 = 0;
    return 0;
}

```

正如读者所见，我们这使用了一个 **NULL** 指针。因为 **0** 决不会是个合法的指针值，所以错误发生，内核进入上面的 **oops** 消息状态。这个调用进程接着就被杀掉了。在 **read** 实现中，**faulty** 模块还有更多有意思的错误状态。

```

char faulty_buf[1024];

ssize_t faulty_read (struct file *filp, char *buf, size_t count,
                    loff_t *pos)
{
    int ret, ret2;
    char stack_buf[4];

    printk(KERN_DEBUG "read: buf %p, count %li\n", buf, (long)count);
    /* the next line oopses with 2.0, but not with 2.2 and later */
    ret = copy_to_user(buf, faulty_buf, count);
    if (!ret) return count; /* we survived */

    printk(KERN_DEBUG "didn't fail: retry\n");
    /* For 2.2 and 2.4, let's try a buffer overflow */
    sprintf(stack_buf, "1234567\n");
    if (count > 8) count = 8; /* copy 8 bytes to the user */
    ret2 = copy_to_user(buf, stack_buf, count);
    if (!ret2) return count;
    return ret2;
}

```

这段程序首先从一个全局缓冲区读取数据，但并不检查数据的长度，然后通过对一个局部缓冲区进行写入操作，制造一次缓冲区溢出。第一个操作仅在 **2.0** 内核会导致 **oops** 的发生，因为后期版本能自动地处理用户拷贝函数。缓冲区溢出则会在所有版本的内核中造成 **oops**；然而，由于 **return** 指令把指令指针带到了不知道的地方，所以这种错误很难跟踪，所能获得的仅是如下的信息：

```

EIP: 0010:[<00000000>]
[...]
Call Trace: [<c010b860>]
Code: Bad EIP value.

```

用户处理 **oops** 消息的主要问题在于，我们很难从十六进制数值中看出什么内在的意义；为了使这些数据对程序员更有意义，需要把它们解析为符号。有两个工具可用来为开发人员完成这样的解析：**klogd** 和 **ksymoops**。前者只要运行就会自行进行符号解码；后者则需要用户有目的地调用。下面的讨论，使用了在我们第一个 **oops** 例子中通过使用 **NULL** 指针而产生的出错信息。

使用 klogd

klogd 守护进程能在 **oops** 消息到达记录文件之前对它们解码。很多情况下，**klogd** 可以为开发者提供所有必要的信息用于捕捉问题的所在，可是有时开发者必须给它一定的帮助。

当 **faulty** 的一个 **oops** 输出送达系统日志时，转储信息看上去会是下面的情况（注意 **EIP** 行和 **stack** 跟踪记录中已经解码的符号）：

```
Unable to handle kernel NULL pointer dereference at virtual address \
00000000
printing eip:
c48370c3
*pde = 00000000
Oops: 0002
CPU: 0
EIP: 0010:[faulty:faulty_write+3/576]
EFLAGS: 00010286
eax: ffffffff ebx: c2c55ae0 ecx: c48370c0 edx: c2c55b00
esi: 0804d038 edi: 0804d038 ebp: c2337f8c esp: c2337f8c
ds: 0018 es: 0018 ss: 0018
Process cat (pid: 23413, stackpage=c2337000)
Stack: 00000001 c01356e6 c2c55ae0 0804d038 00000001 c2c55b00 c2336000 \
00000001
0804d038 bffffbd4 00000000 00000000 bffffbd4 c010b860 00000001 \
0804d038
00000001 00000001 0804d038 bffffbd4 00000004 0000002b 0000002b \
00000004
Call Trace: [sys_write+214/256] [system_call+52/56]
Code: c7 05 00 00 00 00 00 00 00 00 31 c0 89 ec 5d c3 8d b6 00 00
```

klogd 提供了大多数必要信息用于发现问题。在这个例子中，我们看到指令指针（**EIP**）正执行于函数 **faulty_write** 中，因此我们就知道该从哪儿开始检查。字符串 **3/576** 告诉我们处理器正处于函数的第 3 个字节上，而函数整体长度为 576 个字节。注意这些数值都是十进制的，而非十六进制。

然而，当错误发生在可装载模块中时，为了获取错误相关的有用信息，开发者还必须注意一些情况。**klogd** 在开始运行时装入所有可用符号，并随后使用这些符号。如果在 **klogd** 已经对自身初始化之后（一般在系统启动时），装载某个模块，那 **klogd** 将不会有这个模块的符号信息。强制 **klogd** 取得这些信息的办法是，发送一个 **SIGUSR1** 信号给 **klogd** 进程，这种操作在时间顺序上，必须是在模块已经装入（或重新装载）之后，而在进行任何可能引起 **oops** 的处理之前。

还可以在运行 **klogd** 时加上 **-p** 选项，这会使它在任何发现 **oops** 消息的时刻重新读入符号信息。不过，**klogd** 的 **man** 手册不推荐这个方法，因为这使 **klogd** 在出问题之后再向内核查询信息。而发生错误之后，所获得的信息可能是完全错误的了。

为了使 **klogd** 正确地工作，必须给它提供符号表文件 **System.map** 的一个当前复本。通常这个文件在 **/boot** 中；如果从一个非标准的位置编译并安装了一个内核，就需要把 **System.map** 拷贝到 **/boot**，或告知 **klogd** 到什么位置查看。如果符号表与当前内核不匹配，**klogd** 就会拒绝解析符号。

假如一个符号被解析在系统日志中，那么就有理由确信它已被正确解析了。

使用 ksymoops

有些时候，**klogd** 对于跟踪目的而言仍显不足。开发者经常既需要取得十六进制地址，又要获得对应的符号，而且偏移量也常需要以十六进制的形式打印出来。除了地址解码之外，往往还需要更多的信息。对 **klogd** 来说，在出错期间被杀掉，也是常用的事情。在这些情况下，可以调用一个更为强大的 **oops** 分析器，**ksymoops** 就是这样的一个工具。

在 2.3 开发系列之前，**ksymoops** 是随内核源码一起发布的，位于 **scripts** 目录之下。它现在则在自己的 FTP 站点上，对它的维护是与内核相独立的。即使读者所用的仍是较早期的内核，或许还可以从 <ftp://ftp.ocs.com.au/pub/ksymoops> 站点上获取这个工具的升级版本。

为了取得最佳的工作状态，除错误消息之外，**ksymoops** 还需要很多信息；可以使用命令行选项告诉它在什么地方能找到这些各个方面的内容。**ksymoops** 需要下列内容项：

System.map 文件	这个映射文件必须与 oops 发生时正在运行的内核相一致。默认为 /usr/src/linux/System.map 。
模块列表	ksymoops 需要知道 oops 发生时都装入了哪些模块，以便获得它们的符号信息。如果未提供这个列表， ksymoops 会查看 /proc/modules 。
在 oops 发生时已定义好的内核符号表	默认从 /proc/ksyms 中取得该符号表。
当前正运行的内核映像的副本	注意， ksymoops 需要的是一个直接的内核映像，而不是象 vmlinuz 、 zImage 或 bzImage 这样被大多数系统所使用的压缩版本。默认是不使用内核映像，因为大多数人不会保存这样的一个内核。如果手边就有这样一个符合要求的内核的话，就应该采用 -v 选项告知 ksymoops 它的位置。
已装载的任何内核模块的目标文件位置	ksymoops 将在标准目录路径寻找这些模块，不过在开发中，几乎总要采用 -o 选项告知 ksymoops 这些模块的存放位置。

虽然 **ksymoops** 会访问 **/proc** 中的文件来取得它所需的信息，但这样获得的结果是不可靠的。在 **oops** 发生和 **ksymoops** 运行的时间间隙中，系统几乎一定会重新启动，这样取自 **/proc** 的信息就可能与故障发生时的实际状态不符合。只要有可能，最好在引起 **oops** 发生之前，保存 **/proc/modules** 和 **/proc/ksyms** 的副本。

我们强烈建议驱动程序开发人员阅读 **ksymoops** 的手册页，这是一个很好的资料文档。

这个工具命令行中的最后一个参数是 **oops** 消息的位置；如果缺少这个参数，**ksymoops** 会按 Unix 的惯例去读取标准输入设备。运气好的话，消息可以从系统日志中重新恢复；在发生很严重的崩溃情况时，我们可能不得不将这些消息从屏幕上抄下来，然后再敲进去（除非用的是串口控制台，这对内核开发人员来说，是非常棒的工具）。

注意，当 **oops** 消息已经被 **klogd** 处理过时，**ksymoops** 将会陷于混乱。如果 **klogd** 已经运行，而且 **oops** 发生后系统仍在运行，那么经常可以通过调用 **dmesg** 命令来获得一个干净的 **oops**

消息。

如果没有明确地提供全部的上述信息，**ksymoops** 会发出警告。对于载入模块未作符号定义这类的情况，它同样会发出警告。一个不作任何警告的 **ksymoops** 是很少见的。

ksymoops 的输出类似如下：

```
>>EIP; c48370c3 <[faulty]faulty_write+3/20> <=====
Trace; c01356e6 <sys_write+d6/100>
Trace; c010b860 <system_call+34/38>
Code; c48370c3 <[faulty]faulty_write+3/20>
00000000 <_EIP>:
Code; c48370c3 <[faulty]faulty_write+3/20> <=====
  0:  c7 05 00 00 00    movl    $0x0,0x0 <=====
Code; c48370c8 <[faulty]faulty_write+8/20>
  5:  00 00 00 00 00
Code; c48370cd <[faulty]faulty_write+d/20>
  a:  31 c0              xorl    %eax,%eax
Code; c48370cf <[faulty]faulty_write+f/20>
  c:  89 ec              movl    %ebp,%esp
Code; c48370d1 <[faulty]faulty_write+11/20>
  e:  5d                  popl    %ebp
Code; c48370d2 <[faulty]faulty_write+12/20>
  f:  c3                  ret
Code; c48370d3 <[faulty]faulty_write+13/20>
 10:  8d b6 00 00 00    leal    0x0(%esi),%esi
Code; c48370d8 <[faulty]faulty_write+18/20>
 15:  00
```

正如上面所看到的，**ksymoops** 提供的 EIP 和内核堆栈信息与 **klogd** 所做的很相似，不过要更为准确，而且是十六进制形式的。可以注意到，**faulty_write** 函数的长度被正确地报告为 **0x20** 个字节。这是因为 **ksymoops** 读取了模块的目标文件，并从中获得了全部的有用信息。

而且在这个例子中，还可以得到错误发生处代码的汇编语言形式的转储输出。这些信息常被用于确切地判断发生了些什么事情；这里很明显，错误在于一个向 **0** 地址写入数据 **0** 的指令。

ksymoops 的一个有趣特点是，它可以移植到几乎所有 **Linux** 可以运行的平台上，而且还利用了 **bfd**（二进制格式描述）库同时支持多种计算机结构。走出 **PC** 的世界，我们可以看到 **SPARC64** 平台上显示的 **oops** 消息是何等的相似（为了便于排版有几行被打断了）：

```
Unable to handle kernel NULL pointer dereference
tsk->mm->context = 00000000000000734
tsk->mm->pgd = fffff80003499000
      \ /      \ /
      '@' /  .. \ '@'
      /_ | \_ _/ | _ \
      \_ _U_/
ls(16740): Oops
TSTATE: 0000004400009601 TPC: 0000000001000128 TNPC: 0000000000457fbc \
Y: 00800000
g0: 000000007002ea88 g1: 0000000000000004 g2: 0000000070029fb0 \
g3: 0000000000000018
g4: fffff80000000000 g5: 0000000000000001 g6: fffff8000119c000 \
g7: 0000000000000001
o0: 0000000000000000 o1: 000000007001a000 o2: 0000000000000178 \
o3: fffff8001224f168
o4: 0000000001000120 o5: 0000000000000000 sp: fffff8000119f621 \
ret_pc: 0000000000457fb4
```

```

10: ffffff800122376c0 11: ffffffffefea 12: 000000000002c400 \
13: 000000000002c400
14: 0000000000000000 15: 0000000000000000 16: 0000000000019c00 \
17: 0000000070028cbc
i0: ffffff8001224f140 i1: 000000007001a000 i2: 0000000000000178 \
i3: 000000000002c400
i4: 000000000002c400 i5: 000000000002c000 i6: ffffff8000119f6e1 \
i7: 00000000000410114
Caller[00000000000410114]
Caller[000000007007cba4]
Instruction DUMP: 01000000 90102000 81c3e008 <c0202000> \
30680005 01000000 01000000 01000000 01000000

```

请注意，指令转储并不是从引起错误的那个指令开始，而是之前的三条指令：这是因为 RISC 平台以并行的方式执行多条指令，这样可能产生延期的异常，因此必须能回溯最后的几条指令。

下面是当从 TSTATE 行开始输入数据时，ksymoops 所打印出的信息：

```

>>TPC; 0000000001000128 <[faulty].text.start+88/a0> <=====
>>07; 0000000000457fb4 <sys_write+114/160>
>>I7; 00000000000410114 <linux_sparc_syscall+34/40>
Trace; 00000000000410114 <linux_sparc_syscall+34/40>
Trace; 000000007007cba4 <END_OF_CODE+6f07c40d/????>
Code; 000000000100011c <[faulty].text.start+7c/a0>
0000000000000000 <_TPC>:
Code; 000000000100011c <[faulty].text.start+7c/a0>
0: 01 00 00 00 nop
Code; 0000000001000120 <[faulty].text.start+80/a0>
4: 90 10 20 00 clr %o0 ! 0 <_TPC>
Code; 0000000001000124 <[faulty].text.start+84/a0>
8: 81 c3 e0 08 retl
Code; 0000000001000128 <[faulty].text.start+88/a0> <=====
c: c0 20 20 00 clr [ %g0 ] <=====
Code; 000000000100012c <[faulty].text.start+8c/a0>
10: 30 68 00 05 b,a %xcc, 24 <_TPC+0x24> \
0000000001000140 <[faulty]faulty_write+0/20>
Code; 0000000001000130 <[faulty].text.start+90/a0>
14: 01 00 00 00 nop
Code; 0000000001000134 <[faulty].text.start+94/a0>
18: 01 00 00 00 nop
Code; 0000000001000138 <[faulty].text.start+98/a0>
1c: 01 00 00 00 nop
Code; 000000000100013c <[faulty].text.start+9c/a0>
20: 01 00 00 00 nop

```

要打印出上面显示的反汇编代码，我们就必须告知 ksymoops 目标文件的格式和结构(之所以需要这些信息，是因为 SPARC64 用户空间的本地结构是 32 位的)。本例中，使用选项 `-t elf64-sparc -a sparc:v9` 可进行这样的设置。

读者可能会抱怨对调用的跟踪并没带回什么值得注意的信息；然而，SPARC 处理器并不会把所有的调用跟踪记录保存到堆栈中：07 和 I7 寄存器保存了最后调用的两个函数的指令指针，这就是它们出现在调用跟踪记录边上的原因。在这个例子中，我们可以看到，故障指令位于一个由 `sys_write` 调用的函数中。

要注意的是，无论平台/结构是怎样的一种配合情况，用来显示反汇编代码的格式与 `objdump` 程序所使用的格式是一样的。`objdump` 是个很强大的工具；如果想查看发生故障的完整函数，可以调用命令：`objdump -d faulty.o`（再次重申，对于 SPARC64 平台，需要使用特殊选项：`--target`

elf64-sparc-architecture sparc:v9)。

关于 `objdump` 和它的命令行选项的更多信息，可以参阅这个命令的手册页帮助。

学习对 `oops` 消息进行解码，需要一定的实践经验，并且了解所使用的目标处理器，以及汇编语言的表达习惯等。这样的准备是值得的，因为花费在学习上的时间很快会得到回报。即使之前读者已经具备了非 `Unix` 操作系统中 `PC` 汇编语言的专门知识，仍有必要花些时间对此进行学习，因为 `Unix` 的语法与 `Intel` 的语法并不一样。（在 `as` 命令 `infor` 页的“`i386-specific`”一章中，对这种差异进行了很好的描述。）

4.4.2 系统挂起

尽管内核代码中的大多数错误仅会导致一个 `oops` 消息，但有时它们则会将系统完全挂起。如果系统挂起了，任何消息都无法打印。例如，如果代码进入一个死循环，内核就会停止进行调度，系统不会再响应任何动作，包括 `Ctrl-Alt-Del` 组合键。处理系统挂起有两个选择——要么是防范于未然；要么就是亡羊补牢，在发生挂起后调试代码。

通过在一些关键点上插入 `schedule` 调用可以防止死循环。`schedule` 函数（正如读者猜到的）会调用调度器，并因此允许其他进程“偷取”当然进程的 `CPU` 时间。如果该进程因驱动程序的错误而在内核空间陷入死循环，则可以在跟踪到这种情况之后，借助 `schedule` 调用杀掉这个进程。

当然，应该意识到任何对 `schedule` 的调用都可能给驱动程序带来代码重入的问题，因为 `schedule` 允许其他进程开始运行。假设驱动程序进行了合适的锁定，这种重入通常还并不致于带来问题。不过，一定不要在驱动程序持有 `spinlock` 的任何时候调用 `schedule`。

如果驱动程序确实会挂起系统，而又不知该在什么位置插入 `schedule` 调用时，最好的方法是加入一些打印信息，并把它们写入控制台（通过修改 `console_loglevel` 的数值）。

有时系统看起来象挂起了，但其实并没有。例如，如果键盘因某种奇怪的原因被锁住了就会发生这种情况。运行专为探明此种情况而设计的程序，通过查看它的输出情况，可以发现这种假挂起。显示器上的时钟或系统负荷表就是很好的状态监视器；只要它保持更新，就说明 `scheduler` 正在工作。如果没有使用图形显示，则可以运行一个程序让键盘 `LED` 闪烁，或不时地开关软驱马达，或不断触动扬声器（通常蜂鸣声是令人烦恼的，应尽量避免；可改为寻求 `ioctl` 命令 `KDMKTONE`），来检查 `scheduler` 是否工作正常。O'Reilly FTP 站点上可以找到一个例子（`misc-progs/heartbeat.c`），它会使键盘 `LED` 不断闪烁。

如果键盘不接收输入，最佳的处理方法是从网络登录到系统中，杀掉任何违例的进程，或是重新设置键盘（用 `kdb_mode -a`）。然而，如果没有可用的网络用来帮助恢复的话，即使发现了系统挂起是由键盘死锁造成的也没有用了。如果是这样的情况，就应该配置一种可替代的输入设备，以便至少可以正常地重启系统。比起去按所谓的“大红钮”，在你的计算机上，通过替代的输入设备来关机或重启系统要更为容易些，而且它可以免去 `fsck` 对磁盘的长时间扫描。

例如，这种替代输入设备可以是鼠标。1.10 或更新版本的 `gpm` 鼠标服务器可以通过命令行选项支持类似的功能，不过仅限于文本模式。如果没有网络连接，并且以图形方式运行，则建议采用某些自定义的解决方案，比如，设置一个与串口线 `DCD` 针脚相连的开关，并编写一个查询 `DCD` 信

号状态变化的脚本，用于从外界干预键盘已被死锁的系统。

对于上述情形，一个不可缺少的工具是“magic SysRq key”，2.2 和后期版本内核中，在其它体系结构上也可利用得到它。SysRq 魔法键是通过 PC 键盘上的 ALT 和 SysRq 组合键来激活的，在 SPARC 键盘上则是 ALT 和 Stop 组合键。连同这两个键一起按下的第三个键，会执行许多有用动作中的其中一种，这些动作如下：

r

在无法运行 kbd_mode 的情况下，关闭键盘的 raw 模式。

k

激活“留意安全键”（SAK）功能。SAK 将杀掉当前控制台上运行的所有进程，留下一个干净的终端。

s

对所有磁盘进行紧急同步。

u

尝试以只读模式重新挂装所有磁盘。这个操作通常紧接着 s 动作之后被调用，它可以在系统处于严重故障状态时节省很多检查文件系统的时间。

b

立即重启系统。注意先要同步并重新挂装磁盘。

p

打印当前的寄存器信息。

t

打印当前的任务列表。

m

打印内存信息。

还有其它的一些 SysRq 功能：要获得完整列表，可参阅内核源码 Documentation 目录下的 sysrq.txt 文件。注意，SysRq 功能必须明确地在内核配置中被开启，出于安全原因，大多数发行系统并未开启它。不过，对于一个用于驱动程序开发的系统来说，为开启 SysRq 功能而带来的重新编译新内核的麻烦是值得的。SysRq 必须在运行时通过下面的命令启动：

```
echo 1 > /proc/sys/kernel/sysrq
```

在复现系统的挂起故障时，另一个要采取的预防措施是，把所有的磁盘都以只读的方式挂装在系统上（或干脆就卸装它们）。如果磁盘是只读的或者并未挂装，就不会发生破坏文件系统或致使文件系统处于不一致状态的危险。另一个可行方法是，使用通过 NFS（网络文件系统）将其所有文件系统挂装入系统的计算机。这个方法要求内核具有“NFS-Root”的能力，而且在引导时还需传入一些特定参数。如果采用这种方法，即使我们不借助于 SysRq，也能避免任何文件系统的崩溃，因为 NFS 服务器管理文件系统的一致性，而它并不受驱动程序的影响。

4.5 调试器和相关工具

最后一种调试模块的方法就是使用调试器来一步步地跟踪代码，查看变量和计算机寄存器的值。这种方法非常耗时，应该尽量避免。不过，某些情况下通过调试器对代码进行细粒度的分析是很有价值的。

在内核中使用交互式调试器是一个很复杂的问题。出于对系统所有进程的整体利益考虑，内核在自己的地址空间中运行。其结果是，许多用户空间下的调试器所提供的常用功能很难用于内核之中，比如断点和单步调试等。本节着眼于调试内核的几种方法；它们中的每一种都各有利弊。

4.5.1 使用 gdb

gdb 在探究系统内部行为时非常有用。在我们这个层次上，熟练使用调试器，需要掌握 **gdb** 命令、了解目标平台的汇编代码，还要具备对源代码和优化后的汇编码进行匹配的能力。

启动调试器时必须把内核看作是一个应用程序。除了指定未压缩的内核映像文件名外，还应该在命令行中提供“**core** 文件”的名称。对于正运行的内核，所谓 **core** 文件就是这个内核在内存中的核心映像，**/proc/kcore**。典型的 **gdb** 调用如下所示：

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

第一个参数是未经压缩的内核可执行文件的名字，而不是 **zImage** 或 **bzImage** 以及其他任何压缩过的内核。

gdb 命令行的第二个参数是 **core** 文件的名字。与其它 **/proc** 中的文件类似，**/proc/kcore** 也是在被读取时产生的。当在 **/proc** 文件系统中执行 **read** 系统调用时，它会映射到一个用于数据生成而不是数据读取的函数上；我们已在“使用 **/proc** 文件系统”一节中介绍了这个特性。**kcore** 用来按照 **core** 文件的格式表示内核“可执行文件”；由于它要表示对应于所有物理内存的整个内核地址空间，所以是一个非常巨大的文件。在 **gdb** 的使用中，可以通过标准 **gdb** 命令查看内核变量。例如，**p jiffies** 可以打印从系统启动到当前时刻的时钟滴答数。

从 **gdb** 打印数据时，内核仍在运行，不同数据项的值会在不同时刻有所变化；然而，**gdb** 为了优化对 **core** 文件的访问，会将已经读到的数据缓存起来。如果再次查看 **jiffies** 变量，仍会得到和上次一样的值。对通常的 **core** 文件来说，对变量值进行缓存是正确的，这样可避免额外的磁盘访问。但对“动态的”**core** 文件来说就不方便了。解决方法是在需要刷新 **gdb** 缓冲区的时候，执行命令 **core-file /proc/kcore**；调试器将使用新的 **core** 文件并丢弃所有的旧信息。不过，读新数据时并不总是需要执行 **core-file** 命令；**gdb** 以几 KB 大小的小数据块形式读取 **core** 文件，缓存的仅是已经引用的若干小块。

对内核进行调试时，**gdb** 通常能提供的许多功能都不可用。例如，**gdb** 不能修改内核数据；因为在处理其内存映像之前，**gdb** 期望把待调试程序运行在自己的控制之下。同样，也不能设置断点或观察点，或者单步跟踪内核函数。

如果用调试选项（**-g**）编译了内核，产生的 **vmlinux** 会比没有使用 **-g** 选项的更适合于 **gdb**。不过要注意，用 **-g** 选项编译内核需要大量的磁盘空间（每个目标文件和内核自身都会比通常的大三倍

甚至更多)。

在非 PC 类计算机上, 情况则不尽相同。在 Alpha 上, `make boot` 会在生成可启动映像前将调试信息去掉, 所以最终会获得 `vmlinux` 和 `vmlinux.gz` 两个文件。`gdb` 可以使用前者, 后者用来启动。在 SPARC 上, 默认情况则是不把内核 (至少是 2.0 内核) 调试信息去掉。

当用 `-g` 选项编译内核并且和 `/proc/kcore` 一起使用 `vmlinux` 运行调试器时, `gdb` 可以返回很多内核内部信息。例如, 可以使用下面的命令来转储结构数据, 如 `p *module_list`、`p *module_list->next` 和 `p *chrdevs[4]->fops` 等。为了在使用 `p` 命令时取得最好效果, 有必要保留一份内核映射表和随手可及的源码。

利用 `gdb` 可在当前内核上执行的另一个有用任务是, 通过 `disassemble` 命令 (可缩写为 `disass`) 或是 “检查指令” (`x/i`) 命令对函数进行反汇编。`disassemble` 命令的参数可以是函数名或是内存范围; 而 `x/i` 则使用一个内存地址做为参数, 也可以是符号名称的形式。例如, 可以用 `x/20i` 反汇编 20 条指令。注意, 不能反汇编一个模块的函数, 因为调试器作用的是 `vmlinux`, 它并不知道模块的情况。如果试图通过地址反汇编模块代码, `gdb` 很有可能会返回 “Cannot access memory at xxxx (不能访问 xxxx 处的内存)” 这样的信息。基于同样的原因, 也不能查看属于模块的数据项。如果已知道变量的地址, 可以从 `/dev/mem` 中读出它们的值, 但要弄明白从系统内存中分解出的原始数据的含义, 难度是相当大的。

如果需要反汇编模块函数, 最好对模块的目标文件用 `objdump` 工具进行处理。很不幸, 该工具只能对磁盘上的文件复本进行处理, 而不能对运行中的模块进行处理; 因此, 由 `objdump` 给出的地址都是未经重定位的地址, 与模块的运行环境无关。对未经链接的目标文件进行反汇编的另一个不利因素在于, 其中的函数调用仍是未作解析的, 所以就无法轻松地区分是对 `printk` 的调用呢, 还是对 `kmalloc` 的调用。

正如上面看到的, 当目的在于查看内核的运行情况时, `gdb` 是一个有用的工具, 但对于设备驱动程序 的调试, 它还缺少一些至关重要的功能。

4.5.2 kdb 内核调试器

很多读者可能会奇怪这一点, 即为什么不把一些更高级的调试功能直接编译进内核呢。答案很简单, 因为 **Linus** 不信任交互式的调试器。他担心这些调试器会导致一些不良的修改, 也就是说, 修补的仅是一些表面现象, 而没有发现问题的真正原因所在。因此, 没有在内核中内建调试器。

然而, 其他的内核开发人员偶尔也会用到一些交互式的调试工具。`kdb` 就是其中一种内建的内核调试器, 它在 `oss.sgi.com` 上以非正式的补丁形式提供。要使用 `kdb`, 必须首先获得这个补丁 (取得的版本一定要和内核版本相匹配), 然后对当前内核源码进行 `patch` 操作, 再重新编译并安装这个内核。注意, `kdb` 仅可用于 IA-32(x86) 系统 (虽然用于 IA-64 的一个版本在主流内核源码中短暂地出现过, 但很快就被删去了)。

一旦运行的是支持 `kdb` 的内核, 有几个方法可以进入 `kdb` 的调试状态。在控制台上按下 **Pause** (或 **Break**) 键将启动调试。当内核发生 `oops`, 或到达某个断点时, 也会启动 `kdb`。无论是哪一种情况, 都看到下面这样的消息:

```
Entering kdb (0xc1278000) on processor 1 due to Keyboard Entry [1]kdb>
```

注意，当 **kdb** 运行时，内核所做的每一件事情都会停下来。当激活 **kdb** 调试时，系统不应运行其他的任何东西；尤其是，不要开启网络——当然，除非是在调试网络驱动程序。一般来说，如果要使用 **kdb** 的话，最好在启动时进入单用户模式。

作为一个例子，考虑下面这个快速的 **scull** 调试过程。假定驱动程序已被载入，可以象下面这样指示 **kdb** 在 **scull_read** 函数中设置一个断点：

```
[1]kdb> bp scull_read
Instruction(i) BP #0 at 0xc8833514 (scull_read)      is enabled on cpu 1
[1]kdb> go
```

bp 命令指示 **kdb** 在内核下一次进入 **scull_read** 时停止运行。随后我们输入 **go** 继续执行。在把一些东西放入 **scull** 的某个设备之后，我们可以在另一个终端的 **shell** 中运行 **cat** 命令尝试读取这个设备，这样一来就会产生如下的状态：

```
Entering kdb (0xc3108000) on processor 0 due to Breakpoint @ 0xc8833515
Instruction(i) breakpoint #0 at 0xc8833514
scull_read+0x1:  movl  %esp,%ebp
[0]kdb>
```

我们现在正处于 **scull_read** 的开头位置。为了查明是怎样到达这个位置的，我们可以看看堆栈跟踪记录：

```
[0]kdb> bt
      EBP      EIP      Function(args)
0xc3109c5c 0xc8833515  scull_read+0x1
0xc3109fbc 0xfc458b10  scull_read+0x33c255fc( 0x3, 0x803ad78, 0x1000,
0x1000, 0x804ad78)
0xbffffc88 0xc010bec0  system_call
[0]kdb>
```

kdb 试图打印出调用跟踪所记录的每个函数的参数列表。然而，它往往会被编译器所使用的优化技巧弄糊涂。所以在这个例子中，虽然 **scull_read** 实际只有四个参数，**kdb** 却打印出了五个。

下面我们来看看如何查询数据。**mds** 命令是用来对数据进行处理；我们可以用下面的命令查询 **scull_devices** 指针的值：

```
[0]kdb> mds scull_devices 1
c8836104: c4c125c0 ....
```

在这里，我们请求查看的是从 **scull_devices** 指针位置开始的一个字大小（4 个字节）的数据；应答告诉我们设备数据数组的起始地址位于 **c4c125c0**。要查看设备结构自身的数据值，我们需要用到这个地址：

```
[0]kdb> mds c4c125c0
c4c125c0: c3785000 ....
c4c125c4: 00000000 ....
c4c125c8: 00000fa0 ....
c4c125cc: 000003e8 ....
c4c125d0: 0000009a ....
```

```
c4c125d4: 00000000 ....
c4c125d8: 00000000 ....
c4c125dc: 00000001 ....
```

上面的 8 行分别对应于 `Scull_Dev` 结构中的 8 个成员。因此，通过显示的这些数据，我们可以知道，第一个设备的内存是从 `0xc3785000` 开始分配的，链表中没有下一个数据项，量子大小为 4000（十六进制形式为 `fa0`）字节，量子集大小为 1000（十六进制形式为 `3e8`），这个设备中有 154 个字节（十六进制形式为 `9a`）的数据，等等。

`kdb` 还可以修改数据。假设我们要从设备中削减一些数据：

```
[0]kdb> mm c4c125d0 0x50
0xc4c125d0 = 0x50
```

接下来对设备的 `cat` 操作所返回的数据就会少于上次。

`kdb` 还有许多其他的功能，包括单步调试（根据指令，而不是 C 源代码行），在数据访问中设置断点，反汇编代码，跟踪链表，访问寄存器数据等等。加上 `kdb` 补丁之后，在内核源码树的 `Documentation/kdb` 目录可以找到完整的手册页。

4.5.3 集成的内核调试器补丁

有很多内核开发人员为一个名为“集成的内核调试器”的非正式补丁作出过贡献，我们可将其简称为 IKD（integrated kernel debugger）。IKD 提供了很多值得关注的内核调试工具。`x86` 是这个补丁的主要平台，不过它也可以用于其它的结构体系之上。IKD 补丁可以从 <ftp://ftp.kernel.org/pub/linux/kernel/people/andrea/ikd> 下载。它是一个必须应用于内核源码的 `patch` 补丁；因为这个 `patch` 是与版本相关的，所以要确保下载的补丁与正使用的内核版本相一致。

IKD 补丁的功能之一是内核堆栈调试。如果开启这个功能，内核就会在每个函数调用时检查内核堆栈的空闲空间的大小，如果过小的话就会强制产生一个 `oops`。如果内核中的某些事情引起堆栈崩溃，这个工具就能用来帮助查找问题。这其实也就是一种“堆栈计量表”的功能，可以在任何特定的时刻查看堆栈的填充程度。

IKD 补丁还包含了一些用于发现内核死锁的工具。如果某个内核过程持续时间过久而没有得到调度的话，“软件死锁”探测器就会强制产生一个 `oops`。这是简单地通过对函数调用进行计数来实现的，如果计数值超过了一个预定义的阈值，探测器就会动作，并中止一些工作。IKD 的另一个功能是可以连续地把程序计数器打印到虚拟控制台上，这可以作为跟踪死锁的最后手段。“信号量死锁”探测器则是在某个进程的 `down` 调用持续时间过久时强制产生 `oops`。

IKD 中的其它调试功能包括内核的跟踪功能，它可以记录内核代码的执行路径。还有一些内存调试工具，包括一个内存泄漏探测器和一些称为“`poisoner`”的工具，它们在跟踪内存崩溃问题时非常有用。

最后，IKD 也包含前一节讨论过的 `kdb` 调试器。不过，IKD 补丁中的 `kdb` 版本有些老。如果需要 `kdb` 的话，我们推荐直接从 oss.sgi.com 获取当前的版本。

4.5.4 kgdb 补丁

kgdb 是一个在 Linux 内核上提供完整的 **gdb** 调试器功能的补丁，不过仅限于 x86 系统。它通过串口连线以钩子的形式挂入目标调试系统进行工作，而在远端运行 **gdb**。使用 **kgdb** 时需要两个系统——一个用于运行调试器，另一个用于运行待调试的内核。和 **kdb** 一样，**kgdb** 目前可从 oss.sgi.com 获得。

设置 **kgdb** 包括安装内核补丁并引导打过补丁之后的内核两个步骤。两个系统之间需要通过串口电缆（或空调制解调器电缆）进行连接，在 **gdb** 这一侧，需要安装一些支持文件。**kgdb** 补丁把详细的用法说明放在了文件 `Documentation/i386/gdb-serial.txt` 中；我们在这里就不再赘述。建议读者阅读关于“调试模块”的说明：接近末尾的地方，有一些出于这个目的而编写的很好的 **gdb** 宏。

4.5.5 内核崩溃转储分析器

崩溃转储分析器使系统能把发生 **oops** 时的系统状态记录下来，以便在随后空闲的时候查看这些信息。如果是对于一个异地用户的驱动程序进行支持，这些工具就会特别有用。用户可能不太愿意把 **oops** 复制下来，因此安装崩溃转储系统可以使技术支持人员不必依赖于用户的工作，也能获得用于跟踪用户问题的必要信息。也正是出于这样的原因，可供利用的崩溃转储分析器都是由那些对用户系统进行商业支持的公司开发的，这也就不足为奇了。

目前有两个崩溃转储分析器的补丁可以用于 Linux。在编写本节的时候，这两个工具都比较新，而且都处在不断的变化之中。与其提供可能已经过时的详细信息，我们倒不如只是给出一个概观，并指点读者在哪里可以找到更多的信息。

第一个分析器是 **LKCD** (Linux Kernel Crash Dumps, “Linux 内核崩溃转储”)。这个工具仍可以从 oss.sgi.com 上获得。当内核发生 **oops** 时，**LKCD** 会把当前系统状态（主要指内存）写入事先指定好的转储设备中。这个转储设备必须是一个系统交换区。下次重启中（在存储交换功能开启之前）系统会运行一个称为 **LCRASH** 的工具，来生成崩溃的概要记录，并可选择地把转储的复本保存在一个普通文件中。**LCRASH** 可以交互方式地运行，提供了很多调试器风格的命令，用以查询系统状态。

LKCD 目前只支持 Intel 32 位体系结构，并只能用在 **SCSI** 磁盘的交换分区上。

另一个崩溃转储设施可以从 www.missioncriticallinux.com 获得。这个崩溃转储子系统直接在目录 `/var/dumps` 中创建崩溃转储文件，而且并不使用交换区。这样就使某些事情变得更为容易，但也意味着在知道问题已经出现在哪里的时候，文件系统已被系统修改。生成的崩溃转储的格式是标准的 **core** 文件格式，所以可以利用 **gdb** 这类工具进行事后的分析。这个工具包也提供了另外的分析器，可以从崩溃转储文件中解析出比 **gdb** 更丰富的信息。

4.5.6 用户模式的 Linux 虚拟机

用户模式 Linux 是一个很有意思的概念。它作为一个独立的可移植的 Linux 内核而构建，包含在子目录 `arch/um` 中。然而，它并不是运行在某种新的硬件上，而是运行在基于 Linux 系统调用接口所实现的虚拟机之上。因此，用户模式 Linux 可以使 Linux 内核成为一个运行在 Linux 系统

之上单独的、用户模式的进程。

把一个内核的复本当作用户模式下的进程来运行可以带来很多好处。因为它运行在一个受约束的虚拟处理器之上，所以有错误的内核不会破坏“真正的”系统。对软/硬件的不同配置可以在相同的框架中轻易地进行尝试。并且，对于内核开发人员来说最值得瞩目的特点在于，可以很容易地利用 **gdb** 或其它调试器对用户模式 **Linux** 进行处理。归根结底，它只是一个进程。很明显，用户模式 **Linux** 有潜力加快内核的开发过程。

迄今为止，用户模式 **Linux** 虚拟机还未在主流内核中发布；要下载它，必须访问它的 **web** 站点 (<http://user-mode-linux.sourceforge.net>)。需要提醒的是，它仅可以集成到 2.4.0 之后的早期 2.4 内核版本中；当然等到本书出版的时候，版本支持方面可能会做得更好。

目前，用户模式 **Linux** 虚拟机也存在一些重大的限制，不过大部分可能很快就会得到解决。虚拟处理器当前只能工作于单处理器模式；虽然虚拟机可以毫无问题地运行在 **SMP** 系统上，但它仍是把主机模拟成单 **CPU** 模式。不过，对于驱动编写者来说，最大的麻烦在于，用户模式内核不能访问主机系统上的硬件设备。因此，尽管用户模式 **Linux** 虚拟机对于本书中的大多数样例驱动程序的调试非常有用，却无法用于调试那些处理实际硬件的驱动程序。最后一点，用户模式 **Linux** 虚拟机仅能运行在 **IA-32** 体系结构之上。

因为对所有这些问题的修补工作正在进行之中，所以在不久的将来，对于 **Linux** 设备驱动程序的开发人员，用户模式 **Linux** 虚拟机可能会成为一个不可或缺的工具。

4.5.7 Linux 跟踪工具包

Linux 跟踪工具包 (**LTT**) 是一个内核补丁，包含了一组可以用于内核事件跟踪的相关工具集。跟踪内容包括时间信息，而且还能合理地建立在一段指定时间内所发生事件的完整图形化描述。因此，**LTT** 不仅能用于调试，还能用来捕捉性能方面的问题。

在 **Web** 站点 www.opersys.com/LTT 上，可以找到 **LTT** 以及大量的资料。

4.5.8 Dynamic Probes

Dynamic Probes (或 **DProbes**) 是 **IBM** 为基于 **IA-32** 结构的 **Linux** 发布的一种调试工具 (遵循 **GPL** 协议)。它可以在系统的几乎任何一个地方放置一个“探针”，既可以是用户空间也可以是内核空间。这个探针由一些当控制到达指定地点即开始执行的代码 (用一种特别设计的，面向堆栈的语言编写) 组成。这种代码能把信息传送回用户空间，修改寄存器，或者完成许多其它的工作。**DProbes** 很有用的特点是，一旦内核编译进了这个功能，探针就可以插到一个运行系统的任一个位置，而无需重建内核或重新启动。**DProbes** 也可以协同 **LTT** 工具在任意位置插入新的跟踪事件。

DProbes 工具可以从 **IBM** 的开放源码站点，即 oss.software.ibm.com 上下载。

第 5 章 增强的字符驱动程序操作



在第 3 章，我们已经构建了一个结构完整的可读写设备驱动程序，但一个实际可用的设备通常会提供比同步 `read` 和 `write` 更多的功能。我们现在已经有了调试工具，即使出现了什么问题，我们也可以继续实验下去，实现新的操作。

通常，除了读写设备之外，设备驱动程序还需要提供各种各样的硬件控制能力。这些控制操作一般是通过 `ioctl` 方法来支持的，另一种方法是检查写入设备中的数据流，使用特殊序列做为控制命令。后面这种方法应该尽量避免，因为它需要保留一些字符用于控制，在数据中就不能包含这些字符了，另外，这种方法使用起来也比 `ioctl` 复杂。不过尽管如此，作为一种设备控制方法，有时它还是有用的，如 `tty` 和其它一些设备就在使用这种方法。稍后我们会在本章的“非 `ioctl` 的设备控制”一节中介绍这项技术。

正如我们在前一章中所阐述的，`ioctl` 系统调用为设备驱动程序执行“命令”提供了一个设备特有的入口点。与 `read` 等方法不同，`ioctl` 是设备特有的，它允许应用程序访问被驱动硬件的特殊功能，如配置设备、进入或退出某种操作模式等。这些控制操作通常无法通过 `read/write` 文件操作完成。例如，向串口写入的所有东西都作为数据发送，因此无法通过写设备的方法来改变波特率。这就是 `ioctl` 所要做的：控制 I/O 通道。

与 `scull` 不同，实际设备的另一个重要特点是，要读取或写入的数据需要同其他硬件交换而得，这就需要某些同步机制。这就是阻塞型 I/O 和异步通知概念产生的基础，本章将通过改写 `scull` 设备驱动程序介绍这两个概念，这个驱动程序利用不同进程间的交互产生异步事件。与最初的 `scull` 相同，你无需使用特定的硬件来测试驱动程序的工作情况。直到第 8 章“硬件管理”我们才会真正与硬件打交道。

5.1 `ioctl`

在用户空间内调用的 `ioctl` 函数一般具有如下原型：

```
int ioctl(int fd, int cmd, ...);
```

由于使用了一连串的“.”的缘故，这个原型在 Unix 系统调用中显得比较特别，通常这些点代表可变数目的参数表。但是在实际系统中，系统调用不会真正使用可变数目的参数，而是必须有精确

定义参数个数，因为用户程序只能通过硬件“门”才能访问它们，这一点在第 2 章“用户空间与内核空间”中已经指出过了。所以，原型中的这些点并不是数目不定的一串参数，而只是一个可选参数，习惯上用 `char *argp` 定义，这里用点只是为了在编译时防止编译器进行类型检查。第 3 个参数的具体形式依赖于要完成的控制命令，也就是第 2 个参数。某些控制命令不需要参数，某些需要一个整数参数，而某些则需要一个指针参数。使用指针可以向 `ioctl` 传递任意数据，这样设备可以与用户空间交换任意数量的数据。

另一方面，设备驱动程序的 `ioctl` 方法，是按照如下原型获取其参数的：

```
int (*ioctl) (struct inode *inode, struct file *filp,
             unsigned int cmd, unsigned long arg);
```

`inode` 和 `filp` 两个指针的值对应于应用程序传递的文件描述符 `fd`，传给 `open` 方法的也是同一参数。参数 `cmd` 由用户空间不经修改地传递给驱动程序，可选的 `arg` 参数则无论用户程序使用的是指针还是整数值，它都以 `unsigned long` 的形式传递给驱动程序。如果调用程序没有传递第 3 个参数，驱动程序所接收的 `arg` 没有任何意义。

由于对这个附加参数的类型检查被关闭了，如果传递给 `ioctl` 一个非法参数，编译器就无法报警，这样，程序员就有可能漏过这个错误，而直到运行时才会察觉。这个类型检查的缺陷可以视为 `ioctl` 定义中的一个小问题，不过比起它提供的通用性，这也是必要的代价。

读者可能已经想到了，大多数 `ioctl` 的实现中都包括一个 `switch` 语句来根据 `cmd` 参数选择对应的操作。不同的命令被赋予不同的数值，为了简化代码，通常会在代码中使用符号名代替数值，这些符号名都是在预处理中赋值的。定制的设备驱动程序通常会在它们的头文件中声明这些符号，在 `scull.h` 中声明了 `scull` 所使用的符号。为了访问这些符号，用户程序自然也要包含这些头文件。

5.1.1 选择 `ioctl` 命令

在编写 `ioctl` 代码之前，需要选择对应不同命令的编号。遗憾的是，简单地从 1 开始选择号码是不行的。

为了防止对错误的设备使用正确的命令，命令号应该在系统范围内唯一。这种错误匹配并不是不会发生，程序可能发现自己正在试图对 `FIFO` 和 `audio` 等这类非串口设备输入流修改波特率。如果每一个 `ioctl` 命令都是唯一的，应用程序进行这种操作就会得到一个 `EINVAL` 错误，而不是无意间成功地完成了意想不到的操作。

为方便程序员创建唯一的 `ioctl` 命令号，每一个命令号被分为多个位字段。`Linux` 的第一版使用了一个 16 位整数：高 8 位是与设备相关的“幻”数，低 8 位是一个序列号码，在设备内是唯一的。当时采用这种方案是因为，用 `Linus` 的话说，他有点“无头绪”，后来才得到一个更好的位字段分割方案。遗憾的是，相当多的驱动程序仍使用旧的约定，它们不得不这样：修改命令号会使很多已有的程序无法运行。不过在我的源码中，使用了新的命令定义约定。

为了按新方法为驱动程序选择 `ioctl` 号，应该首先看看 `include/asm/ioctl.h` 和 `Documentation/ioctl-number.txt` 这两个文件。头文件定义了位字段：类型（幻数）、基数、传送方向、参数大小等等。`ioctl-number.txt` 文件中罗列了内核使用的幻数，这样，在选择自己的幻数时

就可以避免和内核冲突。这个文件也给出了为什么应该使用这个约定的原因。

现在已经不赞成使用的旧方法非常简单：选择一个 8 位幻数，比如“k”（十六进制为 0x6b），然后加上一个基数，就象这样：

```
#define SCULL_IOCTL1 0x6b01
#define SCULL_IOCTL2 0x6b02
/* .... */
```

如果应用程序和驱动程序都约定使用这些号码，那么只要在驱动程序里实现 `switch` 语句就可以了。但是，不应该再使用这种定义 `ioctl` 号码的传统 Unix 方法。这里介绍旧方法只是想给读者看看一个 `ioctl` 号码大致是个什么样子。

定义号码的新方法使用了 4 个位字段，它们有如下意义。下面所介绍的新符号都定义在 `<linux/ioctl.h>` 中。

类型 (type)

幻数。选择一个号码（记住先仔细阅读 `ioctl-number.txt`），并在整个驱动程序中使用这个号码。这个字段有 8 位宽（`_IOC_TYPEBITS`）。

号码 (number)

序（顺序的）数。它也是 8 位宽（`_IOC_NRBITS`）。

方向 (direction)

如果该命令有数据传输，它定义数据传输的方向。可以使用的值有，`_IOC_NONE`（没有数据传输）、`_IOC_READ`、`_IOC_WRITE` 和 `_IOC_READ | _IOC_WRITE`（双向传输数据）。数据传输是从应用程序的角度看的；`IOC_READ` 意味着从设备中读数据，所以驱动程序必须向用户空间写数据。注意，该字段是一个位掩码，因此可以用逻辑 `AND` 操作从中分解出 `_IOC_READ` 和 `_IOC_WRITE`。

尺寸 (size)

所涉及的用户数据大小。这个字段的宽度与体系结构有关，当前的范围从 8 位到 14 位不等。可以在宏 `_IOC_SIZEBITS` 中找到某种体系结构的具体数值。不过，如果你想保持你的驱动程序的可移植性，你最多只能使用 255，也就是 8 位。系统并不强制使用这个字段。如果需要更大尺度的数据传输，则可以忽略这个字段。稍后我们将介绍如何使用这个字段。

包含在 `<linux/ioctl.h>` 之中的头文件 `<asm/ioctl.h>` 定义了可以用于构造命令号的宏：`_IO(type,nr)`、`_IOR(type,nr,dataitem)`、`_IOW(type,nr,dataitem)` 和 `IOWR(type,nr,dataitem)`。每一个宏都对应一种可能的数据传输方向。`type` 和 `number` 字段通过参数传递，`size` 字段的值使用 `sizeof(dataitem)` 来获得。头文件还定义了解码宏：`_IOC_DIR(nr)`、`_IOC_TYPE(nr)`、`_IOC_NR(nr)` 和 `_IOC_SIZE(nr)`。我不打算详细介绍这些宏，头文件里的定义已经足够清楚了，本节稍后也会给出样例。

下面是 `scull` 中的一些 `ioctl` 命令定义。需要特别指出的是，这些命令设置和获取驱动程序的配置参数。

```
/* Use 'k' as magic number */
#define SCULL_IOC_MAGIC 'k'

#define SCULL_IOCRESET _IO(SCULL_IOC_MAGIC, 0)

/*
 * S means "Set" through a ptr
 * T means "Tell" directly with the argument value
 * G means "Get": reply by setting through a pointer
 * Q means "Query": response is on the return value
 * X means "eXchange": G and S atomically
 * H means "sHift": T and Q atomically
 */
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC, 1, scull_quantum)
#define SCULL_IOCQSET _IOW(SCULL_IOC_MAGIC, 2, scull_qset)
#define SCULL_IOTQUANTUM _IO(SCULL_IOC_MAGIC, 3)
#define SCULL_IOTQSET _IO(SCULL_IOC_MAGIC, 4)
#define SCULL_IOCQQUANTUM _IOR(SCULL_IOC_MAGIC, 5, scull_quantum)
#define SCULL_IOCQQSET _IOR(SCULL_IOC_MAGIC, 6, scull_qset)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC, 7)
#define SCULL_IOCQQSET _IO(SCULL_IOC_MAGIC, 8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, scull_quantum)
#define SCULL_IOCXQSET _IOWR(SCULL_IOC_MAGIC, 10, scull_qset)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC, 11)
#define SCULL_IOCHQSET _IO(SCULL_IOC_MAGIC, 12)
#define SCULL_IOCHARDRESET _IO(SCULL_IOC_MAGIC, 15) /* debugging tool */

#define SCULL_IOC_MAXNR 15
```

最后一条命令，即 `HARDRESET`，用来将模块使用计数器复位为 0，这样就可以在因计数器而发生错误时仍可以卸载模块。实际的源码还定义了从 `IOCHQSET` 到 `HARDRESET` 之间的所有命令，但这里没有列出。

尽管根据已有的约定，`ioctl` 应该使用指针完成数据交换，但我们仍然选择用两种方法实现整数参数传递——通过指针和显式数值。同样，这两种方法还用于返回整数：通过指针或设置返回值。如果返回值是正的，就表示工作正常。从任何一个系统调用返回时，正的返回值是受保护的（如我们在 `read` 和 `write` 所见到的），而负值则被认为是一个错误，并被用来设置用户空间中的 `errno` 变量。

“exchange”和“shift”操作对 `scull` 设备来说并不特别有用。我们实现“exchange”操作是为了示范在驱动程序中如何把分离的操作合并成一个原子操作，而“shift”操作则将“tell”和“query”操作合并在一起。某些时候需要“测试兼设置”这类操作是原子操作——特别是当应用程序需要加锁和解锁时*。

显式的命令序数没什么特别含义，仅仅用来区分命令。其实甚至可以在读命令和写命令中使用同一序数，因为实际 `ioctl` 号中的“方向”位肯定不一样，不过最好还是不要这样做。除了在声明中用到序数外，别的地方我们都不用它，这样就不必为它分配一个符号了。这也就是为什么前面给出的定义中直接使用了数字的原因。例子示范了一种使用命令号的方法，也可以随意使用其它不同的方法。

* 当一段程序代码总是被作为一条单一指令执行，而且执行期间不能被打断（如其它运行代码），就称这段代码是原子的。

当前，内核并未使用 `ioctl` 的 `cmd` 参数的值，以后也不太可能使用。这样，如果想偷懒，可以不使用上面那些复杂的声明，而直接显式地声明一组标量数字。由此带来的问题是，这样将无法从位字段中受益了。头文件 `<linux/kd.h>` 就是这种旧分格的例子，它使用了 16 位的标量数值定义 `ioctl` 命令，这并非由于懒惰，而是那时只有这种方法，现在修改它会引起一大堆兼容性问题。

5.1.2 返回值

`ioctl` 的实现通常就是一个基于命令号的 `switch` 语句。但是如果命令号不能匹配任何合法操作时，默认动作是什么？这问题颇有争议。有些内核函数会返回 `-EINVAL`（“Invalid argument”，非法参数），表示命令参数不是合法参数。然而 POSIX 标准规定，如果使用了不合适的 `ioctl` 命令参数，应该返回 `-ENOTTY`。在 `libc5` 及其以前的 C 库版本中，与这个值相对应的字符串一直都是 “Not a typewriter”，只到 `libc6` 才把消息换成了 “Inappropriate ioctl for device”，这看起来更贴切些。因为绝大多数较新的 Linux 系统都基于 `libc6`，所以我们还是坚持标准，返回 `-ENOTTY` 吧。尽管如此，对非法 `ioctl` 命令返回 `-EINVAL` 仍然是很普遍的做法。

5.1.3 预定义命令

尽管 `ioctl` 系统调用绝大部分用于操作设备，但还有一些命令是可以由内核识别的。要注意，当这些命令用于设备时，它们会在自己的文件操作被调用之前处理。所以，如果为自己的 `ioctl` 命令选用了与这些预定义命令相同的号码，就永远不会收到该命令的请求，而且由于 `ioctl` 号码冲突，应用程序的行为将无法预测。

预定义命令分为三组：

- 可用于任何文件（普通、设备、FIFO 和套接字）的
- 只用于普通文件的
- 用于特定文件系统类型的

最后一组命令只能在宿主(hosting)文件系统上执行（见 `chattr` 命令）。设备驱动程序开发人员只对第一组感兴趣，它们的幻数都是 “T”。分析其它组的工作留给读者做练习。`ext2_ioctl` 是其中最有意思的函数（尽管比你想象的容易的多），因为它实现了只追加（append-only）标志和不可变（immutable）标志。

下列 `ioctl` 命令对任何文件都是预定义的：

FIOCLEX

设置执行时关闭标志（File IOctl CLose on EXec）。设置了这个标志后，当调用进程执行一个新程序时文件描述符将被关闭。

FIONCLEX

清除执行时关闭标志。

FIOASYNC

设置或复位文件异步通知（稍后在本章“异步通知”一节中讨论）。注意直到 Linux 2.2.4 版本的

内核都不正确地使用了这个命令来修改 `O_SYNC` 标志。因为这两个动作都可以通过其它方法完成，所以实际上没有人使用 `FIOASYNC` 命令了，列在这只是为了完整。

FIONBIO

意指“File IOctl Non-Blocking I/O”，即“文件 ioctl 非阻塞型 I/O”（本章稍后在“阻塞型与非阻塞型操作”一节中介绍）。该调用修改 `filp->f_flags` 中的 `O_NONBLOCK` 标志。传递给系统调用的第 3 个参数指明了是设置还是清除该标志。本章稍后我们就可以看到它的作用。注意，`fcntl` 系统调用也可以使用 `F_SETFL` 命令修改这个标志。

上面的最后一项中我们引入了一个新的系统调用，即 `fcntl`，看起来很象 `ioctl`。实际上 `fcntl` 调用也要传递一个命令参数和一个附加的可选参数，在这点上它类似 `ioctl`。它和 `ioctl` 的不同主要是由于历史原因造成的：当 Unix 的开发人员面对控制 I/O 操作的问题时，他们认为文件和设备是不同的。那时，与 `ioctl` 实现相关的唯一设备就是终端，这也解释了为什么非法的 `ioctl` 命令的标准返回值是 `-ENOTTY`。现在情况虽然不同了，但是 `fcntl` 还为了向后兼容而保留下来。

5.1.4 使用 ioctl 参数

在分析 `scull` 驱动程序的 `ioctl` 代码之前我们还有一点要讲解，就是怎样使用那个附加参数。如果它是个整数，很简单，直接用就行了。如果是个指针，就要注意一些问题了。

当用一个指针指向用户空间时，必须确保指向的用户空间是合法的，而且对应的页面也已正确定位。如果内核代码企图越界访问一个地址，处理器就会产生一个异常。在包括 Linux 2.0.x 的以前所有内核版本代码中，这个异常都被转换为 `oops` 消息；2.1 及以后版本处理这个问题则温和许多。无论如何，驱动程序应该负责对每个用到的用户空间地址做适当的检查，如果是非法地址则应该返回一个错误。

内核 2.2.x 及以后版本的地址验证是通过函数 `access_ok` 实现的，它在 `<asm/uaccess.h>` 中声明：

```
int access_ok(int type, const void *addr, unsigned long size);
```

第一个参数应该是 `VERIFY_READ` 或 `VERIFY_WRITE`，取决于要执行的动作是读还是写用户空间内存区。`addr` 参数是一个用户空间地址，`size` 是字节数。例如，如果 `ioctl` 要从用户空间读一个整数，`size` 就是 `sizeof(int)`。如果在指定地址处既要读又要写，则应该用 `VERIFY_WRITE`，它是 `VERIFY_READ` 的超集。

与大多数函数不同，`access_ok` 返回一个布尔值：1 表示成功（访问成功），0 表示失败（访问不成功）。如果返回失败，驱动程序通常要返回 `-EFAULT` 给调用者。

关于 `access_ok` 有两点有趣之处需要注意。第一，它并没有完成验证内存的全部工作，而只检查了引用的内存是否位于进程有合适访问权限的区域内。特别是要确保访问地址没有指向内核空间内存区。第二，大多数驱动程序代码中都不需要真正调用 `access_ok`。因为后面要讲到的内存管理程序会处理它。尽管如此，我们还是示范一下它的使用，既为了理解其过程，也是为了向后兼容的原因。本章末尾还会深入讨论向后兼容问题。

scull 的源码在 `switch` 语句前，通过分析 `ioctl` 号码的位字段来检查参数：

```
int err = 0, tmp;
int ret = 0;

/*
 * extract the type and number bitfields, and don't decode
 * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
 */
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*
 * the direction is a bitmask, and VERIFY_WRITE catches R/W
 * transfers. `Type' is user oriented, while
 * access_ok is kernel oriented, so the concept of "read" and
 * "write" is reversed
 */
if (_IOC_DIR(cmd) & _IOC_READ)
    err = !access_ok(VERIFY_WRITE, (void *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
    err = !access_ok(VERIFY_READ, (void *)arg, _IOC_SIZE(cmd));
if (err) return -EFAULT;
```

在调用 `access_ok` 之后，驱动程序可以安全进行实际的数据传送了。除了 `copy_from_user` 和 `copy_to_user` 函数外，程序员还可以使用已经为最常用的数据尺寸（1、2 或 4 个字节，64 位平台上的 8 字节）优化过的一组函数。这些函数定义在 `<asm/uaccess.h>` 中，列在下面：

```
put_user(datum, ptr) "__put_user(datum, ptr)"
```

这些宏把 `datum` 写到用户空间。它们相对较快，当要传递单个数据时，应该用它而不是用 `copy_to_user`。由于宏展开时不做类型检查，所以可以传递给 `put_user` 任意类型的指针，只要是个用户空间地址就行。传递的数据尺寸依赖于 `ptr` 参数的类型，在编译时由特殊的 `gcc` 伪函数确定，这里就没有必要介绍了。总而言之，如果 `ptr` 是一个字符指针，就传递一个字节，2、4、8 字节的情况类似。

`put_user` 进行检查以确保进程可以写入指定的内存地址。成功返回 0，出错返回 `-EFAULT`。`__put_user` 做的检查少些（它不调用 `access_ok`），但对于某些错误地址仍会出现操作失败。因而，`__put_user` 应该在已经使用 `access_ok` 检验过内存区后再使用。

一般的情况是，实现一个 `read` 方法时，可以调用 `__put_user` 来节省几个时钟周期。或者在复制几项数据之前，调用一次 `access_ok`。

```
get_user(local, ptr) "__get_user(local, ptr)"
```

这些宏用于从用户空间接收一个数据。除了传输方向相反，它们与 `put_user` 和 `__put_user` 差不多。接收的数值保存在局部变量 `local` 中，返回值则指明了操作是否成功。同样，`__get_user` 应该在操作地址已被 `access_ok` 检验后使用。

如果试图使用上面列出的函数传递尺寸不符合任意一个特定值的数值，结果通常是编译器会给出一条奇怪的消息，比如“`conversion to non-scalar type requested`（需要转换为非标量类型）”。这种情况下，必须使用 `copy_to_user` 或者 `copy_from_user`。

5.1.5 权能与受限操作

对设备的访问由设备文件的许可控制，驱动程序通常不进行许可检查。不过也有这种情况，允许用户对设备读/写而其它的操作被禁止。例如，不是所有的磁带驱动器使用者都可以设置它的默认块大小，允许用户使用磁盘设备也并不意味着就可以格式化磁盘。在类似这种情况下，驱动程序必须进行附加检查以确认用户是否有权进行请求的操作。

根据 Unix 系统传统，特权操作仅限于超级用户帐号。这种特权要么全有，要么全没——超级用户几乎可以做任何事，所有其他用户则受到严格的限制。Linux 内核，如 2.2 版本，提供了一个更为灵活的系统，称为权能（capabilities）。基于权能的系统抛弃了那种非有即无的特权分配方式，而是把特权操作划分为独立的组。这样，某个特定的用户或程序可以被授权执行某一指定特权操作，同时又没有执行其它不相关操作的能力。权能在用户空间还很少使用，而内核代码中几乎已经全部使用这种方式了。

全部权能操作都可以在 `<linux/capability.h>` 中找到。对驱动程序开发者有意义的只是其中一部分，罗列如下：

`CAP_DAC_OVERRIDE`

越过文件或目录访问许可的能力

`CAP_NET_ADMIN`

执行网络管理任务的能力，包括那些能影响网络接口的任务。

`CAP_SYS_MODULE`

载入或卸除内核模块的能力

`CAP_SYS_RAWIO`

执行“裸”I/O 操作的能力。例如访问设备端口或直接与 USB 设备通信。

`CAP_SYS_ADMIN`

截获的能力，它提供了访问许多系统管理操作的途径。

`CAP_SYS_TTY_CONFIG`

执行 tty 配置任务的能力。

在执行一项特权操作之前，设备驱动程序应该检查调用进程是否有合适的权能，这是用 `capable` 函数来实现的，它定义在 `<sys/sched.h>` 中：

```
int capable(int capability);
```

在 `scull` 示例程序中，任何用户都允许查询 `quantum` 和 `quantum` 集的大小。但是只有授权用户可以更改这些值，因为不恰当的值会降低系统性能。`scull` 的 `ioctl` 实现了在需要时检查用户的特权级别：

```
if (!capable (CAP_SYS_ADMIN))
    return -EPERM;
```

因为缺少针对该任务的更好的权能定义，所以这里使用了 `CAP_SYS_ADMIN`。

5.1.6 ioctl 命令的实现

scull 的 `ioctl` 实现中只传递设备的可配置参数，很简单：

```
switch(cmd) {
#ifdef SCULL_DEBUG
    case SCULL_IOCTLCHARDRESET:
        /*
         * reset the counter to 1, to allow unloading in case
         * to allow unloading in case
         * because the invoking
         * process has the device open.
         */
        while (MOD_IN_USE)
            MOD_DEC_USE_COUNT;
        MOD_INC_USE_COUNT;
        /* don't break: fall through and reset things */
#endif /* SCULL_DEBUG */

    case SCULL_IOCTLCRESET:
        scull_quantum = SCULL_QUANTUM;
        scull_qset = SCULL_QSET;
        break;

    case SCULL_IOCTLCSQUANTUM: /* Set: arg points to the value */
        if (!capable (CAP_SYS_ADMIN))
            return -EPERM;
        ret = _ _get_user(scull_quantum, (int *)arg);
        break;

    case SCULL_IOCTLQQUANTUM: /* Tell: arg is the value */
        if (!capable (CAP_SYS_ADMIN))
            return -EPERM;
        scull_quantum = arg;
        break;

    case SCULL_IOCTLCGQUANTUM: /* Get: arg is pointer to result */
        ret = _ _put_user(scull_quantum, (int *)arg);
        break;

    case SCULL_IOCTLCQQUANTUM: /* Query: return it (it's positive) */
        return scull_quantum;

    case SCULL_IOCTLCXQUANTUM: /* eXchange: use arg as pointer */
        if (!capable (CAP_SYS_ADMIN))
            return -EPERM;
        tmp = scull_quantum;
        ret = _ _get_user(scull_quantum, (int *)arg);
        if (ret == 0)
            ret = _ _put_user(tmp, (int *)arg);
        break;

    case SCULL_IOCTLCHQUANTUM: /* sHift: like Tell + Query */
        if (!capable (CAP_SYS_ADMIN))
            return -EPERM;
        tmp = scull_quantum;
        scull_quantum = arg;
        return tmp;

    default: /* redundant, as cmd was checked against MAXNR */
        return -ENOTTY;
}
```

```

}
return ret;

```

scull 中还包括 6 个操作 `scull_qset` 的入口，它们和 `scull_quantum` 的相应入口是一样的，这里不再赘述。

从调用方的观点（例如从用户空间）看，传送和接收参数的 6 种途径如下：

```

int quantum;

ioctl(fd, SCULL_IOCSQUANTUM, &quantum);
ioctl(fd, SCULL_IOCTLQUANTUM, quantum);

ioctl(fd, SCULL_IOCQQUANTUM, &quantum);
quantum = ioctl(fd, SCULL_IOCQQUANTUM);

ioctl(fd, SCULL_IOCXQUANTUM, &quantum);
quantum = ioctl(fd, SCULL_IOCHQUANTUM, quantum);

```

当然，正常的驱动程序不会在一个地方就实现这么多调用方式。在这里只是为了示范各种不同的方法。不过，通常情况下数据交换形式应该保持一致，要么都用指针（比较普遍），要么都用数值（用的较少），尽量避免混用。

5.1.7 非 ioctl 的设备控制

有时通过向设备写入控制序列可以更好地控制设备。在控制台驱动程序中就使用了这一技术，它称为“转义序列（escape sequence）”，用于控制移动光标，改变默认颜色，或其它配置任务。用这种方法实现设备控制的好处是用户仅通过写数据就可以控制设备，无需使用（有时还得编写）配置设备的程序。

例如，程序 `setterm` 通过打印转义序列来配置控制台（或某个终端）。这种方法的优点是可以对设备进行远程控制。控制程序可以运行在非被控设备所在的计算机上，然后用一个简单数据流重定向就可以完成配置工作。这项技术已经在终端上使用，但它还可以更通用些。

通过打印序列进行控制的缺点是，它给设备增加了策略限制。例如，只有确认控制序列不会出现写到设备的正常数据中时，才能使用这种技术。而只有部分的终端设备能满足这个要求。尽管文本显示只需显示 ASCII 字符，但有时写数据流中也会出现控制字符，从而影响控制台的设置。例如，对一个二进制文件使用 `grep`，找出的行可能什么都会包含，结果是经常造成控制台的字体错误^{*}。

通过写入来控制的方式，非常适合于那种不传送数据而只响应命令的设备，如机器人。

例如，笔者编写过一个驱动程序，该驱动程序控制相机在两个轴上移动。在这个驱动程序里，“设备”只是一对旧步进马达，不能读写。“发送数据流”的概念对步进马达来说没什么意义。这种情况下，驱动程序将所写的数据解释为 ASCII 命令，并把请求转换为脉冲序列来操纵步进马达。这种思路，与给调制解调器发送 AT 指令以设置通讯的方法基本类似，主要区别就是连接调制解调器的串口还要发送真正的数据。直接设备控制的优点是使用 `cat` 就可以移动相机，而不必编写和

^{*} CTRL-N 设置替换字体，它由图形字符组成，对你的 shell 输入来说是很不友好的；如果碰到这种问题，回显一个 CTRL-O 字符可恢复主字体。

编译用于实现 `ioctl` 调用的代码。

当编写这种“面向命令的”驱动时，没什么必要实现 `ioctl` 方法。在解释器中加一条指令，实现和使用都更简单些。

尽管如此，有时可能需要做相反的事情：不是用“写入”解释器来避免使用 `ioctl`，而是只使用 `ioctl`，完全不使用“写入”。同时，驱动程序附带了一个特定的命令行工具，专门负责把命令送给驱动程序。这种方法把内核空间的复杂性转移到了用户空间，这样处理起来可能会容易些，并且有助于减少驱动程序的尺寸，然而，用户却无法再使用简单的命令如 `cat` 或 `echo` 来操作驱动程序。

5.2 阻塞型 I/O

`read` 的一个问题是当尚无数据可读，而又没有到达文件尾时该怎么办。

默认的回答是“进入睡眠并等待数据”。这一节将介绍如何使进程睡眠，如何唤醒，以及应用程序如何在不盲目执行 `read` 调用或阻塞的情况下查看是否有数据。同一概念也适用于 `write`。

和前面一样，我们在示范实际代码前先解释一些概念。

5.2.1 睡眠和唤醒

当进程等待一个事件（如数据到达或其他进程终止）时，它应该进入睡眠。睡眠使该进程暂时挂起，腾出处理器给其他进程使用。在将来的某个时间，等待的事件发生了，进程被唤醒继续执行。这一节讨论内核 2.4 中进程睡眠和唤醒的机制。以前版本稍后在本章“向后兼容性”一节中讲解。

Linux 中有几种处理睡眠和唤醒的方法，每种分别适合于不同的需求。不过所有方法都要处理同一个基本数据类型：等待队列（`wait_queue_head_t`）。正确地说，一个“等待队列”其实是由正等待事件发生的进程组成的一个队列。等待队列的声明和初始化部分如下：

```
wait_queue_head_t my_queue;
init_waitqueue_head (&my_queue);
```

如果一个等待队列被声明为静态的（比如不是某个过程的自动变量，或某个动态分配的数据结构的一部分），它就可以在编译时初始化：

```
DECLARE_WAIT_QUEUE_HEAD (my_queue);
```

忘记初始化等待队列是一个常见的错误（特别是以前的内核还不要求做这种初始化），如果没有初始化，则可能导致无法预见的错误。

一旦声明了等待队列，完成了初始化，进程就可以使用它进入睡眠。基于睡眠的深度不同，可调用 `sleep_on` 的不同变体函数来完成睡眠。

```
sleep_on(wait_queue_head_t *queue);
```

把进程放入这个队列睡眠。`sleep_on` 有个缺点，就是不能被中断。其结果是，如果进程等待的事

件永远不发生，进程就醒不来了（也杀不掉）。

```
interruptible_sleep_on(wait_queue_head_t *queue);
```

除了睡眠可以被信号中断外，这个变体做的事和 `sleep_on` 类似。在 `wait_event_interruptible`（稍后介绍）出现以前，它也是设备驱动程序开发者一直使用的函数。

```
sleep_on_timeout(wait_queue_head_t *queue, long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t *queue, long timeout);
```

这两个函数和前两个类似，不同之处是到了指定时间后就不再睡眠。时间是用“jiffies”指定的，第 6 章会讲到。

```
void wait_event(wait_queue_head_t queue, int condition);
int wait_event_interruptible(wait_queue_head_t queue, int condition);
```

这两个宏是睡眠的首选方法。它们把等待事件和测试事件是否发生合并了起来，避免了竞态的发生。它们会一直睡眠到 C 布尔表达式 `condition` 为真时为止。这两个宏扩展为 `while` 循环，`condition` 在循环期间不断重新求值——这区别于单个函数调用或简单的宏的行为，它们只会在调用时求一次值。第二个宏实现为一个表达式，如果成功就得 0；如果循环被信号打断就得出 `-ERESTARTSYS`。

值得再重复一遍的是，驱动程序开发人员应该基本只用这些函数/宏中的“可中断的”形式。不可中断的那些只在很少情况下使用，在这些情况下信号不能处理，例如，等待从交换空间取得一个数据页面。大多数驱动程序都不会碰到这类特殊情况。

当然，睡眠只是问题的一半，进程总得在未来某个时刻被唤醒。如果一个驱动程序睡眠了，那么程序中通常还有处理唤醒的部分，一旦等待的事件发生它们就会起作用。典型情况下驱动程序会在新数据到达时的中断处理程序中唤醒睡眠者。当然其他实现方法也是有可能的。

既然有不止一种睡眠方法，当然也就有不止一种唤醒方法。内核提供的用来唤醒进程的高级函数有：

```
wake_up(wait_queue_head_t *queue);
```

这个函数将唤醒在这个事件等待队列中的所有进程。

```
wake_up_interruptible(wait_queue_head_t *queue);
```

`wake_up_interruptible` 只唤醒那些在“可中断睡眠”状态的进程。那些使用不可中断函数或宏而睡眠在等待队列上的进程则继续睡眠。

```
wake_up_sync(wait_queue_head_t *queue);
wake_up_interruptible_sync(wait_queue_head_t *queue);
```

通常，`wake_up` 调用会立即引发一次重新调度，这意味着在 `wake_up` 之前运行的其它进程都会返回。这种“同步”的变体则不同，它让已醒来的进程继续运行，但不重新调度 CPU。在已知当前进程就要进入睡眠因而要引发一次重新调度时，为了避免重调度就可以使用它。注意醒来的进程可能立即会在一个不同的处理器上运行，所以不要指望这些函数提供了互斥性。

如果驱动程序使用的是 `interruptible_sleep_on`，那么用 `wake_up` 或是 `wake_up_interruptible` 几乎没什么区别。不过使用后者是普遍约定，因为在两个调用间保持了一致性。

作为使用等待队列的一个例子，想象一下当进程读设备时进入睡眠而在其他人写设备时被唤醒的情景。下列代码完成这件事：

```

DECLARE_WAIT_QUEUE_HEAD(wq);

ssize_t sleepy_read (struct file *filp, char *buf, size_t count,
    loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) going to sleep\n",
        current->pid, current->comm);
    interruptible_sleep_on(&wq);
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
    return 0; /* EOF */
}

ssize_t sleepy_write (struct file *filp, const char *buf, size_t count,
    loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
        current->pid, current->comm);
    wake_up_interruptible(&wq);
    return count; /* succeed, to avoid retrial */
}

```

这个设备的代码在我们的示例程序中称为 **sleepy**，和其它的一样，可以用 **cat** 或输入/输出重定向来测试它。

关于等待队列，一个要牢记的重点是，被唤醒并不能保证等待的事件已经发生了，进程可能因其它原因被唤醒，大部分的原因是收到了一个信号。从睡眠态返回后，任何睡眠的代码都应该在测试 **condition** 的循环中这么做，就象本章稍后要介绍的“实现示例：**suclpipe**”中讲到的那样。

5.2.2 等待队列的深入分析

大多数驱动程序开发者完成任务时，知道前面所讨论的这些内容已经足够了。也可能还有些人想深入了解一下，这一节可以满足这些人的好奇心，其他人可以直接跳到下一节，不会错过什么重要的东西。

wait_queue_head_t 类型是一个相当简单的结构，在 `<linux/wait.h>` 中定义。它只包括一个锁变量和一个正睡眠进程的链表。链表中的各个数据成员项的类型是 **wait_queue_t**，链表就是在 `<linux/list.h>` 定义的通用链表，在第 10 章的“链表”一节还会讨论。通常，**wait_queue_t** 结构都从堆栈中分配，这通过如 **interruptible_sleep_on** 之类的函数来完成。这些结构之所以在堆栈中，是因为在相关函数中它们都定义为自动变量。一般情况下程序员不用处理它们。

但是在一些高级应用中，可能会要求直接处理 **wait_queue_t** 变量。为此我们浏览一下 **interruptible_sleep_on** 这类函数的内部实现过程。下面是一个简化了的 **interruptible_sleep_on** 的实现，它使进程进入睡眠。

```

void simplified_sleep_on(wait_queue_head_t *queue)
{
    wait_queue_t wait;

    init_waitqueue_entry(&wait, current);
    current->state = TASK_INTERRUPTIBLE;

    add_wait_queue(queue, &wait);
    schedule();
}

```

```
remove_wait_queue (queue, &wait);
}
```

这段代码创建和初始化一个 `wait_queue_t` 变量（即 `wait`，在堆栈中分配）。任务状态被置为 `TASK_INTERRUPTIBLE`，表示处于可中断睡眠中。这个等待队列成员随即被加入队列（由参数 `wait_queue_head_t *queue` 给出）。接着调用 `schedule`，使处理器可以被其它进程使用。只有其它进程唤醒该睡眠进程后 `schedule` 才返回，并设置其进程状态为 `TASK_RUNNING`。接着，这个队列成员从等待队列中删除，睡眠结束。

图 5-1 展示了等待队列数据结构的内部组成，以及如何被进程使用。

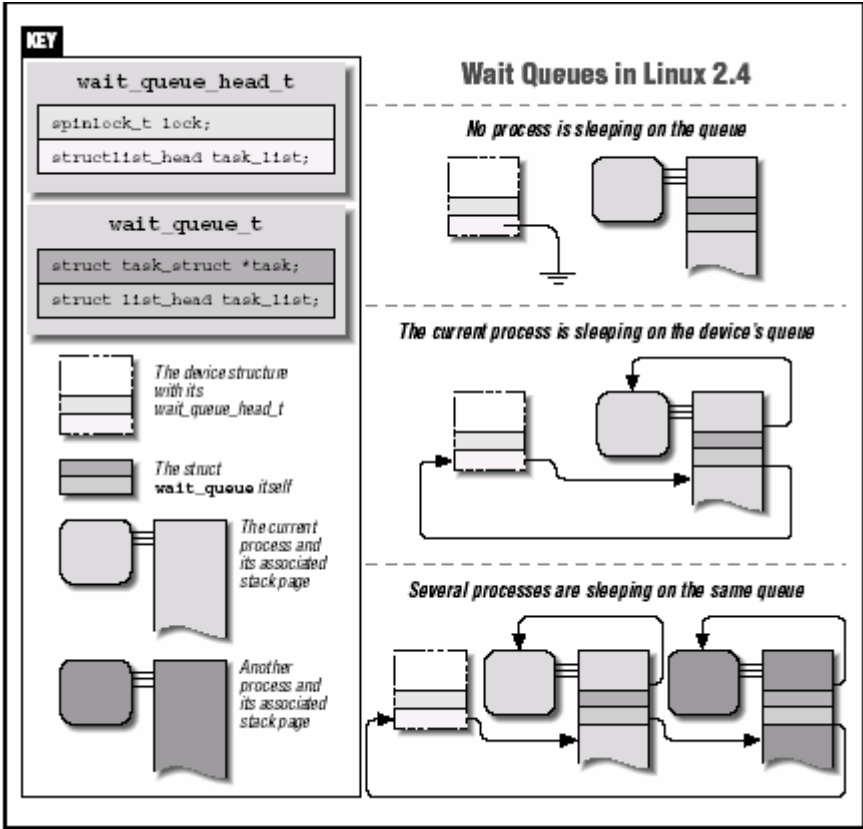


图 5-1: Linux 2.4 的等待队列

快速浏览一下内核代码，会发现非常多的程序都是用类似前面这个例子的方法来“手工”处理睡眠的。其中大多数实现可以追溯到 2.2.3 之前的内核，那时 `wait_event` 还没有引入。前面已经说过，现在处理等待事件的睡眠的首选方法是 `wait_event`，因为使用 `interruptible_sleep_on` 常会引起恼人的竞态。至于为什么会这样，将在第 9 章“无竞争地进入睡眠”中详细阐述。简单地说，就是在驱动程序将要睡眠和实际调用 `interruptible_sleep_on` 之间的时间里，情况可能已经发生了变化。

显式调用调度器的另一个原因是为了执行“排外”的等待。有可能发生下面这种情况，几个进程都在等待同一事件，当 `wake_up` 被调用时，这些进程都试图重新执行。假如该事件表示到达了一个原子性数据，那么只有一个进程可以读取它，所有其它进程仅仅是醒来后发现没有数据，然后接着睡眠。

这种情况有时被称为“哄赶牧群”。在要求高性能的环境，该问题会浪费大量资源。创建许多的进程运行，什么事也不做，还产生了很多上下文切换和处理器负荷，结果什么用也没有。如果这些进程只是简单地继续睡眠下去，情况就好多了。

为此，在 2.3 内核开发系列中增加了“排他睡眠”的概念。如果进程用排外模式睡眠，内核一次只会唤醒其中的一个，这样，可在某些情况下提升系统性能。

实现一个排他睡眠的代码和普通睡眠很类似：

```
void simplified_sleep_exclusive(wait_queue_head_t *queue)
{
    wait_queue_t wait;

    init_waitqueue_entry(&wait, current);
    current->state = TASK_INTERRUPTIBLE | TASK_EXCLUSIVE;

    add_wait_queue_exclusive(queue, &wait);
    schedule();
    remove_wait_queue(queue, &wait);
}
```

在任务状态中增加 **TASK_EXCLUSIVE** 标志表明进程正处于排他睡眠中。调用 **add_wait_queue_exclusive** 也是必须的。该函数把进程加在等待队列的尾部，所有其它进程的后面。目的是让非排他睡眠的进程排在前面，以使它们总是先被唤醒。一旦“唤醒”到了第一个处于排他睡眠中的进程，它就可以停止睡眠了。

细心的读者可能已经注意到了“排外”地操作等待队列和调度器的另一个原因。尽管 **sleep_on** 这样的函数只会在一个等待队列中阻塞进程，但是直接操作队列就意味着允许在多个队列中同时睡眠。当然大多数驱动程序并不需要在多个队列中睡眠，如果有例外，那就需要使用与前面演示的类似代码。

希望进一步了解等待队列的读者可以阅读 `<linux/sched.h>` 和 `kernel/sched.c`。

5.2.3 编写可重入代码

进程睡眠以后，驱动程序仍然是活动的，而且可以由另一个进程调用。考虑一个控制台驱动程序例子。当一个进程在 **tty1** 上等待键盘输入，用户切换到 **tty2**，启动一个新的 **shell**。现在两个 **shell** 都在控制台驱动程序中等待键盘输入，尽管它们睡眠在不同的等待队列上：一个在 **tty1** 的队列上，另一个在 **tty2** 的队列上。两个进程都阻塞在 **interruptible_sleep_on** 函数中，而驱动程序此时仍然可以接收和响应其它 **tty** 的请求。

当然，如果在一个 **SMP** 系统，即便不睡眠，对驱动程序的多并发调用也会发生。

通过编写“可重入代码”可以轻松地处理这种情况。可重入代码是指这样的代码：其中不使用任何全局变量来记录状态信息，因而可以处理交织的调用，而不会造成混淆。如果所有状态信息都是进程特有的，就不会发生冲突。

如果状态信息需要记录，它既可以保存在驱动程序函数的局部变量中（每个进程在内核空间中都有不同的堆栈来保存局部变量），也可以保存在访问文件用的 `filp` 的私有数据结构（`private_data`）中。由于同一个 `filp` 有时会被两个进程共享（通常是父进程和子进程），所以建议最好使用局部变量。

如果需要保存很多状态数据，可以只在局部变量中保存一个指针，并用 `kmalloc` 获取实际存储空间。这种情况下别忘了用 `kfree` 释放空间，因为在内核空间工作时可没有“进程终止时会释放所有资源”的说法。在局部变量中存放很多数据并不是一个好方法，因为为堆栈分配的一页内存可能放不下这些数据。

在下列两种情况需要使用可重入函数。第一，调用了 `schedule`，这可能是通过调用了 `sleep_on` 或者 `wake_up` 引起。第二，和用户空间交换了数据。因为访问用户空间可能产生页面失效，内核处理缺页时进程会进入睡眠状态。每个调用了这类可重入函数的函数也必须是可重入的。举个例子，如果 `sample_read` 调用了 `sample_getdata`，后者可能随即被阻塞，由于调用 `sample_read` 的进程睡眠后别的进程有可能再调用 `sample_read`，所以 `sample_read` 和 `sample_gendata` 都必须是可重入的。

当然，最后要记住的是，进程睡眠时系统中可能会发生任何事情。驱动程序应该仔细检查睡眠期间运行环境可能发生的所有变化。

5.2.4 阻塞和非阻塞型操作

在分析功能完整的 `read` 和 `write` 方法之前我们还要看看另一个问题，就是 `filp->f_flags` 中的 `O_NONBLOCK` 标志的作用。这个标志在 `<linux/fcntl.h>` 中定义，这个头文件自动包含在 `<linux/fs.h>` 中。

这个标志的名字取自“非阻塞打开（`open-nonblock`）”，因为它可以在打开时指定（而且本来也只能在那时指定）。浏览一下源代码，会发现一些对 `O_NDELAY` 标志的引用，这是 `O_NONBLOCK` 的另一个名字，是为保持和 `System V` 代码的兼容性而设计的。这个标志默认是的，因为等待数据的进程一般都要睡眠。在阻塞型操作（这是默认的）的情况下，应该实现下列动作以保持和标准语义一致：

如果一个进程调用了 `read` 但是还没有数据可读，进程必须阻塞。数据到达时进程被唤醒，并把数据返回给调用者。即使数据数目少于 `count` 参数指定的数目也是如此。

如果进程调用了 `write` 但缓冲区没有空间，进程必须阻塞。而且必须睡眠在与读进程不同的等待队列上。当向硬件设备写入一些数据，腾出了部分输出缓冲区后，进程被唤醒，`write` 调用成功。即使缓冲区中可能没有所要求的 `count` 字节的空间而只写入了部分数据，也是如此。

上面的描述假设输入和输出缓冲区都存在，实际上它们也确实存在于绝大多数设备中。输入缓冲区用于当数据已到达而又无人读取时，把数据暂存起来避免丢失，相反，如果调用 `write` 时系统不能接收数据，它们会保留在用户空间缓冲区而不会丢失。除此以外，输出缓冲区几乎总是可以提高硬件的性能。

在驱动程序中实现输出缓冲区可以提高性能，这得益于减少了上下文切换和用户级/内核级转换的

次数。假设一个慢速设备没有输出缓冲区，那么每次系统调用只能接收一个或几个字符，然后进程在 `write` 睡眠，另一个进程开始运行（这里有一次上下文切换），当前一个进程被唤醒后，它重新开始运行（引起另一次上下文切换），`write` 返回（内核/用户转换），接着进程重复系统调用写入更多数据（用户/内核转换），接着调用又阻塞，再次进行以上的循环。如果输出缓冲区足够大，那么 `write` 调用首次操作就成功了——缓存的数据可以在以后的中断时间送给设备——而不必返回用户空间为第二次或第三次的 `write` 调用进行控制。显然，输出缓冲区多大才合适是与设备相关的。

在 `scull` 中没有使用输入缓冲区，因为调用 `read` 时，数据已经就绪了。类似地，输出缓冲区也没有，因为数据只是简单地复制到与设备对应的内存区。其实该设备本身就是一个缓冲区，不必实现另外的缓冲区了。在第 9 章的“中断驱动的 I/O”一节我们将介绍缓冲区的使用。

如果指定了 `O_NONBLOCK` 标志，`read` 和 `write` 的行为会有所不同。如果在数据没有就绪时调用 `read` 或是在缓冲区没有空间时调用 `write`，该调用简单地返回 `-EAGAIN`。

读者可能已经想到，非阻塞型操作会立即返回，使得应用程序可以查询数据。在处理非阻塞型文件时，应用程序调用 `stdio` 函数必须非常小心，因为很容易把一个非阻塞返回误认为是 `EOF`。所以必须始终检查 `errno`。

自然，`O_NONBLOCK` 在 `open` 方法中也是有用的。它用于在 `open` 调用可能会阻塞很长时间的场合。例如打开一个还没有进程向其中写入的 `FIFO` 或是访问一个被锁住的磁盘文件。通常情况下打开一个设备不是成功就是失败，不必等待外部事件。但是有时候打开设备需要很长时间的初始化，这时就可以选择在 `open` 方法中支持 `O_NONBLOCK` 标志，如果该标志被置位，在设备开始初始化后会立刻返回一个 `-EAGAIN`（“try it again, 再试一次”）。驱动程序中也可以实现阻塞型 `open` 以支持和文件锁方式类似的访问策略。在本章的“替代 `EBUSY` 的阻塞型 `open`”一节中会看到这样一个实现。

有些驱动程序还为 `O_NONBLOCK` 实现了特殊的语义。例如，在磁带还没有插入时打开一个磁带设备通常会阻塞，如果磁带驱动程序是用 `O_NONBLOCK` 打开的，不管磁带在不在，`open` 都会立即返回成功。

只有 `read`，`write` 和 `open` 文件操作受非阻塞标志的影响。

5.2.5 一个样例实现：sculpipe

`/dev/sculpipe` 设备（默认有 4 个设备）是 `scull` 模块的一部分，用来示范如何实现阻塞型 I/O。

在驱动程序内部，阻塞在 `read` 调用的进程在数据到达时被唤醒；通常硬件会发出一个中断来通知这个事件，作为中断处理的一部分，驱动程序会唤醒等待进程。`scull` 驱动程序的工作方法则不同，它不需要任何特殊的硬件或是中断处理程序就可以运行。我们选择使用另一个进程来产生数据并唤醒读进程；类似地，读进程用来唤醒睡眠的写进程。除了名称外，这种实现类似于一个 `FIFO`（或命名管道）文件系统节点。

该设备驱动程序使用了一个包含两个等待队列和一个缓冲区的设备结构。缓冲区大小可以用通常的方式配置（编译、加载和运行时）。

```
typedef struct Scull_Pipe {
    wait_queue_head_t inq, outq; /* read and write queues */
    char *buffer, *end; /* begin of buf, end of buf */
    int buffersize; /* used in pointer arithmetic */
    char *rp, *wp; /* where to read, where to write */
    int nreaders, nwriters; /* number of openings for r/w */
    struct fasync_struct *asynch_queue; /* asynchronous readers */
    struct semaphore sem; /* mutual exclusion semaphore */
    devfs_handle_t handle; /* only used if devfs is there */
} Scull_Pipe;
```

read 实现了阻塞型和非阻塞型输入，如下所示（函数的第一行可能令人有些迷惑，稍后在“定位设备”一节中解释）：

```
ssize_t scull_p_read (struct file *filp, char *buf, size_t count,
                     loff_t *f_pos)
{
    Scull_Pipe *dev = filp->private_data;

    if (f_pos != &filp->f_pos) return -ESPIPE;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    while (dev->rp == dev->wp) { /* nothing to read */
        up(&dev->sem); /* release the lock */
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("\'%s\' reading: going to sleep\n", current->comm);
        if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
            return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
        /* otherwise loop, but first reacquire the lock */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    /* ok, data is there, return something */
    if (dev->wp > dev->rp)
        count = min(count, dev->wp - dev->rp);
    else /* the write pointer has wrapped, return data up to dev->end */
        count = min(count, dev->end - dev->rp);
    if (copy_to_user(buf, dev->rp, count)) {
        up (&dev->sem);
        return -EFAULT;
    }
    dev->rp += count;
    if (dev->rp == dev->end)
        dev->rp = dev->buffer; /* wrapped */
    up (&dev->sem);

    /* finally, awaken any writers and return */
    wake_up_interruptible(&dev->outq);
    PDEBUG("\'%s\' did read %li bytes\n", current->comm, (long)count);
    return count;
}
```

可以看到代码中保留了一些 **PDEBUG** 语句。编译该驱动程序时可以打开消息以便跟踪不同进程间的交互。

需要再次注意的是代码中用于保护临界区的信号量的使用。**scull** 的代码必须小心避免在持有一个信号量的同时进入睡眠——那样写进程将永远无法写入数据，从而会陷入死锁。如果需要的话，代码中使用 **wait_event_interruptible** 来等待数据，尽管等待结束后必须再次检查数据是否已就绪。因为别的进程有可能在本进程醒来和取回信号量的间隙中把数据取走。

值得再重复一次的是，一个进程在直接或间接调用 `schedule` 以及和用户空间交换数据的时候都可能进入睡眠。在后一种情况下进程还可能因用户数据结构不在内存中而引起睡眠。如果 `scull` 在内核和用户空间复制数据的时候睡眠了，它会在睡眠中持有设备信号量。在这种情况下保持信号量已被证明是正确的，因为它不会造成系统死锁，而且，重要的是，驱动程序睡眠时设备内存区不会发生变化。

接着 `interruptible_sleep_on` 的 `if` 语句用于信号处理。这条语句确保对信号进行正确的预定的响应，该信号可能是用来唤醒进程的（因为进程处于可中断睡眠中）。如果一个信号到达而且没有被进程阻塞，正确的动作是让内核的上层去处理事件。为此，驱动程序返回给调用者 `-ERESTARTSYS`，这个值由虚拟文件系统层（VFS）内部使用，它或者重启系统调用，或者给用户空间返回 `-EINTR`。我们将在所有的 `read` 和 `write` 实现中使用同样的语句进行信号处理。由于 `signal_pending` 是在内核 2.1.57 版本引入的，为保持代码的可移植性，`sysdep.h` 定义了它，使它在早期内核中也能使用。

`write` 的实现和 `read` 非常类似（它的第一行同样留待后面解释）。它唯一特别的地方就是从不会完全填满缓冲区，总是留下至少一个字节的空间。因此，当缓冲区空的时候 `wp` 和 `rp` 是相等的；有数据的时候它们总是不相等。

```
static inline int spacefree(Scull_Pipe *dev)
{
    if (dev->rp == dev->wp)
        return dev->buffer_size - 1;
    return ((dev->rp + dev->buffer_size - dev->wp) % dev->buffer_size) - 1;
}

ssize_t scull_p_write(struct file *filp, const char *buf, size_t count,
                    loff_t *f_pos)
{
    Scull_Pipe *dev = filp->private_data;

    if (f_pos != &filp->f_pos) return -ESPIPE;

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

    /* Make sure there's space to write */
    while (spacefree(dev) == 0) { /* full */
        up(&dev->sem);
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("%s\n" writing: going to sleep\n", current->comm);
        if (wait_event_interruptible(dev->outq, spacefree(dev) > 0))
            return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    /* ok, space is there, accept something */
    count = min(count, spacefree(dev));
    if (dev->wp >= dev->rp)
        count = min(count, dev->end - dev->wp); /* up to end-of-buffer */
    else /* the write pointer has wrapped, fill up to rp-1 */
        count = min(count, dev->rp - dev->wp - 1);
    PDEBUG("Going to accept %li bytes to %p from %p\n",
        (long)count, dev->wp, buf);
    if (copy_from_user(dev->wp, buf, count)) {
        up(&dev->sem);
        return -EFAULT;
    }
}
```

```

    }
    dev->wp += count;
    if (dev->wp == dev->end)
        dev->wp = dev->buffer; /* wrapped */
    up(&dev->sem);

    /* finally, awaken any reader */
    wake_up_interruptible(&dev->inq); /* blocked in read() and select() */

    /* and signal asynchronous readers, explained later in Chapter 5 */
    if (dev->async_queue)
        kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
    PDEBUG("\'%s\' did write %li bytes\n", current->comm, (long)count);
    return count;
}

```

这个设备，依照构想，没有实现阻塞型的 `open`，所以比一个实际的 FIFO 要简单。如果想看看实际 FIFO 的代码，可以看内核源码中的 `fs/pipe.c`。

要测试 `scullpipe` 设备的阻塞型操作，可以在它上面运行些程序，并使用输入/输出重定向，就象通常那样。测试非阻塞活动要麻烦些，因为一般的程序不会做非阻塞型操作。`misc-progs` 源码目录中包含一个简单程序 `nbtest`，用来测试非阻塞型操作，代码罗列如下。它所做的全部事情就是用非阻塞型 I/O 把输入复制到输出并在期间稍做延迟。延迟时间由命令行传递，默认是 1 秒。

```

int main(int argc, char **argv)
{
    int delay=1, n, m=0;

    if (argc>1) delay=atoi(argv[1]);
    fcntl(0, F_SETFL, fcntl(0,F_GETFL) | O_NONBLOCK); /* stdin */
    fcntl(1, F_SETFL, fcntl(1,F_GETFL) | O_NONBLOCK); /* stdout */

    while (1) {
        n=read(0, buffer, 4096);
        if (n>=0)
            m=write(1, buffer, n);
        if ((n<0 || m<0) && (errno != EAGAIN))
            break;
        sleep(delay);
    }
    perror( n<0 ? "stdin" : "stdout");
    exit(1);
}

```

5.3 poll 和 select

使用非阻塞型 I/O 的应用程序也经常使用 `poll` 和 `select` 系统调用。`poll` 和 `select` 的功能本质上是一样的：都允许进程决定是否可以对一个或多个打开的文件做非阻塞的读或写。因此它们常常用于那些要使用多个输入或输出流而又不阻塞其中任何一个的应用程序中。同一功能之所以要由两个分离的函数分别提供，是因为它们几乎是同时在两个不同的 Unix 团体中分别实现的：`select` 由 BSD Unix 实现，`poll` 由 System V 实现。

对其中任一个系统调用的支持都要求设备驱动程序提供对应函数的支持，在内核版本 2.0 中设备驱动程序实现方法是基于 `select` 模型的（用户程序不能使用 `poll`）；从 2.1.23 版本开始两种方法都提供了，并且设备驱动程序方法改为基于新引入的 `poll` 系统调用，`poll` 提供了比 `select` 更精确的控制。

`poll` 方法可用来实现 `poll` 和 `select` 系统调用。它的原型如下：

```
unsigned int (*poll) (struct file *, poll_table *);
```

该驱动方法在用户空间程序执行 `poll` 或 `select` 系统调用时被调用，包括传递一个与设备相关的文件描述符。设备方法分为两步处理：

- 在一个或多个指明了 `poll` 状态变化的等待队列上调用 `poll_wait`。
- 返回一个用来描述操作是否可以立即无阻塞执行的位掩码。

这些操作通常简单明了，各个驱动程序的这些操作看起来也非常类似。然而，实际上它们依赖于只有驱动程序才能提供的信息，因此必须为每个驱动程序分别实现对应的操作。

传递给 `poll` 方法的第二个参数，`poll_table` 结构，用于在内核中实现 `poll` 和 `select` 系统调用。它在 `<linux/poll.h>` 中声明，驱动程序代码必须包含这个头文件。驱动程序编写者不需要了解该结构的细节，会用就可以了。它被传递给驱动程序方法，以使每个可以唤醒进程和修改 `poll` 操作状态的事件队列都可以被加入 `poll_table` 结构中，这通过调用函数 `poll_wait` 完成：

```
void poll_wait (struct file *, wait_queue_head_t *, poll_table *);
```

`poll` 方法执行的第二项任务是返回描述哪个操作可以立即执行的位掩码，这也很直观。例如，如果设备已有数据就绪，一个 `read` 操作可以立刻完成而不用睡眠，那么 `poll` 方法应该指出这种情况。几个标志（在 `<linux/poll.h>` 定义）用来指明可能的操作：

POLLIN

如果设备可以无阻塞读，就置该位。

POLLRDNORM

如果“通常”的数据已经就绪，可以读取，就置该位。一个可读设备返回 `(POLLIN | POLLRDNORM)`。

POLLRDBAND

这一位指示可以从设备读取 **out-of-band**（频带之外）的数据。它当前只在 Linux 内核的一个地方（DECnet 代码）中使用，通常不用于设备驱动程序。

POLLPRI

可以无阻塞地读取高优先级（即 **out-of-band**）的数据。置该位会导致 `select` 报告文件发生一个异常，这是由于 `select` 把“**out-of-band**”的数据作为异常对待。

POLLHUP

当读设备的进程到了文件尾，驱动程序必须设置 `POLLHUP`（挂起）位。依照 `select` 的功能描述，调用 `select` 的进程会被告知设备是可读的。

POLLERR

设备发生了错误。如果调用 `poll`，会报告设备既可读也可以写，因为读写都会无阻塞地返回一个错误码。

POLLOUT

如果设备可以无阻塞写就在返回值中置该位。

POLLWRNORM

该位和 **POLLOUT** 的意义一样，有时就是同一个数字。一个可写的设备返回 (**POLLOUT** | **POLLWRNORM**)。

POLLWRBAND

就象 **POLLRDBAND**，这一位表示具有非零优先级的数据可以写入设备。只有数据报 (datagram) 的 poll 实现中使用了这一位，因为数据报可以传输 out-of-band 数据。

POLLRDBAND 和 **POLLWRBAND** 只在套接字相关的文件描述符中才是有意义的。设备驱动程序通常用不到这两个标志。

描述 poll 很费事，实际的使用则相对简单多了。考虑一下 sculpipe 的 poll 实现：

```
unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
    Scull_Pipe *dev = filp->private_data;
    unsigned int mask = 0;

    /*
     * The buffer is circular; it is considered full
     * if "wp" is right behind "rp". "left" is 0 if the
     * buffer is empty, and it is "1" if it is completely full.
     */
    int left = (dev->rp + dev->buffersize - dev->wp) % dev->buffersize;

    poll_wait(filp, &dev->inq, wait);
    poll_wait(filp, &dev->outq, wait);
    if (dev->rp != dev->wp) mask |= POLLIN | POLLRDNORM; /* readable */
    if (left != 1) mask |= POLLOUT | POLLWRNORM; /* writable */

    return mask;
}
```

这段代码简单地增加两个 sculpipe 等待队列到 poll_table 中，然后根据数据可读或可写设置合适的位掩码。

展示的 poll 代码中没有处理文件尾的部分。在设备已到文件尾的时候 poll 方法应该返回 POLLHUP。如果调用者使用 select 系统调用，则会报告文件是可读的。两种情况下应用程序都能知道它一定可以执行无阻塞的 read，而 read 方法将会返回 0 来指示已到了文件尾。

在真正的 FIFO 实现中，读进程在所有的写进程都关闭了文件后就能看到文件尾。然而在 sculpipe 中读进程却永远看不到文件结束符。之所以有这种不同，是因为 FIFO 一般作为两个进程的通讯通道使用，而 sculpipe 就是一个垃圾桶——只要还有一个读进程存在，任何进程都可以往里面扔数据。此外，重新实现内核中已有的东西也没什么意义。

象 FIFO 那样实现文件尾意味着要在 read 和 write 中检查 dev->nwrites，如果没有进程为写而打开设备，就报告文件结束。不过遗憾的是，如果读进程在写进程之前打开了 sculpipe 设备，它马上就会看到文件尾，没有机会等待数据到达。修正这个问题的最好方法是实现阻塞的 open 操

作。这个任务作为练习留给读者。

5.3.1 与 read 和 write 的交互

`poll` 和 `select` 调用的目的是确定接下来的 I/O 操作是否会阻塞。从这个方面来说，它们是 `read` 和 `write` 的补充。`poll` 和 `select` 的更重要的用途是它们可以使应用程序同时等待多个数据流，尽管在 `scull` 的例子中没有使用这个特点。

为了使应用程序正常工作，正确实现这三个调用是非常重要的。所以尽管下面的规则多多少少已经提过了，我们还是在这里总结一下。

从设备读取数据

如果输入缓冲区有数据，即使就绪的数据比程序请求的少，并且驱动程序保证剩下的数据马上就能到达，`read` 调用仍然应该以难以察觉的延迟立即返回。如果为了某种方便（比如我们的 `scull`），`read` 甚至可以一直返回比请求数目少的数据，当然，前提是至少也得返回一个字节。

如果输入缓冲区中没有数据，默认情况下 `read` 必须阻塞等待，直到至少一个字节到达。另一方面，如果设置了 `O_NONBLOCK` 标志，`read` 立即返回，返回值是 `-EAGAIN`（有些 System V 的老版本返回 0）。在这种情况下 `poll` 必须报告设备不可读，直到至少有一个字节到达。一旦缓冲区中有了数据，我们就回到了前一种情况。

如果已经到了文件尾，`read` 应该立即返回 0，无论 `O_NONBLOCK` 是否设置。此时 `poll` 应该报告 `POLLHUP`。

向设备写数据

如果输出缓冲区中有空间，`write` 应该无延迟地立即返回。它可以接收比请求少的数据，但至少要接收一个字节。这种情况下，`poll` 报告设备可写。

如果输出缓冲区已满，默认情况下 `write` 被阻塞直到有空间释放。如果设置了 `O_NONBLOCK` 标志，`write` 立即返回，返回值是 `-EAGAIN`（旧的 System V 系统返回 0）。这时 `poll` 应该报告文件不可写。另一方面，如果设备不能再接受任何数据，`write` 返回 `-ENOSPC`（“No space left on device，设备无可可用空间”），而不管 `-ENOSPC` 标志是否设置。

永远不要让 `write` 调用在返回前等待数据传输结束，即使 `O_NONBLOCK` 标志被清除。这是因为许多应用程序用 `select` 来检查 `write` 是否会阻塞。如果报告设备可以写，调用就不能被阻塞。如果使用设备的程序需要保证输出缓冲区中的数据确实已经传送出去，驱动程序必须提供一个 `fsync` 方法。例如，可移动设备就应该有一个 `fsync` 的入口点。

尽管这些已经是一个很好的通用规则集合，还是应该承认每个设备都有独特之处，所以有时候规则需要稍稍改变一下。例如，面向记录的设备（如磁带机）不能执行部分写（必须以记录为单位）。

刷新待处理输出

我们已经看到了为什么 `write` 方法不能满足所有数据输出的需求，`fsync` 函数可以弥补这一空缺，

通过同名系统调用来调用它。该方法的原型是

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

如果应用程序需要确保数据已经被送到设备上，就必须实现 `fsync` 方法。一个 `fsync` 调用只有在设备已被完全刷新（输出缓冲区全空）时才会返回，即使这要花一些时间。`O_NONBLOCK` 标志是否设置对此没有影响。参数 `datasync` 在 2.4 内核才出现，用于区分 `fsync` 和 `fdatasync` 这两个系统调用。这里它只和文件系统的代码有关，驱动程序可以忽略它。

`fsync` 方法没有什么特别的地方。这个调用对时间没有严格要求，所以每个驱动程序都可以按照作者的喜好实现它。大多数时候，字符设备驱动程序在它们的 `fops` 只有一个 `NULL` 指针。而块设备总是用通用的 `block_fsync` 来实现这个方法，`block_fsync` 会依次刷新设备的所有缓冲块，一直到 I/O 结束。

5.3.2 底层的数据结构

`poll` 和 `select` 系统调用的实现是相当简单的。当用户应用程序调用了其中一个函数，内核会调用由该系统调用引用的全部文件的 `poll` 方法，并向它们传递同一个 `poll_table`。这个数据结构是由 `poll_table_entry` 结构组成的数组，每个结构都是为特定的 `poll` 或 `select` 调用而分配的。每个 `poll_table_entry` 包括一个指向被打开设备的 `struct file` 类型的指针，一个 `wait_queue_head_t` 指针以及一个 `wait_queue_t` 入口。当一个驱动程序调用 `poll_wait` 时，这些入口中的一个就会填入由驱动程序提供的信息，然后该入口项被放到驱动程序队列中。`wait_queue_head_t` 指针用来跟踪当前 `poll_table_entry` 注册的等待队列，在等待队列被唤醒之前，`free_wait` 用这个指针把入口项从队列中删除。

如果轮询（`poll`）时没有一个驱动指明可以进行非阻塞 I/O，这个 `poll` 调用就进入睡眠，直到睡在其上的某个（或多个）等待队列唤醒它为止。

`poll` 实现中最有趣的地方是可以用空指针 `NULL` 作为 `poll_table` 参数来调用文件操作。有两个原因可以导致产生这种情况。如果一个正调用 `poll` 的应用程序提供的 `timeout` 值为 0（表明不做等待），那就不需要加入等待队列，系统不用处理它。任何被轮询的驱动程序指明可以进行 I/O 之后，`poll_table` 指针也被立即置为 `NULL`。因为内核知道那时不会发生等待，所以不再建立一个等待队列的列表。

当 `poll` 调用完成后，`poll_table` 结构被重新分配，所有先前加入轮询表（如果有的话）的等待队列成员项在轮询表和它们的等待队列中删除。

实际的情况比这里描述的更复杂些。因为轮询表不是一个简单的数组而是一个或多个页面的集合，每一个页面容纳了一个数组。这么复杂的实现是为了使包括在 `poll` 或者 `select` 系统调用中的文件描述符的最大数目不会被限制得过低（现在是由页面大小决定）。

在图 5-2 中显示了与轮询相关的数据结构；该图是实际数据结构的简化表示，其中忽略了轮询表的多页特性，也省略了每个 `poll_table_entry` 中的文件指针。推荐对实际实现感兴趣的读者阅读 `<linux/poll.h>` 和 `fs/select.c` 的相关代码。

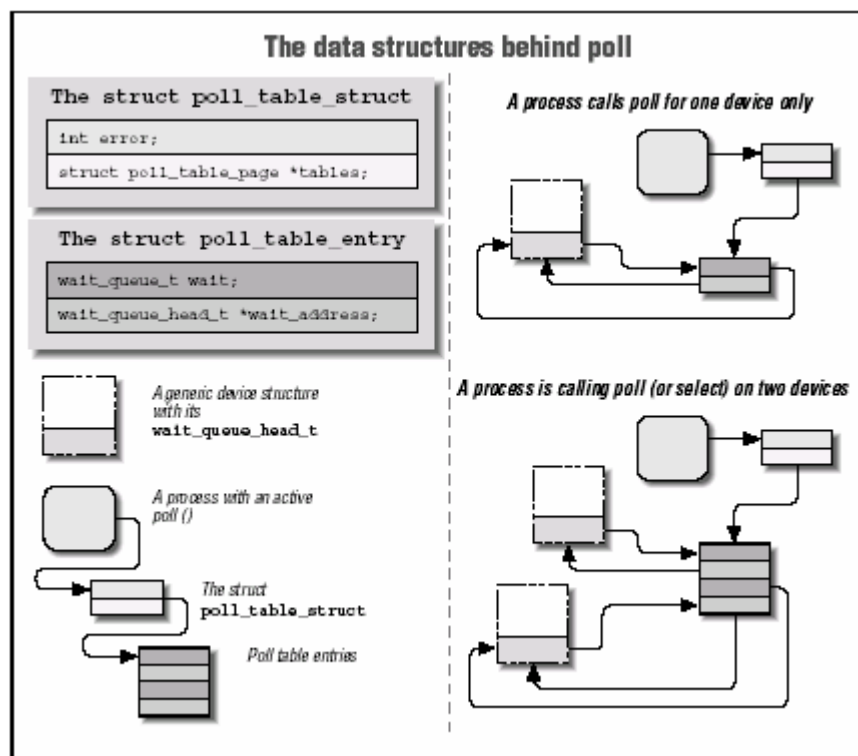


图 5-2: poll 的数据结构

5.4 异步通知

尽管大多数时候阻塞型和非阻塞型操作的组合以及 **select** 方法可以有效地查询设备，但某些时候用这种技术处理就效率不高了。

例如，我们想象一下，一个进程在低优先级执行长的循环计算，但又需要尽可能快地处理输入数据。如果输入是通过键盘，可以向进程发送信号（使用“INTR”字符，一般是 **CTRL-C**），但这种信号是 **tty** 抽象层的一部分，通常的字符驱动程序没有这样一种软件层。我们需要的异步通知与此不同。而且，任何输入数据都应该产生中断，而不仅仅是 **CTRL-C**。

为了打开文件的异步通知机制，用户程序必须执行两个步骤。首先，它们指定一个进程作为文件的“属主（owner）”。当进程使用 **fcntl** 系统调用执行 **F_SETOWN** 命令时，属主进程的进程 ID 号就被保存在 **filp->f_owner** 中。这一步是必需的，是为了让内核知道该通知谁。然后，为了真正打开异步通知机制，用户程序还必须在设备中设置 **FASYNC** 标志，这是通过 **fcntl** 命令 **F_SETFL** 完成的。

执行完这两步之后，输入文件就可以在新数据到达时请求发送一个 **SIGIO** 信号。该信号被送到存放在 **filp->f_owner** 的进程（如果是负值就是进程组）。

例如，用户程序中的如下代码段打开了 **stdin** 输入文件到当前进程的异步通知机制：

```
signal(SIGIO, &input_handler); /* dummy sample; sigaction() is better */
fcntl(STDIN_FILENO, F_SETOWN, getpid());
oflags = fcntl(STDIN_FILENO, F_GETFL);
```

```
fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

示例源码中名为 `asynctest` 的程序就是这样一个读 `stdin` 的例子。它可以用来测试 `sculpipe` 的异步功能。该程序类似 `cat`，但不会在文件尾终止。它只响应输入，没有输入时就没有响应。

要注意的是，不是所有的设备都支持异步通知，我们也可以选择不提供异步通知功能。应用程序通常假设只有套接字和终端才有异步通知能力。象管道和 `FIFO` 就不支持，至少现在的内核是这样。鼠标提供了异步通知，因为某些程序需要鼠标能象终端那样发送 `SIGIO` 信号。

还有一个问题。当进程收到了 `SIGIO`，它不知道是哪个输入文件有了新的输入。如果有多于一个文件可以异步通知处理输入的进程，程序仍然必须借助 `poll` 或 `select` 来确定输入的来源。

5.4.1 从驱动程序的角度看

对我们来讲，一个更重要的话题是驱动程序怎样实现异步信号。下面列出的是从内核角度看的详细操作过程。

- `F_SETOWN` 被调用时对 `filp->f_owner` 赋值，此外什么也不做。
- 在执行 `F_SETFL` 启用 `FASYNC` 时，调用驱动程序的 `fasync` 方法。只要 `filp->f_flags` 中的 `FASYNC` 标志发生了变化，就会调用该方法，以便把这个变化通知驱动程序，使其能正确响应。文件打开时，`FASYNC` 标志默认是清除的。我们一会再来看看这个驱动程序方法的标准实现。
- 数据到达时，所有注册为异步通知的进程都会被发送一个 `SIGIO` 信号。

第一步的实现很简单，在驱动程序部分没什么可做的。其它步骤则要涉及维护一个动态数据结构，以跟踪不同的异步读进程，这种进程可能会有好几个。不过，这个动态数据结构并不依赖于特定的设备，内核已经提供了一套合适的通用实现方法，无需为每个驱动程序重写同一代码了。

Linux 的这种通用方法基于一个数据结构和两个函数（它们要在前面提到的第二步和第三步中调用）。含有相关声明的头文件是 `<linux/fs.h>`，这对我们来说并不新鲜，那个数据结构称为 `struct fasync_struct`。前面在处理等待队列的时候，就需要把一个该类型的指针插入设备相关的数据结构中去。其实在“样例实现：`sculpipe`”一节中就已经看过这个成员了。

驱动程序要调用的两个函数的原型如下：

```
int fasync_helper(int fd, struct file *filp,
                  int mode, struct fasync_struct **fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

当一个已打开的文件的 `FASYNC` 标志被修改时，调用 `fasync_helper` 从相关的进程列表中增加或删除文件。它的所有参数除了最后一个外都被提供给 `fasync` 方法，而且可以直接传递。`kill_fasync` 在数据到达时通知所有相关进程。它的参数包括要发送的信号（通常是 `SIGIO`）和约束方式，后者几乎总是 `POLL_IN`（除了在网络编程中用来发送“紧急”或 `out-of-band` 的数据）。

`sculpipe` 中是这样实现 `fasync` 方法的：

```
int scull_p_fasync(fasync_file fd, struct file *filp, int mode)
{
    Scull_Pipe *dev = filp->private_data;

    return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

显然所有工作都由 `fasync_helper` 完成。不过，如果没有驱动程序中提供的方法，它是不可能实现这一功能的。因为函数 `helper` 需要访问正确的 `struct fasync_struct *` 类型（这里是 `&dev->async_queue`）的指针，只有驱动程序才能提供这些信息。

接着，当数据到达时，必须执行下面的语句来通知异步读进程。由于供给 `scullpipe` 的读进程的新数据是由一个进程调用 `write` 产生的，所以这条语句是在 `scullpipe` 的 `write` 方法中：

```
if (dev->async_queue)
    kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

看起来差不多了，不过还漏了一件事。当文件关闭时必须调用 `fasync` 方法，以便从活动的异步读进程列表中删除该文件。尽管这个调用只在 `filp->f_flags` 设置了 `FASYNC` 标志时才是必须的，但不管什么情况，调用它不会有什么坏处，并且这也是普遍的实现方法。例如下面的代码是 `scullpipe` 的 `close` 方法中的一段：

```
/* remove this filp from the asynchronously notified filp's */
scull_p_fasync(-1, filp, 0);
```

异步通知使用的数据结构和 `struct wait_queue` 使用的几乎是相同的，因为两种情况都涉及等待事件。不同之处在于前者用 `struct file` 替换了 `struct task_struct`。队列中的 `file` 结构用来获取 `f_owner`，以便给进程发送信号。

5.5 定位设备

本章的难点已经都讨论过了，下面我们快速地浏览一下 `llseek` 方法，它很有用，并且易于实现。

5.5.1 llseek 实现

`llseek` 方法实现了 `lseek` 和 `llseek` 系统调用。前面已经提过如果设备操作中没有 `llseek` 方法，内核默认的实现是通过修改 `filp->f_pos` 从文件头或当前位置开始进行定位，`filp->f_pos` 是文件的当前读/写位置。请注意为了使 `lseek` 系统调用能正确工作，`read` 和 `write` 方法中必须通过更新它们收到的偏移量参数（该参数通常是指向 `filp->f_ops` 的指针）来配合。

如果 `seek` 操作对应于设备的一个物理操作，或者能够实现基于文件尾的定位（默认方法中没有实现）的话，可能就需要提供自己的 `llseek` 方法。在 `scull` 的驱动程序中可以看到一个简单的例子：

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    Scull_Dev *dev = filp->private_data;
    loff_t newpos;

    switch(whence) {
```

```

case 0: /* SEEK_SET */
    newpos = off;
    break;

case 1: /* SEEK_CUR */
    newpos = filp->f_pos + off;
    break;

case 2: /* SEEK_END */
    newpos = dev->size + off;
    break;

default: /* can't happen */
    return -EINVAL;
}
if (newpos < 0) return -EINVAL;
filp->f_pos = newpos;
return newpos;
}

```

这里唯一的设备相关的操作就是从设备中获得文件长度。在 `scull` 中 `read` 和 `write` 方法需要相互配合，就象第 3 章“读取和写入”中介绍的那样。

上面的实现对 `scull` 是有意义的，因为它处理一个明确定义的数据区。然而大多数设备只提供了数据流（就象串口和键盘），而不是数据区，定位这些设备是没有意义的。这种情况下，不能简单地不声明 `llseek` 操作。相反，应该使用如下代码：

```

loff_t scull_p_llseek(struct file *filp, loff_t off, int whence)
{
    return -ESPIPE; /* unseekable */
}

```

该函数来自 `scullpipe` 设备，它是不可定位的。错误码应被解释为“`Illegal seek`，非法定位”，尽管从字面上看它的意思是“`is a pipe`，是个管道”。因为当前位置对不可定位设备是没有意义的，所以 `read` 和 `write` 在数据传输时都不需要更新它。

要注意的一点是，因为 `pread` 和 `pwrite` 已经添加到被支持的系统调用集中，所以用户空间程序用来定位文件的方法就不仅仅是 `lseek` 设备方法了。不可定位设备的正确实现应该允许通常的 `read` 和 `write` 调用而禁止使用 `pread` 和 `pwrite`。这由下列代码完成——在 `scullpipe` 的 `read` 和 `write` 方法中的第一行——当时介绍那些方法时没有解释它们的作用：

```

if (f_pos != &filp->f_pos) return -ESPIPE;

```

5.6 设备文件的访问控制

提供访问控制对于设备节点的可靠性有时是至关重要的。不仅不允许未授权的用户使用设备（这可以通过设置文件系统的许可位实现），而且某些情况下一次只能允许一个授权用户打开设备。

使用终端的问题与此类似。每当一个用户登录系统，`login` 进程就修改设备节点的属主，以防止其他用户干扰或获取这个终端的数据流。然而如果仅仅为了保证独占设备，而在每次打开它时都用特权程序修改设备属主，是不现实的。

到现在为止，我们还没有看到能超越文件系统许可位而实现任意访问控制的代码。如果 `open` 系统调用将请求转给驱动程序，`open` 就成功了。现在来介绍一些实现某些附加检查的技术。

本节的每个设备都和空 `scull` 设备（它实现了一个持久的内存区）具有相同的功能，但有不同的访问控制，这是在 `open` 和 `close` 操作中实现的。

5.6.1 独享设备

最粗暴的访问控制方法是一次只允许一个进程打开设备（独享）。这种技术最好避免使用，因为它制约了用户的灵活性。用户可能会希望在同一设备上运行不同的进程，一个用来读状态信息，其它的写数据。有时候，用户通过一个 `shell` 脚本运行几个可以同时访问设备的简单程序就完成很多工作。换句话说，独享是策略而非机制，它的实现方法只考虑用户“做什么”。

一次只允许一个进程打开设备有很多令人不快的特性，不过这也是设备驱动程序中最容易实现的访问控制方法。下面给出了代码。这些代码摘自 `scullsingle` 设备。

`open` 调用基于一个全局的整数类型标志来拒绝访问：

```
int scull_s_open(struct inode *inode, struct file *filp)
{
    Scull_Dev *dev = &scull_s_device; /* device information */
    int num = NUM(inode->i_rdev);

    if (!filp->private_data && num > 0)
        return -ENODEV; /* not devfs: allow 1 device only */
    spin_lock(&scull_s_lock);
    if (scull_s_count) {
        spin_unlock(&scull_s_lock);
        return -EBUSY; /* already open */
    }
    scull_s_count++;
    spin_unlock(&scull_s_lock);
    /* then, everything else is copied from the bare scull device */

    if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
        scull_trim(dev);
    if (!filp->private_data)
        filp->private_data = dev;
    MOD_INC_USE_COUNT;
    return 0; /* success */
}
```

另一方面，`close` 调用则标记设备不再忙。

```
int scull_s_release(struct inode *inode, struct file *filp)
{
    scull_s_count--; /* release the device */
    MOD_DEC_USE_COUNT;
    return 0;
}
```

通常，建议把打开标志 `scull_s_count`（这里还使用了 `spinlock` 和 `scull_s_lock`，下一小节解释它们的作用）放在设备结构中，因为从概念上来讲它就属于设备。不过 `scull` 驱动程序使用了单独的变量保存标志和锁，这是为了保持与空 `scull` 设备使用同样的设备结构和方法，减少代码重复。

5.6.2 关于竞态的问题

考虑一下在刚才的 `scull_s_count` 变量上做个测试。做两个分离的操作：（1）测试变量值，如果不是 0 就拒绝打开；（2）变量自增以标识设备占用。在一个单处理器系统上，测试可以安全通过，因为在两个操作之间不会有其他进程运行。

但在 SMP 系统就有问题了。如果分别在两个处理器上有两个进程同时试图打开一个设备，就有可能同时测试 `scull_s_count` 的值，而此时两个进程都还没有修改它。这种情况下就可以发现，独享设备的语义并没有实现，这还是最好的结果；最糟糕的情况下，意外的并发访问会造成数据结构损坏，接着是系统崩溃。

换句话说，这里有另一个竞态存在。它可以用与我们已经在第 3 章介绍过的几乎完全相同的方法来解决。那些竞态是由于访问了可能共享的数据结构的状态变量而引起的，可以通过使用信号量来解决。然而，通常情况下使用信号量的代价是高昂的，因为它们会使调用的进程睡眠。为了保护状态变量的一次快速查询就使用它们，未免有点牛刀杀鸡了。

所以，`scullsingle` 使用了不同的锁机制，称为自旋锁（`spinlock`）。自旋锁永远不会使进程睡眠。相反，如果锁不可用，自旋锁原语只是简单地不断重试（正如“`spin`”的意思）直到锁被释放。因此自旋锁引起的开销很少，但是如果某个进程老是上着锁不释放，它们也有可能引起处理器被锁住很长时间。对比信号量，自旋锁的另一个优点是，在单处理器系统上编译代码时自旋锁的实现部分都是空的（因为那些在 SMP 系统上的竞态不会发生）。信号量则是一个更通用的资源，在单处理器和 SMP 系统都有意义，所以在单处理器的情况下它们不会被优化掉。

对于较小的临界区，自旋锁是理想的解决机制。进程应该尽可能减少持有自旋锁的时间，而且绝不能在持锁时睡眠。因此，主 `scull` 驱动程序由于和用户空间交换数据可能会引起睡眠，并不适合用自旋锁的解决方案。但是自旋锁在控制访问 `scull_s_single` 时就工作得很好（不过它们还不是最佳方案，这在第 9 章会看到）。

`spinlock` 声明为 `spinlock_t` 类型，它在头文件 `<linux/spinlock.h>` 中定义。使用之前，必须先初始化：

```
spin_lock_init(&spinlock_t *lock);
```

进入临界区的进程用 `spin_lock` 来获得锁：

```
spin_lock(&spinlock_t *lock);
```

调用 `spin_unlock` 后，锁被释放：

```
spin_unlock(&spinlock_t *lock);
```

自旋锁的使用还可以比这更复杂，那些我们在第 9 章才会去详细地介绍。这里展示的简化例子可以满足现在的需求，`scull` 的所有带有访问控制的变种都使用这种简化的自旋锁。

细心的读者可能已经注意到，`scull_s_open` 在增加 `scull_s_count` 标志之前要先获得 `scull_s_lock` 锁，而 `scull_s_close` 则没有这个步骤。这其实是安全的，因为如果 `scull_s_count` 不

是 0 的话，不会有别的代码段去修改它。所以这里的特定情况下没有冲突。

5.6.3 限制每次只由一个用户访问

建立了系统范围内单一的锁之后的步骤是，让一个用户可以在多个进程中打开一个设备，但是一次只允许一个用户打开设备。这种方法使得测试设备比较简单，因为用户可以从几个进程读和写设备，前提是由用户负责在多进程访问中维护数据的完整性。这通过在 `open` 方法中加入检查来完成，这种检查在正常的许可检查之后进行，提供了比文件属主和属组许可位更严格的访问控制。这种策略和终端使用的访问策略相同，不过它无需借助于一个外部的特权程序。

这些访问策略比实现独享策略要有一些技巧。此时需要两个数据项：一个打开记数和设备属主的 UID。同样的，这些数据项最好是保存在设备结构内部；不过，我们的例子用了全局变量，原因在前面的 `scullsingle` 已解释过了。设备名字是 `sculluid`。

`open` 调用在第一次打开时授权，但它记录下设备的属主。这意味着一个用户可以多次打开设备，允许几个互相协作的进程并发地在设备上操作。同时，其他用户不能打开设备，这就避免了外部干扰。因为这个函数版本和上一个基本相同，所以只列出相关部分：

```
spin_lock(&scull_u_lock);
if (scull_u_count &&
    (scull_u_owner != current->uid) && /* allow user */
    (scull_u_owner != current->euid) && /* allow whoever did su */
    !capable(CAP_DAC_OVERRIDE)) { /* still allow root */
    spin_unlock(&scull_u_lock);
    return -EBUSY; /* -EPERM would confuse the user */
}

if (scull_u_count == 0)
    scull_u_owner = current->uid; /* grab it */

scull_u_count++;
spin_unlock(&scull_u_lock);
```

虽然代码完成了许可检查，我们还是选择返回 `-EBUSY` 而不是 `-EPERM`，以便给访问被拒绝的用户正确的提示信息。返回“许可拒绝 (Permission denied)”通常是检查 `/dev` 文件的状态和属主的结果，而“设备忙 (Device busy)”提示用户设备已经被进程使用。

代码中还检查了试图打开设备的进程是否有越过文件访问许可的能力；如果是这样，允许它进行打开操作，即使这个进程不是设备属主。在这种情况下 `CAP_DAC_OVERRIDE` 正适合于完成这项任务。

`close` 的代码这里没有列出，它做的仅仅是把使用记数减 1 而已。

5.6.4 替代 `EBUSY` 的阻塞型 `open`

当设备不能访问时返回一个错误，通常这是最合理的方式，但有些情况下可能需要让进程等待设备。

例如，如果一个用来发送定时报告（用 `crontab`）的数据通道同时也根据人们的需要而临时使用，那么这个定时报告最好稍微延迟一会儿，而不是因为通道忙就返回失败。

这是在设计设备驱动程序时程序员必须作出的选择，根据所解决的问题不同答案也不一样。

读者可能已经想到，代替 **EBUSY** 的另一个方法是实现阻塞型 **open**。

scullwuid 设备和 **sculluid** 的不同是，**open** 时等待设备而不是返回 **-EBUSY**。它和 **sculluid** 只在 **open** 操作的下列部分不同：

```
spin_lock(&scull_w_lock);
while (scull_w_count &&
      (scull_w_owner != current->uid) && /* allow user */
      (scull_w_owner != current->euid) && /* allow whoever did su */
      !capable(CAP_DAC_OVERRIDE)) {
    spin_unlock(&scull_w_lock);
    if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
    interruptible_sleep_on(&scull_w_wait);
    if (signal_pending(current)) /* a signal arrived */
        return -ERESTARTSYS; /* tell the fs layer to handle it */
    /* else, loop */
    spin_lock(&scull_w_lock);
}
if (scull_w_count == 0)
    scull_w_owner = current->uid; /* grab it */
scull_w_count++;
spin_unlock(&scull_w_lock);
```

这里的实现又是基于等待队列。创建等待队列是为了维护一个因等待事件而睡眠的进程的列表，所以在这里使用非常合适。

接下来，**release** 方法唤醒所有等待的进程：

```
int scull_w_release(struct inode *inode, struct file *filp)
{
    scull_w_count--;
    if (scull_w_count == 0)
        wake_up_interruptible(&scull_w_wait); /* awaken other uid's */
    MOD_DEC_USE_COUNT;
    return 0;
}
```

阻塞型 **open** 实现中的问题是，对于交互用户来说它是很令人不愉快的，用户可能会在等待中猜测设备出了什么问题。交互用户通常使用象 **cp** 和 **tar** 这样的预先编译好的命令，它们都没有在 **open** 调用中加入 **O_NONBLOCK** 选项。隔壁一些正使用磁带机做备份的人可能更愿意得到一条清晰的消息“设备或资源忙”，而不是在 **tar** 扫描磁盘的时候坐在一边猜想为什么今天硬盘这么安静。

这类问题（对同一设备的不同的、不兼容的策略）最好通过为每一种访问策略实现一个设备节点的方法来解决。这种实现的一个例子是 **Linux** 的磁带设备驱动程序，它为同一个设备提供了多个设备文件。不同的设备文件会使设备以不同的方式工作，例如是否以压缩方式记录，在设备关闭时是否自动回卷磁带，等等。

5.6.5 在打开时复制设备

另一个实现访问控制的方法是，在进程打开设备时创建设备的不同私有副本。

显然这种方法只有在设备没有绑定到某个硬件对象时才能实现。**scull** 就是这样一个“软设备”的例子。**/dev/tty** 内部也使用了类似的技术，以提供给它的进程一个不同于 **/dev** 入口点表现出的“情景”。如果复制的设备是由软件驱动程序创建的，我们称它们为“虚拟设备”——就象所有的虚拟终端都使用同一个物理终端设备一样。

虽然这种访问控制并不常见，但它的实现，展示了内核代码可以轻松地改变应用程序看到的外部环境（如计算机）。实际上，这个主题相当怪异，所以如果读者不感兴趣，可以直接跳到下一节。

scull 包中的 **/dev/scullpriv** 设备节点实现了虚拟设备。在 **scullpriv** 的实现中，使用当前进程控制终端的次设备号作为访问虚拟设备的键值。不过这个来源可以很容易地修改成任意整数值作为键值，每个不同的值导致不同的策略。例如，使用 **uid** 会导致给每个用户复制不同的虚拟设备，使用 **pid** 则会导致给每个访问进程复制一个新设备。

使用控制终端意味着可以通过输入/输出重定向来简化测试设备：运行在某一个虚拟终端的所有命令共享设备，这个设备与在另一个终端上运行的命令所看到的设备互相独立。

open 方法的代码如下。它必须找到正确的虚拟终端，也许还需要创建一个。函数的最后一部分没有列出，因为它是从空 **scull** 中复制过来的，我们已经看过了。

```
/* The clone-specific data structure includes a key field */
struct scull_listitem {
    Scull_Dev device;
    int key;
    struct scull_listitem *next;
};

/* The list of devices, and a lock to protect it */
struct scull_listitem *scull_c_head;
spinlock_t scull_c_lock;

/* Look for a device or create one if missing */
static Scull_Dev *scull_c_lookfor_device(int key)
{
    struct scull_listitem *lptr, *prev = NULL;

    for (lptr = scull_c_head; lptr && (lptr->key != key); lptr = lptr->next)
        prev=lptr;
    if (lptr) return &(lptr->device);

    /* not found */
    lptr = kmalloc(sizeof(struct scull_listitem), GFP_ATOMIC);
    if (!lptr) return NULL;

    /* initialize the device */
    memset(lptr, 0, sizeof(struct scull_listitem));
    lptr->key = key;
    scull_trim(&(lptr->device)); /* initialize it */
    sema_init(&(lptr->device.sem), 1);

    /* place it in the list */
    if (prev) prev->next = lptr;
    else     scull_c_head = lptr;

    return &(lptr->device);
}
```

```

int scull_c_open(struct inode *inode, struct file *filp)
{
    Scull_Dev *dev;
    int key, num = NUM(inode->i_rdev);

    if (!filp->private_data && num > 0)
        return -ENODEV; /* not devfs: allow 1 device only */

    if (!current->tty) {
        PDEBUG("Process \"%s\" has no ctl tty\n", current->comm);
        return -EINVAL;
    }
    key = MINOR(current->tty->device);

    /* look for a scullc device in the list */
    spin_lock(&scull_c_lock);
    dev = scull_c_lookfor_device(key);
    spin_unlock(&scull_c_lock);

    if (!dev) return -ENOMEM;

    /* then, everything else is copied from the bare scull device */

```

release 方法没有做什么特殊处理。它在最后一次关闭时释放设备，但是为了简化测试，这里没有维护一个打开设备的计数器。如果设备在最后一次关闭时释放了，在写设备后将不能再从中读出同样的数据，除非有一个后台进程保持把它打开。驱动程序样例使用了比较简单的方法来保存数据，所以在下一次打开设备时还能找到那些数据。设备在 **scull_cleanup** 被调用时释放。

这里是 `/dev/scullpriv` 的 **release** 的实现。对于设备方法的讨论也到此结束。

```

int scull_c_release(struct inode *inode, struct file *filp)
{
    /*
     * Nothing to do, because the device is persistent.
     * A 'real' cloned device should be freed on last close
     */
    MOD_DEC_USE_COUNT;
    return 0;
}

```

5.7 向后兼容性

随着新内核版本的发布，本章涉及的许多设备驱动程序的 **API** 都发生了变化。如果需要使驱动程序在 **Linux 2.0** 或者 **2.2** 工作，这里是一个简要的提纲，列出了会遇到的差别。

5.7.1 Linux 2.2 和 2.0 中的等待队列

本章涉及的资料在 **2.3** 开发系列中相对变化较少。一个重要的更改是在等待队列方面。**2.2** 版本内核的等待队列的实现有所不同，相对简单一些，不过缺少一些重要的象排他睡眠这样的功能。等待队列的新的实现是在内核 **2.3.1** 版本引入的。

2.2 的等待队列的实现中使用了一个 `struct wait_queue *` 类型的变量，而不是用 `wait_queue_head_t`。该指针在使用之前必须初始化为 **NULL**。等待队列的一个典型的声明和初始化过程是这样：

```
struct wait_queue *my_queue = NULL;
```

除了队列本身的变量类型不同，其它处理睡眠和唤醒的函数看起来也差不多。所以，编写在 2.x 系列内核运行的代码是很容易的，只要包含了类似下面的代码就可以了。用来编译示例代码的头文件 `sysdep.h` 中就包含了它们。

```
# define DECLARE_WAIT_QUEUE_HEAD(head) struct wait_queue *head = NULL
typedef struct wait_queue *wait_queue_head_t;
# define init_waitqueue_head(head) (*(head)) = NULL
```

`wake_up` 的同步版本是在 2.3.29 加入的，`sysdep.h` 提供了同名的宏使代码在保持兼容性的同时使用该功能。这些替代的宏扩展为通常的 `wake_up`，因为早期的内核中没有相关的机制。`sleep_on` 的 `timeout` 版本是在内核 2.1.127 加入的。等待队列的其余接口相对保持不变。头文件 `sysdep.h` 定义了所需的宏以使模块在 Linux 2.2 和 Linux 2.0 也能编译运行，这样就不用代码中加入大量的 `#ifdef` 条件编译语句。

内核 2.0 中没有 `wait_event` 宏。在 `sysdep.h` 中提供了它的实现。

5.7.2 异步通知

在异步通知的工作方式上，2.2 和 2.4 版本都有些小的改动。

`kill_fasync` 是在 Linux 2.3.21 增加了第三个参数的。在此以前，`kill_fasync` 是这样调用的：

```
kill_fasync(struct fasync_struct *queue, int signal);
```

`sysdep.h` 处理了这个问题。

在 2.2 版本，`fasync` 方法的第一个参数的类型已经变了。在 2.0 内核是传递一个指向设备对应的 `inode` 结构的指针，而不是整型的文件描述符：

```
int (*fasync) (struct inode *inode, struct file *filp, int on);
```

为解决这个不兼容的问题，我们使用了在 `read` 和 `write` 中用过的同样方法：当模块在 2.0 头文件下编译时，使用一个包装函数。

`fasync` 方法的 `inode` 参数在调用 `release` 方法时还要传递给它，而不是象后来的内核那样使用 `-1`。

5.7.3 fsync 方法

传给 `fsync` 的 `file_operations` 方法的第三个参数（整数值 `datasync`）是在 2.3 开发系列中加入的，所以可移植的代码一般需要包括一个给老内核用的包装函数。然而对试图编写可移植的 `fsync` 方法的人来说有个陷阱：至少有一个不知名的发行商，把 2.4 的 `fsync` 的 API 补丁打进了它的 2.2 内核中。内核开发人员通常（通常）会避免修改一个稳定版本系列的 API，但他们不能控制发行商的行为。

5.7.4 在 Linux 2.0 中访问用户空间

2.0 内核中的内存访问方法是不同的。那时 Linux 虚拟内存系统还没有开发好，内存访问方法有一点不同。新的内存系统是 2.1 开发系列的关键改进，带来了性能的显著提高。不幸的是，对驱动程序开发人员来说，它也带来了另外一大堆令人头痛的兼容性问题。

Linux 2.0 的内存访问函数是象下面这样的：

```
verify_area(int mode, const void *ptr, unsigned long size);
```

这个函数的工作方式和 `access_ok` 类似，但是进行了更多的检查，所以慢一些。成功时函数返回 0，出错时返回 `-EFAULT`。近来的内核头文件仍然定义了该函数，但现在只是 `access_ok` 的一个包装函数。使用 2.0 内核时，调用 `verify_area` 绝不是可有可无的，没有预先的、清楚的检验确认的话，是不能安全地访问用户空间的。

```
put_user(datum, ptr)
```

`put_user` 宏看起来很像它的现在的同名宏。然而它们的不同之处在于，没有检验确认，也没有返回值。

```
get_user(ptr)
```

这个宏取得给定地址的值，并把它作为自己的返回值返回。同样，宏执行时不做检验确认。

由于用户空间的复制函数都不做检查，必须显式地调用 `verify_area`。Linux 2.1 在引入了 `get_user` 和 `put_user` 函数中造成不兼容的修改的同时，带来的好消息是检验用户地址的任务留给硬件去完成，因为内核现在能够捕获和处理在数据复制到用户空间时期产生的处理器异常。

作为使用旧调用的例子，再来看一下 `scull`。使用 2.0 API 的 `scull` 版本这样调用 `verify_area`：

```
int err = 0, tmp;

/*
 * extract the type and number bitfields, and don't decode
 * wrong cmds: return ENOTTY before verify_area()
 */
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*
 * the direction is a bit mask, and VERIFY_WRITE catches R/W
 * transfers. `Type' is user oriented, while
 * verify_area is kernel oriented, so the concept of "read" and
 * "write" is reversed
 */
if (_IOC_DIR(cmd) & _IOC_READ)
    err = verify_area(VERIFY_WRITE, (void *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
    err = verify_area(VERIFY_READ, (void *)arg, _IOC_SIZE(cmd));
if (err) return err;
```

然后可以象下面这样使用 `get_user` 和 `put_user`：

```
case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer */
    tmp = scull_quantum;
    scull_quantum = get_user((int *)arg);
```

```

    put_user(tmp, (int *)arg);
    break;

default: /* redundant, as cmd was checked against MAXNR */
    return -ENOTTY;
}
return 0;

```

只列出了 `ioctl` 的 `switch` 代码的一小部分，因为和 2.2 及其它版本的差别很小。

对那些兼容性敏感的驱动程序的开发人员来说，如果不是在所有的 Linux 版本中 `put_user` 和 `get_user` 都用宏实现而且接口也不同的话，也许会轻松很多。现在的结果是，直接修改宏无法解决问题。

一个可能的解决方法是定义一套新的与版本无关的宏。`sysdep.h` 使用的办法包括定义大写的宏：`GET_USER`，`_GET_USER`，等等。参数和 Linux 2.4 的内核使用的宏相同，但是调用者必须确定先前已经调用过 `verify_area` 了（因为为 2.0 编译时该调用是必须的）。

5.7.5 2.0 中的权能

2.0 内核根本不支持权能的抽象概念。所有的许可检查仅仅是看调用进程是否作为超级用户在运行，如果是就允许操作。函数 `suser` 就用于此目的，它无需参数，如果进程有超级用户特权就返回一个非 0 值。

后来的内核中 `suser` 仍然存在，但已强烈不建议使用。更好的办法是为 2.0 定义一个 `capable` 版本，`sysdep.h` 中就是这样：

```
# define capable(anything) suser()
```

这样，代码具有了可移植性，而且在现在的基于权能的系统上也能工作。

5.7.6 Linux 2.0 的 select 方法

2.0 内核不支持 `poll` 系统调用，只有 BSD 风格的 `select` 调用可以使用。相应的设备驱动程序方法都是调用 `select`，尽管完成的动作几乎是相同的，操作起来还是略有区别。

`select` 方法被传给一个指向 `select_table` 类型的指针，只有在调用进程将要等待要求的条件（`SEL_IN`、`SEL_OUT` 或 `SEL_EX` 中的一个）时才把该指针传给 `select_wait`。

`scull` 驱动程序通过声明一个特定的 `select` 方法来解决兼容问题，该方法在为 2.0 版本内核编译时使用：

```

#ifdef __USE_OLD_SELECT__
int scull_p_poll(struct inode *inode, struct file *filp,
                int mode, select_table *table)
{
    Scull_Pipe *dev = filp->private_data;

    if (mode == SEL_IN) {
        if (dev->rp != dev->wp) return 1; /* readable */
        PDEBUG("Waiting to read\n");
    }
}

```

```

select_wait(&dev->inq, table); /* wait for data */
return 0;
}
if (mode == SEL_OUT) {
    /*
     * The buffer is circular; it is considered full
     * if "wp" is right behind "rp". "left" is 0 if the
     * buffer is empty, and it is "1" if it is completely full.
     */
    int left = (dev->rp + dev->buffersize - dev->wp) % dev->buffersize;
    if (left != 1) return 1; /* writable */
    PDEBUG("Waiting to write\n");
    select_wait(&dev->outq, table); /* wait for free space */
    return 0;
}
return 0; /* never exception-able */
}
#else /* Use poll instead, already shown */

```

这里的 `__USE_OLD_SELECT__` 预处理符号是在 `sysdep.h` 包含文件中根据内核版本设置的。

5.7.7 Linux 2.0 的设备定位

在 Linux 2.1 之前，`llseek` 设备方法叫作 `lseek`，它接收的参数与当前版本的也不一样。因此，在 Linux 2.0 是不允许定位一个超过 2GB 限制的文件或设备的。现在的 `llseek` 系统调用则已经支持了。

2.0 内核中该文件操作的原型如下：

```
int (*lseek) (struct inode *inode, struct file *filp, off_t off, int whence);
```

驱动程序通常通过为 `seek` 方法的两个接口定义互相分离的实现来保持 2.0 和 2.2 的兼容。

5.7.8 2.0 和 SMP

因为 Linux 2.0 仅仅为 SMP 系统提供了最低限度的支持，本章提到的那些类型的竞态通常不会发生。2.0 内核倒是也有一个自旋锁的实现，但是因为一次只有一个处理器可以运行内核代码，所以很少用到锁。

5.8 快速参考

本章介绍了如下这些符号和头文件：

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

这个头文件声明了用于定义 `ioctl` 命令的所有的宏。它现在包含在 `<linux/fs.h>` 中。

```

_IOC_NRBITS
_IOC_TYPEBITS
_IOC_SIZEBITS
_IOC_DIRBITS

```

`ioctl` 命令的不同位字段的可用位数。还有 4 个宏定义了不同的 MASK（掩码），4 个宏定义了不同的 SHIFT（偏移），但它们基本仅用于内部使用。由于 `_IOC_SIZEBITS` 在不同体系结构上的值不同，它是一个需要检查的重要的值。


```
_IOC_NONE
_IOC_READ
_IOC_WRITE
```

“方向”位字段的可能值。“读”和“写”是不同的位，可以 OR 在一起实现读/写。这些值都是基于 0 的。

```
_IOC(dir,type,nr,size)
_IO(type,nr)
_IOR(type,nr,size)
_IOW(type,nr,size)
_IOWR(type,nr,size)
```

用于生成 `ioctl` 命令的宏。

```
_IOC_DIR(nr)
_IOC_TYPE(nr)
_IOC_NR(nr)
_IOC_SIZE(nr)
```

用于解码 `ioctl` 命令的宏。特别地，`_IOC_TYPE(nr)` 是 `_IOC_READ` 和 `_IOC_WRITE` 的 OR 的结果。

```
#include <asm/uaccess.h>
int access_ok(int type, const void *addr, unsigned long size);
```

这个函数验证指向用户空间的指针是否可用。如果访问允许，`access_ok` 返回非 0 值。

```
VERIFY_READ
VERIFY_WRITE
```

`access_ok` 中 `type` 参数可取的值。`VERIFY_WRITE` 是 `VERIFY_READ` 的超集。

```
#include <asm/uaccess.h>
int put_user(datum,ptr);
int get_user(local,ptr);
int _put_user(datum,ptr);
int _get_user(local,ptr);
```

用于向用户空间存取单个数据的宏。传送的字节数目由 `sizeof(*ptr)` 决定。前两个要先调用 `access_ok`，后两个（`_put_user` 和 `_get_user`）则假设 `access_ok` 已经调用过了。

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

为 Linux 2.2 和以后版本的权能操作定义了不同的 `CAP_` 符号。

```
int capable(int capability);
```

如果进程具有指定的权能，返回非 0 值。

```
#include <linux/wait.h>
typedef struct { /* ... */ } wait_queue_head_t;
void init_waitqueue_head(wait_queue_head_t *queue);
DECLARE_WAIT_QUEUE_HEAD(queue);
```

Linux 等待队列的已定义类型。`wait_queue_head_t` 类型必须显式地初始化，初始化方法可以在运行时用 `init_waitqueue_head`，或者在编译时用 `declare_waitqueue_head`。

```
#include <linux/sched.h>
void interruptible_sleep_on(wait_queue_head_t *q);
void sleep_on(wait_queue_head_t *q);
void interruptible_sleep_on_timeout(wait_queue_head_t *q, long timeout);
void sleep_on_timeout(wait_queue_head_t *q, long timeout);
```

调用这些函数都使当前进程在队列上睡眠。实现阻塞型的 `read` 和 `write` 时，通常选择 `interruptible` 形式的函数。

```
void wake_up(struct wait_queue **q);
void wake_up_interruptible(struct wait_queue **q);
void wake_up_sync(struct wait_queue **q);
void wake_up_interruptible_sync(struct wait_queue **q);
```

这些函数唤醒正睡眠在队列 `q` 上的进程。`_interruptible` 形式的只能唤醒可中断的进程。`_sync` 版本的函数在返回前不会重新调度 CPU。

```
typedef struct { /* ... */ } wait_queue_t;
init_waitqueue_entry(wait_queue_t *entry, struct task_struct *task);
```

`wait_queue_t` 类型用于不通过调用 `sleep_on` 而进行的睡眠。等待队列的入口必须在使用前先初始化；`task` 参数几乎总是 `current`。

```
void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
void add_wait_queue_exclusive(wait_queue_head_t *q, wait_queue_t *wait);
void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
```

这些函数在等待队列中增加一个成员。`add_wait_queue_exclusive` 把成员加到队列的尾部以实现排外等待。睡眠结束后，这些成员要通过调用 `remove_wait_queue` 从队列中删除。

```
void wait_event(wait_queue_head_t q, int condition);
int wait_event_interruptible(wait_queue_head_t q, int condition);
```

这两个宏使进程在指定的队列上睡眠，直到指定条件的值为真。

```
void schedule(void);
```

这个函数从运行队列中选择一个可运行进程。被选中的进程可以是 `current` 或另一个不同进程。通常不会直接调用 `schedule`，因为 `sleep_on` 函数的内部已经调用了它。

```
#include <linux/poll.h>
void poll_wait(struct file *filp, wait_queue_head_t *q, poll_table *p)
```

这个函数把当前进程放入一个等待队列而不立即调度。它设计为供设备驱动程序的 `poll` 方法使用。

```
int fasync_helper(struct inode *inode, struct file *filp, int mode, struct fasync_struct **fa);
```

这个函数“帮助”实现 `fasync` 设备方法。参数 `mode` 被直接传给 `fasync` 方法，`fa` 指向一个设备相关的 `fasync_struct *` 类型。

```
void kill_fasync(struct fasync_struct *fa, int sig, int band);
```

如果驱动程序支持异步通知，这个函数可以用来发送一个信号给注册在 `fa` 中的进程。

```
#include <linux/spinlock.h>
typedef struct { /* ... */ } spinlock_t; "void spin_lock_init(spinlock_t *lock);
spinlock_t 类型定义了一个自旋锁，它在使用前必须先初始化（用 spin_lock_init）。
```

```
spin_lock(spinlock_t *lock);
spin_unlock(spinlock_t *lock);
```

`spin_lock` 锁住指定的锁，然后等待，直到它可用为止。这个锁随后用 `spin_unlock` 来释放。

第 6 章 时间流



至此，我们基本知道怎样编写一个功能完整的字符模块了。现实中的设备驱动程序，除了实现必需的操作外还要做更多工作，如计时、内存管理，硬件访问等等。幸好，内核中提供的许多机制可以简化驱动程序开发者的工作，我们将在后面几章陆续讨论驱动程序可以访问的一些内核资源。本章，我们先来看看内核代码是如何对时间问题进行处理。按复杂程度递增排列，该问题包括：

- 理解内核时间机制
- 如何获得当前时间
- 如何将操作延迟指定的一段时间
- 如何调度异步函数到指定的时间后执行

6.1 内核中的时间间隔

我们首先要涉及的是时钟中断，操作系统通过时钟中断来确定时间间隔。中断是异步事件，通常由外部硬件触发。中断发生时，CPU 停止正在进行的任务，转而执行另一段特殊的代码（即中断服务例程，又称 ISR）来响应这个中断。中断和 ISR 的实现将在第 9 章讨论。

时钟中断由系统计时硬件以周期性的间隔产生，这个间隔由内核根据 HZ 的值设定，HZ 是一个与体系结构有关的常数，在文件 `<linux/param.h>` 中定义。当前的 Linux 版本为大多数平台定义的 HZ 的值是 100，某些平台上是 1024，IA-64 仿真器上是 20。驱动程序开发者不应使用任何特定的 HZ 值来计数，不管你的平台使用的是哪一个值。

当时钟中断发生时，变量 `jiffies` 的值就增加。`jiffies` 在系统启动时初始化为 0，因此，`jiffies` 值就是自操作系统启动以来的时钟滴答的数目，`jiffies` 在头文件 `<linux/sched.h>` 中被定义为数据类型为 `unsigned long volatile` 型变量，这个变量在经过长时间的连续运行后有可能溢出（不过现在还没有哪种平台会在运行不到 16 个月就使 `jiffies` 溢出）。为了保证 `jiffies` 溢出时内核仍能正常工作，人们已做了很多努力。驱动程序开发人员通常不用考虑 `jiffies` 的溢出问题，知道有这种可能性就行了。

如果想改变系统时钟中断发生的频率，可以修改 HZ 值。有人使用 Linux 处理硬实时任务，他们增加了 HZ 值以获得更快的响应时间，为此情愿忍受额外的时钟中断产生的系统开销。总而言之，时钟中断的最好方法是保留 HZ 的缺省值，因为我们可以完全相信内核的开发者们，他们一定已经为我们挑选了最佳值。

6.1.1 处理器特有的寄存器

如果需要度量非常短的时间，或是需要极高的时间精度，可以使用与特定平台相关的资源，这是将时间精度的重要性凌驾于代码的可移植性之上的做法。

大多数较新的 CPU 都包含一个高精度的计数器，它每个时钟周期递增一次。这个计数器可用于精确地度量时间。由于大多数系统中的指令执行时间具有不可预测性（由于指令调度、分支预测、缓存等等），在运行具有很小时间粒度的任务时，使用这个时钟计数器是唯一可靠的计时方法。为适应现代处理器的高速度，满足衡量性能指标的紧迫需求，同时由于 CPU 设计中的多层缓存引起的指令时间的不可预测性，CPU 的制造商们引入了记录时钟周期这一测量时间的简单可靠的方法。所以绝大多数现代处理器都包含一个随时钟周期不断递增的计数寄存器。

基于不同的平台，在用户空间，这个寄存器可能是可读的，也可能不可读；可能是可写的，也可能不可写；可能是 64 位的也可能是 32 位的。如果是 32 位的，还得注意处理溢出的问题。无论该寄存器是否可以置 0，我们都强烈建议不要重置它，即使硬件允许这么做。因为总可以通过多次读取该寄存器并比较读出数值的差异来完成要做的事，我们无须要求独占该寄存器并修改它的当前值。

最有名的计数器寄存器就是 TSC（timestamp counter，时间戳计数器），从 x86 的 Pentium 处理器开始提供该寄存器，并包括在以后的所有 CPU 中。它是一个 64 位寄存器，记录 CPU 时钟周期数，内核空间和用户空间都可以读取它。

包含了头文件 `<asm/msr.h>`（意指“machine-specific registers，机器特有的寄存器”）之后，就可以使用如下的宏：

```
rdtsc(low,high);
rdtscl(low);
```

前一个宏原子性地把 64 位的数值读到两个 32 位变量中；后一个只把寄存器的低半部分读入一个 32 位变量，在大多数情况，这已经够用了。举例来说，一个 500MHz 的系统使一个 32 位计数器溢出需 8.5 秒，如果要处理的时间肯定比这短的话，那就没有必要读出整个寄存器。

下面这段代码可以测量该指令自身的运行时间：

```
unsigned long ini, end;
rdtscl(ini); rdtsc(end);
printf("time lapse: %li\n", end - ini);
```

其他一些平台也提供了类似的功能，在内核头文件中还有一个与体系结构无关的函数可以代替 rdtsc，它就是 `get_cycles`，是在 2.1 版的开发过程中引入的。其原型是：

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

```
cycles_t get_cycles(void);
```

在各种平台上都可以使用这个函数，在没有时钟周期计数寄存器的平台上它总是返回 0。`cycles_t` 类型是能装入对应 CPU 单个寄存器的合适的无符号类型。选择能装入单个寄存器的类型意味着，举例来说，`get_cycles` 用于 Pentium 的时钟周期计数器时只返回低 32 位。这种选择是明智的，它避免了多寄存器操作的问题，与此同时并未阻碍对该计数器的正常用法，即用来度量很短的时间间隔。

除了这个与体系结构无关的函数外，我们还将示例使用一段内嵌的汇编代码。为此，我们来给 MIPS 处理器实现一个 `rdtscl` 函数，功能就象 x86 的一样。

这个例子之所以基于 MIPS，是因为大多数 MIPS 处理器都有一个 32 位的计数器，在它们的内部“coprocessor 0”中命名为 register 9 寄存器。为了从内核空间读取该寄存器，可以定义下面的宏，它执行“从 coprocessor 0 读取”的汇编指令：^{*}

```
#define rdtsc(dest) \
    __asm__ __volatile__ ("mfc0 %0,$9; nop" : "=r" (dest))
```

通过使用这个宏，MIPS 处理器就可以执行和前面所示用于 x86 的相同的代码了。

gcc 内嵌汇编的有趣之处在于通用寄存器的分配使用是由编译器完成的。这个宏中使用的 %0 只是“参数 0”的占位符，参数 0 由随后的“作为输出(=)使用的任意寄存器(r)”定义。该宏还说明了输出寄存器要对应于 C 表达式 `dest`。内嵌汇编的语法功能强大但也比较复杂，特别是在对各寄存器使用有限制的平台上更是如此，如 x86 系列。完整的语法描述在 gcc 文档中提供，一般在 info 中就可找到。

这节展示的短小的 C 代码段已经在一个 K7 类的 x86 处理器和一个 MIPS VR4181 处理器（使用了刚才的宏）上运行过了。前者给出的时间消耗为 11 时钟周期，后者仅为 2 时钟周期。这是可以理解的，因为 RISC 处理器通常每时钟周期运行一条指令。

6.2 获取当前时间

内核一般通过 `jiffies` 值来获取当前时间。该数值表示的是自最近一次系统启动到当前的时间间隔，它和设备驱动程序不怎么相关，因为它的生命期只限于系统的运行期(uptime)。但驱动程序可以利用 `jiffies` 的当前值来计算不同事件间的时间间隔(比如在输入设备驱动程序中就用它来分辨鼠标的单双击)。简而言之，利用 `jiffies` 值来测量时间间隔在大多数情况下已经足够了，如果还需要测量更短的时间，就只能使用处理器特有的寄存器了。

驱动程序一般不需要知道墙钟时间（指日常生活使用的时间），通常只有象 `cron` 和 `at` 这样用户程序才需要墙钟时间。需要墙钟时间的情形是使用设备驱动程序的特殊情况，此时可以通过用户程序来将墙钟时间转换成系统时钟。直接处理墙钟时间常常意味着正在实现某种策略，应该仔细审视一下是否该这样做。

^{*} `nop` 指令是必需的，防止了编译器在指令 `mfc0` 之后立刻访问目标寄存器。这种互锁(interlock)在 RISC 处理器中是很典型的，在延迟期间编译器仍然可以调度其它指令执行。我们在这里使用 `nop`，是因为内嵌汇编指令对编译器来说是个黑盒，不能进行优化。

如果驱动程序真的需要获取当前时间，可以使用 `do_gettimeofday` 函数。该函数并不返回今天是本周的星期几或类似的信息；它是用秒或微秒值来填充一个指向 `struct timeval` 的指针变量，`gettimeofday` 系统调用中用的也是同一变量。`do_gettimeofday` 的原型如下：

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

源码中描述 `do_gettimeofday` 在许多体系结构上有“接近微秒级的分辨率”，然而实际精度是随不同的平台而变化的，在旧版本的内核中还会低些。当前时间也可以通过 `xtime` 变量(类型为 `struct timeval`)获得(但精度差些)，但是，并不鼓励直接使用该变量，因为除非关闭中断，否则无法原子性地访问 `timeval` 变量的两个成员 `tv_sec` 和 `tv_usec`。在 2.2 版的内核中，一个快捷安全的获得时间的办法(可能精度会差些)是使用 `get_fast_time`：

```
void get_fast_time(struct timeval *tv);
```

获取当前时间的代码可见于 `jit` (“Just In Time”)模块，源文件可以从 O'Reilly 公司的 FTP 站点获得。`jit` 模块将创建 `/proc/currenttime` 文件，读取该文件将以 ASCII 码的形式返回三项：

- 由 `do_gettimeofday` 返回的当前时间
- 从 `xtime` 钟获得的当前时间
- `jiffies` 的当前值

我们选择用动态的 `/proc` 文件，是因为这样模块代码量会小些——不值得为返回三行文本而写一个完整的设备驱动程序。

如果用 `cat` 命令在一个时钟滴答内多次读该文件，就会发现 `xtime` 和 `do_gettimeofday` 两者的差异了，`xtime` 更新的次数不那么频繁：

```
morgana% cd /proc; cat currenttime currenttime currenttime
gettime: 846157215.937221
xtime:   846157215.931188
jiffies: 1308094
gettime: 846157215.939950
xtime:   846157215.931188
jiffies: 1308094
gettime: 846157215.942465
xtime:   846157215.941188
jiffies: 1308095
```

6.3 延迟执行

设备驱动程序经常需要将某些特定代码延迟一段时间后执行——通常是为了让硬件能完成某些任务。这一节将介绍许多实现延迟的不同技术，哪种技术最好取决于实际环境中的具体情况。我们将介绍所有的这些技术并指出各自的优缺点。

一件需要考虑的很重要的事情是所需的延迟长度是否多于一个时钟滴答。较长的延迟可以利用系统时钟；较短的延迟通常必须通过软件循环来获得。

6.3.1 长延迟

如果能把执行延迟若干个时钟滴答，或者对延迟的精度要求不高(比如，想延迟整数数目的秒数)，最简单的也是最笨的实现如下，也就是所谓的“忙等待”：

```
unsigned long j = jiffies + jit_delay * HZ;

while (jiffies < j)
    /* nothing */;
```

这种实现当然要避免。我们在这里提到它，只是因为读者可能某时需要运行这段代码，以便更好地理解其他的延迟技术。

还是先看看这段代码是如何工作的。因为内核的头文件中 `jiffies` 被声明为 `volatile` 型变量，每次 C 代码访问它时都会重新读取它，因此该循环可以起到延迟的作用。尽管也是“正确”的实现，但这个忙等待循环在延迟期间会锁住处理器，因为调度器不会中断运行在内核空间的进程。更糟糕的是，如果在进入循环之前正好关闭了中断，`jiffies` 值就不会得到更新，那么 `while` 循环的条件就永远为真，这时，你不得不按下那只大的红按钮(指电源按钮)。

这种延迟和下面的几种延迟方法都在 `jit` 模块中实现了。由该模块创建的所有 `/proc/jit*` 文件每次被读取时都延迟整整 1 秒。如果你想测试忙等待代码，可以读 `/proc/jitbusy` 文件，当该文件的 `read` 方法被调用时它将进入忙等待循环，延迟 1 秒；而象 `dd if=/proc/jitbusy bs=1` 这样的命令每次读一个字符就要延迟 1 秒。

可以想见，读 `/proc/jitbusy` 文件会大大影响系统性能，因为此时计算机要到 1 秒后才能运行其他进程。

更好的延迟方法如下，它允许其他进程在延迟的时间间隔内运行，尽管这种方法不能用于硬实时任务或者其他对时间要求很严格的场合：

```
while (jiffies < j)
    schedule();
```

这个例子和下面各例中的变量 `j` 应是延迟到达时的 `jiffies` 值，计算方法和忙等待一样。

这种循环(可以通过读 `/proc/jitsched` 文件来测试它)延迟方法还不是最优的。系统可以调度其他任务；当前任务除了释放 CPU 之外不做任何工作，但是它仍在任务队列中。如果它是系统中唯一的可运行的进程，它还会被运行(系统调用调度器，调度器选择同一个进程运行，此进程又再调用调度器，然后...)。换句话说，机器的负载(系统中运行的进程平均数)至少为 1，而 `idle` 进程(进程号为 0，由于历史原因被称为“swapper”)绝不会被运行。尽管这个问题看来无所谓，当系统空闲时运行 `idle` 进程可以减轻处理器负载，降低处理器温度，延长处理器寿命，如果是手提电脑，还能延长电池的寿命。而且，延迟期间实际上进程是在执行的，因此延迟消耗的所有时间都是记在它的运行时间上的。运行命令 `time cat /proc/jitsched` 就可以发现这一点。

另一种情况下，如果系统很忙，驱动程序等待的时间可能会比预计多得多。一旦一个进程在调度时让出了处理器，无法保证以后的某个时间就能重新分配给它。如果可接受的延迟时间有上限的话，

用这种方式调用 `schedule`，对驱动程序来说并不是一个安全的解决方案。

尽管有些毛病，这种循环延迟还是提供了一种有点“脏”但比较快的监视驱动程序工作的途径。如果模块中的某个 `bug` 会锁死整个系统，则可在每个用于调试的 `printk` 语句后添加一小段延迟，这样可以保证在处理器碰到令人厌恶的 `bug` 而被锁死之前，所有的打印消息都能进入系统日志。如果没有这样的延迟，这些消息只能进入内存缓冲区，但在 `klogd` 得到运行前系统可能已经被锁住了。

获得延迟的最好方法，是请求内核为我们实现延迟。根据驱动程序是否在等待其他事件，有两种设置短期延迟的办法。

如果驱动程序使用等待队列等待某个事件，而你又想确保在一段时间后一定运行该驱动程序，可以使用 `sleep` 函数的超时版本，这在第 5 章“睡眠和唤醒”一节中已介绍过了：

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t *q,
                               unsigned long timeout);
```

两种实现都能让进程在指定的等待队列上睡眠，而在超时期限（用 `jiffies` 表示）未到时的任何事件都会将其唤醒。由此它们就实现了一种有上限的不会永远持续下去的睡眠。注意超时值表示要等待的 `jiffies` 数量，而不是绝对的时间值。这种方式的延迟可以在 `/proc/jitqueue` 的实现中看到：

```
wait_queue_head_t wait;

init_waitqueue_head (&wait);
interruptible_sleep_on_timeout(&wait, jit_delay*HZ);
```

在通常的驱动程序中，可以以下列两种方式重新获得执行：在等待队列上调用一个 `wake_up`，或者 `timeout` 超时。在这个特定实现中，没人会调用 `wake_up`（毕竟其它代码根本就不知道这件事），所以进程总是因 `timeout` 超时而被唤醒。这是一个完美有效的实现，不过，如果驱动程序无须等待其它事件，可以用一种更直接的方式获取延迟，即使用 `schedule_timeout`：

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (jit_delay*HZ);
```

上述代码行（在 `/proc/jitself` 中实现）使进程进入睡眠直到指定时间。`schedule_timeout` 也是处理一个时间增量而不是一个 `jiffies` 的绝对值。和前面一样，在从超时到进程实际被调度执行之间，可能会消耗一些毫无价值的额外时间。

6.3.2 短延迟

有时驱动程序需要非常短的延迟来和硬件同步。此时，使用 `jiffies` 值无法达到目的。

这时就要用内核函数 `udelay` 和 `mdelay`^{*}。

它们的原型如下：

^{*} `u` 表示希腊字母“mu” (μ)，它代表“微”。

```
#include <linux/delay.h>
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

该函数在绝大多数体系结构上是作为内联函数编译的。前者使用软件循环延迟指定数目的微秒数，后者使用 `udelay` 做循环，用于方便程序开发。`udelay` 函数里要用到 **BogoMips** 值：它的循环基于整数值 `loops_per_second`，这个值是在引导阶段计算 **BogoMips** 时得到的结果。

`udelay` 函数只能用于获取较短的时间延迟，因为 `loops_per_second` 值的精度只有 8 位，所以，当计算更长的延迟时会积累出相当大的误差。尽管最大能允许的延迟将近 1 秒(因为更长的延迟就要溢出)，推荐的 `udelay` 函数的参数的最大值是取 1000 微秒(1 毫秒)。延迟大于 1 毫秒时可以使用函数 `mdelay`。

要特别注意的是 `udelay` 是个忙等待函数（所以 `mdelay` 也是），在延迟的时间段内无法运行其他的任务，因此要十分小心，尤其是 `mdelay`，除非别无他法，要尽量避免使用。

目前在支持大于几个微秒和小于 1 个时钟滴答的延迟时还是很低效的，但这通常不是个问题，因为延迟需要足够长，以便能够让人或者硬件注意到。对人来说，百分之一秒的时间间隔是比较适合的精度，而 1 毫秒对硬件动作来说也足够长了。

`mdelay` 在 Linux 2.0 中并不存在，头文件 `sysdep.h` 弥补了这一缺陷。

6.4 任务队列

许多驱动程序需要将任务延迟到以后处理，但又不想借助中断。Linux 为此提供了三种方法：任务队列、`tasklet`（从内核 2.3.43 开始）和内核定时器。任务队列和 `tasklet` 的使用很灵活，可以或长或短地延迟任务到以后处理，在编写中断处理程序时非常有用，我们还将第 9 章“`Tasklet` 和底半部处理”一节中继续讨论。内核定时器则用来调度任务在未来某个指定时间执行，将在本章的“内核定时器”一节中讨论。

使用任务队列或 `tasklet` 的一个典型情形是，硬件不产生中断，但仍希望提供阻塞型的读取。此时需要对设备进行轮询，同时要小心地不使 CPU 负担过多无谓的操作。将读进程以固定的时间间隔唤醒（例如，使用 `current->timeout` 变量）并不是个很好的方法，因为每次轮询需要两次上下文切换（一次是切换到读进程中运行轮询代码，另一次是返回执行实际工作的某个进程），而且通常来讲，恰当的轮询机制应该在进程上下文之外实现。

类似的情形还有象不时地给简单的硬件设备提供输入。例如，有一个直接连接到并口的步进马达，要求该马达能一步步地移动，但马达每次只能移动一步。在这种情况下，由控制进程通知设备驱动程序进行移动，但实际上，移动是在 `write` 返回后，才在周期性的时间间隔内一步一步进行的。

快速完成这类不定操作的恰当方法是注册任务在未来执行。内核提供了对“任务队列”的支持，任务可以累积，而在运行队列时被“消耗”。我们可以声明自己的任务队列，并且在任意时刻触发它，或者也可以将任务注册到预定义的任务队列中去，由内核来运行（触发）它。

这一节将首先概述任务队列，然后介绍预定义的任务队列，这使读者可以开始一些有趣的测试（如果出错也可能挂起系统），最后介绍如何运行自己的任务队列。接着，我们来看看新的 **tasklet** 接口，在 2.4 内核中它在很多情况下取代了任务队列。

6.4.1 任务队列的本质

任务队列其实是一个任务链表，每个任务用一个函数指针和一个参数表示。任务运行时，它接受一个 **void *** 类型的参数，返回值类型为 **void**，而指针参数可用来将一个数据结构传入函数，或者可以被忽略。队列本身是一个结构（即任务）链表，并由声明和操纵它们的内核模块所拥有。模块要全权负责这些数据结构的分配和释放，为此一般使用静态的数据结构。

队列元素由下面这个结构来描述，这段代码是直接从头文件 `<linux/tqueue.h>` 拷贝下来的：

```
struct tq_struct {
    struct tq_struct *next;      /* linked list of active bh's */
    int sync;                   /* must be initialized to zero */
    void (*routine)(void *);    /* function to call */
    void *data;                 /* argument to function */
};
```

第一个注释中的 **bh** 指的是底半部（**bottom-half**）。底半部是“中断处理程序的一半部”，我们将在第 9 章的“**tasklet** 和底半部”一节中介绍中断时详细讨论。现在，我们只要知道底半部是驱动程序实现的一种机制就可以了，它用于处理异步任务，这些任务通常比较大，不适于在处理硬件中断时完成。本章并不要求你理解底半部处理，但必要时也会偶尔提及。

上面的数据结构中最重要的成员是 **routine** 和 **data**。为了将随后执行的任务排队，必须先设置好结构的这些成员，并把 **next** 和 **sync** 两个字段清零。结构中的 **sync** 标志位由内核使用，以避免同一任务被插入多次，因为这会破坏 **next** 指针。一旦任务被排队，该数据结构就被认为由内核“拥有”了，不能再被修改，直到任务开始运行。

与任务队列有关的其他数据结构还有 **task_queue**，目前它实现为指向 **tq_struct** 结构的指针，如果将来需要扩充 **task_queue**，只要用 **typedef** 将该指针定义为其符号就可以了。在使用之前，必须将 **task_queue** 指针初始化为 **NULL**。

下面汇总了所有可以在任务队列和 **tq_struct** 结构上执行的操作。

```
DECLARE_TASK_QUEUE(name);
```

这个宏用给定的名称 **name** 声明了一个任务队列，并把它初始化为空。

```
int queue_task(struct tq_struct *task, task_queue *list);
```

正如该函数的名字，它用于将任务排进队列中。如果队列中已有该任务，返回 0，否则返回非 0。

```
void run_task_queue(task_queue *list);
```

run_task_queue 函数用于运行累积在队列上的任务。除非你要声明和维护自己的任务队列，否则不必调用本函数。

在讨论使用任务队列的细节之前，我们先看一下它们在内核中是怎样工作的。

6.4.2 任务队列的运行

如前所述，一个任务队列，实际上是一个函数链表。当调用 `run_task_queue` 运行某个队列时，列表中的每一项都会被执行。在编写和任务队列有关的函数时，一定要记住，当内核调用 `run_task_queue` 时，实际的上下文将限制能够进行的操作。也不应对队列中任务的运行顺序做任何假定，它们每个都是独立完成自己的任务的。

那么任务队列在什么时候运行呢？如果使用的是下面一节介绍的预定义的任务队列，则答案是“在内核轮到它那里时”。不同的队列在不同的时间运行，只要内核没有其他更紧要的任务，它们总是会运行的。

更重要的是，当对任务进行排队的进程运行时，任务队列几乎肯定是不运行的，相反，它们是异步执行的。到现在为止，示例驱动例程中所有的事情都是在这个执行系统调用的进程上下文中完成的。但当任务队列运行时，这个进程可能正在睡眠，或正在另一个处理器上运行，甚至可能已经完全退出了。

这种异步执行类似于硬件中断发生时的情景（我们会在第 9 章详细讨论）。实际上，任务队列常常是作为“软件中断”的结果而运行的。在中断模式（或中断期间）下，代码的运行会受到许多限制。我们现在介绍这些限制，这些限制还会在本书后面多次出现。我们也会多次重复，中断模式下的这些规则必须遵守，否则系统会有大麻烦。

许多动作需要在进程上下文中才能执行。如果处于进程上下文之外（比如在中断模式下），则必须遵守如下规则：

- 不允许访问用户空间。因为没有进程上下文，无法将进程与用户空间关联起来。
- `current` 指针在中断模式下是无效的，不能使用。
- 不能执行睡眠或调度。中断模式代码不可以调用 `schedule` 或者 `sleep_on`；也不能调用任何可能引起睡眠的函数。例如，调用 `kmalloc(..., GFP_KERNEL)` 就不符合本规则。信号量也不能用，因为可能引起睡眠。

内核代码可以通过调用函数 `in_interrupt()` 来判断自己是否正运行于中断模式，该函数无需参数，如果处理器在中断期间运行就返回非 0 值。

当前的任务队列实现还有一个特性，队列中的一个任务可以将自己重新插回到它原先所在的队列。举个例子，定时器队列中的任务可以在运行时将自己插回到定时器队列中去，从而在下一个定时器滴答又再次被运行。这是通过调用 `queue_task` 把自己放回队列来实现的。由于在处理任务队列之前，是先用 `NULL` 指针替换队列的头指针，因此才可能进行不断的重新调度。结果是，一旦旧的队列开始执行，就有一个新的队列被建立。

尽管一遍遍地重新调度同一个任务看起来似乎没什么意义，但有时这也有些用处。例如，步进马达每次移动一步直到目的地，它的驱动程序就可以通过让任务在定时器队列上不断地重新调度自己来实现。其他的例子还有 `jq` 模块，该模块中的打印函数通过重新调度自己来产生输出——结果是利用定时器队列产生多次迭代。

6.4.3 预定义的任务队列

延迟任务执行的最简单方法是使用由内核维护的任务队列。这种队列有好几种，但驱动程序只能使用下面列出的其中三种。任务队列的定义在头文件 `<linux/queue.h>` 中，驱动程序代码需要包含该头文件。

调度器队列

调度器队列在预定义任务队列中比较独特，它运行在进程上下文中，这意味着该队列中的任务可以更多的事情。在 Linux 2.4，该队列由一个专门的内核线程 `keventd` 管理，通过函数 `schedule_task` 访问。在较老的内核版本，没有用 `keventd`，所以该队列（`tq_scheduler`）是直接操作的。

`tq_timer`

该队列由定时器处理程序（定时器嘀哒）运行。因为该处理程序（见函数 `do_timer`）是在中断期间运行的，因此该队列中的所有任务也是在中断期间运行的。

`tq_immediate`

立即队列是在系统调用返回时或调度器运行时得到处理，以便尽可能快地运行该队列。该队列在中断期间得到处理。

还有其它的预定义队列，但驱动程序开发中通常不会涉及到它们。

使用任务队列的一个设备驱动程序的执行流程可见图 6-1。该图演示了设备驱动程序是如何在中断处理程序中将一个函数插入 `tq_immediate` 队列中的。

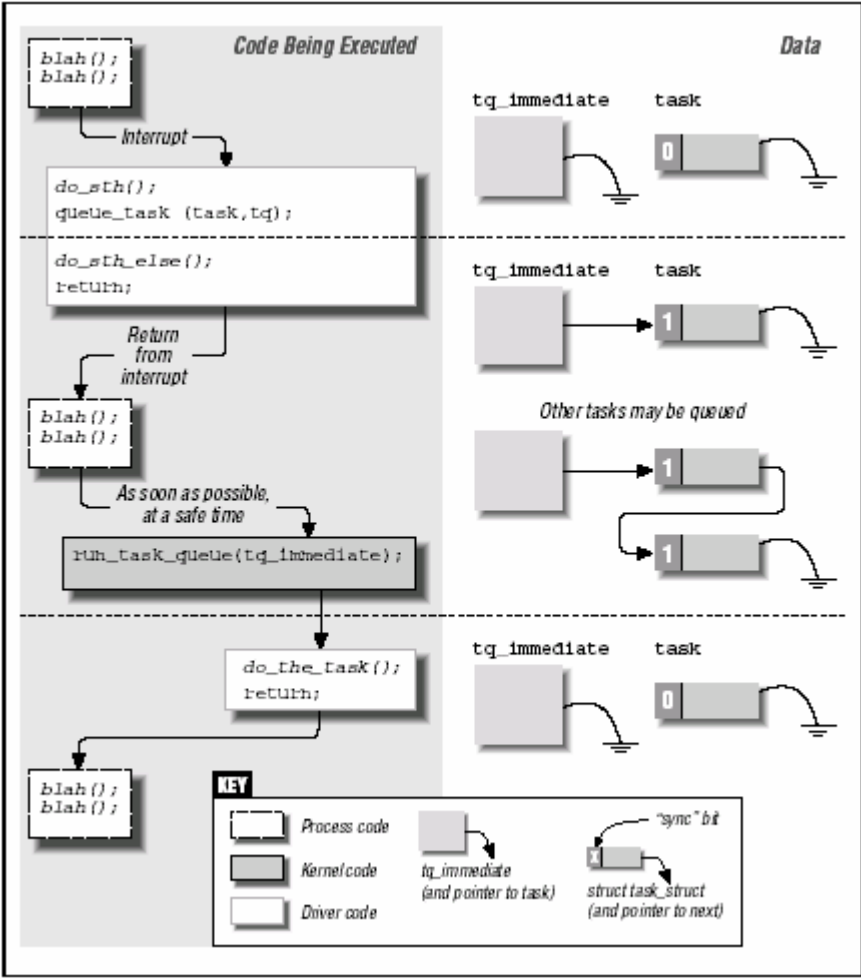


图 6-1: task_queue 的使用流程

示例程序是如何工作的

延迟计算的示例程序包含在 `jiq`(Just In Queue)模块中，本节中抽取了它的部分源码。该模块创建 `/proc` 文件，可以用 `dd` 或者其他工具来读，这点上与 `jit` 模块很相似。读 `jiq` 文件的进程被转入睡眠状态直到缓冲区满*。

睡眠是由一个简单的等待队列处理的，声明为

```
DECLARE_WAIT_QUEUE_HEAD (jiq_wait);
```

缓冲区由不断运行的任务队列来填充。任务队列的每次运行都会要在要填充的缓冲区中添加一个字符串，该字符串记录了当前时间（`jiffies` 值），当前进程以及 `in_interrupt` 的返回值。

填充缓冲区的代码都在 `jiq_print_tq` 函数中，任务队列的每遍运行都要调用它。打印函数没什么意思，不在这里列出，我们还是来看看插入队列的任务的初始化代码：

```
struct tq_struct jiq_task; /* global: initialized to zero */
```

* `/proc` 文件的缓冲区是内存中的一页：4KB，或对应于使用平台的尺寸。

```
/* these lines are in jiq_init() */
jiq_task.routine = jiq_print_tq;
jiq_task.data = (void *)&jiq_data;
```

这里没必要对 `jiq_task` 结构的 `sync` 成员和 `next` 成员清零，因为静态变量已由编译器初始化为零了。

调度器队列

最容易使用的任务队列是调度器（scheduler）队列，因为该队列中的任务不会在中断模式运行，因此可以做更多事，特别是它们还能睡眠。内核中有多处使用该队列完成各种任务。

在内核 2.4.0-test11，实际实现调度器队列的任务队列被内核的其余部分隐藏了。使用这个队列的代码必须调用 `schedule_task` 把任务放入队列，而不能直接使用 `queue_task`：

```
int schedule_task(struct tq_struct *task);
```

其中的 `task` 当然就是要调度的任务。返回值直接来自 `queue_task`：如果任务不在队列中就返回非零。

再提一次，从版本 2.4.0-test11 开始内核使用了一个特殊进程 `keventd`，它唯一的任务就是运行 `scheduler` 队列中的任务。`keventd` 为它运行的任务提供了可预期的进程上下文，而不象以前的实现，任务是在完全随机的进程上下文中运行的。

对于 `keventd` 的执行有几点是值得牢记的。首先，这个队列中的任务可以睡眠，一些内核代码就使用了这一优点。但是，好的代码应该只睡眠很短的时间，因为在 `keventd` 睡眠的时候，调度器队列中的其他任务就不会再运行了。还有一点需要牢记，你的任务是和其它任务共享调度器队列，这些任务也可以睡眠。正常情况下，调度器队列中的任务会很快运行（也许甚至在 `schedule_task` 返回之前）。但如果其它某个任务睡眠了，轮到你的任务执行时，中间流逝的时间会显得很久。所以那些有严格的执行时限的任务应该使用其它队列。

`/proc/jiqsched` 文件是使用调度器队列的示例文件，该文件对应的 `read` 函数以如下的方式将任务放进队列中：

```
int jiq_read_sched(char *buf, char **start, off_t offset,
                  int len, int *eof, void *data)
{
    jiq_data.len = 0;                /* nothing printed, yet */
    jiq_data.buf = buf;              /* print in this place */
    jiq_data.jiffies = jiffies;      /* initial time */

    /* jiq_print will queue_task() again in jiq_data.queue */
    jiq_data.queue = SCHEDULER_QUEUE;

    schedule_task(&jiq_task);        /* ready to run */
    interruptible_sleep_on(&jiq_wait); /* sleep till completion */

    *eof = 1;
    return jiq_data.len;
}
```

读取 `/proc/jiqsched` 文件产生如下输出：

```
time delta interrupt pid cpu command
601687 0 0 2 1 keventd
601687 0 0 2 1 keventd
601687 0 0 2 1 keventd
601687 0 0 2 1 keventd
601687 0 0 2 1 keventd
601687 0 0 2 1 keventd
601687 0 0 2 1 keventd
601687 0 0 2 1 keventd
601687 0 0 2 1 keventd
```

上面的输出中，**time** 域是任务运行时的 **jiffies** 值，**delta** 是自任务最近一次运行以来 **jiffies** 的增量，**interrupt** 是 `in_interrupt` 函数的输出，**pid** 是运行进程的 ID，**cpu** 是正被使用的 CPU 的编号（在单处理器系统中始终为 0），**command** 是当前进程正在运行的命令。

在这个例子中，我们看到，任务总是在 `keventd` 进程中运行，而且运行得非常快，一个不断把自己重复提交给调度器队列的任务可以在一次定时器滴答中运行数百甚至数千次。即使是在一个负载很重的系统，调度器队列的延迟也是非常小的。

定时器队列

定时器队列的使用方法和调度器队列不同，它（`tq_timer`）是可以直接操作的。还有，定时器队列是在中断模式下执行的。另外，该队列一定会在下一个时钟滴答被运行，这消除了可能因系统负载造成的延迟。

示例代码使用定时器队列实现了 `/proc/jiqtimer`。使用这个队列要用到 `queue_task` 函数。

```
int jiq_read_timer(char *buf, char **start, off_t offset,
                  int len, int *eof, void *data)
{
    jiq_data.len = 0;          /* nothing printed, yet */
    jiq_data.buf = buf;        /* print in this place */
    jiq_data.jiffies = jiffies; /* initial time */
    jiq_data.queue = &tq_timer; /* reregister yourself here */

    queue_task(&jiq_task, &tq_timer); /* ready to run */
    interruptible_sleep_on(&jiq_wait); /* sleep till completion */

    *eof = 1;
    return jiq_data.len;
}
```

下面是在我的系统在编译一个新内核时运行命令 `head /proc/jiqtimer` 输出的结果：

```
time delta interrupt pid cpu command
45084845 1 1 8783 0 cc1
45084846 1 1 8783 0 cc1
45084847 1 1 8783 0 cc1
45084848 1 1 8783 0 cc1
45084849 1 1 8784 0 as
45084850 1 1 8758 1 cc1
45084851 1 1 8789 0 cpp
45084852 1 1 8758 1 cc1
45084853 1 1 8758 1 cc1
```



```
45084854 1 1 8758 1 cc1
45084855 1 1 8758 1 cc1
```

注意，这次在任务的每次执行之间正好都经过了一个定时器滴答，而且正在运行的可能是任意一个进程。

立即队列

最后一个可由模块代码使用的预定义队列是立即队列。这个队列通过底半处理机制运行，所以要用它还需额外的步骤。底半处理程序只有在通知内核需要它运行时才会运行，这是通过“标记”底半部完成的。对于 `tq_immediate`，必须调用 `mark_bh(IMMEDIATE_BH)`。注意必须在任务插入队列后才能调用 `mark_bh`，否则可能在任务还没加入队列时内核就开始运行队列了。

立即队列是系统处理得最快的队列——它反应最快并且在中断期间运行。立即队列既可以由调度器执行，也可以在一个进程从系统调用返回时被尽快地执行。典型的输出大致如下：

```
time delta interrupt pid cpu command
45129449 0 1 8883 0 head
45129453 4 1 0 0 swapper
45129453 0 1 601 0 X
45129453 0 1 601 0 X
45129453 0 1 601 0 X
45129453 0 1 601 0 X
45129454 1 1 0 0 swapper
45129454 0 1 601 0 X
45129454 0 1 601 0 X
45129454 0 1 601 0 X
45129454 0 1 601 0 X
45129454 0 1 601 0 X
45129454 0 1 601 0 X
45129454 0 1 601 0 X
45129454 0 1 601 0 X
```

显然该队列不能用于延迟任务的执行——它是个“立即”队列。相反，它的目的是使任务尽快地得以执行，但是要在“安全的时间”内。这对中断处理非常有用，因为它提供了在实际的中断处理程序之外执行处理程序代码的一个入口点，例如接收网络包的机制就类似这样。

注意不要把任务重新注册到立即队列中（尽管 `/proc/jiqimmed` 为了演示而这么做），这种做法没什么好处，而且在某些版本 / 平台的搭配上运行时会锁死计算机。因为在有些实现中会不断重运行立即队列直到它空为止。这种情况出现过，例如在 PC 上运行 2.0 版本的时候。

6.4.4 运行自己的工作队列

声明新的任务队列并不困难。驱动程序可以随意地声明一个甚至多个新任务队列。这些队列的使用和我们前面讨论过的预定义队列差不多。

与预定义队列不同的是，内核不会自动处理定制的任务队列。定制的任务队列要由程序员自己维护，并安排运行方法。

下面的宏声明一个定制队列并扩展为变量声明。最好把它放在文件开头的地方，所有函数的外面：

```
DECLARE_TASK_QUEUE(tq_custom);
```

声明完队列，就可以调用下面的函数对任务进行排队。上面的宏和下面的调用相匹配：

```
queue_task(&custom_task, &tq_custom);
```

当要运行累积的任务队列时，执行下面一行，运行 `tq_custom` 队列：

```
run_task_queue(&tq_custom);
```

如果现在想测试定制的任务队列，则需要在某个预定义的队列中注册一个函数来触发这个队列。尽管看起来象绕了弯路，但其实并非如此。当需要累积任务以便同时得到执行时，定制的任务队列是非常有用的，尽管需要用另一个队列来决定这个“同时”。

6.4.5 Tasklets

就在 2.4 内核发布之前，开发者们增加了一种用于内核任务延迟的新机制。这种新机制称为 **tasklet**，现在是实现底半任务的推荐方法。实际上，现在的底半处理程序本身就是用 **tasklet** 实现的。

tasklets 在很多方面类似任务队列。它们都是把任务延迟到安全时间执行的一种方式，都在中断期间运行。象任务队列一样，即使被调度多次，**tasklet** 也只运行一次，不过 **tasklet** 可以在 SMP 系统上和其它（不同的）**tasklet** 并行地运行。在 SMP 系统上，**tasklet** 还被确保在第一个调度它的 CPU 上运行，因为这样可以提供更好的高速缓存行为，从而提高性能。

每个 **tasklet** 都与一个函数相联系，当 **tasklet** 要运行的时候该函数被调用。该函数只有一个 **unsigned long** 类型的参数，这多少使一些内核开发者的生活变得轻松；但对那些宁愿传递一个指针的开发人员来说肯定是增加了苦恼。把 **long** 类型的参数转换为一个指针类型在所有已支持的平台上都是安全的操作，在内存管理中（第 13 章讨论）更是普遍使用。这个 **tasklet** 的函数的类型是 **void**，无返回值。

tasklet 的实现部分在 `<linux/interrupt.h>` 中，它自己必须用下列中的一种来声明：

```
DECLARE_TASKLET(name, function, data);
```

用指定的名字 **name** 声明一个 **tasklet**，在该 **tasklet** 执行时（后面要讲到），指定的函数 **function** 被调用，传递的参数值为 **(unsigned long) data**。

```
DECLARE_TASKLET_DISABLED(name, function, data);
```

和上面一样声明一个 **tasklet**，不过初始状态是“禁止的”，意味着可以被调度但不会执行，直到被“使能”以后才能执行。

用 2.4 的头文件编译 **jiq** 示例驱动程序，可以实现 `/proc/jiqtasklet`，它和其他的 **jiq** 入口工作类似，只不过使用了 **tasklet**。我们并没有在 `sysdep.h` 中为旧版本模拟实现 **tasklet**。该模块如下定义它的 **tasklet**：

```
void jiq_print_tasklet (unsigned long);
DECLARE_TASKLET (jiq_tasklet, jiq_print_tasklet, (unsigned long)
    &jiq_data);
```

当驱动程序要调度一个 `tasklet` 运行的时候，它调用 `tasklet_schedule`：

```
tasklet_schedule(&jiq_tasklet);
```

一旦一个 `tasklet` 被调度，它就肯定会在一个安全时间运行一次（如果已经被使能）。`tasklet` 可以重新调度自己，其方式和任务队列一样。在多处理器系统上，一个 `tasklet` 无须担心自己会在多个处理器上同时运行，因为内核采取了措施确保任何 `tasklet` 都只能在一个地方运行。但是，如果驱动程序中实现了多个 `tasklet`，那么就可能会有多个 `tasklet` 在同时运行。在这种情况下，需要使用自旋锁来保护临界区代码（信号量是可以睡眠的，因为 `tasklet` 是在中断期间运行，所以不能用于 `tasklet`）。

`/proc/jiqtasklet` 的输出如下：

```
time delta interrupt pid cpu command
45472377 0 1 8904 0 head
45472378 1 1 0 0 swapper
45472379 1 1 0 0 swapper
45472380 1 1 0 0 swapper
45472383 3 1 0 0 swapper
45472383 0 1 601 0 X
45472383 0 1 601 0 X
45472383 0 1 601 0 X
45472383 0 1 601 0 X
45472389 6 1 0 0 swapper
```

注意这个 `tasklet` 总是在同一个 CPU 上运行，即使输出来自双 CPU 系统。

`tasklet` 子系统提供了一些其它的函数，用于高级的 `tasklet` 操作：

```
void tasklet_disable(struct tasklet_struct *t);
```

这个函数禁止指定的 `tasklet`。该 `tasklet` 仍然可以用 `tasklet_schedule` 调度，但执行被推迟，直到重新被使能。

```
void tasklet_enable(struct tasklet_struct *t);
```

使能一个先前被禁止的 `tasklet`。如果该 `tasklet` 已经被调度，它很快就会运行（但一从 `tasklet_enable` 返回就直接运行）。

```
void tasklet_kill(struct tasklet_struct *t);
```

该函数用于对付那些无休止地重新调度自己的 `tasklet`。`tasklet_kill` 把指定的 `tasklet` 从它所在的所有队列删除。为避免与正重新调度自己的 `tasklet` 产生竞态，该函数会等到 `tasklet` 执行，然后再把它移出队列。这样就可以确保 `tasklet` 不会在中途被打断。然而，如果目标 `tasklet` 当前既没有运行也没有重新调度自己，`tasklet_kill` 会挂起。`tasklet_kill` 不能在中断期间被调用。

6.5 内核定时器

内核中最终的计时资源还是定时器。定时器用于调度函数（定时器处理程序）在未来某个特定时间执行。与任务队列和 `tasklet` 不同，我们可以指定某个函数在未来何时被调用，但不能确定队列中的会在何时执行。另外，内核定时器与任务队列相似的是，注册的处理函数只执行一次——定时器

不是循环执行的。

有时候要执行的操作不在任何进程上下文内，比如关闭软驱马达和中止某个耗时的关闭操作，在这些情况下，延迟从 `close` 调用的返回对于应用程序不合适，而且这时也没有必要使用任务队列，因为已排队的任务在必要的时间过去之后还要不断重新注册自己。

这时，使用定时器就方便得多。注册处理函数一次，当定时器超时时内核就调用它一次。这种处理一般较适合由内核完成，但有时驱动程序也需要，就象软驱马达的例子。

内核定时器被组织成双向链表。这意味着我们可以加入任意多的定时器。定时器包括它的超时值(单位是 `jiffies`)和超时时要调用的函数。定时器处理程序需要接收一个参数，该参数和处理程序函数指针本身一起存放在一个数据结构中。

定时器的数据结构如下，取自头文件 `<linux/timer.h>`:

```
struct timer_list {
    struct timer_list *next;          /* never touch this */
    struct timer_list *prev;          /* never touch this */
    unsigned long expires;            /* the timeout, in jiffies */
    unsigned long data;               /* argument to the handler */
    void (*function)(unsigned long); /* handler of the timeout */
    volatile int running;             /* added in 2.4; don't touch */
};
```

定时器的超时值是个 `jiffies` 值，当 `jiffies` 值大于等于 `timer->expires` 时，`timer->function` 函数就要运行。`timeout` 值是个绝对数值，它通常是用 `jiffies` 的当前值加上需要的延迟量计算出来的。

一旦完成对 `timer_list` 结构的初始化，`add_timer` 函数就将它插入一张有序链表中，该链表每秒钟会被查询 100 次左右。即使某些系统（如 Alpha）使用更高的时钟中断频率，也不会更频繁地检查定时器列表。因为如果增加定时器分辨率，遍历链表的代价也会相应增加。

用于操作定时器的有如下函数：

```
void init_timer(struct timer_list *timer);
```

该内联函数用来初始化定时器结构。目前，它只将 `prev` 和 `next` 指针清零（在 SMP 系统上还有运行标志）。强烈建议程序员使用该函数来初始化定时器而不要显式地修改结构内的指针，以保证向前兼容。

```
void add_timer(struct timer_list *timer);
```

该函数将定时器插入活动定时器的全局队列。

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

如果要更改定时器的超时时间则调用它，调用后定时器使用新的 `expires` 值。

```
int del_timer(struct timer_list *timer);
```

如果需要在定时器超时前将它从列表中删除，则应调用 `del_timer` 函数。但当定时器超时时，系统会自动地把它从链表中删除。

```
int del_timer_sync(struct timer_list *timer);
```

该函数的工作类似 `del_timer`，不过它还确保了当它返回时，定时器函数不在任何 CPU 上运行。当一个定时器函数在无法预料的时间运行时，使用 `del_timer_sync` 可避免产生竞态，大多数情况下都应该使用这个函数。调用 `del_timer_sync` 时还必须保证定时器函数不会使用 `add_timer` 把它自己重新加入队列。

使用定时器的一个例子是 `jiq` 示例模块。`/proc/jitimer` 文件使用一个定时器来产生两行数据，所使用的打印函数和前面任务队列中用到的是同一个。第一行数据是由 `read` 调用产生的（由查看 `/proc/jitimer` 的用户进程调用），而第二行是 1 秒后后定时器函数打印出的。

用于 `/proc/jitimer` 文件的代码如下所示：

```
struct timer_list jiq_timer;

void jiq_timedout(unsigned long ptr)
{
    jiq_print((void *)ptr);          /* print a line */
    wake_up_interruptible(&jiq_wait); /* awaken the process */
}

int jiq_read_run_timer(char *buf, char **start, off_t offset,
                       int len, int *eof, void *data)
{
    jiq_data.len = 0;      /* prepare the argument for jiq_print() */
    jiq_data.buf = buf;
    jiq_data.jiffies = jiffies;
    jiq_data.queue = NULL; /* don't requeue */

    init_timer(&jiq_timer);          /* init the timer structure */
    jiq_timer.function = jiq_timedout;
    jiq_timer.data = (unsigned long)&jiq_data;
    jiq_timer.expires = jiffies + HZ; /* one second */

    jiq_print(&jiq_data); /* print and go to sleep */
    add_timer(&jiq_timer);
    interruptible_sleep_on(&jiq_wait);
    del_timer_sync(&jiq_timer); /* in case a signal woke us up */

    *eof = 1;
    return jiq_data.len;
}
```

运行命令 `head /proc/jitimer` 得到如下输出结果：

time	delta	interrupt	pid	cpu	command
45584582	0	0	8920	0	head
45584682	100	1	0	1	swapper

从输出中可以发现，打印出最后一行的定时器函数是在中断模式运行的。

可能看起来有点奇怪的是，定时器总是可以正确地超时，即使处理器正在执行系统调用。我在前面曾提到，运行在内核态的进程不会被调出，但时钟中断是个例外，它与当前进程无关，独立完成了自己的任务。读者可以试试同时在后台读 `/proc/jitbusy` 文件和在前台读 `/proc/jitimer` 文件会发生什么。这时尽管看起来系统似乎被忙等待的系统调用给锁死住了，但定时器队列和内核定时器还是

能不断得到处理。

因此，定时器是另一个竞态资源，即使是在单处理器系统中。定时器函数访问的任何数据结构都要进行保护以防止并发访问，保护方法可以用原子类型（第 10 章讲述）或者用自旋锁。

删除定时器时也要小心避免竞态。考虑这样一种情况：某一模块的定时器函数正在一个处理器上运行，这时在另一个处理器上发生了相关事件（文件被关闭或模块被删除）。结果是，定时器函数等待一种已不再出现的状态，从而导致系统崩溃。为避免这种竞态，模块中应该用 `del_timer_sync` 代替 `del_timer`。如果定时器函数还能够重新启动自己的定时器（这是一种普遍使用的模式），则应该增加一个“停止定时器”标志，并在调用 `del_timer_sync` 之前设置。这样定时器函数执行时就可以检查该标志，如果已经设置，就不会用 `add_timer` 重新调度自己了。

还有一种会引起竞态的情况是修改定时器：先用 `del_timer` 删除定时器，再用 `add_timer` 加入一个新的以达到修改目的。其实在这种情况下简单地使用 `mod_timer` 是更好的方法。

6.6 向后兼容性

任务队列和时间机制的实现多年来基本保持着相对的稳定。不过，还是有一些值得注意的改进。

`sleep_on_timeout`、`interruptible_sleep_on_timeout` 和 `schedule_timeout` 这几个函数是在 2.2 版本内核才加入的。在使用 2.0 的时期，超时值是通过 `task` 结构中的一个变量（`timeout`）处理的。作一个比较，现在的代码是这样进行调用的：

```
interruptible_sleep_on_timeout(my_queue, timeout);
```

而以前则是如下这样编写：

```
current->timeout = jiffies + timeout;
interruptible_sleep_on(my_queue);
```

头文件 `sysdep.h` 为 2.4 以前的内核重建了 `schedule_timeout`，所以可以在 2.0 和 2.2 版本使用新语法并正常运行：

```
extern inline void schedule_timeout(int timeout)
{
    current->timeout = jiffies + timeout;
    current->state = TASK_INTERRUPTIBLE;
    schedule();
    current->timeout = 0;
}
```

2.0 版本还有另外两个函数可把函数放入任务队列。中断被禁止时可以用 `queue_task_irq` 代替 `queue_task`，这会损失一点性能。`queue_task_irq_off` 更快些，但在任务已经插入队列或正在运行时出错，所以只有在确保这类情况不会发生时才能使用。这两个函数在提升性能方面都没什么好处，从内核 2.1.30 开始把它们去掉了。任何情况下，使用 `queue_task` 都能在所有内核版本下工作。（要注意一点，在 2.2 及其以前内核中，`queue_task` 返回值的类型是 `void`）

2.4 内核之前不存在 `schedule_task` 函数及 `keventd` 进程，使用的是另一个预定义任务队列 `tq_scheduler`。`tq_scheduler` 队列中的任务在 `schedule` 函数中执行，所以总是运行在进程上下文中。然而，“提供”上下文的进程总是不同的，它有可能是当时正被 CPU 调度运行的任何一个进程。`tq_scheduler` 通常有比较大的延迟，特别是对那些会重复提交自己的任务更是如此。`sysdep.h` 在 2.0 和 2.2 系统上对 `schedule_task` 的实现如下：

```
extern inline int schedule_task(struct tq_struct *task)
{
    queue_task(task, &tq_scheduler);
    return 1;
}
```

前面已经提到，2.3 内核系列中增加了 `tasklet` 机制。在此之前，只有任务队列可以用于“立即延迟”的执行。底半处理部分也改动了，不过大多数改动对驱动程序开发人员是透明的。`sysdep.h` 中不再模拟 `tasklet` 在旧内核上的实现，它们对驱动程序操作来说并非严格必要。如果要保持向后兼容，要么编写自己的模拟代码，要么用任务队列代替。

Linux 2.0 中没有 `in_interrupt` 函数，代替它的是一个全局变量 `intr_count`，记录着正运行的中断处理程序的个数。查询 `intr_count` 的语法和调用 `in_interrupt` 差不多，所以在 `sysdep.h` 中保持兼容性是很容易实现的。

函数 `del_timer_sync` 在内核 2.4.0-test2 之前还没有引入。`sysdep.h` 中进行了一些替换，以便使用旧的内核头文件也可以编译。2.0 版本内核也没有 `mod_timer`。这个问题也在兼容性头文件中得以解决。

6.7 快速参考

本章引入如下符号：

```
#include <linux/param.h>
HZ
```

`HZ` 符号指出每秒钟产生的时钟滴答数。

```
#include <linux/sched.h>
volatile unsigned long jiffies
```

`jiffies` 变量每个时钟滴答后加 1，因此它每秒增加 `HZ` 次。

```
#include <asm/msr.h>
rdtsc(low,high);
rdtscl(low);
```

读取时间戳计数器或其低半部分。头文件和宏是 PC 类处理器特有的，其它平台可能需要用汇编语句实现类似功能。

```
extern struct timeval xtime;
```

当前时间，由最近一次定时器滴答计算出。

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
void get_fast_time(struct timeval *tv);
```

这两个函数返回当前时间。前者具有很高的分辨率，后者更快些，但分辨率较差。

```
#include <linux/delay.h>
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

这两个函数引入整数数目的微秒或毫秒的延迟。前一个应用于不超过 1 毫秒的延迟；后一个使用时要格外慎重，因为它们使用的都是忙等待循环。

```
int in_interrupt();
```

如果处理器正在中断模式运行，就返回非 0 值。

```
#include <linux/tqueue.h>
DECLARE_TASK_QUEUE(variablename);
```

该宏声明一个新的变量并作初始化。

```
void queue_task(struct tq_struct *task, task_queue *list);
```

该函数注册一个稍后执行的任务。

```
void run_task_queue(task_queue *list);
```

该函数运行任务队列。

```
task_queue tq_immediate, tq_timer;
```

这些预定义的任务队列在内核调度新的进程前（`tq_immediate`）尽快地，或者在每个时钟滴答后（`tq_timer`）得到执行。

```
int schedule_task(struct tq_struct *task);
```

调度一个任务在调度器队列运行。

```
#include <linux/interrupt.h>
DECLARE_TASKLET(name, function, data)
DECLARE_TASKLET_DISABLED(name, function, data)
```

声明一个 `tasklet` 结构，运行时它将调用指定的函数 `function`（并将指定参数 `unsigned long data` 传递给函数）。第二种形式把 `tasklet` 初始化为禁止状态，直到明确地使能后 `tasklet` 才能运行。

```
void tasklet_schedule(struct tasklet_struct *tasklet);
```

调度指定的 `tasklet` 运行。如果该 `tasklet` 没有被禁止，它将在调用了 `tasklet_schedule` 的 CPU 上很快得到执行。

```
tasklet_enable(struct tasklet_struct *tasklet);
tasklet_disable(struct tasklet_struct *tasklet);
```

这两个函数分别使能和禁止指定的 `tasklet`。被禁止的 `tasklet` 可以被调度，但只有使能后才能运行。

```
void tasklet_kill(struct tasklet_struct *tasklet);
```

使一个正“无休止重新调度”的 `tasklet` 停止执行。该函数可以阻塞，而且不能在中断期间调用。

```
#include <linux/timer.h>
void init_timer(struct timer_list * timer);
```

该函数初始化新分配的定时器。

```
void add_timer(struct timer_list * timer);
```

该函数将定时器插入待处理定时器的全局队列。


```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

该函数用于更改一个已调度的定时器结构中的超时时间。

```
int del_timer(struct timer_list * timer);
```

del_timer 函数将定时器从待处理定时器队列中删除。如果队列中存在该定时器，**del_timer** 返回 1，否则返回 0。

```
int del_timer_sync(struct timer_list *timer);
```

该函数类似 **del_timer**，但是确保定时器函数当前不在其它 CPU 上运行。

第 7 章 获取内存



到目前为止，我们已经使用过 `kmalloc` 和 `kfree` 分配和释放内存，但 Linux 内核实际提供了更加丰富的内存分配原语集。本章我们将会介绍设备驱动程序中使用内存的一些其它方法，还会介绍如何最好地利用系统内存资源。我们不会讨论不同体系结构是如何实际管理内存的。因为内核为设备驱动程序提供了一致的内存管理接口，模块不需要涉及分段，分页等问题。另外，本章也不会描述内存管理的内部细节，这些问题将留到第 13 章的“Linux 的内存管理”一节讨论。

7.1 `kmalloc` 函数的内幕

`kmalloc` 内存分配引擎是个功能很强的工具，由于和 `malloc` 相似，学习它也很容易。除非被阻塞，这个函数运行得很快，而且不对获取的内存空间清零，也就是说，分配给它的区域仍然保持着原有的数据。分配的区域在物理内存中也是连续的。在下面几节中，我们将详细讨论 `kmalloc` 函数，读者可以把它和后面将要讨论的其它一些内存分配技术作个比较。

7.1.1 `flags` 参数

`kmalloc` 的第一个参数是要分配的块的大小，第二个参数是分配标志 (`flags`)，更有意思的是，它能够以多种方式控制 `kmalloc` 的行为。

最常用的标志是 `GFP_KERNEL`，它表示内存分配（最终总是调用 `get_free_pages` 来实现实际的分配，这就是 `GFP_` 前缀的由来）是代表运行在内核空间的进程执行的。换句话说，这意味着调用它的函数正代表某个进程执行系统调用。使用 `GFP_KERNEL` 允许 `kmalloc` 在空闲内存较少时把当前进程转入睡眠以等待一个页面。因此，使用 `GFP_KERNEL` 分配内存的函数必须是可重入的。在当前进程睡眠时，内核会采取适当的行动，或者是把缓冲区的内容刷到硬盘上，或者是从一个用户进程换出内存，以获取一个内存页面。

`GFP_KERNEL` 分配标志并不是始终适用，有时 `kmalloc` 是在进程上下文之外调用的，例如在中断处理程序、任务队列或者内核定时器中调用。这种情况下当前进程就不应该睡眠，驱动程序则应该换用 `GFP_ATOMIC` 标志。内核通常会为原子性的分配预留一些空闲页面。使用 `GFP_ATOMIC` 标志时，`kmalloc` 甚至可以用掉最后一个空闲页面。不过如果连最后一页都没有了，分配就返回失败。

除了 `GFP_KERNEL` 和 `GFP_ATOMIC` 外，还有一些其他的标志可用于替换或补充这两个标志，不过这两个标志已经可以满足大多数驱动程序的需要了。所有的标志都定义在 `<linux/mm.h>` 中：单个标志使用两个下划线作为前缀，比如 `__GFP_DMA`；而标志的组合则没有这个前缀，并且有时称为“分配优先级”。

`GFP_KERNEL`

内核内存的通常分配方法，可能引起睡眠。

`GFP_BUFFER`

用于管理高速缓冲区，这个优先级允许调用进程睡眠。它不同于 `GFP_KERNEL` 的地方在于很少通过把脏页刷到硬盘来获取空闲内存；其目的是避免 I/O 子系统需要内存的时候造成死锁。

`GFP_ATOMIC`

用于在中断处理程序或其它运行于进程上下文之外的代码中分配内存，不会睡眠。

`GFP_USER`

代表用户分配内存。可能会睡眠，是一个低优先级的请求。

`GFP_HIGHUSER`

很象 `GFP_USER`，不过如果有高端内存的话就从那里分配。高端内存的问题在下一小节讨论。

`__GFP_DMA`

这个标志为设备的 DMA（直接内存访问）数据传输申请内存。它的实际意义是与特定平台相关的，可以和 `GFP_KERNEL` 或 `GFP_ATOMIC` 经过“或”运算后一起使用。

`__GFP_HIGHMEM`

该标志申请高端内存，并且与平台相关，在不支持它的平台上无效。它是 `GFP_HIGHUSER` 掩码的一部分，在别的地方几乎没什么用。

内存区段

`__GFP_DMA` 和 `__GFP_HIGHMEM` 的使用与平台相关，尽管在所有平台上都可以用这两个标志。

2.4 内核把内存分为三个区段：可用于 DMA 的内存、常规内存以及高端内存。通常的内存分配都发生在常规内存区，但通过设置上面介绍过的标志也可以请求在其它区段中分配。其思路是每种计算平台都必须知道如何把自己特定的内存范围归到这三个区段中，而不是认为所有的 RAM 都一样。

可用于 DMA 的内存区段是唯一可以和外设进行 DMA 数据传输的内存。这种限制是为了避免用于连接外设到处理器的地址总线和用于访问内存的地址总线发生冲突。例如，x86 平台上的 ISA 总线设备只能访问 0 到 16 MB 地址的内存。其它平台上也有类似限制，不过通常没有 ISA 总线那么严格*。

高端内存是需要一些特殊处理才能访问的内存。这种处理在内核内存管理中直到 2.3 开发版本中实现对 Pentium II 虚拟内存扩展的支持时才出现，它可以访问到 64 GB 的物理内存。高端内存

* 有趣的是这种限制仅限于 ISA 总线，插到 PCI 总线的 x86 设备可以在所有的常规内存区段进行 DMA 传输。

的概念只在 x86 和 SPARC 平台上才有，而且这两者的实现也不一样。

当一个新页面应 `kmalloc` 的要求被分配时，内核会创建一个内存区段的列表以供搜索。如果指定了 `_GFP_DMA` 标志，则只有 DMA 区段会被搜索：如果低地址段上没有可用内存，分配就会失败。如果没有指定特定的标志，常规区段和 DMA 区段都会被搜索；而如果设置了 `_GFP_HIGHMEM` 标志，所有三个区段都会被搜索以获取一个空闲页。

如果平台上没有高端内存的概念，或在内核配置中被禁止，`_GFP_HIGHMEM` 定义为 0，不起任何作用。

内存区段机制是在 `mm/page_alloc.c` 中实现的，区段的初始化是平台相关的，通常在对应的 `arch` 目录下的 `mm/init.c` 中。第 13 章还会再次讨论这个问题。

7.1.2 size 参数

内核负责管理系统物理内存，物理内存只能按页面进行分配。其结果是，`kmalloc` 和典型的用户空间的 `malloc` 的实现有很大的差别。简单的基于堆的内存分配技术会遇到麻烦，因为页面边界的处理成为一个很棘手的问题。因此内核使用了特殊的基于页的分配技术，以最佳地利用系统 RAM。

Linux 处理内存分配的方法是，创建一系列的内存对象池，每个池中的内存块大小固定一样。处理分配请求时，就直接在包含有足够大的内存块的池中传递一个整块给请求者。内存管理方法相当复杂，其细节对设备驱动程序开发人员也不重要，所以就不仔细讨论了。毕竟我们可以修改实现方法却不会影响提供给内核其它部分的接口，比如在 2.1.38 内核中就修改过。

驱动程序开发人员应该记住一点，就是内核只能分配一些预定义的固定大小的字节数组。如果申请任意数量的内存，那么得到的很可能会多一些，最多会到申请数量的两倍。另外，程序员应该记住 `kmalloc` 处理的最小的内存块是 32 或者 64，到底是哪个则依赖于当前体系结构使用的页面大小。

这些预定义的内存大小通常是 2 的某次方。在 2.0 内核，使用的数字实际上稍小于 2 的次方，因为管理系统加入了控制标志。记住这一点有助于更有效地利用内存。例如，在 Linux 2.0 下需要一个 2000 字节左右的缓冲区，那么最好是申请 2000 字节而不是 2048。在 2.1.38 版本以前的内核中，申请恰好是 2 的幂次的内存空间是最糟糕的情况，内核会分配两倍于你所申请大小的内存。这也就是为什么 `scull` 每次都是使用 4000 字节而不是 4096 的原因。

可以在文件 `mm/kmalloc.c` (2.0 内核) 或者 `mm/slab.c` (当前内核) 中找到预定义的分配块大小的确切数值，但要记住它们可能会未经通知而再次改变。`scull` 中使用了一个技巧，即每次都分配小于 4 K 的内存，这在所有 2.x 的内核中都能很好地工作，但不能保证将来也是最佳的。

任何情况下，`kmalloc` 可以分配的最大内存是 128 KB——在 2.0 内核中还要稍微少一点。但是如果需要更多内存，那么有下面介绍的比 `kmalloc` 更好的方法。

7.2 后备式高速缓存

设备驱动程序常常会反复地分配很多同一大小的内存块。既然内核已经维护了一组拥有同一大小内

存块的内存池，为什么不为这些反复使用的块增加某些特殊的内存池呢？实际上，内核的确实实现了这种后备式的高速缓存（lookaside cache）。设备驱动程序通常不会涉及这种使用后备式高速缓存的内存行为，但也有例外，Linux 2.4 的 USB 和 ISDN 就使用了高速缓存。

Linux 内存高速缓存具有类型 `kmem_cache_t`，通过调用 `kmem_cache_create` 创建：

```
kmem_cache_t * kmem_cache_create(const char *name, size_t size,
    size_t offset, unsigned long flags,
    void (*constructor)(void *, kmem_cache_t *,
        unsigned long flags),
    void (*destructor)(void *, kmem_cache_t *,
        unsigned long flags) );
```

该函数创建一个新的高速缓存对象，其中可以容纳任意数目的内存区域。这些区域的大小都相同，由 `size` 参数指定。参数 `name` 与这个高速缓存相关联，功能是保管一些信息以便追踪问题，通常它就设置为将要缓存的结构类型的名字，包括结束符在内，`name` 的最大长度是 20 个字符。

`offset` 参数是页面中第一个对象的偏移量，它可以用来确保对已分配的象进行某种特殊的对齐，但是最常用的就是 0，表示使用默认值。`flags` 控制如何完成分配，是一个位掩码，可取的值如下：

SLAB_NO_REAP

设置这个标志可以保护高速缓存在系统寻找内存的时候不会被减少。通常不需要设置这个标志。

SLAB_HWCACHE_ALIGN

这个标志要求所有数据对象跟缓存线（cache line）对齐，实际的操作则依赖于宿主平台的 cache 设计。设置它通常是个好的选择。

SLAB_CACHE_DMA

这个标志要求每个数据对象都从可用于 DMA 的内存区段中分配。

`constructor` 和 `destructor` 参数是可选的函数（但是不能只有 `constructor` 而没有 `destructor`）；前者用于初始化新分配的对象，而后者用于“清除”对象——在内存空间整个释放给系统之前。

`constructor` 和 `destructor` 很有用，不过使用时有一些限制。`constructor` 函数是当分配供一组对象使用的内存时调用的，因为这些内存中可能会包含好几个对象，`constructor` 函数可能会被调用多次。不能认为分配一个对象后随之就是一次 `constructor` 调用。类似的，`destructor` 函数也有可能不是在一个对象释放后立即就调用，而是在将来的某个未知的时间才调用。`constructor` 和 `destructor` 可能允许也可能不允许睡眠，这要看是否向它们传递了 `SLAB_CTOR_ATOMIC` 标志（CTOR 是 `constructor` 的简写）。

为了简便，程序员可以使用同一个函数同时作为 `constructor` 和 `destructor` 使用；当调用的是一个 `constructor` 函数的时候，分配程序总是传递 `SLAB_CTOR_CONSTRUCTOR` 标志。

一旦某种内存块对象的高速缓存被创建，就可以调用 `kmem_cache_alloc` 从中分配内存：

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
```

这里，参数 `cache` 是先前创建的高速缓存；参数 `flags` 和传递给 `kmalloc` 的相同，并且如果

`kmem_cache_alloc` 自己需要分配更多内存的时候会利用这个参数。

释放一个对象使用 `kmem_cache_free`:

```
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

如果驱动程序代码中和高速缓存有关的部分已经处理完了，（一个典型情况是模块被卸载的时候）这时驱动程序应该释放它的高速缓存，如下所示：

```
int kmem_cache_destroy(kmem_cache_t *cache);
```

这个释放操作只有在从高速缓存中分配的所有对象都已经归还后才能成功。所以一个模块应该检查 `kmem_cache_destroy` 的返回状态，如果是失败，则表明模块中发生了内存泄漏（因为有一些对象被漏掉了）。

使用暂时性的高速缓存带来的另一个好处是内核可以统计高速缓存的使用情况。在内核配置选项中甚至专门有一项用来收集额外的统计信息，但运行时的开销很大。高速缓存的使用统计情况可以从 `/proc/slabinfo` 获得。

7.2.1 基于高速缓存的 `scull`: `sculc`

现在举个例子。`sculc` 是 `scull` 模块的一个缩减版本，只实现了基本的设备——即持久的内存区。与 `scull` 使用 `kmalloc` 不同的是，`sculc` 使用内存高速缓存。数据对象的大小可以在编译或加载时修改，但不能在运行时修改——那样需要创建一个新的内存高速缓存，而这里不必处理那些不需要的细节问题。示例模块不能在 2.0 内核版本编译，因为 2.0 没有内存高速缓存支持，稍后在本章“向后兼容”一节中解释。

`sculc` 是一个完整的例子，可以用于测试。它和 `scull` 的不同之处只有几行代码。它这样分配内存块：

```
/* Allocate a quantum using the memory cache */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] =
        kmem_cache_alloc(sculc_cache, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, sculc_quantum);
}
```

下面的代码释放内存：

```
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        kmem_cache_free(sculc_cache, dptr->data[i]);
kfree(dptr->data);
```

为了支持 `sculc_cache` 的使用，这几行被加入文件中：

```
/* declare one cache pointer: use it for all devices */
kmem_cache_t *sculc_cache;
```

```

/* init_module: create a cache for our quanta */
scullc_cache =
    kmem_cache_create("scullc", scullc_quantum,
        0, SLAB_HWCACHE_ALIGN,
        NULL, NULL); /* no ctor/dtor */
if (!scullc_cache) {
    result = -ENOMEM;
    goto fail_malloc2;
}

/* cleanup_module: release the cache of our quanta */
kmem_cache_destroy(scullc_cache);

```

和 `scull` 相比，`scullc` 的最主要差别是运行速度略有提高，并且对内存的利用更好。由于数据对象是从内存池中分配的，而内存池中所有内存块都是同样大小，所以这些数据对象在内存中的位置排列达到了最大程度的密集，相反的，`scull` 的数据对象则会引入不可预测的内存碎片。

7.3 `get_free_page` 和相关函数

如果模块需要分配大块的内存，使用面向页的分配技术会更好些。整页的分配还有其它优点，以后会在第 13 章的“`mmap` 设备操作”一节介绍。

分配页面可使用下面的函数：

`get_zeroed_page`

返回指向新页面的指针并将页面清零。

`_ _get_free_page`

类似于 `get_zeroed_page`，但不清零页面。

`_ _get_free_pages`

分配若干（物理连续的）页面，并返回指向该内存区域第一个字节的指针，不清零页面。

`_ _get_dma_pages`

类似于 `get_free_pages`，但是保证分配的内存可用于 DMA 传输。如果使用 2.2 或更新的内核版本，可以简单地使用 `_ _get_free_pages` 并传递 `_ _GFP_DMA` 标志；如果希望向后兼容 2.0 版本，那么需要调用 `_ _get_dma_pages`。

这些函数的原型如下：

```

unsigned long get_zeroed_page(int flags);
unsigned long _ _get_free_page(int flags);
unsigned long _ _get_free_pages(int flags, unsigned long order);
unsigned long _ _get_dma_pages(int flags, unsigned long order);

```

参数 `flags` 的作用和 `kmalloc` 中的一样；通常使用 `GFP_KERNEL` 或 `GFP_ATOMIC`，也许还会加上 `_ _GFP_DMA` 标志（申请可用于直接内存访问的内存）或者 `_ _GFP_HIGHMEM` 标志（使用高端内存）。参数 `order` 是要申请或释放的页面数的以 2 为底的对数（即 \log_2 ）。例如，`order` 为 0 表示一个页面，`order` 为 3 表示 8 个页面。如果 `order` 太大，而又没有那么大的连续区域可以分配，就会返回失败。在 Linux 2.0 中 `order` 的最大值是 5（32 个页面），最近的版本是 9（512

个页面，大部分平台上是 2 MB)。总之，**order** 越大，分配失败的可能性就越大。

当程序不再需要使用页面，它可以使用下列函数之一来释放它们。第一个函数是一个宏，展开后就是对第二个函数的调用：

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

如果试图释放和先前分配数目不等的页面，内存映射关系会被破坏，随后系统就会出错。

值得强调的是，只要符合和 **kmalloc** 同样的规则，**get_free_pages** 和其它函数可以在任何时间调用。某些情况下函数分配内存时会失败，特别是在使用了 **GFP_ATOMIC** 的时候。因此，调用了这些函数的程序在分配出错时都应提供相应的处理。

如果想冒险的话，可以假定按优先权 **GFP_KERNEL** 调用 **kmalloc** 和底层的 **get_free_pages** 的操作永远不会失败。一般来说这是对的，但有时未必：内存有限的小型系统可能会运行不正常。驱动程序开发人员不应冒这种危险。

尽管 **kmalloc(GFP_KERNEL)** 在没有空闲内存时有时会失败，但内核总是尽可能满足这个内存分配请求。因此，如果分配太多内存，系统的响应性能很容易就会降下来。例如，如果往 **scull** 设备写入大量数据，计算机可能就会死掉；当系统为满足 **kmalloc** 分配请求而试图换出尽可能多的内存页时，就会变得很慢。所有资源都被贪婪的设备所吞噬，计算机很快就变的无法使用了；此时甚至已经无法为解决这个问题而生成新进程了。我们没有在 **scull** 模块中提到这个问题，因为它只是个例子，并不能真正在多用户系统中使用。但作为一个编程者必须要小心，因为模块是特权代码，会带来系统的安全漏洞(比如说，很可能会造成 **DoS** (denial-of-service, 拒绝服务攻击) 安全漏洞。

7.3.1 使用一整页的 **scull**: **scullp**

为了实际测试页面分配，我们编写了 **scullp** 模块。就象前面介绍的 **sculc** 一样，它是一个缩减了的 **scull**。

scullp 分配的页面单位是一整页或数个页：**scullp** 的 **order** 变量默认为 0，可以在编译或加载时指定。

下列代码展示了它如何分配内存：

```
/* Here's the allocation of a single quantum */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] =
        (void *)_get_free_pages(GFP_KERNEL, dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

scullp 中释放内存的代码如下：

```
/* This code frees a whole quantum set */
for (i = 0; i < qset; i++)
```



```
if (dptr->data[i])
    free_pages((unsigned long)(dptr->data[i]),
               dptr->order);
```

从用户的角度看，可以感觉到的差别就是速度快了一些，并且内存利用得更有效，因为不会有内部的内存碎片。我们运行了测试程序，把 4M 字节的数据从 `scull0` 拷贝到 `scull1`，然后再从 `scullp0` 拷贝到 `scullp1`；结果表明处理器在内核空间的使用率有所提高。

但性能提高的并不多，因为 `kmalloc` 也运行得很快。基于页的分配策略的优点实际不在速度上，而是更有效地使用了内存。按页分配不会浪费内存空间，而用 `kmalloc` 函数则会因分配粒度的原因而浪费一定数量的内存。

但是使用 `__get_free_page` 函数的最大优点是这些分配的页面完全属于你，而且在理论上可以通过适当地调整页表将它们合并成一个线性区域。例如，可以允许用户进程对这些单一但互不相关的页面分配得到的内存区域进行 `mmap`。我们将在第 13 章的“`mmap` 设备操作”一节中讨论这种操作，那时将演示 `scullp` 如何提供内存映射，`scull` 不能提供这种操作。

7.4 vmalloc 与相关函数

下面要介绍的内存分配函数是 `vmalloc`，它分配虚拟地址空间的连续区域。尽管这段区域在物理上可能是不连续的(要访问其中的每个页面都必须独立地调用函数 `__get_free_page`)，内核却认为它们在地址上是连续的。`vmalloc` 在发生错误时返回 0 (NULL 地址)，成功时返回一个指针，该指针指向一个线性的、大小最少为 `size` 的内存区域。

该函数及其相关函数 (`ioremap`，严格地说不是一个分配函数，稍后讨论) 的原型如下：

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

void * vmalloc(unsigned long size);
void vfree(void * addr);
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

要强调说明的是，由 `kmalloc` 和 `get_free_pages` 返回的也是虚拟地址。实际值仍然要由 MMU (内存管理单元，通常是 CPU 的组成部分) 处理才能转为物理内存地址^{*}。`vmalloc` 在如何使用硬件上没有区别，区别在于内核如何执行分配任务上。

`kmalloc` 和 `get_free_pages` 使用的虚拟地址范围与物理内存是一一对应的，可能会有基于常量 `PAGE_OFFSET` 的一个位移；函数不需要为该地址段而修改页表。但 `vmalloc` 和 `ioremap` 使用的地址范围完全是虚拟的，每次分配都是通过设置页表来建立虚拟内存区的。

可以通过比较内存分配函数返回的指针来发现这种差别。某些平台上 (如 x86)，`vmalloc` 返回的地址仅仅是比 `kmalloc` 返回的地址高一些。其它平台上 (如 MIPS 和 IA-64)，它们就完全属于不同的地址范围了。`vmalloc` 可以获得的地址在 `VMALLOC_START` 到 `VMALLOC_END` 的范围

^{*} 实际上，某些体系结构定义了保留的虚拟地址段用来寻址物理内存。这种情况下 Linux 内核会利用这一特点，结果是内核地址和 `get_free_pages` 返回的地址都会落在这些内存范围内。这种差别对设备驱动程序和不直接参与内核内存管理子系统的程序都是透明的。

中。这两个符号都在 `<asm/pgtable.h>` 中定义。

用 `vmalloc` 分配得到的地址是不能在微处理器之外使用的，因为它们只有在处理器的内存管理单元之上才有意义。驱动程序需要真正的物理地址时（象外设用以驱动系统总线的 DMA 地址），就不能使用 `vmalloc` 了。使用 `vmalloc` 函数的正确场合是在分配一大块连续的、只在软件中存在的、用于缓冲的内存区域的时候。注意 `vmalloc` 的开销要比 `__get_free_pages` 大，因为它不但获取内存还要建立页表。因此，用 `vmalloc` 函数分配仅仅一页的内存空间是不值得的。

使用 `vmalloc` 函数的一个例子函数是 `create_module` 系统调用，它利用 `vmalloc` 函数来获取被创建模块需要的内存空间。在调用 `insmod` 来重定位模块代码后，接着会调用 `copy_from_user` 函数把模块代码和数据复制到分配而得的空间内。这样，模块看来象是在连续的内存空间内。但检查 `/proc/ksyms` 文件就能发现模块导出的内核符号和内核本身导出的符号分布在不同的内存范围上。

用 `vmalloc` 分配得到的内存空间要用 `vfree` 函数来释放，这就象要用 `kfree` 函数来释放 `kmalloc` 函数分配得到的内存空间一样。

和 `vmalloc` 一样，`ioremap` 也建立新的页表，但和 `vmalloc` 不同的是，`ioremap` 并不实际分配内存。`ioremap` 的返回值是个特殊的虚拟地址，可以用来访问指定的物理内存区域；这个虚拟地址最后要调用 `iounmap` 来释放掉。注意 `ioremap` 的返回值在所有平台上都不能直接使用，而应该使用 `readb` 这类函数。细节可以参考第 8 章的“直接映射的内存”一节。

`ioremap` 更多用于映射（物理）PCI 缓冲区地址到（虚拟）内核空间。例如，可以用来处理 PCI 显示设备的帧缓冲区；该缓冲区通常被映射到高物理地址，超出了系统初始化时建立的页表地址范围。PCI 的细节将在第 15 章的“PCI 接口”一节中讨论。

要注意的是为了保持可移植性，不应把 `ioremap` 返回的地址当作指向内存的指针而直接访问。相反，应该使用 `readb` 或其它 I/O 函数（在第 8 章“使用 I/O 内存”一节介绍）。这是因为在如 Alpha 的一些平台上，由于 PCI 规范和 Alpha 处理器在数据传输方式上的差异，不能直接把 PCI 内存区映射到处理器的地址空间。

对 `vmalloc` 函数可分配的和 `ioremap` 可访问的内存空间大小并没有什么限制，不过为了能检测到程序员犯的一些错误，`vmalloc` 不允许分配超过物理内存大小的内存空间。但是应该记住，调用 `vmalloc` 函数请求过多的内存空间会导致类似调用 `kmalloc` 函数那样的问题。

`ioremap` 和 `vmalloc` 函数都是面向页的（它们都会修改页表）；因此定位或分配的内存空间实际上都会上调为最近的一个页边界。而且，`ioremap` 在 Linux 2.0 的实现中没有考虑重新映射那些不是从页边界开始的物理地址。新内核通过把重新映射的地址向下圆整到页边界并返回在第一个重新映射页面中的位移的方法实现了对这种物理地址的映射。

`vmalloc` 函数的小缺点是它不能在中断时间内使用，因为它的内部实现调用了 `kmalloc(GFP_KERNEL)` 来获取页表的存储空间，因而可能睡眠。但这不是什么问题——如果 `__get_free_page` 函数都还不能满足中断处理程序的需求的话，那还是先修改一下软件设计吧。

7.4.1 使用虚拟地址的 scull: scullv

scullv 模块使用了 **vmalloc**。和 **scullp** 一样，这个模块也是 **scull** 的一个缩减版本，只是使用了不同的分配函数来获取给设备储存数据的内存空间。

该模块每次分配 16 页的内存。这里的内存分配使用了较大的数据块以获取比 **scullp** 更好的性能，并且展示了为什么使用其它分配技术会更耗时。用 `_get_free_pages` 函数来分配一页以上的内存空间容易出错，而且即使成功了也比较慢。前面我们已经看到，用 **vmalloc** 分配几个页时比其他函数要快一些，但由于存在建立页表的开销，只分配一页时却会慢一些。**scullv** 设计得和 **scullp** 很相似。**order** 参数指定分配的内存空间的“幂”，默认为 4。**scullv** 和 **scullp** 的唯一差别是在分配管理上。下面一段代码用 **vmalloc** 获取内存：

```
/* Allocate a quantum using virtual addresses */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] =
        (void *)vmalloc(PAGE_SIZE << dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

这段代码释放内存：

```
/* Release the quantum set */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        vfree(dptr->data[i]);
```

如果在编译这两个模块时都打开了调试开关，就可以通过读它们在 **/proc** 下创建的文件来查看它们进行的数据分配。下面的快照取自两个不同系统：

```
salma% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem

Device 0: qset 500, order 0, sz 1048576
  item at e00000003e641b40, qset at e000000025c60000
    0:e00000003007c000
    1:e000000024778000
salma% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1048576
  item at e0000000303699c0, qset at e000000025c87000
    0:a0000000000034000
    1:a0000000000078000
salma% uname -m
ia64

rudo% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem

Device 0: qset 500, order 0, sz 1048576
  item at c4184780, qset at c71c4800
    0:c262b000
    1:c2193000
rudo% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1048576
  item at c4184b80, qset at c71c4000
    0:c881a000
    1:c882b000
rudo% uname -m
```

这些数值说明了二者行为上的差别。在 IA-64 平台，物理地址和虚拟地址被映射到完全不同的地址范围（0xE 和 0xA），而在 x86 平台上 `vmalloc` 返回的虚拟地址就在用于映射物理内存的地址之上。

7.5 引导时的内存分配

如果的确需要连续的大量的内存用作缓冲区，就要在系统引导分配。这种技术比较粗暴也很不灵活，但也是最不容易失败的。显然，模块不能在引导时分配内存；只有直接连接到内核的设备驱动程序才能在引导时分配内存。

在引导时就进行分配是获得大量连续内存页面的唯一方法，绕过了 `get_free_pages` 在缓冲区大小上的最大尺寸和固定粒度的双重限制。在引导时分配缓冲区有点“脏”，因为它通过保留私有内存池而跳过了内核内存管理机制。

还有一个问题是对于普通用户来说它不是一个可用的选项：代码直接连接入内核映像；要安装或替换使用了这种分配技术的驱动程序，就只能重新编译内核，重启计算机。幸好还有两种其它的方法，我们随后介绍。

尽管我们不推荐在引导时分配内存，但它还是值得在此提及的，因为在 `_GFP_DMA` 被引入之前，这种技术曾是 Linux 的早期版本里分配可用于 DAM 传输的缓冲区的唯一方法。

7.5.1 在引导时获得专用缓冲区

内核引导时，它可以访问系统所有的物理内存。然后调用各个子系统的初始化函数进行初始化，它允许初始化代码分配私有的缓冲区，同时减少留给常规系统操作的 RAM 数量。

在内核的 2.4 版本，这种分配通过调用下列函数进行：

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

这些函数要么分配完整的页（以 `_pages` 结尾），要么分配没有对齐页边界的内存区。它们分配低端内存或常规内存（参考本章前面关于内存区段的讨论）。常规内存区的分配返回 `MAX_DMA_ADDRESS` 之上的内存地址；低端内存分配返回该值之下的地址。

这些接口在内核 2.3.23 引入。更早的版本使用不那么精确的接口，就如 Unix 书籍中描述的那样。基本上，几个内核子系统的初始化函数接收两个 `unsigned long` 参数，表示当前空闲内存区的边界。每个这样的函数都可以从这个区“偷”一部分，再返回新的下边界。由于驱动程序是在系统引导时进行内存分配的，所以可以从空闲 RAM 的线性数组获取连续的内存空间。

这种老式的、在引导期间分配内存的机制的主要问题是，不是所有的初始化函数都可以修改内存下

界，所以编写一个需要这种分配技术的驱动程序，通常意味着用户必须给内核打补丁。另一方面，`alloc_bootmem` 在引导时还可能被任何内核子系统的初始化函数调用。

除了不能释放得到的缓冲区外，这种分配方法还有些别的缺点。驱动程序得到这些内存页后，就无法将它们再放到空闲页面池中了；页面池是在物理内存的分配已经结束后才建立起来的，而且我们也不推荐这样地“hack”内存管理的内部数据结构。但另一方面，这种技术的优势是，它可以获取一段连续的物理内存，来用于 DMA 传输等用途。目前这也是分配超过 32 页的连续内存缓冲区的唯一“安全”的方式，32 这个值是源于 `get_free_pages` 函数参数 `order` 可取的最大值为 5。但是如果要用到的多个内存页物理上可以是不连续的，那么最好还是用 `vmalloc` 函数。

如果要在引导时获取内存的话，必须修改内核代码中的 `init/main.c` 文件。关于 `main.c` 文件的更多细节参见第 16 章。

注意，这种“分配”只能是按页面大小的倍数进行，页面数不必是 2 的某个幂次。

7.5.2 bigphysarea 补丁

另一个在驱动程序中获取大量连续内存区的办法是使用 `bigphysarea` 补丁。这个非官方补丁已经在网上流传多年了；它广为人知而且十分有效，某些发行版本甚至直接把它加入了默认安装的内核映像中。补丁的原理是在引导时分配内存，在运行时提供给设备驱动程序使用。需要向内核传递一个命令行选项以指定引导时保留的内存数量。

该补丁如今在 <http://www.polyware.nl/~middelink/En/hob-v4l.html> 维护。包括了描述设备驱动程序如何使用它提供的分配接口的文档。内核 2.4 中的 Zoran 36120 帧捕获设备驱动程序（在 `drivers/char/zr36120.c` 中）使用了 `bigphysarea` 扩展，是示范如何使用该接口的很好的例子。

7.5.3 保留高端 RAM 地址

分配连续内存区的最后一个办法，也许是最容易的，就是在物理内存的“尾部”保留一段内存区（`bigphysarea` 是在物理内存的首部保留内存）。为此需要给内核传递一个命令行选项以限制它管理的内存数量。例如，在一个 128MB 内存的系统上，笔者使用了 `mem=126M` 来保留 2M 内存。稍后在运行时，这段内存可以分配给设备驱动程序使用。

作为样例代码的一部分，`allocator` 模块也在 O'Reilly 的 FTP 站点上，它提供了未被 Linux 内核使用的高端内存的分配接口。在第 13 章“做自己的分配”一节中详细介绍了这个模块。

`allocator` 相比 `bigphysarea` 补丁的优点是不需要修改官方内核代码。缺点是当系统 RAM 数量变化时必须修改送给内核的命令行选项。另一个缺点是高端内存不能用于某些任务，如 ISA 设备的 DMA 缓冲区。

7.6 向后兼容性

Linux 内存管理子系统自 2.0 内核出现以来已经发生了很大的变化。不过幸好它的编程接口变化很少，而且容易处理。

`kmalloc` 和 `kfree` 从 Linux 2.0 到 2.4 基本上保持不变。对高端内存的访问和 `__GFP_HIMEM` 标志是从内核 2.3.23 加入的。`sysdep.h` 弥补了这个缺陷，允许 2.4 的语义在 2.2 和 2.0 中使用。

后备式高速缓冲区函数是在 Linux 2.1.23 引入的，在 2.0 内核中不可使用。要保持对 2.0 的兼容性就只能使用 `kmalloc` 和 `kfree` 了。另外，`kmem_destroy_cache` 是在 2.3 开发版本引入的，只能向后移植到 2.2.18。因此 `sculc` 不能在比 2.2.18 更早的 2.2 内核下编译。

Linux 2.0 的 `__get_free_pages` 有第三个整数参数，称为 `dma`。它和现在内核中的 `__GFP_DMA` 标志的作用相同，但没有合并到 `flags` 参数中。为此 `sysdep.h` 传递 0 作为 2.0 的该函数的第三个参数。如果需要请求 DMA 页面，又要保持和 2.0 向后兼容，需要调用 `get_dma_pages` 来代替 `__GFP_DMA`。

在 2.x 内核中 `vmalloc` 和 `vfree` 没有变化。不过，`ioremap` 函数在 2.0 中称为 `vremap`。而且没有 `iounmap`。`vremap` 获取的 I/O 映射是用 `vfree` 释放的。在 2.0 中也没有头文件 `<linux/vmalloc.h>`；相应函数都是在 `<linux/mm.h>` 中声明的。与前面一样，`sysdep.h` 使得 2.4 的代码可以在以前的内核下工作；如果包含了 `<linux/mm.h>`，它就包含 `<linux/vmalloc.h>`，因此隐藏了差异。

7.7 快速参考

与内存分配有关的函数和符号列在下面。

```
#include <linux/malloc.h>
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
```

最常用的内存分配接口。

```
#include <linux/mm.h>
GFP_KERNEL
GFP_ATOMIC
__GFP_DMA
__GFP_HIGHMEM
```

`kmalloc` 的标志。`__GFP_DMA` 和 `__GFP_HIGHMEM` 是标志位，可以和 `GFP_KERNEL` 或者 `GFP_ATOMIC` 中的一个作“或”运算。

```
#include <linux/malloc.h>
kmem_cache_t *kmem_cache_create(char *name, size_t size, size_t offset, unsigned long
    flags, constructor(), destructor());
int kmem_cache_destroy(kmem_cache_t *cache);
```

创建和销毁一个包含了同样大小的内存块的高速缓存区。可以从这个高速缓存区中分配同样大小的对象。

```
SLAB_NO_REAP
SLAB_HWCACHE_ALIGN
SLAB_CACHE_DMA
```

创建高速缓存时指定的标志。

```
SLAB_CTOR_ATOMIC
SLAB_CTOR_CONSTRUCTOR
```

分配器传递给 **constructor** 和 **destructor** 函数的标志。

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

从高速缓存中分配和释放一个对象。

```
unsigned long get_zeroed_page(int flags);
unsigned long __get_free_page(int flags);
unsigned long __get_free_pages(int flags, unsigned long order);
unsigned long __get_dma_pages(int flags, unsigned long order);
```

基于页的分配函数。**get_zeroed_page** 返回一个已清零页面。其它调用返回不进行初始化的页面。**__get_dma_pages** 是唯一在 Linux 2.2 及其后版本间（可使用 **__GFP_DMA** 代替）兼容的宏。

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

这些函数释放基于页分配的内存。

```
#include <linux/vmalloc.h>
void * vmalloc(unsigned long size);
void vfree(void * addr);
#include <asm/io.h>
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void *addr);
```

这些函数分配或释放连续虚拟地址空间。**ioremap** 通过虚拟地址访问物理内存，**vmalloc** 分配空闲页面。使用 **ioremap** 映射的区域用 **iounmap** 释放，用 **vmalloc** 获得的页面用 **vfree** 释放。

```
#include <linux/bootmem.h>
void *alloc_bootmem(unsigned long size);
void *alloc_bootmem_low(unsigned long size);
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

只有在 2.4 版本内核才能使用这些函数。这些函数在引导时分配内存，只有直接连接入内核映像的驱动程序才能使用这些函数。

第 8 章 硬件管理



尽管摆弄 `scull` 和其他一些玩具程序对理解 Linux 设备驱动程序的软件接口很有帮助，但实现真正的设备还是要涉及实际硬件。设备驱动程序是软件概念和硬件电路之间的一个抽象层，因此，两者都需要谈谈。到现在为止，我们已经详细讨论了软件上的一些细节；本章将完成另外一部分，介绍驱动程序是如何在保持可移植性的前提下访问 I/O 端口和 I/O 内存的。

和前面一样，本章尽可能不针对特定设备。在需要示例的场合，我们使用简单的数字 I/O 端口（比如标准 PC 并口）来讲解 I/O 指令，使用普通的帧缓存显示内存来讲解内存映射 I/O。

我们选择使用并口是因为它是最简单的输入/输出端口。几乎所有的计算机上都有并口，并实现了原始的 I/O：写到设备的数据位出现在输出引脚上，而输入引脚的电压值可以由处理器直接获取。实践中，我们必须将 LED 连接到并口上才能真正“看到”数字 I/O 操作的结果，相关底层硬件也非常容易使用。

8.1 I/O 端口和 I/O 内存

每种外设都通过读写寄存器进行控制。大部分外设都有几个寄存器，不管是在内存地址空间还是在 I/O 地址空间，这些寄存器的访问地址都是连续的。

在硬件级上，内存区域和 I/O 区域没有概念上的区别：它们都通过向地址总线和控制总线发送电平信号进行访问（比如读和写信号）^{*}，再通过数据总线读写数据。

一些 CPU 制造厂商在它们的芯片中使用单一地址空间，另一些则为外设保留了独立的地址空间以便和内存区分开来。一些处理器（主要是 x86 家族的）还为 I/O 端口的读和写使用分离的连线，并且使用特殊的 CPU 指令访问端口。

因为外设要与外围总线相匹配，而最流行的 I/O 总线是基于个人计算机模型的，所以即使原本没有独立的 I/O 端口地址空间的处理器，在访问外设时也要虚拟成读写 I/O 端口。这通常是由外部芯片组或 CPU 核心中的附加电路来实现的。后一种方式只在嵌入式的微处理器中比较多见。

^{*} 并非所有的计算机平台都使用读和写信号；有些使用不同的方式处理外部电路。不过这些区别对软件是无关的，

基于同样的原因，Linux 在所有的计算机平台上都实现了 I/O 端口，包括使用单一地址空间的 CPU 在内。端口操作的具体实现则依赖于宿主计算机的特定模型和制造了（因为不同的模型使用不同的芯片组把总线操作映射到内存地址空间）。

即使外设总线为 I/O 端口保留了分离的地址空间，也不是所有设备都会把寄存器映射到 I/O 端口。ISA 设备普遍使用 I/O 端口，大多数 PCI 设备则把寄存器映射到某个内存地址区段。这种 I/O 内存通常是首选方案，因为不需要特殊的处理器指令；而且 CPU 核心访问内存更有效率，访问内存时，编译器在寄存器分配和寻址方式选择上也有更多的自由。

8.1.1 I/O 寄存器和常规内存

尽管硬件寄存器和内存非常相似，程序员在访问 I/O 寄存器的时候必须注意避免由于 CPU 或编译器不恰当的优化而改变预期的 I/O 动作。

I/O 寄存器和 RAM 的最主要区别就是 I/O 操作具有边际效应，而内存操作则没有：内存写操作的唯一结果就是在指定位置存储一个数值；内存读操作则仅仅返回指定位置最后一次写入的数值。由于内存访问速度对 CPU 的性能至关重要，而且也没有边际效应，所以可用多种方法进行优化，如使用高速缓存保存数值，重新排序读/写指令等。

编译器能够将数值缓存在 CPU 寄存器中而不写入内存，即使存储数据，读写操作也都能在高速缓存中进行而不用访问物理 RAM。无论在编译器一级或是硬件一级，指令的重新排序都有可能发生：一个指令序列如果以不同于程序文本中的次序运行常常能执行得更快，例如在防止 RISC 处理器流水线的互锁时就是如此。在 CISC 处理器上，耗时的操作则可以和运行较快的操作并发执行。

在对常规内存进行这些优化的时候，优化过程是透明的，而且效果良好（至少在单处理器系统上是这样）。但对 I/O 操作来说这些优化很可能造成致命的错误，因为它们会干扰“边际效应”，而这却是驱动程序访问 I/O 寄存器的主要目的。处理器无法预料到某些其它进程（在另一个处理器上运行，或在某个 I/O 控制器中）是否会依赖于内存访问的顺序。因此驱动程序必须确保不会使用高速缓存，并且在访问寄存器时不会发生读或写指令的重新排序：编译器或 CPU 可能会自作聪明地重新排序所要求的操作，结果是发生奇怪的错误，并且很难调试。

由硬件自身缓存引起的问题很好解决：底层硬件配置成（可以是自动的或是由 Linux 初始化代码完成）访问 I/O 区域时（不管是内存还是端口）禁止硬件缓存就行了。

由编译器优化和硬件重新排序引起的问题的解决办法是，在从硬件角度看必须以特定顺序执行的操作之间设置内存屏障。Linux 提供了 4 个宏来解决所有可能的排序问题。

```
#include <linux/kernel.h>
void barrier(void)
```

这个函数通知编译器插入一个内存屏障，但对硬件无效。编译后的代码会把当前 CPU 寄存器中的所有修改过的数值存到内存，需要这些数据的时候再重新读出来。

为简化讨论，这里假定所有平台都用读和写信号。

```
#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);
```

这些函数在已编译的指令流中插入硬件内存屏障；具体的插入方法是平台相关的。**rmb**（读内存屏障）保证了屏障之前的读操作一定会在后来的读操作执行之前完成。**wmb** 保证写操作不会乱序，**mb** 指令保证了两者都不会。这些函数都是 **barrier** 的超集。

设备驱动程序中使用内存屏障的典型格式如下：

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb();
writel(dev->registers.control, DEV_GO);
```

在这个例子中，最重要的是要确保控制某特定操作的所有设备寄存器一定要在操作开始之前正确设置。其中的内存屏障会强制写操作以必需的次序完成。

因为内存屏障会影响系统性能，所以应该只用于真正需要的地方。不同类型的内存屏障影响性能的方面也不同，所以最好尽可能使用针对需要的特定类型。例如在当前的 **x86** 体系结构上，由于处理器之外的写不会重新排序，**wmb** 就没什么用。可是读会重新排序，所以 **mb** 就会比 **wmb** 慢一些。

注意其它大多数的处理同步的内核原语，如 **spinlock** 和 **atomic_t** 操作，也能作为内存屏障使用。

在有些体系结构上允许把赋值语句和内存屏障进行合并以提高效率。**2.4** 版本内核提供了几个执行这种合并的宏；它们默认情况下定义如下：

```
#define set_mb(var, value) do {var = value; mb();} while 0
#define set_wmb(var, value) do {var = value; wmb();} while 0
#define set_rmb(var, value) do {var = value; rmb();} while 0
```

在适当的地方，**<asm/system.h>** 中定义的这些宏可以利用体系结构特有的指令更快地完成任务。

头文件 **sysdep.h** 中定义了本节介绍的这些宏，可供缺少这些宏的平台和内核版本使用。

8.2 使用 I/O 端口

I/O 端口是驱动程序与许多设备的之间通信方式——至少在部分时间是这样。本节讲解了使用 I/O 端口的不同函数，另外也涉及到一些可移植性问题。

我们先回忆一下，I/O 端口必须先分配，然后才能由驱动程序使用。这在第 2 章的“I/O 端口 和 I/O 内存”一节已经讨论过了，用来分配和释放端口的函数是：

```
#include <linux/ioport.h>
int check_region(unsigned long start, unsigned long len);
struct resource *request_region(unsigned long start,
    unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
```

驱动程序请求了需要使用的 I/O 端口范围后，它必须读并且/或者写这些端口。为此，大多数硬件都把 8 位、16 位和 32 位的端口区分开来。它们不能象访问系统内存那样混淆*。因此，C 语言程序必须调用不同的函数来访问大小不同的端口。如前一节所述，那些只支持映射到内存的 I/O 寄存器的计算机体系结构通过把 I/O 端口地址重新映射到内存地址来模拟端口 I/O，并且为了易于移植，内核对驱动程序隐藏了这些细节。Linux 内核头文件中(就在与体系结构相关的头文件 `<asm/io.h>` 中)定义了如下一些访问 I/O 端口的内联函数。



从现在开始，如果我使用 `unsigned` 而不进一步指定类型信息的话，那么就是在谈及一个与体系结构相关的定义，此时不必关心它的准确特性。这些函数基本是可移植的，因为编译器在赋值时会自动进行强制类型转换 (`cast`)——强制转换成 `unsigned` 类型防止了编译时出现的警告信息。只要程序员赋值时注意避免溢出，这种强制类型转换就不会丢失信息。在本章剩余部分将会一直保持这种“不完整的类型定义”的方式。

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

按字节(8 位宽度)读写端口。`port` 参数在一些平台上定义为 `unsigned long`，而在另一些平台上定义为 `unsigned short`。不同平台上 `inb` 返回值的类型也不相同。

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

这些函数用于访问 16 位端口(“字宽度”);不能用于 M68k 或 S390 平台，因为这些平台只支持字节宽度的 I/O 操作。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

这些函数用于访问 32 位端口。`longword` 参数根据不同平台定义成 `unsigned long` 类型或 `unsigned int` 类型。和字宽度 I/O 一样，“长字” I/O 在 M68k 和 S390 平台上也不能用。

注意这里没有定义 64 位的 I/O 操作。即使在 64 位的体系结构上，端口地址空间也只使用最大 32 位的数据通路。

上面这些函数主要是提供给设备驱动程序使用的，但它们也可以在用户空间使用，至少在 PC 类计算机上可以使用。GNU 的 C 库在 `<sys/io.h>` 中定义了这些函数。如果要在用户空间代码中使用 `inb` 及其相关函数，必须满足下面这些条件：

编译该程序时必须带 `-O` 选项来强制内联函数的展开。

必须用 `ioperm` 或 `iopl` 来获取对端口进行 I/O 操作的许可。`ioperm` 用来获取对单个端口的操作许可，而 `iopl` 用来获取对整个 I/O 空间的操作许可。这两个函数都是 Intel 平台特有的。

必须以 `root` 身份运行该程序才能调用 `ioperm` 或 `iopl`*。或者，该程序的某个祖先已经以 `root` 身份获取了对端口操作的权限。

* 有时 I/O 端口是和内存一样对待的，(例如)可以将 2 个 8 位的操作合并成一个 16 位的操作。例如，PC 的显示卡就可以，但一般来说不能认为一定具有这种特性。

* 从技术上说，必须有 `CAP_SYS_RAWIO` 的权能，不过这与在当前系统以 `root` 身份运行是一样的。

如果宿主平台没有 `ioperm` 和 `iopl` 系统调用，用户空间程序仍然可以使用 `/dev/port` 设备文件访问 I/O 端口。不过要注意，该设备文件的含义和平台的相关性是很强的，并且除 PC 上以外，它几乎没有用处。

示例程序 `misc-progs/inp.c` 和 `misc-progs/outp.c` 是在用户空间通过命令行读写端口的一个小工具。它们会以多个名字安装（如 `inpb`、`inpw`，`inpl`）并且按用户调用的名字分别操作字节端口、字端口或双字端口。如果没有 `ioperm`，它们就使用 `/dev/port`。

如果想冒险，可以将它们设置上 `SUID` 位，这样，不用显式地获取特权就可以使用硬件了。

8.2.1 串操作

以上的 I/O 操作都是一次传输一个数据，作为补充，有些处理器上实现了一次传输一个数据序列的特殊指令，序列中的数据单位可以是字节、字或双字。这些指令称为串操作指令，它们执行这些任务时比一个 C 语言写的循环语句快得多。下面列出的宏实现了串 I/O，它们或者使用一条机器指令实现，或者在没有串 I/O 指令的平台上使用紧凑循环实现。M68k 和 S390 平台上没有定义这些宏。这不会影响可移植性，因为这些平台通常不会和其它平台使用同样的设备驱动程序，它们的外设总线不同。

串 I/O 函数的原型如下：

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

从内存地址 `addr` 开始连续读写 `count` 数目的字节。只对单一端口 `port` 读取或写入数据。

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

对一个 16 位端口读写 16 位数据。

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

对一个 32 位端口读写 32 位数据。

8.2.2 暂停式 I/O

某些平台，特别是 i386 平台上，当处理器和总线之间的数据传输太快时会引起问题。因为相对于 ISA 总线，处理器的时钟频率太快，当设备板卡速度太慢时，这个问题就会暴露出来。解决方法是，如果一条 I/O 指令后还跟着另一条 I/O 指令，就在两条指令间插入一小段延迟。如果有设备丢失数据的情况，或为了防止设备可能会丢失数据的情况，可以使用暂停式的 I/O 函数来取代通常的 I/O 函数。这些暂停式的 I/O 函数很象前面已经列出的那些 I/O 函数，不同之处是它们的名字用 `_p` 结尾；如 `inb_p`，`outb_p`，等等。在 Linux 支持的大多数平台上都定义了这些函数，不过它们常常扩展为和非暂停式 I/O 同样的代码，因为如果某种体系结构不使用过时的外设总线，就不需要额外的暂停。

8.2.3 平台相关性

由于自身的特性，I/O 指令是与处理器密切相关的。因为它们的工作涉及到处理器移入移出数据的细节，所以隐藏平台间的差异非常困难。因此，大部分与 I/O 端口有关的源代码都与平台相关。

回头看看前面的函数列表，可以看到一处不兼容的地方：数据类型。函数的参数类型根据各平台体系结构上的不同要相应地使用不同的数据类型。例如，`port` 参数在 `x86` 平台（处理器只支持 64KB 字节的 I/O 空间）上定义为 `unsigned short`，但在其它平台上定义为 `unsigned long`。在那些平台上端口是与内存存在同一地址空间内的一些特定区域。

其他一些与平台相关的问题来源于处理器基本结构上的差异，因此也无法避免。因为本书假定读者不会在不了解底层硬件的情况下为特定的系统编写驱动程序，所以不会详细讨论这些差异。下面是内核 2.4 版本支持的体系结构可以使用的函数的总结：

IA-32 (x86)

该体系结构支持本章提到的所有函数。端口号的类型是 `unsigned short`。

IA-64 (Itanium)

支持所有函数；端口类型是 `unsigned long`（映射到内存）。串操作函数是用 C 语言实现的。

Alpha

支持所有函数，而 I/O 端口是映射到内存的。基于不同的 Alpha 平台上使用的芯片组的不同，端口 I/O 操作的实现也有所不同。串操作是用 C 语言实现的，在文件 `arch/alpha/lib/io.c` 中定义。端口类型是 `unsigned long`。

ARM

端口映射到内存，支持所有函数；串操作用 C 语言实现。端口类型是 `unsigned int`。

M68k

端口映射到内存，只支持字节类型的函数。不支持串操作，端口类型是 `unsigned char *`。

MIPS

MIPS64

MIPS 端口支持所有函数。因为该处理器不提供机器一级的串 I/O 操作，所以串操作是用汇编语言写的紧凑循环（tight loop）实现的。端口映射到内存；端口类型在 32 位处理器上是 `unsigned int`，在 64 位处理器上是 `unsigned long`。

PowerPC

支持所有函数；端口类型为 `unsigned char *`。

S390

类似于 M68k，该平台的头文件只支持字节宽度的端口 I/O，不支持串操作。端口类型是字符型（char）指针，映射到内存。

Super-H

端口类型是 `unsigned int`（映射到内存），支持所有函数。

SPARC

SPARC64

和前面一样，I/O 空间映射到内存。端口操作函数的 `port` 参数类型是 `unsigned long`。

感兴趣的读者可以从 `io.h` 文件获得更多信息，除了在本章介绍的函数，一些与体系结构相关的函数有时也由该文件定义。不过要注意这些文件阅读起来会比较困难。

值得提及的是，x86 家族之外的处理器都不为端口提供不同的地址空间，尽管使用其中几种处理器的机器带有 ISA 和 PCI 插槽（两种总线都实现了不同的 I/O 和内存地址空间）。

除此以外，一些处理器（特别是早期的 Alpha 处理器）没有一次传输 1 或 2 个字节的指令^{*}。因此，它们的外设芯片通过把端口映射到内存地址空间的特殊地址范围来模拟 8 位和 16 位的 I/O 访问。这样，对同一个端口的 `inb` 和 `inw` 指令实现为两个 32 位的读不同内存地址的操作。幸好，本章前面介绍的宏的内部实现对驱动程序开发人员隐藏了这些细节，不过这个特点还是很有趣的。想进一步深入的读者可以看 `include/asm-alpha/core_lca.h` 中的例子。

I/O 操作在各个平台上执行的细节在对应平台的编程手册中有详细的叙述；也可从 Web 上下载这些手册的 PDF 文件。

8.3 使用数字 I/O 端口

我们用来演示设备驱动程序的端口 I/O 的示例代码工作于通用的数字 I/O 端口上；这种端口在大多数计算机平台上都能找到。

数字 I/O 端口最普通的形式是一个字节宽度的 I/O 区域，它或者映射到内存，或者映射到端口。当数值写入到输出区域时，输出引脚上的电平信号随着写入的各位发生相应变化。从输入区域读到的数据则是输入引脚各位当前的逻辑电平值。

这类 I/O 端口的具体实现和软件接口是因系统而异的。大多数情况下，I/O 引脚是由两个 I/O 区域控制的：一个区域中可以选择用于输入和输出的引脚，另一个区域中可以读写实际逻辑电平。不过有时候情况简单些，每个位不是输入就是输出（不过在这种情况下不能再称为“通用 I/O”了）；所有个人计算机上都能找到的并口就是这样的非通用的 I/O 端口。我们随后介绍的示例代码要用到这些 I/O 引脚。

8.3.1 并口简介


因为假定大多数读者使用的都是称为“个人计算机”的 x86 平台，所以解释一下 PC 并口的设计是必要的。并口也是在个人计算机上运行的数字 I/O 示例代码选用的外设接口。尽管许多读者可能已经有了并口规格说明，为了方便还是在这里概括一下。

并口的最小配置（不涉及 ECP 和 EPP 模式）由 3 个 8 位端口组成。PC 标准中第一个并口的 I/O 端口是从地址 `0x378` 开始，第二个端口是从地址 `0x278` 开始。第一个端口是一个双向的数据寄存器；它直接连接到物理插口的 2 到 9 号引脚上。第二个端口是一个只读的状态寄存器；当

^{*} 单字节 I/O 操作并没有想象中那么重要，因为这种操作很少发生。为了读写任意地址空间的单个字节，需要实现一条从寄存器组数据总线低位到外部数据总线任意字节地址的数据通路。这种数据通路在每一次数据传输中都需要额外的逻辑门。不使用这类字节宽度的存取指令可以提升系统总体性能。

并口连接到打印机时，该寄存器报告打印机的状态，如是否在线、缺纸、正忙等等。第三个端口是一个只用于输出的控制寄存器，它的作用之一是控制是否打开中断。

在并行通信中使用的电平信号是标准的 TTL 电平：0 伏和 5 伏，逻辑阈值大约为 1.2 伏；端口要求至少满足标准的 TTL LS 电流规格，而现代的大部分并口电流和电压都超过这个规格。

 并口插座没有和计算机的内部电路隔离，这一点在试图把逻辑门直接连到端口时很有用。但要注意正确连线；否则在测试自己定制的电路时，并口很容易被烧毁。如果担心会破坏主板的话，可以选用可插拔的并行接口。

位规范显示在图 8-1 中。可以读写 12 个输出位和 5 个输入位，其中一些位在它们的信号通路上有逻辑上的翻转。唯一一个不与任何信号引脚有联系的位是 2 号端口的第 4 位(0x10)，它打来自并口的中断。我们将在第 9 章“中断处理”中的一个中断处理程序实现中使用到它。

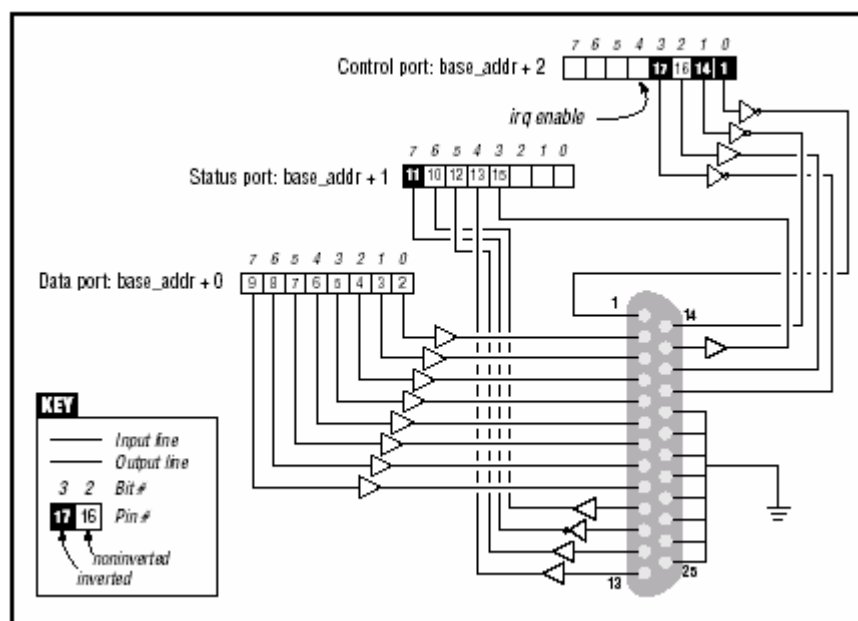


图 8-1：并口的插线引脚

8.3.2 示例驱动程序

下面要介绍的驱动程序叫做 **short** (Simple Hardware Operations and Raw Tests，简单硬件的操作和原始测试)。它所做的就是读写几个 8 位端口，其中第一个是加载时选定的。默认情况下它使用的就是分配给 PC 并口的端口范围。每个设备节点（拥有唯一的次设备号）访问一个不同的端口。**short** 设备没有任何实际用途，使用它只是为了能用一条指令来从外部对端口进行操作。如果读者不太了解端口 I/O，那么可以通过使用 **short** 来熟悉它，可以测量它传输数据时消耗的时间或者进行其它的测试。

为使 **short** 在系统上工作，它必须能自由地访问底层硬件设备（默认情况就是并口），因此不能有其他驱动程序在使用同一设备。现在的大多数 Linux 发布版本将并口驱动程序作为模块安装，并且只在需要用到时才加载，所以一般不会发生争夺 I/O 地址的问题。不过，如果 **short** 给出一个“can't get I/O address，无法获得 I/O 地址”错误（可能在控制台或者系统日志文件中）的

话，说明可能已经有其它驱动程序占用了这个端口。通过检查 `/proc/iports` 一般可以找出这是哪个驱动程序。这种情况一样适用于并口之外的其它 I/O 设备。

从现在开始，为简化讨论，我们所指的设备都是并口。不过也可以在模块加载时通过设置参数 `base` 把 `short` 重定向到其它 I/O 设备。这样示例代码可以在任何拥有对数字 I/O 接口访问权限的 Linux 平台上运行，这些接口必须是能用 `outb` 和 `inb` 进行访问的（尽管实际硬件在除 x86 的所有平台上都是映射到内存的）。在随后的“使用 I/O 内存”中，我们还将展示 `short` 是如何用于通用的映射到内存的数字 I/O 的。

为了观察并口插座上发生了什么，并且如果读者喜欢操作硬件，那么可以焊几个 LED 到输出引脚上。每个 LED 都要串联一个 1KΩ 的电阻到一个接地的引脚上（除非使用的 LED 已经有内建电阻）。如果将输出引脚接到输入引脚上，就可以产生自己的输入供输入端口读取。

注意不能仅仅通过把打印机连到并口来观察送给 `short` 的数据。因为这个驱动程序只实现了简单的 I/O 端口访问，不能提供打印机操作数据时所需的握手信号。

如果读者想将 LED 焊到 D 型插座上来观察并行数据，建议不要使用 9 号和 10 号引脚，因为在运行第 9 章的示例代码时我们要连上它们。

至于 `short`，它通过 `/dev/short0` 读写位于 I/O 地址 `base`（除非加载时修改，否则就是 0x378）的 8 位端口。`/dev/short1` 写位于 `base+1` 的 8 位端口，依此类推，直到 `base+7`。

`/dev/short0` 实际执行的输出操作是一个使用 `outb` 的紧凑循环。这里还使用了内存屏障指令来确保输出操作会实际执行而不是被优化掉。

```
while (count--) {
    outb(*(ptr++), address);
    wmb();
}
```

可以运行下面的命令来使 LED 发光：

```
echo -n "any string" > /dev/short0
```

每个 LED 监控输出端口的一个位。注意只有最后写的字符数据才会在输出引脚上稳定地保持下来而被观察到。因此，建议将 `-n` 选项传给 `echo` 程序来制止输出字符后的自动换行。

读端口也是使用类似的函数，只是用 `inb` 代替了 `outb`。为了从并口读取“有意义的”值，需要将某个硬件连到并口插座的输入引脚上来产生信号。如果没有输入信号，只会读到始终是相同字节的无穷输出流。如果选择从输出端口读入，将会取回写到该端口的最后一个值（对并口和其它大多数普通数字 I/O 电路都是如此）。因此，不想摆弄烙铁的读者可以运行下面的命令在端口 0x378 读取当前的输出值：

```
dd if=/dev/short0 bs=1 count=1 | od -t x1
```

为了示范所有 I/O 指令的使用，每个 `short` 设备都提供了 3 个变种：`/dev/short0` 执行的是上面

的循环；`/dev/short0p` 使用了 `outb_p` 和 `inb_p` 来替代前者使用的“较快的”函数，`/dev/short0s` 使用串指令。这样的设备共有 8 个，从 `short0` 到 `short7`。PC 并口只有三个端口，如果读者使用了其它不同的 I/O 设备进行测试，就可能需要更多的端口。

虽然 `short` 驱动程序只完成了最低限度的硬件控制，但这对演示 I/O 端口指令的使用已经足够了。感兴趣的读者可以去看 `parport` 和 `parport_pc` 两个模块的源码，看看实际上为支持使用并口的设备（打印机、磁带备份，网络接口）所需的复杂工作。

8.4 使用 I/O 内存

除了 x86 上普遍使用的 I/O 端口，和设备通信的另一种主要机制是通过使用映射到内存的寄存器或设备内存。这两种都称为 I/O 内存，因为寄存器和内存的差别对软件是透明的。

I/O 内存仅仅是类似 RAM 的一个区域，在那里处理器可以通过总线访问设备。这种内存有很多用途，比如存放视频数据或网络包；这些用设备寄存器也能实现，其行为类似于 I/O 端口（比如，读写时有边际效应）。

访问 I/O 内存的方法和计算机体系结构、总线，以及设备是否正在使用有关，不过原理都是相同的。本章主要讨论 ISA 和 PCI 内存，同时也试着介绍一些通用的知识。尽管这里介绍了 PCI 内存的访问，但关于 PCI 的详细讨论将放到第 15 章中进行。

根据计算机平台和所使用总线的不同，I/O 内存可能是，也可能不是通过页表访问的。如果访问是经由页表进行的，内核必须首先安排物理地址使其对设备驱动程序可见（这通常意味着在进行任何 I/O 之前必须先调用 `ioremap`）。如果访问无需页表，那么 I/O 内存区域就很象 I/O 端口，可以使用适当形式的函数读写它们。

不管访问 I/O 内存时是否需要调用 `ioremap`，都不鼓励直接使用指向 I/O 内存的指针。尽管（在“I/O 端口和 I/O 内存”介绍过）I/O 内存存在硬件一级是象普通 RAM 一样寻址的，但在“I/O 寄存器和常规内存”中描述过的那些需要额外小心的情况中已经建议不要使用普通指针。相反，使用“包装的”函数访问 I/O 内存，一方面在所有平台上都是安全的，另一方面，在可以直接对指针指向的内存区域执行操作的时候，该函数是经过优化的。

因此，即使在 x86 上直接使用指针（现在）可以工作（而不是使用正确的宏），这种做法也会影响驱动程序的可移植性和可读性。

在第 2 章中说过，设备内存区域在使用前必须先分配。这和 I/O 端口注册过程类似，是由下列函数完成的：

```
int check_mem_region(unsigned long start, unsigned long len);
void request_mem_region(unsigned long start, unsigned long len,
char *name);
void release_mem_region(unsigned long start, unsigned long len);
```

传给函数的 `start` 参数是内存区的物理地址，此时还没有发生任何重映射。这些函数通常的使用方式如下：

```

if (check_mem_region(mem_addr, mem_size)) {
    printk("drivername: memory already in use\n");
    return -EBUSY;
}

request_mem_region(mem_addr, mem_size, "drivername");

[...]

release_mem_region(mem_addr, mem_size);

```

8.4.1 直接映射的内存

几种计算机平台上保留了部分内存地址空间留给 I/O 区域，并且自动禁止对该内存范围内的任何（虚拟）地址进行内存管理。

用在个人数字助理（PDA）中的 MIPS 处理器就是这种配置的一个有趣的实例。两个各为 512 MB 的地址段直接映射到物理地址，对这些地址范围内的任何内存访问都绕过 MMU，也绕过缓存。这些 512 MB 地址段中的一部分是为外设保留的，驱动程序可以用这些无缓存的地址范围直接访问设备的 I/O 内存。

其它平台使用另外的方式提供直接映射的地址段：有些使用特殊的地址空间来解析物理地址（例如，SPARC64 就使用了一个特殊的“地址空间标识符”），还有一些则使用虚拟地址，这些虚拟地址被设置成访问时绕过处理器缓存。

当需要访问直接映射的 I/O 内存区时，仍然不应该直接使用 I/O 指针指向的地址——即使在某些体系结构这么做也能正常工作。为了编写出的代码在各种系统和内核版本都能工作，应该避免使用直接访问的方式，而代之以下列函数。

```

unsigned readb(address);
unsigned readw(address);
unsigned readl(address);

```

这些宏用来从 I/O 内存接收 8 位、16 位和 32 位的数据。使用宏的好处是不用考虑参数的类型：参数 `address` 是在使用前才强制转换的，因为这个值“不清楚是整数还是指针，所以两者都要接收”（摘自 `asm-alpha/io.h`）。读函数和写函数都不会检查参数 `address` 是否合法，因为这在解析指针指向区域的同时就能知道（我们已经知道有时它们确实扩展成指针的反引用操作）。

```

void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);

```

类似前面的函数，这些函数（宏）用来写 8 位、16 位和 32 位的数据。

```

memset_io(address, value, count);

```

当需要在 I/O 内存上调用 `memset` 时，这个函数可以满足需要，同时它保持了原来的 `memset` 的语义。

```

memcpy_fromio(dest, source, num);
memcpy_toio(dest, source, num);

```

这两个函数用来和 I/O 内存交换成块的数据，功能类似于 C 库函数 `memcpy`。

在较新的内核版本中，这些函数在所有体系结构中都是可用的。当然具体实现会有不同：在一些平

台上是扩展成指针操作的宏，在另一些平台上是真正的函数。不过作为驱动程序开发人员，不需要关心它们具体是怎样工作的，只要会用就行了。

一些 64 位平台还提供了 `readq` 和 `writel` 用于 PCI 总线上的 4 字（8 字节）内存操作。这个 4 字（quad-word）的命名是个历史遗留问题，那时候所有的处理器都只有 16 位的字。实际上，现在把 32 位的数值命名为 L（长字）已经是不正确的了，不过如果对所有东西都重新命名，只会把事情搞得更复杂。

8.4.2 在 short 中使用 I/O 内存

前面介绍的 `short` 示例模块访问的是 I/O 端口，它也可以访问 I/O 内存。为此必须在加载时通知它使用 I/O 内存，另外还要修改 `base` 的地址以使其指向 I/O 区域。

例如，我们用下列命令在一块 MIPS 开发板上点亮调试用的 LED：

```
mips.root# ./short_load use_mem=1 base=0xb7fffc0
mips.root# echo -n 7 > /dev/short0
```

在 `short` 中使用 I/O 内存和使用 I/O 端口是一样的；不过，因为没有给 I/O 内存使用的暂停式指令和串操作指令，所以访问 `/dev/short0p` 和 `/dev/short0s` 时，操作和 `/dev/short0` 是一样的。

下列片段显示了 `short` 写内存区域时使用的循环：

```
while (count--) {
    writeb(*(ptr++), address);
    wmb();
}
```

注意这里用了写内存屏障。因为在许多体系结构上 `writeb` 会转化成直接赋值语句，为确保写操作按照预想顺序执行，使用内存屏障是必要的。

8.4.3 通过软件映射的 I/O 内存

尽管 MIPS 类的处理器使用直接映射的 I/O 内存，但这种方式在现在的平台中是相当少见的；特别是当使用外设总线处理映射到内存的设备时更是如此。

使用 I/O 内存时最普遍的硬件和软件处理方式是这样的：设备对应于某些约定的物理地址，但是 CPU 并没有预先定义访问它们的虚拟地址。这些约定的物理地址可以是硬连接到设备上的，也可以是在启动时由系统固件（如 BIOS）指定的。前一种的例子有 ISA 设备，它的地址或者是固化在设备的逻辑电路中，因而已经在局部设备内存中静态赋值，或者是通过物理跳线设置；后一种的例子有 PCI 设备，它的地址是由系统软件赋值并写入设备内存的，只在设备加电时才存在。

不管哪种方式，为了让软件可以访问 I/O 内存，必须有一种把虚拟地址赋于设备的方法。这个任务是由 `ioremap` 函数完成的，我们在“`vmalloc` 和相关函数”中已有介绍。这个函数因为与内存的使用相关，所以已经在前面的章节中讲解过了，它就是为了把虚拟地址指定到 I/O 内存区域而专门设计的。此外，由内核开发人员实现的 `ioremap` 在用于直接映射的 I/O 地址时不起任何作用。

一旦有了 `ioremap` 和 `iounmap`，设备驱动程序就能访问任何 I/O 内存地址，而不管它是否直接映射到虚拟地址空间。不过要记住，这些地址不能直接引用，而应该使用象 `readb` 这样的函数。这样，在设置了 `use_mem` 参数时，通过在 `short` 模块中使用 `ioremap/iounmap` 调用，就可以让 `short` 既能在 MIPS 的 I/O 内存方式下工作，也能在更普通的 ISA/PCI x86 I/O 内存方式下工作。

在示范 `short` 如何调用这些函数之前，先复习一下函数的原型，同时介绍一些在前面章节中忽略的细节。

这些函数定义如下：

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

首先，注意新函数 `ioremap_nocache`。第 7 章中没有具体讲解它，因为它的含义是与硬件相关的。引用内核中的一个头文件的描述：“如果有某些控制寄存器在这个区域，并且不希望发生写操作合并或读缓存的话，可以使用它。”实际上，在大多数计算机平台上这个函数的实现和 `ioremap` 是完全一样的：因为在所有 I/O 内存都已经可以通过非缓存地址访问的情况下，就不必实现一个单独的，非缓存的 `ioremap` 了。

`ioremap` 的另一个重要特点是在内核 2.0 中它的行为和后来内核中的不同。在 Linux 2.0 中，该函数(那时称为 `vremap`)不能映射任何没有对齐页边界的内存区。这是个明智的选择，因为在 CPU 一级所有操作都是以页面大小的粒度进行的。但是，有时候需要映射小的 I/O 寄存器区域，而这些寄存器的(物理)地址不是按页面对齐的。为适应这种新需求，内核 2.1.131 及后续版本中允许重映射未对齐的地址。

`short` 模块为了保持和 2.0 的兼容，同时为了能够访问非页面对齐的寄存器，没有直接调用 `ioremap`，而是使用了下列代码：

```
/* Remap a not (necessarily) aligned port region */
void *short_remap(unsigned long phys_addr)
{
    /* The code comes mainly from arch/any/mm/ioremap.c */
    unsigned long offset, last_addr, size;

    last_addr = phys_addr + SHORT_NR_PORTS - 1;
    offset = phys_addr & ~PAGE_MASK;

    /* Adjust the begin and end to remap a full page */
    phys_addr &= PAGE_MASK;
    size = PAGE_ALIGN(last_addr) - phys_addr;
    return ioremap(phys_addr, size) + offset;
}

/* Unmap a region obtained with short_remap */
void short_unmap(void *virt_addr)
{
    iounmap((void *)((unsigned long)virt_addr & PAGE_MASK));
}
```

8.4.4 1M 地址空间之下的 ISA 内存

最广为人知的 I/O 内存区之一就是个人计算机上的 ISA 内存段。它的内存范围在 640(0xA0000) KB 到 1(0x100000)MB 之间。因此它正好出现在常规系统 RAM 的中间。这种地址安排看上去可能有点奇怪；因为这个设计决策是 80 年代早期作出的，在当时看来没有人会用到 640 KB 以上的内存。

这个内存段属于非直接映射一类的内存^{*}。可以利用 `short` 模块在该内存段中读写几个字节，前面介绍过，在加载模块时要设置 `use_mem` 标志。

尽管 ISA I/O 内存只存在于 x86 类的计算机上，我们还是介绍一下，并附以一个示例程序。

本章不讨论 PCI 内存，因为它是 I/O 内存中最“干净”的一种：只要知道了物理地址，就能简单地重映射并访问它。PCI I/O 内存的“问题”在于，它不适合于用作本章的工作示例，因为无法预先知道 PCI 内存会映射到哪一段物理地址，也就不知道访问这些地址段是否安全。这里选择讲解 ISA 内存段，是因为它不那么“干净”，更适合运行示例代码。

为了示范对 ISA 内存的访问，我们要用到另一个有点“愚笨”的小模块（是示例源码的一部分）。实际上这个模块就叫作 `silly`，是“Simple Tool for Unloading and Printing ISA Data，卸载及打印 ISA 数据的简单工具”的简称。

这个模块补充了 `short` 的功能，它可以访问整个 384 KB 的内存空间，还演示了所有不同的 I/O 函数。该模块包括四个用了不同的数据传输函数来完成相同任务的设备节点。`silly` 设备就象 I/O 内存之上的一个窗口，与 `/dev/mem` 的工作有些类似。对该设备可以读、写数据或 `lseek` 到一个任意的 I/O 内存地址。

因为 `silly` 提供对 ISA 内存的访问，所以启动它时必须把物理 ISA 地址映射到内核虚拟地址中。在较早的 Linux 内核中，只需简单地把要用的 ISA 地址赋值给一个指针，然后直接解析它就可以了。但在现在的内核中，必须配合虚拟内存系统工作，首先重新映射该地址段。这种映射是由 `ioremap` 完成的，这在前面讲解 `short` 时已经介绍过了：

```
#define ISA_BASE    0xA0000
#define ISA_MAX     0x100000 /* for general memory access */

/* this line appears in silly_init */
io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);
```

`ioremap` 返回一个指针值，以供 `readb` 或其它在“直接映射的内存”一节中介绍的函数使用。

现在回头看看示例代码中这些函数是如何使用的。`/dev/sillyb` 的次设备号是 0，通过 `readb` 和 `writb` 访问 I/O 内存。下面代码展示了读操作的实现，其中地址段 0xA0000-0xFFFFF 作为 0-0x5FFFF 段的一个虚拟文件对待。`read` 函数中包括一个 `switch` 语句来处理不同的访问模式。这里是 `sillyb` 的 `case` 语句：

^{*} 实际并非完全如此。因为该内存段很小而且使用频繁，所以内核在启动时就建立了访问这些地址的页表。但是，访问它们使用的虚拟地址和实际物理地址并不相同，所以无论如何都是要使用 `ioremap` 的。另外，内核 2.0 对该地址段是直接映射的，见“向后兼容”与 2.0 版本相关的部分。

```
case M_8:
    while (count) {
        *ptr = readb(add);
        add++; count--; ptr++;
    }
    break;
```

下面的两个设备是 `/dev/sillyw` (次设备号为 1) 和 `/dev/sillyl` (次设备号为 2)。它们和 `/dev/sillyb` 差不多, 只不过分别使用了 16 位和 32 位的函数。下面是 `sillyl` 的 `write` 的实现, 是 `switch` 语句中的一部分:

```
case M_32:
    while (count >= 4) {
        writel(*(u32 *)ptr, add);
        add+=4; count-=4; ptr+=4;
    }
    break;
```

最后一个设备是 `/dev/sillycp` (次设备号为 3), 它使用 `memcpy_io` 函数完成相同任务。它的 `read` 实现的核心部分如下:

```
case M_memcpy:
    memcpy_fromio(ptr, add, count);
    break;
```

因为使用了 `ioremap` 来提供对 ISA 内存区的访问, `silly` 模块卸载时必须调用 `iounmap`:

```
iounmap(io_base);
```

8.4.5 isa_readb 及相关函数

看看内核源代码, 可以发现一组函数, 它们的名字类似于 `isa_readb`。实际上, 上面描述的每个函数都有一个等价的以 `isa_` 开头的函数。这些函数提供了一种不需要单独的 `ioremap` 步骤就能访问 ISA 内存的方法。不过内核开发人员解释说, 这些函数只是暂时性的, 用于帮助移植驱动程序, 将来它们会消失。所以, 最好避免使用这些函数。

8.4.6 探测 ISA 内存

尽管现在的大多数设备都是基于更好的 I/O 总线结构的, 比如 PCI, 但是有时程序员还是得对付 ISA 设备和它们的 I/O 内存, 所以我们为此花些篇幅。我们不涉及高端的 ISA 内存 (称为 **memory hole**, 内存洞, 在 14 MB 到 16 MB 的物理地址段中), 因为现在那种 I/O 内存已经极其少见, 而且现在主流的主板和内核都不支持它了。为访问这种 I/O 内存段需要修改内核初始化代码, 所以这里不再讨论了。

当使用内存映射的 ISA 设备时, 驱动程序开发人员常常会忽略对应的 I/O 内存存在物理地址空间的位置, 因为实际地址通常是由用户从一个可能的地址范围中分配的。否则检查一个指定地址上是否存在设备就很简单了。

内存资源管理配置是有助于内存探测的, 因为它可以识别已经由其它设备使用的内存区段。但是,

资源管理器不能分辨哪些设备的驱动程序已经加载，或者一个给定的区域是否包含有你感兴趣的设备。虽然如此，在实际探测内存、检查地址内容时它仍然是必需的。可能会遇到 3 种截然不同的情况：映射到目标地址上的是 **RAM**，或者是 **ROM**（例如 **VGA BIOS**），或者该区域是空闲的。

skull 示例代码示范了处理这些内存的一种方法，由于 **skull** 和任何物理设备都不相关，它只是打印出 **640 KB** 到 **1 MB** 内存段的信息，然后就退出了。不过其中用来分析内存的代码是值得描述一下的，它示范了如何进行内存探测。

检查 **RAM** 段的代码使用 **cli** 关闭了中断，因为这些内存段只能通过物理地写入数据随后重新读出的方法才能识别，而在测试过程中，真正 **RAM** 中的内容可能被中断处理程序修改。下列的代码并不总是正确，因为如果一个设备正在写它自己的内存段，同时测试代码又正在扫描这个区段，测试程序就会误认为该板卡的 **RAM** 内存段是一个空的区段。不过，这种情况并不常见。

```
unsigned char oldval, newval; /* values read from memory */
unsigned long flags;         /* used to hold system flags */
unsigned long add, i;
void *base;

/* Use ioremap to get a handle on our region */
base = ioremap(ISA_REGION_BEGIN, ISA_REGION_END - ISA_REGION_BEGIN);
base -= ISA_REGION_BEGIN; /* Do the offset once */

/* probe all the memory hole in 2-KB steps */
for (add = ISA_REGION_BEGIN; add < ISA_REGION_END; add += STEP) {
    /*
     * Check for an already allocated region.
     */
    if (check_mem_region (add, 2048)) {
        printk(KERN_INFO "%lx: Allocated\n", add);
        continue;
    }
    /*
     * Read and write the beginning of the region and see what happens.
     */
    save_flags(flags);
    cli();
    oldval = readb (base + add); /* Read a byte */
    writeb (oldval^0xff, base + add);
    mb();
    newval = readb (base + add);
    writeb (oldval, base + add);
    restore_flags(flags);

    if ((oldval^newval) == 0xff) { /* we reread our change: it's RAM */
        printk(KERN_INFO "%lx: RAM\n", add);
        continue;
    }
    if ((oldval^newval) != 0) { /* random bits changed: it's empty */
        printk(KERN_INFO "%lx: empty\n", add);
        continue;
    }
}

/*
 * Expansion ROM (executed at boot time by the BIOS)
 * has a signature where the first byte is 0x55, the second 0xaa,
 * and the third byte indicates the size of such ROM
 */
if ( (oldval == 0x55) && (readb (base + add + 1) == 0xaa) ) {
    int size = 512 * readb (base + add + 2);
    printk(KERN_INFO "%lx: Expansion ROM, %i bytes\n",
           add, size);
}
```

```

    add += (size & ~2048) - 2048; /* skip it */
    continue;
}

/*
 * If the tests above failed, we still don't know if it is ROM or
 * empty. Since empty memory can appear as 0x00, 0xff, or the low
 * address byte, we must probe multiple bytes: if at least one of
 * them is different from these three values, then this is ROM
 * (though not boot ROM).
 */
printk(KERN_INFO "%lx: ", add);
for (i=0; i<5; i++) {
    unsigned long radd = add + 57*(i+1); /* a "random" value */
    unsigned char val = readb (base + radd);
    if (val && val != 0xFF && val != ((unsigned long) radd&0xFF))
        break;
}
printk("%s\n", i==5 ? "empty" : "ROM");
}

```

只需要注意恢复探测内存时修改的字节的原始值，这种探测并不会造成和其它设备的冲突。要注意的是，写入另一个设备的内存可能会引发该设备的一些不可预测的动作。一般情况下，只要有可能，应该尽量避免使用这种探测内存的方法，但在处理旧设备时经常不得不这样做。

8.5 向后兼容性

幸好，在基本硬件的访问方面变化很少。编写向后兼容的驱动程序时只需要记住有限的几点就行了。

硬件内存屏障在内核 2.0 版本是没有的。那时支持的平台上，不需要这类处理指令排序的功能。通过在驱动程序中包含 `sysdep.h` 头文件可以修正这个问题，它把硬件屏障定义为与软件屏障相同。

类似的，在旧内核中并不是所有的端口访问函数（`inb` 和相关函数）在所有体系结构上都能支持。特别是串操作指令，常常没有。我们没有在 `sysdep.h` 中提供这些函数：这不是个容易完成的任务，而且也不太值得，因为这些函数依赖于具体的硬件。

在 Linux 2.0 中，`ioremap` 和 `iounmap` 分别称为 `vremap` 和 `vfree`。参数和功能则完全相同。因此，通常把这两个函数定义成映射到旧的对应函数就行了。

不幸的是，尽管 `vremap` 在提供对“高端”内存（如 PCI 卡上的内存）的访问上和 `ioremap` 别无二致，它却不能重映射 ISA 内存段。在以前，对该内存段的访问是通过直接使用指针完成的，所以不需要重映射该地址空间。因此，一个更完整的 x86 平台、Linux 2.0 上实现 `ioremap` 的解决方法如下：

```

extern inline void *ioremap(unsigned long phys_addr, unsigned long size)
{
    if (phys_addr >= 0xA0000 && phys_addr + size <= 0x100000)
        return (void *)phys_addr;
    return vremap(phys_addr, size);
}

extern inline void iounmap(void *addr)
{

```



```

    if ((unsigned long)addr >= 0xA0000
        && (unsigned long)addr < 0x100000)
        return;
    vfree(addr);
}

```

如果在驱动程序中包含了 `sysdep.h` 头文件，就可以使用 `ioremap` 了，即使在访问 ISA 内存时也不会出问题。

内存区段的分配（`check_mem_region` 及相关函数）是在内核 2.3.17 引入的。在 2.0 和 2.2 内核没有这种内存分配的工具。如果包含了 `sysdep.h` 头文件，就可以随意使用这些宏了，因为在 2.0 和 2.2 上编译时，这三个宏是空的。

8.6 快速参考

本章引入下列与操纵硬件有关的符号：

```

#include <linux/kernel.h>
void barrier(void)

```

这个“软件”内存屏障要求编译器考虑执行到该指令时相关的所有内存中的变化。

```

#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);

```

硬件内存屏障。要求 CPU（和编译器）执行该指令时检查所有必须的内存读、写（或二者兼有）已经执行完毕。

```

#include <asm/io.h>
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);

```

这些函数读写 I/O 端口。如果用户空间的程序有访问端口的权限，也可以调用这些函数。

```

unsigned inb_p(unsigned port);
...

```

有时候需要用到 `SLOW_DOWN_IO` 来处理 x86 平台上的低速 ISA 板卡。如果 I/O 操作之后需要一小段延时，可以用上面介绍的函数的 6 个暂停式的变体。这些暂停式的函数都以 `_p` 结尾。

```

void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);

```

这些“串操作函数”为输入端口与内存区之间的数据传输做了优化。这类传输是通过对同一端口连续读写 `count` 次实现的。

```

#include <linux/ioport.h>

```

```
int check_region(unsigned long start, unsigned long len);
void request_region(unsigned long start, unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
```

为 I/O 端口分配资源的函数。**check** 函数在成功时返回 0，出错时返回负值。

```
int check_mem_region(unsigned long start, unsigned long len);
void request_mem_region(unsigned long start, unsigned long len, char *name);
void release_mem_region(unsigned long start, unsigned long len);
```

这些函数处理对内存区的资源分配。

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void *virt_addr);
```

ioremap 把一个物理地址段重新映射到处理器的虚拟地址空间，以供内核使用。**iounmap** 用来解除这个映射。

```
#include <linux/io.h>
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
memset_io(address, value, count);
memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);
```

用这些函数可以访问 I/O 内存区，包括低端的 ISA 内存和高端的 PCI 缓冲区。

第 9 章 中断处理



尽管有些设备仅通过它们的 I/O 寄存器就可以控制，但现实中的大部分设备却要比这复杂一些。设备需要与外部世界打交道，比如象旋转的磁盘、绕卷的磁带、远距离连接的电缆等等。这些设备的许多工作通常是在与处理器完全不同的时间周期内完成的，并且总是要比处理器慢。这种让处理器等待外部事件的情况总是不能令人满意的，所以必须有一种方法，可以让设备在产生某个事件时通知处理器。

这种方法就是中断。一个“中断”仅仅是一个信号，当硬件需要获得处理器对它的关注时，就可以发送这个信号。Linux 处理中断的方式很大程度上与它在用户空间处理信号是一样的。大多数情况下，一个驱动程序只需要为它自己设备的中断注册一个处理程序，并且在中断到达时进行正确的处理。当然，这个过程看似简单，但还是有一些复杂性的。需要特别指出的是，随着中断处理程序运行方式的不同，它们所能执行的动作将会受到不同的限制。

如果没有一个真正的硬件设备产生中断，就很难示范中断的使用方法。因而，本章中的样例代码利用并口来产生中断。我们将使用上一章的 `short` 模块来示范，作一些小的改动就可以通过并口来产生中断并处理中断。模块的名字 `short`，实际是指 `short int` (很象 C 语言)，提醒我们这个模块要处理中断。

9.1 中断的整体控制

由于设计上和硬件上的改变，Linux 处理中断的方法近几年来有所变化。早期 PC 中的中断处理很简单，中断的处理仅仅涉及到一个处理器和 16 条中断信号线，而现代硬件则可以有更多的中断，并且还可能装配价格高昂的高级可编程中断控制器 (APIC)，该控制器可以以一种智能 (和可编程) 的方式在多个处理器之间分发中断。

令人高兴的是，Linux 能够处理所有这些变化，但在驱动程序级却没有引入太多的非兼容性问题。这样，在不同的内核版本中，本章所描述的接口只有少许差别。有些情况下，问题可以得到很好的解决。

多年来，Unix 系列系统一直采用 `cli` 和 `sti` 函数来禁止和使能中断。在现代的 Linux 系统中，却不鼓励直接使用它们，对任意一个例程来讲，想要知道在它被调用时，中断是否被打开，变得越来越

不可能。在每个例程返回时使用 `sti` 打开中断并不是好习惯，因为你的函数可能会返回到一个期望中断仍然被禁止的函数。

因而，如果必须禁止中断，使用下列调用会是较好的选择：

```
unsigned long flags;
save_flags(flags);
cli();
/* This code runs with interrupts disabled */
restore_flags(flags);
```

注意 `save_flags` 是一个宏，并且保存标志的变量被直接赋值而没有“与”操作。使用这些宏也有一个重要的限制：`save_flags` 和 `restore_flags` 必须在同一个函数内被调用，换句话说，除非其它的函数是内联的，否则不能将 `flags` 传给其它的函数。忽略这一限制的代码可能会在某些体系结构上正常工作，但在其它的体系结构上会失败。

但是，无论在什么地方使用，我们都不鼓励读者使用前述的示例代码。在一个多处理器的系统中，关键代码是不能仅仅通过禁止中断来保护的，一些锁机制还是必须使用的。例如 `spin_lock_irqsave` 函数（本章后面“使用自旋锁”部分会论述）可以一起提供锁和中断控制。如果要在存在中断的情况下控制并发操作，这些函数是唯一真正安全的方法。

`cli` 期间，禁止系统上所有处理器的中断，因此会在总体上影响系统性能^{*}。因而，显式调用 `cli` 及其相关函数的代码正逐渐在大多数的内核中消失。需要在设备驱动中使用它们的场合目前也为数不多。在调用 `cli` 之前，应该考虑是否真的需要禁止系统上所有的中断。

9.2 准备并口

尽管并行口的接口很简单，但它也可以触发中断。打印机就是利用这种能力来通知 `lp` 驱动程序它已经准备好接受缓冲区中的下一个字符了。

就象大多数设备一样，在没有设定产生中断之前，并口是不会产生中断的；并口标准规定设置端口 `2` (`0x37a`、`0x27a` 或者其它端口) 的第 `4` 比特位将启动中断。`short` 模块在初始化的时候调用 `outb` 来设置该位。

在中断被使能的情况下，每当引脚 `10`（叫做 `ACK` 位）的电平发生从低到高的改变，并口就会产生一个中断，在没有把打印机连到端口上的情况下，强制接口产生中断的最简单的方法是连接并口插座的 `9` 脚和 `10` 脚。将一根短电线插入系统后面并口插座上的对应的孔，就可以连接这两个引脚。并口的引出线在图 `8-1` 中说明。

引脚 `9` 是并口数据字节中的最高位，如果将二进制数据写入 `/dev/short0`，就会引发几个中断，将 `ASCII` 码文本写入端口则不会产生中断，因为此时没有设置这个最高位。

如果读者手头有一台打印机，并且想要避免焊接电线，则可以运行本章后面针对真实打印机的中断处理程序。注意，我们将要介绍的探测函数依赖于在引脚 `9` 和 `10` 之间适当的跳线，因此，在使用

^{*} 实际情况要复杂一点，如果你正在处理一个中断，`cli` 调用只会在当前处理器上禁止中断。

这些代码作探测试验的时候需要它。

9.3 安装中断处理程序

如果读者确实想“看到”产生的中断，仅仅通过向硬件设备写入是不够的，必须要在系统中安装一个软件处理程序。如果 Linux 内核没有被通知硬件中断的发生，那么内核只会简单应答并忽略该中断。

中断信号线是非常珍贵且有限的资源，尤其是在系统上只有 15 根或 16 根中断信号线时更为如此。内核维护了一个中断信号线的注册表，它类似于 I/O 端口的注册表。模块在使用中断前要先申请一个中断通道（或者中断请求 IRQ），然后在使用后释放该通道。我们将会在后面看到，在很多场合下，模块也希望可以和其它的驱动程序共享中断信号线。下列在头文件<linux/sched.h>中声明的函数实现了该接口：

```
int request_irq(unsigned int irq,
               void (*handler)(int, void *, struct pt_regs *),
               unsigned long flags,
               const char *dev_name,
               void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

通常，从 request_irq 函数返回到调用函数的值，为 0 时表示申请成功，负值表示错误码。函数返回 -EBUSY 表示已经有另一个驱动程序占用了你要申请的中断信号线。该函数的参数如下：

```
unsigned int irq
```

这是要申请的中断号。

```
void (*handler)(int, void *, struct pt_regs *)
```

这是要安装的中断处理函数指针，我们会在本章的后面部分讨论这个函数的参数含义。

```
unsigned long flags
```

如读者所想，这是一个与中断管理有关的位掩码选项（后面描述）

```
const char *dev_name
```

传递给 request_irq 的字符串，用来在 /proc/interrupts 中显示中断的拥有者（参看下节）。

```
void *dev_id
```

这个指针用于共享的中断信号线。在释放中断信号线时，它是标识设备的唯一标识符，驱动程序也可以使用它指向驱动程序自己的私有数据区（用来识别哪个设备产生中断）。在没有强制使用共享方式时，dev_id 可以设置为 NULL，总之用它来指向设备的数据结构是一个比较好的思路。我们会在本章后面的“实现处理程序”中看到 dev_id 的实际应用。

可以在标志中设置的位如下所示：

```
SA_INTERRUPT
```

当该位被设置时，表明这是一个“快速”的中断处理程序，快速处理程序是运行在中断禁止状态下的（更详细的主题将在本章后面的“快速和慢速处理程序”小节中讨论）

SA_SHIRQ

该位表示中断可以在设备之间共享。共享的概念将在本章后面的“中断共享”小节描述。

SA_SAMPLE_RANDOM

该位指出产生的中断能对 `/dev/random` 设备和 `/dev/urandom` 设备使用的熵池有贡献。从这些设备读取，将会返回真正的随机数，从而有助于应用软件选择用于加密的安全密钥。这些随机数是从一个熵池中得到的，各种随机事件都会对该熵池作出贡献，如果读者的设备以真正随机的周期产生中断，就应该设置该标志位。另一方面，如果中断是可预期的（例如，帧捕获卡的垂直消隐），就不值得设置这个标志位——它对系统的熵没有任何贡献。能受到攻击者影响的设备不应该设置该位，例如，网络驱动程序会被外部的事件影响到预定的数据包的时间周期，因而也不会对熵池有贡献，更详细的信息请参见 `drivers/char/random.c` 文件中的注释。

中断处理程序可在驱动程序初始化时或者设备第一次打开时安装。虽然在模块初始化函数中安装中断处理程序看起来是个好主意，实际上并非如此。因为中断信号线的数量是非常有限的，我们不会想着肆意浪费。计算机拥有的设备通常要比中断信号线多得多，如果一个模块在初始化时请求了 `IRQ`，即使驱动程序只是占用它而从未使用，也将会阻止任意一个其它的驱动程序使用该中断。而在设备打开的时候申请中断，则可以共享这些有限的资源。

这种情况很可能出现，例如，在运行一个与调制解调器共用同一中断的帧捕获卡驱动程序时，只要不同时使用两个设备就是可以的。用户在系统启动时装载特殊的设备模块是一种普遍作法，即使该设备很少使用。数据捕获卡可能会和第二个串口使用相同的中断，我们可以在捕获数据时，避免使用调制解调器连接到互联网服务供应商（ISP），但是如果为了使用调制解调器而不得不卸载一个模块，却总是令人不快的。

调用 `request_irq` 的正确位置应该是在设备第一次打开、硬件被告知产生中断之前，调用 `free_irq` 的位置是最后一次设备关闭、硬件被告知不要再中断处理器之后。这种技术的缺点是必须为每个设备保存一个打开计数。如果使用一个模块控制两个或者更多的设备，那么仅仅使用模块计数是不够的。

尽管我们已经讨论了不应该在装载模块时调用 `request_irq`，但 `short` 模块还是在装载时申请了它的中断信号线，这样做的方便之处是，我们可以直接运行测试程序，而不需要额外运行其它的进程来保持设备的打开状态。因此，`short` 在它自己的初始化函数 `short_init` 中申请中断，而不是象真正的设备驱动那样在 `short_open` 中申请中断。

下面这段代码要申请的中断是 `short_irq`，对这个变量（例如，决定使用哪个 `IRQ`）的实际赋值操作会在后面给出，因此它与当前的讨论无关。`short_base` 是并口使用的 I/O 地址空间的基地址，向并口的 2 号寄存器写入，可以打开中断报告。

```
if (short_irq >= 0) {
    result = request_irq(short_irq, short_interrupt,
                        SA_INTERRUPT, "short", NULL);
    if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n",
               short_irq);
        short_irq = -1;
    }
    else { /* actually enable it -- assume this *is* a parallel port */
        outb(0x10, short_base+2);
    }
}
```

```
}

```

从代码能够看出，已经安装的中断处理程序是一个快速的处理程序（SA_INTERRUPT），不支持中断共享（没有设置 SA_SHIRQ），并且对系统的熵（也没有设置 SA_SAMPLE_RANDOM）没有贡献。最后，代码执行 `outb` 调用打开并口的中断报告。

9.3.1 /proc 接口

当硬件的中断到达处理器，一个内部计数会加 1，这为检查设备是否按预期工作提供了一种方法，产生的中断报告显示在文件 `/proc/interrupts` 中。下面是一个双处理器的奔腾系统启动几天后该文件的快照：

```

      CPU0      CPU1
0:   34584323   34936135   IO-APIC-edge timer
1:   224407    226473    IO-APIC-edge keyboard
2:      0       0        XT-PIC cascade
5:   5636751   5636666   IO-APIC-level eth0
9:      0       0    IO-APIC-level acpi
10:   565910    565269   IO-APIC-level aic7xxx
12:   889091    884276   IO-APIC-edge PS/2 Mouse
13:      1       0        XT-PIC fpu
15:  1759669   1734520   IO-APIC-edge idel
NMI:  69520392  69520392
LOC:  69513717  69513716
ERR:      0

```

第一列是 IRQ 号，其中明显缺少一些中断，这说明该文件只会显示那些已经安装了中断处理程序的中断。例如，第一个串口（使用中断号 4）没有显示，说明我没有使用调制解调器，实际上，即使早些时候已经使用过调制解调器，而在文件快照的时候没有使用的话，也不会出现在文件中。串口驱动程序具有良好的行为，当设备关闭的时候会释放它们的中断处理程序。

文件 `/proc/interrupts` 给出了已经发送到系统上每一个 CPU 的中断数量，正如读者能从输出中看到的，Linux 内核试图将中断均匀地分配到各个处理器上，而且比较成功。最后一列给出可编程中断控制器（驱动作者不需要关心的该控制器）处理中断的信息，还有已经注册了中断处理程序的设备名称（这和传递给 `request_irq` 的参数 `dev_name` 一样）。

`/proc` 树结构中还包含其它与中断相关的文件，象 `/proc/stat`。你有时会发现某个文件很有用，有时又会更喜欢使用其它的文件。`/proc/stat` 记录了一些系统活动的底层统计信息，包括（但不仅限于）从系统启动开始接收到的中断数量，`stat` 文件的每行都以一个字符串开始，它是这行的关键字。`intr` 标记正是我们需要的，下列（被截断和分行）快照是在前一个快照不久之后获得的：

```

intr 884865 695557 4527 0 3109 4907 112759 3 0 0 0 11314
    0 17747 1 0 34941 0 0 0 0 0 0 0

```

第一个数是所有中断的总数，而其它的每个数都代表一个单独的 IRQ 信号线，从中断 0 开始。这个快照显示 4 号中断已经发生 4907 次，尽管它的处理程序当前没有安装，如果测试的驱动程序在每次打开、关闭设备的循环内请求和释放中断的话，读者就会发现 `/proc/stat` 比 `/proc/interrupts` 更有用。

两个文件的另一个不同之处是 `interrupts` 文件不依赖于体系结构，而 `stat` 文件是依赖的：字段的数

量依赖于内核之下的硬件。可用的中断数量从 **sparc** 体系结构上的 15 个，到 **IA-64** 结构和一些其它系统上的 256 个之间变化。值得注意的是，当前 **x86** 体系结构上定义的中断数量是 224 个，不是读者猜测的 16 个。这可以从头文件 `include/asm-386/irq.h` 中得到解释，它取决于 **Linux** 使用的体系结构的限制而不是特定实现的限制（象 16 个中断源的老式 **PC** 中断控制器）。

下面是文件 `/proc/interrupts` 在一个 **IA-64** 系统上的快照，正如读者看见的，除了将常见中断源递交不同的处理器处理之外，这里没有任何的平台依赖性。

	CPU0	CPU1	
27:	1705	34141	IO-SAPIC-level qla1280
40:	0	0	SAPIC perfmon
43:	913	6960	IO-SAPIC-level eth0
47:	26722	146	IO-SAPIC-level usb-uhci
64:	3	6	IO-SAPIC-edge ide0
80:	4	2	IO-SAPIC-edge keyboard
89:	0	0	IO-SAPIC-edge PS/2 Mouse
239:	5606341	5606052	SAPIC timer
254:	67575	52815	SAPIC IPI
NMI:	0	0	
ERR:	0		

9.3.2 自动检测 IRQ 号

驱动程序初始化时，最迫切的问题之一就是如何决定设备将要使用哪条 **IRQ** 信号线。驱动程序需要这个信息以便正确地安装处理程序，尽管程序员可以要求用户在装载时指定中断号，但这不是一个好习惯，因为大部分时间用户不知道这个中断号，或者是因为他没有配置跳线或者是因为设备是无跳线的。因此，中断号的自动检测对于驱动程序可用性来说是一个基本要求。

有时，自动检测依赖于一些设备拥有的默认特性。既然如此，驱动程序可以假定设备使用了这些默认值。**Short** 在检测并口时默认就是这么做的，正如 **short** 的代码所给出的，实现是很简单的：

```
if (short_irq < 0) /* not yet specified: force the default on */
{
    switch(short_base) {
        case 0x378: short_irq = 7; break;
        case 0x278: short_irq = 2; break;
        case 0x3bc: short_irq = 5; break;
    }
}
```

这段代码根据选定的 **I/O** 地址的基地址分配中断号，也允许用户在装载时用 `insmod ./short.o short_irq=x` 来覆盖默认值。**Short_base** 默认为 0x378，所以 **short_irq** 默认为 7。

有些设备设计的更先进，会简单地“声明”它们要使用的中断。这样，驱动程序就可以通过从设备的某个 **I/O** 端口或者 **PCI** 配置空间中读出一个状态字来获得中断号。当目标设备有能力告诉驱动程序它将使用的中断号时，自动检测 **IRQ** 号只是意味着探测设备，不需要额外的工作来探测中断。

值得注意的是，现代的设备提供了它们的中断配置信息，**PCI** 标准通过要求外围设备声明它们要使用的中断信号线的方法来解决这个问题，**PCI** 标准将在第 15 章讨论。

令人遗憾的是，不是所有的设备都对程序员很友好，自动检测可能还是需要做一些探测工作。这技术上很简单：驱动程序通知设备产生中断并观察会发生什么，如果一切正常，那么只有一条中断信号线被激活。

尽管从理论上讲，探测过程很简单，但实际上实现起来可就不清晰了。我们看看执行该任务的两种方法：调用内核定义的辅助函数，或者实现我们自己的版本。

内核帮助下的探测

Linux 内核提供了一系列底层设施来探测中断号，它们只能在非共享中断的模式下工作，但是大多数硬件有能力工作在共享中断的模式下，并提供更好的找到配置中断号的方法。内核提供的这一设施由两个函数组成，在头文件<linux/interrupt.h>中声明（该文件也描述了探测方法）：

```
unsigned long probe_irq_on(void);
```

这个函数返回一个未分配中断的位掩码。驱动程序必须保存返回的位掩码，并且将它传递给后面的 `probe_irq_off` 函数，调用该函数之后，驱动程序要安排设备产生至少一次中断。

```
int probe_irq_off(unsigned long);
```

在请求设备产生中断之后，驱动程序调用这个函数，并将前面 `probe_irq_on` 返回的位掩码作为参数传递给它。`probe_irq_off` 返回“`probe_irq_on`”之后发生的中断次数，如果没有中断发生，就返回 0（因此，IRQ 0 不能被探测，但在任何已支持的体系结构上，没有任何设备能够使用 IRQ 0）。如果产生了多次中断（出现二义性），`probe_irq_off` 会返回一个负值。

程序员要注意在调用 `probe_irq_on` 之后启动设备上的中断，并在调用 `probe_irq_off` 之前禁止中断。此外，要记住在 `probe_irq_off` 之后，需要处理设备上的待处理的中断。

Short 模块演示了如何进行这样的探测。如果指定 `probe=1` 装载模块，并且并口连接器的引脚 9 和 10 相连，就会执行下面的代码进行中断信号线的检测：

```
int count = 0;
do {
    unsigned long mask;
    mask = probe_irq_on();
    outb_p(0x10, short_base+2); /* enable reporting */
    outb_p(0x00, short_base); /* clear the bit */
    outb_p(0xFF, short_base); /* set the bit: interrupt! */
    outb_p(0x00, short_base+2); /* disable reporting */
    udelay(5); /* give it some time */
    short_irq = probe_irq_off(mask);
    if (short_irq == 0) { /* none of them? */
        printk(KERN_INFO "short: no irq reported by probe\n");
        short_irq = -1;
    }
} /*
 * If more than one line has been activated, the result is
 * negative. We should service the interrupt (no need for lpt port)
 * and loop over again. Loop at most five times, then give up
 */
while (short_irq < 0 && count++ < 5);
if (short_irq < 0)
    printk("short: probe failed %i times, giving up\n", count);
```

在调用 `probe_irq_off` 之前应该注意 `udelay` 的用法，这取决于所使用的处理器速度，读者可能不得不安排一个很短的延时，以保证留给中断足够的传递时间。

如果读者钻研过内核源代码，也许会偶然发现其中提及了一对不同的函数：

```
void autoirq_setup(int waittime);
```

该函数设置中断探测，在这里忽略了 `waittime` 参数。

```
int autoirq_report(int waittime);
```

该函数延迟给定的时间（以 `jiffies` 计算），然后返回自调用 `autoirq_setup` 以后产生的 IRQ 数量。

这些函数最初是在网络驱动程序的代码中使用，由于历史原因，它们现在用 `probe_irq_on` 和 `probe_irq_off` 实现。通常没有必要使用 `autoirq_` 函数，而应该使用 `probe_irq_`。

探测是一个很耗时的任务，尽管 `short` 的探测耗时不多，但是像帧捕获卡的探测就至少需要 20 毫秒的延迟（这对处理器来说，已经是很长的时间了），而探测其它的设备可能要花费更多的时间。因此，最好的方法就是只在模块初始化的时候探测中断信号线一次，这与是否在设备打开时（应该这样做），或者在初始化函数内（不推荐这样做）安装中断处理程序，是相互独立的。

值得注意的是，在一些平台（PowerPC、M68k、大部分 MIPS 的实现以及两个 SPARC 版本）上，探测是没有必要的，因此前面的函数只是一些空的占位符，有时叫做“`useless ISA nonsense`”，在其它的平台上探测只是为 ISA 设备实现的，总之，大多数体系结构都定义了函数（甚至是空的）来简化现有的设备驱动程序的移植。

一般而言，探测是一种“黑客”行为，而象 PCI 总线这样成熟的体系结构会提供所有必要的信息。

DIY 探测

探测也可以由驱动程序自己实现。如果装载时指定 `probe=2`，`short` 模块将对 IRQ 信号线进行 DIY 探测。

这种机制与先前描述的内核帮助下的探测是一样的：启动所有未被占用的中断，然后等着看会发生什么。但是，我们要充分发挥对设备的了解。通常，设备可以使用 3 或 4 个 IRQ 号中的一个来进行配置，探测这些 IRQ 号，使我们能够不必测试所有可能的 IRQ 就检测到正确的 IRQ 号。

在 `short` 的实现中，我们假定可能的 IRQ 值是 3、5、7 和 9，这些数实际上是一些并口设备允许用户选择的值。

下面的代码通过测试所有“可能”的中断并观察将要发生的事情来进行中断探测。`trials` 数组列出了以 0 作为结束标志的需要测试的 IRQ，`tried` 数组用来记录哪个处理程序被驱动程序注册了。

```
int trials[] = {3, 5, 7, 9, 0};
int tried[] = {0, 0, 0, 0, 0};
int i, count = 0;

/*
 * Install the probing handler for all possible lines. Remember
 * the result (0 for success, or -EBUSY) in order to only free
 * what has been acquired
 */
for (i=0; trials[i]; i++)
    tried[i] = request_irq(trials[i], short_probing,
                          SA_INTERRUPT, "short probe", NULL);
```

```

do {
    short_irq = 0; /* none obtained yet */
    outb_p(0x10, short_base+2); /* enable */
    outb_p(0x00, short_base);
    outb_p(0xFF, short_base); /* toggle the bit */
    outb_p(0x00, short_base+2); /* disable */
    udelay(5); /* give it some time */

    /* the value has been set by the handler */
    if (short_irq == 0) { /* none of them? */
        printk(KERN_INFO "short: no irq reported by probe\n");
    }
    /*
     * If more than one line has been activated, the result is
     * negative. We should service the interrupt (but the lpt port
     * doesn't need it) and loop over again. Do it at most 5 times
     */
} while (short_irq <= 0 && count++ < 5);

/* end of loop, uninstall the handler */
for (i=0; trials[i]; i++)
    if (tried[i] == 0)
        free_irq(trials[i], NULL);

if (short_irq < 0)
    printk("short: probe failed %i times, giving up\n", count);

```

有时，我们无法预知“可能的”IRQ 值，这种情况下，需要探测所有的空闲中断号，而不仅是一些由 `trials[]` 数组列出的中断号。为了探测所有的中断，不得不从 IRQ 0 探测到 IRQ NR_IRQS-1，NR_IRQS 是在头文件 `<asm/irq.h>` 中定义的具有平台相关性的常数。

现在我们就剩下探测处理程序本身了，处理程序的任务是根据实际收到的中断号更新 `short_irq` 变量，`short_irq` 的值为 0 意味着“什么也没有”，负值意味着存在“二义性”，我们选择这些值是为了和 `probe_irq_off` 保持一致，并可以在 `short.c` 中使用同样的代码调用任何一种探测方法。

```

void short_probing(int irq, void *dev_id, struct pt_regs *regs)
{
    if (short_irq == 0) short_irq = irq; /* found */
    if (short_irq != irq) short_irq = -irq; /* ambiguous */
}

```

处理程序的参数稍后介绍。只要了解参数 `irq` 是要处理的中断号，就足以理解上面的函数了。

9.3.3 快速和慢速处理程序

老版本的 Linux 内核做了很多努力才区分出“快速”和“慢速”中断。快速中断是那些可以很快被处理的中断，然而处理慢速中断就会明显花费更长的时间。当慢速中断正被处理时，慢速中断要求处理器可以再次启动中断，否则，需要快速处理的任务可能会被延迟过长。

在现代的内核里面，很多快速中断和慢速中断的区别已经消失了。剩下的只有一个：快速中断（使用 SA_INTERRUPT 标志申请的中断）执行时，当前处理器上的其它所有中断都被禁止，注意其它的处理器仍然可以处理中断，尽管从来不会看到两个处理器同时处理同一 IRQ 的情况。

总结慢速中断和快速中断的执行环境：快速处理程序在微处理器的中断报告被禁止的状态下运行，并且在中断控制器上，正在处理的中断被禁止。虽然如此，中断处理程序也可以通过调用 `sti` 来启

动处理器上的中断报告。

一个慢速处理程序在使能处理器中断报告的状态下运行，并且在中断控制器上，正在处理的中断被禁止。

那么，读者的驱动程序应该使用哪种中断处理程序呢？在现代的系统中，`SA_INTERRUPT` 只是在少数几种特殊情况（例如定时器中断）下使用，读者不应该使用 `SA_INTERRUPT` 标志，除非有足够必要的原因想要在其它中断被禁止的时候，运行自己的中断处理程序。

这段论述足以满足大多数读者，但有些熟悉硬件或者对计算机有着强烈兴趣的读者，需要深入了解一些信息。如果不想了解内部细节，可以跳过下一节。

x86 平台上中断处理的内幕

下面的描述是从 2.4 内核中的三个文件 `arch/i386/kernel/irq.c`、`arch/i386/kernel/i8259.c` 和 `include/asm-i386/hw_irq.h` 得出的。虽然基本概念是相同的，但是硬件细节还是与其它平台有所区别。

最底层的中断处理是在头文件 `hw_irq.h` 中声明为宏的一些汇编代码，这些宏在文件 `i8259.c` 中展开。每一个中断都被联系到文件 `irq.c` 中定义的函数 `do_IRQ`。

`do_IRQ` 做的第一件事是应答中断，这样中断控制器就可以继续处理其它的事情了。然后该函数对于给定的 `IRQ` 号获得一个自旋锁，这样就阻止了任何其它的 `CPU` 处理这个 `IRQ`。接着清除几个状态位（包括一个我们很快就会讲到的 `IRQ_WAITING`），然后寻找这个特定 `IRQ` 的处理程序。如果没有处理程序，就什么也不做，自旋锁被释放，任何待处理的 `tasklet` 和底半部处理程序会运行，最后 `do_IRQ` 返回。

通常，如果设备有一个已注册的处理程序并且发生了中断，函数 `handle_IRQ_event` 会被调用以实际调用处理程序。它首先检测一个全局中断锁的位来启动，如果这个位是设置的，处理器会等待该位被清除，而调用 `cli` 可以设置这个位，这样就阻塞了中断的处理。正常的中断处理机制是不设置这个位的，这允许对中断的进一步处理。如果处理程序是慢速类型，将重新启动硬件中断，并调用处理程序，然后只是做一些清理工作，接着运行 `tasklet` 和底半部处理程序，最后返回到常规工作。作为中断的结果（例如，处理程序可以 `wake_up` 一个进程），“常规工作”可能已经改变，所以，从中断返回时发生的最后一件事情，可能就是一次处理器的重新调度。

`IRQ` 的探测是通过为每个缺少中断处理程序的 `IRQ` 设置 `IRQ_WAITING` 状态位来完成的。当中断产生时，因为没有处理程序被注册，`do_IRQ` 清除该位然后返回。当 `probe_irq_off` 被一个驱动程序调用的时候，只需要搜索那些没有设置 `IRQ_WAITING` 位的 `IRQ`。

9.4 实现中断处理程序

迄今为止，我们已经学会了如何注册一个中断处理程序，但是还没有编写过中断处理程序，实际上，处理程序没有什么与众不同的地方——它们也是普通的 `C` 代码。

唯一特殊的地方就是处理程序是在中断时间内运行的，因此它的行为会受到一些限制。这些限制与我们在任务队列中看到的一样，处理程序不能向用户空间发送或者接收数据，因为它不是在任何进程的上下文中执行的，处理程序也不能做任何可能发生睡眠的操作，例如调用 `sleep_on`，使用不带 `GFP_ATOMIC` 标志的分配内存操作，或者锁住一个信号量等等。最后，处理程序不能调用 `schedule` 函数。

中断处理程序的功能就是将有关中断接收的信息反馈给设备，并根据正在服务的中断的不同含义对数据进行相应的读写。第一步通常要清除接口卡上的一个位，大多数硬件设备在它们的“`interrupt-pending`”位被清除之前不会产生其它的中断。有些设备不需要这个步骤，因为它们没有“中断挂起”位，这样的设备是很少的，但并口设备却是其中的一种。由于这个原因，`short` 不需要清除这样的位。

中断处理程序的一个典型任务就是：如果中断通知进程所等待的事件已经发生，比如新的数据到达，就会唤醒在该设备上睡眠的进程。

还是举帧捕获卡的例子，一个进程通过连续地读设备来获取一系列图象，在读每一帧数据前，`read` 调用都是阻塞的，每当新的数据帧到达时，中断处理程序就会唤醒进程。这假定捕获卡中断处理器以通知每一帧数据的成功到达。

无论是快速还是慢速处理程序，程序员都应该编写执行时间尽可能短的处理例程。如果需要执行一个长时间的计算任务，最好的方法是使用 `tasklet` 或者任务队列在更安全的时间内调度计算任务（具体请参阅第 6 章的“任务队列”）。

`short` 的示例代码中，中断处理程序调用了 `do_gettimeofday`，并打印当前时间到大小为一页的循环缓冲区中，然后唤醒任何一个读进程，因为现在有新的数据可以读取。

```
void short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct timeval tv;
    int written;
    do_gettimeofday(&tv);
    /* Write a 16-byte record. Assume PAGE_SIZE is a multiple of 16 */
    written = sprintf((char *)short_head,"%08u.%06u\n",
                      (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* wake any reading process */
}
```

尽管上述代码很简单，却代表了中断处理程序的典型工作流程。它所调用的 `short_incr_bp` 函数定义如下：

```
static inline void short_incr_bp(volatile unsigned long *index,
                                int delta)
{
    unsigned long new = *index + delta;
    barrier (); /* Don't optimize these two together */
    *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
}
```

这个函数的实现非常仔细，它可以将指针限制在循环缓冲区的范围之内，并且不会因为传递一个不正确的值而返回。通过在最后赋值，并放置一个障碍函数（`barrier()`）来阻止编译器进行优化，这

样，在不使用锁的情况下，就可以安全地操作循环缓冲区指针。

用来读取我们在中断时间内填充的缓冲区的设备文件是 `/dev/shortint`。我们在第 8 章里并没有介绍这个设备特殊文件以及 `/dev/shortprint`，因为它们用法只是针对中断处理的。`/dev/shortint` 的内部实现是专门针对中断的产生和报告的。每向设备写一个字节就产生一次中断，而读设备时则给出每次中断报告产生的时间。

如果读者连接并口连接器的引脚 9 和 10，通过拉高并口数据字的高位就会产生中断，这可以通过向设备文件 `/dev/short0` 写入二进制数据或者向设备文件 `/dev/shortint` 写入任何数据来实现。^{*}

下面的代码为 `/dev/shortint` 实现了 `read` 和 `write` 系统调用。

```
ssize_t short_i_read (struct file *filp, char *buf, size_t count,
                     loff_t *f_pos)
{
    int count0;
    while (short_head == short_tail) {
        interruptible_sleep_on(&short_queue);
        if (signal_pending (current)) /* a signal arrived */
            return -ERESTARTSYS; /* tell the fs layer to handle it */
        /* else, loop */
    }
    /* count0 is the number of readable data bytes */
    count0 = short_head - short_tail;
    if (count0 < 0) /* wrapped */
        count0 = short_buffer + PAGE_SIZE - short_tail;
    if (count0 < count) count = count0;
    if (copy_to_user(buf, (char *)short_tail, count))
        return -EFAULT;
    short_incr_bp (&short_tail, count);
    return count;
}

ssize_t short_i_write (struct file *filp, const char *buf, size_t count,
                      loff_t *f_pos)
{
    int written = 0, odd = *f_pos & 1;
    unsigned long address = short_base; /* output to the parallel
                                         data latch */
    if (use_mem) {
        while (written < count)
            writeb(0xff * ((++written + odd) & 1), address);
    } else {
        while (written < count)
            outb(0xff * ((++written + odd) & 1), address);
    }
    *f_pos += count;
    return written;
}
```

其它的设备特殊文件，如 `/dev/shortprint`，使用并口来驱动一个打印机，如果读者想避免在 D-25 连接器的引脚 9 和 10 之间焊接一根电线的话就可以使用打印机，`shortprint` 的 `write` 实现，使用了一个循环缓冲区来存储被打印的数据，而 `read` 的实现是刚才介绍的那一种（所以读者可以读出打印机读入每个字符的时间）。

为了支持打印机操作，上面列出的中断处理程序被做了少许修改，增加了发送下一个数据字到打印

^{*} `shortint` 设备通过交替地向并口写入 0x00 和 0xff 来完成他的任务。

机的能力（如果有更多的数据需要传送的话）。

9.4.1 使用参数

虽然 `short` 没有对参数进行处理，但还是有三个参数传给了中断处理程序：`irq`、`dev_id` 和 `regs`，让我们看看每个参数的意义。

如果存在任何可以打印到日志的消息时，中断号（`int irq`）是很有用的，它主要用于 2.0 之前的内核（还没有 `dev_id` 时），现在，`dev_id` 可以更好地完成这一工作。

第二个参数 `void *dev_id`，是 `ClientData` 类型（即驱动程序可用的私有数据）。传递给 `request_irq` 函数的 `void *` 参数，会在中断发生时作为参数传回处理程序。

通常在 `dev_id` 中传递一个指向自己设备的数据结构指针，所以一个管理若干同样设备的驱动程序，在中断处理程序中不需要任何额外的代码，就可以找出哪个设备产生了当前的中断事件。中断处理程序中参数的典型用法如下：

```
static void sample_interrupt(int irq, void *dev_id, struct pt_regs
                           *regs)
{
    struct sample_dev *dev = dev_id;
    /* now `dev' points to the right hardware item */
    /* .... */
}
```

与这个处理程序相关联的典型 `open` 代码如下所示：

```
static void sample_open(struct inode *inode, struct file *filp)
{
    struct sample_dev *dev = hwinfo + MINOR(inode->i_rdev);
    request_irq(dev->irq, sample_interrupt,
    0 /* flags */, "sample", dev /* dev_id */);
    /*....*/
    return 0;
}
```

最后的参数 `struct pt_reg *regs` 很少使用，它保存了处理器进入中断代码之前的处理器上下文快照，寄存器可被用来监视和调试，对一般的设备驱动程序任务来说通常不是必须的。

9.4.2 打开和禁止中断

我们已经了解了 `sti` 和 `cli` 函数，它们可以打开和禁止所有的中断。有时，对于驱动程序来说，仅仅启动和禁止自己的 `IRQ` 信号线还是很有用的，内核为这一目的提供了三个函数，全部在头文件 `<asm/irq.h>` 中声明：

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

调用这些函数中的任何一个都会更新可编程中断控制器（`PIC`）中指定中断的掩码，因而，这样就可以在所有的处理器上禁止或者启动 `IRQ`。对这些函数的调用是可以嵌套的——如果 `disable_irq`

被成功调用两次，在 `IRQ` 真正重新打开之前，需要执行两次 `enable_irq` 调用。从一个中断处理程序中调用这些函数是可以的，但是在处理某个 `IRQ` 时再打开它，并不是一个好习惯。

`disable_irq` 不但会禁止给定的中断，而且也会等待当前正在执行的中断处理程序完成。另一方面，`disable_irq_nosync` 是立即返回的。这样，使用后者将会更快，但是可能会让你的驱动程序处于竞态下。

但为什么还要禁止中断呢？还是举并口的例子，先看看 `plip` 网络接口。一个 `plip` 设备使用裸的并口传送数据，因为并口连接器上只有 5 个位可以读，它们被解释为 4 个数据位和一个时钟/握手信号。当发起者（发送数据包的那个接口）送出一个包的头 4 个位时，时钟信号升高，造成接收方接口去中断处理器，然后，`plip` 的处理程序就会被调用，以便处理最新到达的数据。

在设备被通知之后，数据的传输将继续进行。这里，`plip` 使用握手信号线和接收方保持同步（这可能不是最好的实现，但是和其它使用并口的数据包驱动程序保持兼容是必要的）。如果接收接口每接收一个字节都要处理两次中断，那么性能显然是不可忍受的。因此驱动程序在接收数据包的时候禁止中断，否则，需要使用“轮询并延迟”循环来接收数据。

同样地，因为接收方到发送方的握手信号被用来应答数据的接收，所以发送接口也要在发送数据包时禁止它自己的 `IRQ` 信号。

最后，值得注意的是在 `SPARC` 和 `M68k` 实现中，符号 `disable_irq` 和 `enable_irq` 被定义为指针而不是函数，这个技巧允许内核在启动时根据实际运行的平台给指针赋值。在所有的 `Linux` 系统上，不管是否使用这个技巧，函数在 `C` 语言中的语义都是相同的，这帮助我们避免编写那些单调乏味的条件编译代码。

9.5 tasklet 和底半部处理

中断处理的一个主要问题是怎样在处理程序内完成耗时的任务。响应一次设备中断需要完成一定数量的工作，但是中断处理程序需要尽快结束而不要使中断阻塞的时间过长，这两个需求（工作和速度）彼此冲突，让驱动程序的作者有点困扰。

`Linux`（连同很多其它的系统）通过将中断处理程序分成两部分来解决这个问题，叫做“顶半部”的部分，是实际响应中断的例程，也就是用 `request_irq` 注册的中断例程；“底半部”是一个被顶半部调度，并在稍后更安全的时间内执行的例程。在 2.4 内核中，底半部这一术语的使用有一点混乱，底半部有时指的是中断处理程序的第二部分，有时指一种用来实现这个第二部分的机制，或者两者都是。当我们提到一个“底半部（bottom half）”时，通常我们的话题正围绕一个底半部。过去老的 `Linux` 底半部实现被提及时，用只取首字母的缩写词 `BH` 来表示。

但是底半部有什么用处呢？

顶半部处理程序和底半部处理程序之间最大的不同，就是当底半部处理程序执行时，所有的中断都是打开的——这就是所谓的在更安全时间内运行。典型的情况是顶半部保存设备的数据到一个设备特有的缓冲区并调度它的底半部，然后退出：这些处理是非常快的。然后，底半部执行其它必要的工作，例如唤醒进程，启动另外的 `I/O` 操作等等。这种方式允许在底半部工作期间，顶半部还可以

继续为新的中断服务。

任何一个严格的中断处理程序都是以这种方式分成两部分的。例如，当一个网络接口报告有新数据包到达时，处理程序仅仅接收数据并将它推到协议层上，实际的包处理过程是在底半部执行的。

另外一个值得注意的是，所有应用于中断处理程序的限制也在底半部处理中适用。这样，底半部不可以睡眠，不可以访问用户空间，不可以调用调度器。

Linux 内核有两种不同的机制可以用来实现底半部处理。**tasklet** 将在稍后的 2.3 开发系列中介绍，它们现在是进行底半部处理的首选方法，但是它们不能移植到早期的内核版本上。尽管 2.4 内核用 **tasklet** 实现了底半部，但较老的底半部（BH）实现存在于非常老的内核中，因此，我们将在这一节讨论这两种机制。一般而言，如果可能的话，编写新代码的设备驱动程序应该尽量选择 **tasklet** 作为它们的底半部处理手段，但如果从可移植性上考虑，则需要用 BH 机制替代 **tasklet**。

下面再用 **short** 驱动程序来进行我们的讨论。在使用某个模块选项装载时，可以通知 **short** 模块使用一个 **tasklet** 或者底半部处理程序，并且使用顶/底半部的模式进行中断处理。因此，顶半部执行的就很快，因为它仅保存当前时间并调度底半部处理，然后底半部管理这些时间的编码，并唤醒可能等待数据的任何用户进程。

9.5.1 tasklet

我们已经在第 6 章中介绍了 **tasklet**，这里首先回顾一下。记住 **tasklet** 是一个可以在中断上下文、在由系统决定的安全时刻被调度运行的特殊函数。它们可以被调度运行多次，但是实际只会运行一次。不会有同一 **tasklet** 的多个实例并行地运行，因为它们只运行一次，但是 **tasklet** 可以与其它 **tasklet** 并行地运行在对称多处理器系统上。这样，如果驱动程序有多个 **tasklet**，它们必须使用某种锁来避免彼此冲突。

tasklet 可确保和第一次调度它们的函数运行在同样的 CPU 上。这样，因为 **tasklet** 在中断处理程序结束前并不会开始运行，所以，此时的中断处理程序是安全的。不管怎样，在 **tasklet** 运行时，当然可以有其它的中断发生，因此在 **tasklet** 和中断处理程序之间的锁还是需要的。

必须使用宏 **DECLARE_TASKLET** 声明 **tasklet**：

```
DECLARE_TASKLET(name, function, data);
```

name 是给 **tasklet** 起的名字，**function** 是执行 **tasklet** 时调用的函数（它带有一个 **unsigned long** 型的参数并且返回 **void**），**data** 是一个用来传递给 **tasklet** 函数的 **unsigned long** 类型的值。

驱动程序 **short** 如下声明它自己的 **tasklet**：

```
void short_do_tasklet (unsigned long);
DECLARE_TASKLET (short_tasklet, short_do_tasklet, 0);
```

函数 **tasklet_schedule** 用来调度一个 **tasklet** 运行。如果指定 **tasklet=1** 装载 **short**，它会安装一个不同的中断处理程序，这个处理程序保存数据并如下调度 **tasklet**：

```

void short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    do_gettimeofday((struct timeval *) tv_head); /* cast to stop
    'volatile' warning */
    short_incr_tv(&tv_head);
    tasklet_schedule(&short_tasklet);
    short_bh_count++; /* record that an interrupt arrived */
}

```

实际的 **tasklet** 例程, **short_do_tasklet**, 将会在系统方便时得到执行。就象先前提到的, 这个例程执行中断处理的大多数任务, 看起来这样:

```

void short_do_tasklet (unsigned long unused)
{
    int savecount = short_bh_count, written;
    short_bh_count = 0; /* we have already been removed from queue */
    /*
    * The bottom half reads the tv array, filled by the top half,
    * and prints it to the circular text buffer, which is then consumed
    * by reading processes
    */

    /* First write the number of interrupts that occurred before
    this bh */

    written = sprintf((char *)short_head,"bh after %6i\n",savecount);
    short_incr_bp(&short_head, written);

    /*
    * Then, write the time values. Write exactly 16 bytes at a time,
    * so it aligns with PAGE_SIZE
    */

    do {
        written = sprintf((char *)short_head,"%08u.%06u\n",
            (int)(tv_tail->tv_sec % 100000000),
            (int)(tv_tail->tv_usec));
        short_incr_bp(&short_head, written);
        short_incr_tv(&tv_tail);
    } while (tv_tail != tv_head);

    wake_up_interruptible(&short_queue); /* wake any reading process */
}

```

在其它动作之外, 这个 **tasklet** 记录了自从它上次被调用以来产生了多少次中断的记录。一个类似于 **short** 的设备可以在很短的时间内产生很多次中断, 所以, 在底半部被执行前, 肯定会有多次中断发生。驱动程序必须一直对这种情况有所准备, 并且必须能根据顶半部保留的信息知道有多少工作需要完成。

9.5.2 BH 机制

不象 **tasklet**, 旧式的 **BH** 底半部存在的时间已经和内核本身一样长了, 它们以很多种方式表现了它们的长期存在。例如, 所有 **BH** 底半部都在内核中被预先定义, 并且可以最多有 32 个, 因为它们都是预先定义的, 所以底半部不能直接被模块使用, 但是实际上这不是问题。

无论何时, 只要有代码想要调度一个底半部去运行, 它可以调用 **mark_bh**。在较老的 **BH** 实现中, **mark_bh** 会设置位掩码中的一个位, 使相应的底半部处理程序在需要运行的时刻可以快速定位。在现代的内核中, 它只是调用 **tasklet_schedule** 来调度底半部例程执行。

标记底半部的函数在头文件 `<linux/interrupt.h>` 中这样定义：

```
void mark_bh(int nr);
```

这里，`nr` 是要激活的 BH 的“数量”，这个数是一个定义在头文件 `<linux/interrupt.h>` 中的符号常量，用来识别要运行的底半部。对应于每个底半部的函数，是由拥有底半部的驱动程序提供的。例如，当 `mark_bh(SCSI_BH)` 被调用时，被调度执行的函数是 `scsi_bottpm_half_handler`，它是 SCSI 驱动程序中的一部分。

前面提过，底半部是静态对象，所以一个模块化的驱动程序不能注册自己的 BH。目前还不支持 BH 底半部的动态分配，当然也未必永远是这样，幸运的是，我们可以使用立即任务队列来替代。

本节其余部分列出了一些有意思的底半部。它们描述了内核是怎样运行 BH 底半部的，为了正确使用底半部，首先需要理解这些底半部。

内核声明了若干值得我们关注的 BH 底半部处理，先前曾提到，其中有几个甚至可以由驱动程序使用。下面简单描述几个最重要的 BH：

IMMEDIATE_BH

该底半部对驱动程序开发者来讲最为重要。被调度的函数通过调用 `run_task_queue` 函数而执行 `tq_immediate` 任务队列。一个不拥有自身底半部处理的驱动程序，比如某个定制模块，可以将该立即队列看成是自己所拥有的 BH 那样使用。驱动程序在该队列当中注册了一个任务之后，它必须标记该 BH，以便使其代码得到实际执行。我们已经在第 6 章“立即对列”中对此进行过描述。

TQUEUE_BH

当有任务注册于 `tq_timer` 队列当中时，每一个时钟嘀哒都会激活该 BH。实际上，驱动程序可以利用 `tq_timer` 实现自己的 BH。我们在第 6 章“定时器队列”当中介绍过的定时器队列就是一个 BH，但没有必要为该底半部处理调用 `mark_bh`。

TIMER_BH

该 BH 由负责管理时钟嘀哒的 `do_timer` 函数标记。该 BH 所执行的函数也就是驱动内核定时器的函数。除使用 `add_timer` 之外，驱动程序无法通过其它的途径使用这一内核设施。

其余的 BH 底半部处理由特定的内核驱动程序使用。对模块而言，没有任何的入口点可以使用这些底半部处理，实际上也没有任何意义去定义这样的入口点。随着越来越多的驱动程序转而使用 `tasklet`，这种底半部处理的数目正在稳步减少。

一旦某个 BH 被标记，该 BH 将在 `bh_active` (`kernel/softirq.c`) 被调用时得到执行，而这个过程发生在 `tasklet` 得以运行的时候。整个过程又发生在某个进程从系统调用当中退出，或者某个中断处理过程退出时。`tasklet` 始终作为定时器中断的一部分而执行，所以，当驱动程序调度某个底半部例程之后，通常该例程能够在最多 10ms 之后得以执行。

9.5.3 编写 BH 底半部

从前面“BH 机制”一节给出的可利用的底半部列表中可以明显看出，一个实现了底半部的驱动程

序，可以通过使用立即队列来将它的代码挂在 `IMMEDIATE_BH` 上。

当 `IMMEDIATE_BH` 被标记，负责立即底半部的函数就会去处理立即队列。如果我们的中断处理程序将它的 `BH` 处理程序排入 `tq_immediate` 队列，并且标记了 `IMMEDIATE_BH` 底半部，排队的任务将会在适当的时间被调用。因为在所有我们感兴趣的内核中，都可以将相同的任务多次排队而不会破坏任务队列，因此，可以在每次顶半部运行的时候将自己的底半部排队。我们接下来就可以看到这种方法。

需要特殊配置的驱动程序（多个底半部或者其它不能简单地用普通的 `tq_immediate` 来处理的驱动程序）可以使用定制的任务队列。中断处理程序将任务排入自己的任务队列，并且当它准备运行这些任务时，就将一个简单的对任务队列进行处理的函数插入立即队列。具体细节请看第 6 章“运行自己的任务队列”一节。

现在让我们看看 `short` 模块的 `BH` 实现。装载时如果指定 `bh=1`，那么模块就会安装一个使用了 `BH` 底半部的中断处理程序：

```
void short_bh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    /* cast to stop 'volatile' warning */
    do_gettimeofday((struct timeval *) tv_head);
    short_incr_tv(&tv_head);
    /* Queue the bh. Don't care about multiple enqueueing */
    queue_task(&short_task, &tq_immediate);
    mark_bh(IMMEDIATE_BH);
    short_bh_count++; /* record that an interrupt arrived */
}
```

正如所料，这段代码调用 `queue_task` 而没有检查任务是否已经排入队列。

然后，底半部处理余下的工作。这个 `BH` 实际上是同前面提到的 `short_do_tasklet` 是一样的。

下面是装载 `short` 模块时指定 `bh=1` 读者所看到的输出结果：

```
morgana% echo 1122334455 > /dev/shortint ; cat /dev/shortint
bh after      5
50588804.876653
50588804.876693
50588804.876720
50588804.876747
50588804.876774
```

读者所看到的实际时间会有所不同，当然，这取决于读者自己的系统。

9.6 中断共享

“IRQ 冲突”这种说法和“PC 架构”几乎是同义语。通常，PC 上的 IRQ 信号线不能为一个以上的设备服务，它们从未都是不够用的，结果，许多没有经验的用户总是花费很多时间，试图找到一种方法使所有的硬件能够协同工作，这样，他们不得不总是打开着自己的计算机。

但是实际上，从硬件本身的设计上看，并没有表明中断信号线不能被共享，问题出在软件这边。随

着 PCI 总线的出现，系统软件的作者不得不花费更多的功夫，因为所有的 PCI 中断都可以被共享。Linux 支持中断的共享——而且是在所有它所关注的总线上，不仅仅是 PCI，这样，ISA 设备的驱动程序也可以共享 IRQ 信号线。

在 ISA 总线下，中断共享的问题引出了边缘触发中断信号线和水平触发中断信号线之间的问题。尽管前者对共享来说是安全的，但如果未经正确处理，还是可能导致软件锁住。而边缘触发的中断对共享来说不是安全的，ISA 总线的中断是边缘触发类型的，因为这个触发信号动作在硬件级别上易于实现，因而在 19 世纪 80 年代是通常的选择。这个问题与电平高低无关，为了支持共享，不管是水平触发还是边缘触发，中断信号线必须能够被多个中断源驱动激活。

具有水平触发中断信号线的外围设备在软件清除待处理的中断（通常通过向设备的寄存器写入数据）之前，会一直保持 IRQ 信号。因此，如果有多个设备同时激活信号线，那么只要 IRQ 是打开的，CPU 就会发出中断直到所有的驱动程序已经完成对设备的服务。这种行为对于共享是安全的，但如果一个驱动程序忘记清除它的中断源就可能导致锁住。

而在使用边缘触发类型的中断时，中断就有可能丢失：如果一个设备拉起信号线的时间过长，而这时恰好有另一个设备试图拉起信号线时，就不会产生边缘，这样处理器会忽略第二个请求。一个共享的处理程序也许正好没有看到第二个中断，而且如果硬件没有释放该 IRQ 线，则处理程序根本就不会注意到其它共享设备的中断。

由于这个原因，即使中断共享在 ISA 上是支持的，却可能无法正常运行。当某些设备在一个时钟周期内激活了 IRQ 信号线，而其它的设备却不能很好地配合，这样就会导致驱动程序编写者很难实现共享 IRQ。我们不会在这个问题上进一步探讨，在本节余下的部分我们假定主机总线支持共享，或者读者知道自己在做什么。

为了开发能够处理共享中断信号线的驱动程序，需要考虑到一些细节。正如后面讨论的，使用共享中断的设备不能使用本章描述的一些特性。只要有可能，最好还是支持共享，因为这样对最终用户比较方便，某些情况下（例如，当使用 PCI 总线），中断共享是强制的。

9.6.1 安装共享的处理程序

就像普通中断一样，共享的中断也是通过 `request_irq` 安装的，但是有两处不同：

- 申请中断时，必须指定 `flags` 参数中的 `SA_SHIRQ` 位。
- `dev_id` 参数必须是唯一的。任何指向模块地址空间的指针都可以，`dev_id` 不能设置成 `NULL`。

内核为每个中断维护了一个共享处理程序的列表，这些处理程序的 `dev_id` 各不相同，就像是设备的签名。如果两个驱动程序在同一个中断上都注册 `NULL` 作为它们的签名，那么在卸载的时候会混淆起来，当中断到达时造成内核出现 oop 消息。由于这个原因，在注册共享中断时如果传递了值为 `NULL` 的 `dev_id`，现代内核会给出警告。

当请求一个共享中断时，如果中断信号线空闲，或者任何已经注册了该中断的处理程序标识了 IRQ 是共享的，那么 `request_irq` 就会成功。在内核 2.0，所有共享中断的处理程序都必须是快速的或都是慢速的——两种模式不能被混合使用。

无论何时当两个或者更多的驱动程序共享同一根中断信号线，而硬件又通过这根信号线中断处理器时，内核调用每一个为这个中断注册的处理程序，并将它们自己的 `dev_id` 传回去。因此，一个共享的处理程序必须能够识别属于自己的中断，并且在自己的设备没有中断的时候迅速退出。

如果读者在申请中断请求信号线之前需要探测设备的话，内核不会有所帮助，对于共享的处理程序是没有探测函数可以利用的。仅当要使用的中断信号线处于空闲时，标准的探测机制才能工作。但如果信号线已经被其它具有共享特性的驱动程序占用的话，即使你的驱动已经可以很好的工作了，探测也会失败。

在探测共享信号线时，唯一可以利用的技术是 **DIY**。驱动程序为每一个可能的 **IRQ** 信号线申请共享处理程序，然后检查中断在何处报告，与 **DIY** 探测不同的是，探测处理函数必须检查设备是否真的发生中断，因为它可能为了响应在同一根线上的其它设备的中断，而已经被调用了。

释放处理程序同样是通过执行 `release_irq` 来实现的。这里 `dev_id` 参数被用来从该中断的共享处理程序列表中选择正确的处理程序来释放，这就是为什么 `dev_id` 指针必须唯一的原因。

使用共享处理程序的驱动程序需要小心一件事情：不能使用 `enable_irq` 和 `disable_irq`。如果使用了，共享中断信号线的其它设备就无法正常工作了。通常，程序员必须记住他的驱动程序并不独占 **IRQ**，所以它的行为必须比独占中断信号线时更“社会化”。

9.6.2 运行处理程序

如上所述，当内核收到中断，所有已注册的处理程序都被调用，一个共享处理程序必须能够将要处理的中断和其它设备产生的中断区分开。

装载 `short` 时，如果指定 `shared=1`，将安装下面的处理程序而不是默认的处理程序：

```
void short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int value, written;
    struct timeval tv;

    /* If it wasn't short, return immediately */
    value = inb(short_base);
    if (!(value & 0x80)) return;

    /* clear the interrupting bit */
    outb(value & 0x7F, short_base);

    /* the rest is unchanged */

    do_gettimeofday(&tv);
    written = sprintf((char *)short_head, "%08u.%06u\n",
                     (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* wake any reading process */
}
```

解释如下，因为并口没有“`interrupt-pending`”位可以检查，为此处理程序使用 **ACK** 位，如果该位为高，报告的中断就是送给 `short` 的，然后处理程序清除该位。

处理程序通过将并口的数据端口的高位清零来重新设置该位——**short** 假定并口的引脚 9 和 10 是连接在一起的。如果一个与 **short** 共享 IRQ 的其它设备产生了中断，**short** 知道它自己的信号线没有被激活，所以不会做任何工作。

一个功能完整的驱动程序可能会将任务分成顶半部和底半部，当然这很容易添加，并且对于实现共享的代码没有任何影响。一个真正的驱动程序或许会使用 **dev_id** 参数来判断产生中断的某个或多个设备。

注意，如果读者使用一个打印机（顶替跳线）来检验 **short** 的中断管理，这个共享的中断处理程序不会按预期工作，因为打印机协议不允许共享，而且驱动程序也无从知道中断是否是由打印机产生的。

9.6.3 /proc 接口

在系统上安装共享的中断处理程序不会对 **/proc/stat** 造成影响，它甚至不知道哪些处理程序是共享的，但是，**/proc/interrupts** 会有稍许改变。

所有为同一个中断号安装的处理程序会出现在 **/proc/interrupts** 文件的同一行上，下面的输出说明了共享的中断处理程序是怎样显示的：

```

          CPU0      CPU1
0:    22114216    22002860   IO-APIC-edge timer
1:     135401     136582   IO-APIC-edge keyboard
2:         0         0      XT-PIC cascade
5:    5162076    5160039   IO-APIC-level eth0
9:         0         0   IO-APIC-level acpi, es1370
10:    310450     312222   IO-APIC-level aic7xxx
12:    460372     471747   IO-APIC-edge PS/2 Mouse
13:         1         0      XT-PIC fpu
15:    1367555    1322398   IO-APIC-edge idel
NMI:    44117004    44117004
LOC:    44116987    44116986
ERR:         0
```

这里，被共享的中断信号线是 **IRQ 9**，活动的处理程序被列在同一行，用逗号分割。这里是由电源管理子系统（“**acpi**”）与声卡（“**es1370**”）共享这个 **IRQ**。内核不能区分中断是从这两个中断源中的哪个产生的，然后会为每次中断调用每一个驱动程序的中断处理程序。

9.7 中断驱动的 I/O

如果与驱动程序管理的硬件之间的数据传输因为某种原因被延迟的话，驱动程序作者应该实现缓冲。数据缓冲区有助于将数据的传送和接收与系统调用 **write** 和 **read** 分离开来，从而提高系统的整体性能。

一个好的缓冲机制需要采用中断驱动的 **I/O**，这种模式下，一个输入缓冲区在中断时间内被填充，并由读取该设备的进程取走缓冲区内的数据；一个输出缓冲区由写设备的进程填充，并在中断时间内取走数据。一个中断驱动输出的例子是 **/dev/shortint** 的实现。

中断驱动的数据传输要正确的进行，要求硬件应该能按照下面的语义来产生中断：

对于输入来说，当新的数据已经到达并且处理器准备好接收它时，设备就中断处理器。实际执行的

动作取决于设备使用的是 I/O 端口、内存映射还是 DMA。

对于输出来说，当设备准备好接收新数据或者对成功的数据传送进行应答时，就要发出中断。内存映射和具有 DMA 能力的设备，通常通过产生中断来通知系统它们对缓冲区的处理已经结束。

read 或者 write 与实际数据到达之间的时序关系已经在第 5 章“阻塞和非阻塞操作”一节中介绍过。但是中断驱动的 I/O 引入了并发访问共享数据项的同步问题，以及所有涉及到竞态的问题。下一节将进一步讨论这个主题。

9.8 竞态

我们已经看到竞态在前一章出现了很多次。鉴于竞态可以在对称多处理器系统中的任何时刻出现，因此，尽管在单处理器系统上很少出现，但也不得不考虑竞态了。^{*}

不管怎样，中断可以带来一系列新的竞态，甚至在单处理器系统上也是这样。因为中断可以在任何时间产生，所以它会造成中断处理程序能在任何一段驱动程序代码的中间执行，这样，任何使用中断工作的设备驱动程序——它们中的大部分——必须非常仔细地处理竞态。因为这个原因，我们将在本章更详细地讨论竞态，并讨论如何防止它们。

处理竞态是编程中最麻烦的部分，因为相关的程序错误很微妙，并且难以再现，所以很难分辨出在中断代码和驱动程序的方法之间什么时候会有竞态。程序员必须特别小心地避免数据或元数据的冲突。

防止数据冲突可以使用不同的技术，接下来我们会介绍几种最常用的方法。我们不会给出完整的代码，因为各种情况下，最好的代码取决于被驱动的设备的操作模式，以及程序员的不同风格。本书中所有的驱动程序都做了保护来防止竞态，在样例代码中找到这样的例子。

最通用的保护数据并发访问的方法象下面这样：

- 使用循环缓冲区并且避免共享变量
- 使用自旋锁强制实现互斥访问
- 使用原子地增加或减少的锁变量

注意，信号量没有在这里列出，因为锁住一个信号量可能会造成一个处理器睡眠，信号量不应该在中断处理程序中使用。

无论读者选择什么方法，在访问一个可以在中断时间内被修改的变量时，都需要决定如何处理。在简单的情况下，这样的变量可以简单地声明为 **volatile** 的，来阻止编译器对该值的访问进行优化（例如，它阻止编译器在整个函数的运行期内将这个值放在寄存器中）。但是，在使用 **volatile** 变量之后，编译器会产生不理想的代码，所以读者可以选择求助于某种锁来替代，在更多复杂的情况下，没有其它选择，只能使用某种锁。

^{*} 注意，不管怎样，内核开发者正在认真考虑即使是在单处理器系统上，也应该确保所有的内核代码几乎在任意的

9.8.1 使用循环缓冲区

使用循环缓冲区是一种可以有效处理并发访问问题的方法，处理并发访问最好的方法是不允许并发访问。

循环缓冲区使用一种叫做“生产者和消费者”的算法：一个进程将数据放入缓冲区，另一个进程将数据取出。如果只有一个生产者和一个消费者，那就避免了并发访问，在 **short** 中有两个关于生产者和消费者的例子。其中一个情形是，读进程等待取出中断时间内产生的数据。另一个情形是，底半部取出顶半部产生的数据。

两个指针被用来寻址循环缓冲区：**head** 和 **tail**。**head** 是指向数据写入的位置并且只能被数据的生产者更新的指针，数据被从 **tail** 读出，它只能被消费者更新。象前面提到的，如果数据在中断时间内写入，当多次访问 **head** 时读者必须小心，或者作为 **volatile** 声明或者使用某种锁。

循环缓冲区在填满之前运行的很好，如果缓冲区满了，就可能出现问题的，读者可以选择不同的解决方法。**short** 的实现是并不检查溢出而直接丢弃数据。如果 **head** 超过 **tail**，整个缓冲区的数据被丢弃，可替代的实现是丢弃最旧的数据，覆盖缓冲区的 **tail**，就像 **printk** 那样（请见第 4 章“消息是如何记录的”一节）；或者阻塞生产者，就像 **scullpipe** 那样；或者分配临时的附加缓冲区来备份主缓冲区。最好的解决方案取决于数据的重要程度和其它一些具体问题，所以我们就不在这里讨论了。

尽管循环缓冲区能解决并发访问的问题，但是当 **read** 函数睡眠的时候还是有可能产生竞态。这段代码给出在 **short** 中这个问题出现的位置：

```
while (short_head == short_tail) {
    interruptible_sleep_on(&short_queue);
    /* ... */
}
```

当执行到这一语句时，新的数据有可能在 **while** 条件被测试为真之后和进程进入睡眠之前到达，中断中携带的信息就不会被进程读取，即使此时 **head!=tail**，进程也会进入睡眠，直到下一个数据项到达时才能被唤醒。

我们没有为 **short** 实现正确的锁，因为 **short_read** 的源代码在第 8 章“样例驱动程序”一节中已经包括了，在那时还没有讨论这一点，而且 **short** 所处理的数据也不值得这么做。

尽管 **short** 采集的数据不是很重要，而且在两条连续指令之间发生中断的可能性也常常可以忽略，有些时候还是不能在有数据待处理的时候冒险地进入睡眠。这个问题通常还是值得特别对待的，我们将留在本章的后面“无竞争地进入睡眠”一节进行详细地讨论。

值得注意的是，循环缓冲区只能解决生产者和消费者的情形。程序员必须经常处理更复杂的数据结构来解决并发访问的问题。生产者/消费者情形实际上是这类问题中最简单的一种。其它的数据结构，例如链表，就不能简单的借用循环缓冲区的实现方案。

时刻都是可抢占的，同时应该遵循强制性的锁。

9.8.2 使用自旋锁

我们已经在前面看到过自旋锁，例如，在 `scull` 驱动程序中。之前的讨论只是给出了自旋锁的少许用法，本节中我们将介绍更多的细节。

记住，一个自旋锁基于一个共享变量来工作。函数可以通过给变量设置一个特殊的值来获得锁，任何其它需要锁的函数就会查询它并知道锁现在不可用，然后在一个忙等待的循环中“自旋”直到锁可用为止。因此自旋锁应该小心使用，持有自旋锁过长时间的函数会浪费更多的时间，因为其它的 CPU 被强制等待。

自旋锁使用类型 `spinlock_t` 来描述，连同各种各样的自旋函数，都是在头文件 `<asm/spinlock.h>` 中声明的。通常的，一个自旋锁象下面这行一样被声明和初始化为不加锁状态：

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

或者，如果需要在运行时初始化一个自旋锁，可以使用 `spin_lock_init`：

```
spin_lock_init(&my_lock);
```

有很多处理自旋锁的函数（实际上是宏）：

```
spin_lock(spinlock_t *lock);
```

获得给定的锁，如果需要的话自旋，直到锁成为可用状态为止。在 `spin_lock` 返回之后，调用函数将拥有该锁。

```
spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
```

这个版本也是获得锁，另外，它还在本地处理器上禁止中断并在 `flags` 中保存当前的中断状态。注意所有的自旋锁原语都被定义为宏，因此，`flags` 参数是直接传递的，而不是作为指针。

```
spin_lock_irq(spinlock_t *lock);
```

该函数除了不保存当前的中断状态之外，其它动作类似 `spin_lock_irqsave`。这个版本比 `spin_lock_irqsave` 更为有效，但是它应该最好用在如下情形下，即读者知道将来不希望中断已经被禁止。

```
spin_lock_bh(spinlock_t *lock);
```

获得给定的锁并且阻止底半部的执行。

```
spin_unlock(spinlock_t *lock);
spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
spin_unlock_irq(spinlock_t *lock);
spin_unlock_bh(spinlock_t *lock);
```

这些函数与前面描述的各种处理锁的原语是配对的。`spin_unlock` 解开给定的锁而不做其它的工作，

`spin_unlock_irqrestore` 是否启动中断取决于 `flags` 的值（由 `spin_lock_irqsave` 保存的），`spin_unlock_irq` 无条件地启动中断，`spin_unlock_bh` 重新启动底半部处理。在每种情况下，函数在调用解锁函数之前应该是占有锁的，否则就会导致严重的混乱。

```
spin_is_locked(spinlock_t *lock);
spin_trylock(spinlock_t *lock);
spin_unlock_wait(spinlock_t *lock);
```

`spin_is_locked` 查询自旋锁的状态而不改变它，如果当前锁忙它将返回非零值。如果不想等待而试图获得一个锁，可以使用 `spin_trylock`，如果操作失败它将返回非零（锁处于忙状态）。`spin_unlock_wait` 将等待直到锁变为空闲，但是不占有它。

很多自旋锁的用户都喜欢使用 `spin_lock` 和 `spin_unlock`。如果在中断处理程序中使用了自旋锁，无论如何，用户都应该在非中断代码中使用 IRQ 被禁止的版本（通常是 `spin_lock_irqsave` 和 `spin_unlock_irqsave`），否则将陷入死锁的情形下。

现在我们考虑一个示例驱动程序。假定驱动程序正在它的 `read` 方法中运行，并且它用 `spin_lock` 获得了锁，当 `read` 方法持有锁时，设备产生了中断，中断处理程序在同一个处理器上执行。如果它试图使用同一个锁，它就会进入忙等待循环，因为读者的 `read` 方法已经占有了这个锁。但是因为中断例程抢先 `read` 方法而执行，锁将永远不会被释放并且处理器死锁，这可能是读者不希望看到的。

当锁被持有时，这个问题可以通过使用 `spin_lock_irqsave` 禁止本地处理器上的中断来避免。在拿不准的时候，使用 `_irqsave` 版本的函数，就不需要担心死锁了。记住，从 `spin_lock_irqsave` 返回的 `flags` 值绝对不能传给其它的函数。

常规的自旋锁对于大多数设备驱动程序作者遇到的情形都能很好地工作。在某些情况下，一种对临界区数据的特殊访问模式需要特殊对待。如果你有多个线索（进程、中断处理程序、底半部例程）需要以只读的方式访问一个临界区数据，这时，你可能会担心使用自旋锁的开支。众多的读者之间不会彼此干预，而只有写者会造成问题。此时，允许所有的读者同时地访问数据会是更有效的方法。

Linux 有一种不同类型的自旋锁，称为“读者/写者自旋锁”，这些锁是一种可以被初始化为 `RW_LOCK_UNLOCKED` 的 `rwlock_t` 类型。任意数量的线程可以在同一时间持有该锁进行读操作，当一个写进程出现，它会等待直到它可以获得互斥的访问。

处理读者/写者锁的函数如下：

```
read_lock(rwlock_t *lock);
read_lock_irqsave(rwlock_t *lock, unsigned long flags);
read_lock_irq(rwlock_t *lock);
read_lock_bh(rwlock_t *lock);
```

函数的行为与常规的自旋锁是一样的。

```
read_unlock(rwlock_t *lock);
read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
read_unlock_irq(rwlock_t *lock);
read_unlock_bh(rwlock_t *lock);
```

这些是释放读取锁的不同方法。

```
write_lock(rwlock_t *lock);
write_lock_irqsave(rwlock_t *lock, unsigned long flags);
write_lock_irq(rwlock_t *lock);
write_lock_bh(rwlock_t *lock);
```

作为写者来获得锁。

```
write_unlock(rwlock_t *lock);
write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
write_unlock_irq(rwlock_t *lock);
write_unlock_bh(rwlock_t *lock);
```

释放写者获得的锁。

如果中断处理程序仅仅使用读锁，那么不需要禁止中断，就可以让所有的代码用 `read_lock` 来获得读锁。任何写锁必须用 `write_lock_irqsave` 来获得，但是，应该避免死锁。

值得注意的是，在为单处理器系统创建的内核中，自旋锁函数扩展为空，因而它们在这样的系统上没有什么开支（除了可能禁止中断以外），这里它们不是必须的????。

9.8.3 使用锁变量

内核提供了一组函数可以用来对变量提供原子性（不可中断的）的访问。在需要执行的操作很简单的时候，使用这些函数可以减少使用更复杂的锁方案的需求。通过手工的测试和循环，原子性的操作可用来提供一种“穷人的自旋锁”。直接使用自旋锁通常比较好，因为它们已经为此目的进行了优化。

Linux 内核导出了两组函数来处理锁：位操作和对“原子的”数据类型的访问。

位操作

使用单个位的锁变量或者在中断时间内更新设备状态标志（而某个进程可能正在访问它们），都是十分普遍的。内核提供一组原子地修改或检查单个位的函数。因为整个操作发生在一个单一指令中，没有中断（或者其它处理器）能打断该操作。

原子的位操作是非常快的，只要底层平台允许这样做，它们甚至可以在不禁止中断的同时，使用单条机器指令来完成操作。这些函数在头文件 `<asm/bitops.h>` 中定义，并且依赖于体系结构。甚至在对称多处理器的计算机上，也能保证这些操作是原子的，并且它们有利于保持处理器间的一致性。

不幸的是，这些函数的数据类型也是体系结构相关的。`nr` 参数大部分都定义为 `int`，但在少数体系结构上定义为 `unsigned long`。这是位操作的列表，它们出现在 2.1.37 和之后的内核中：

```
void set_bit(nr, void *addr);
```

这个函数在 `addr` 指向的数据项中设置 `nr` 位。函数作用在 `unsigned long` 变量上，尽管 `addr` 是 `void` 类型的指针。

```
void clear_bit(nr, void *addr);
```

函数清除 `addr` 指向的 `unsigned long` 数据中的指定位。它的语义与 `set_bit` 相反。

```
void change_bit(nr, void *addr);
```

这个函数切换该位。

```
test_bit(nr, void *addr);
```

这个函数是唯一不需要原子性的位操作函数，它简单地返回该位当前的值。

```
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

就像前面列出的那些函数一样，这些函数原子地运行，只是它们返回该位的前一个值。

当这些函数被用来访问和修改一个共享的标志时，除了调用这些函数之外，不需要做其它工作。而使用位操作来管理锁变量（它控制对某个共享变量的访问）时，则更复杂些，需要举个例子加以说明。大多数现代代码不会以下面这种方式使用位操作，但是内核中却仍然存在类似的代码。

需要访问一个共享数据项的代码段通过使用 `test_and_set_bit` 或者 `test_and_clear_bit` 来尝试原子地获得锁。通常的实现如下，它假定锁位于地址 `addr` 的 `nr` 位上，还假定当锁空闲的时候该位为 0，锁忙的时候为非零。

```
/* try to set lock */
while (test_and_set_bit(nr, addr) != 0)
    wait_for_a_while();

/* do your work */

/* release lock, and check... */
if (test_and_clear_bit(nr, addr) == 0)
    something_went_wrong(); /* already released: error */
```

如果阅读内核源代码，就会发现类似例子程序这样工作的代码。正如前面提及的，不管怎样，最好在新的代码中使用自旋锁，除非在等待锁被释放时需要执行有用的工作（例如，在 `wait_for_a_while()` 中列出的指令）。

原子性的整数操作

内核程序员经常需要在中断处理程序和其它函数之间共享整型变量。内核已经提供了一组独立的函数以实现这种共享，它们在头文件 `<asm/atomic.h>` 中定义。

`atomic.h` 提供的设施比刚刚讨论的位操作更健壮。`atomic.h` 定义了一个新的数据类型，`atomic_t`，它只能通过原子操作来访问。在所有支持的体系结构上一个 `atomic_t` 都保存一个 `int` 值。因为在某些处理器上，这种类型的工作方法可能不会用到全部的整数类型范围，这样，你不应该期望一个 `atomic_t` 可以保存多于 24 个位。下面的操作是为这个数据类型定义的，能保证 SMP 系统上的所有处理器都是原子地对它进行访问。这些操作都很快，因为它们会尽可能地编译成单条机器指令。

```
void atomic_set(atomic_t *v, int i);
```

设置原子变量 `v` 为整型值 `i`。

```
int atomic_read(atomic_t *v);
```

返回 `v` 的当前值。

```
void atomic_add(int i, atomic_t *v);
```

将 `v` 指向的原子变量加上 `i`，返回值是 `void` 类型，因为大多数时间不需要知道新的值。这个函数被网络代码用来更新套接字对象在内存使用上的统计信息。

```
void atomic_sub(int i, atomic_t *v);
```

从 `*v` 减去 `i`。

```
void atomic_inc(atomic_t *v);
```

```
void atomic_dec(atomic_t *v);
```

对一个原子变量进行增加或者减少操作。

```
int atomic_inc_and_test(atomic_t *v);
```

```
int atomic_dec_and_test(atomic_t *v);
```

```
int atomic_add_and_test(int i, atomic_t *v);
```

```
int atomic_sub_and_test(int i, atomic_t *v);
```

这些函数的行为就像前面列出的那些函数的一样，只是它们还返回原子数据类型的先前值。

如前所述，`atomic_t` 数据项只能通过这些函数来访问。如果你将原子数据项传递给一个要求参数类型为整型的函数，就会得到一个编译错误。

9.8.4 无竞争地进入睡眠

到目前为止，我们一直忽略了一个竞态，即进入睡眠的问题。通常来说，在驱动程序决定睡眠，并且在 `sleep_on` 调用实际执行之间，会出现这个问题。有时候，进入睡眠的条件也许会在真正睡眠之前来到，这将导致一个超过预期的长时间睡眠。这是一个比中断驱动的 I/O 问题更为普遍的问题，而有效的解决方案需要对 `sleep_on` 的内部实现有些了解。

作为一个例子，考虑下面从 `short` 驱动程序中摘出的代码：

```
while (short_head == short_tail) {
    interruptible_sleep_on(&short_queue);
    /* ... */
}
```

在上面的代码中，`short_head` 的值可能在 `while` 语句中的检查和调用 `interruptible_sleep_on` 之间改变。在这种情况下，即使新的数据可用，驱动程序也会进入睡眠；在最好的情况下，这个条件会延迟，而在最坏的情况下，设备会被锁住。

解决这个问题的方法是在执行检查之前进入一半睡眠。其想法是，进程可以将自己放入等待队列，声明自己将要睡眠，然后执行它的检查。下面是典型的实现：

```
wait_queue_t wait;
init_waitqueue_entry(&wait, current);
add_wait_queue(&short_queue, &wait);
while (1) {
    set_current_state(TASK_INTERRUPTIBLE);
    if (short_head != short_tail) /* whatever test your driver needs */
        break;
    schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(&short_queue, &wait);
```

这段代码有点象是将 `sleep_on` 的内部实现展开了，这里我们将逐步说明上述代码。

代码首先声明了一个 `wait_queue_t` 变量，初始化它，然后将它加入到驱动程序的等待队列（读者可能还有印象，它是一种 `wait_queue_head_t`）。一旦这些步骤被执行，一个在 `short_queue` 上对 `wake_up` 的调用将会唤醒这个进程。

然而进程还没有睡眠。在调用 `set_current_state` 之后，将接近睡眠的状态，该函数设置进程状态为 `TASK_INTERRUPTIBLE`。接下来，系统将认为进程已经睡眠，调度器不会尝试运行它。在“进入睡眠”的过程中，这是重要的一步，但是事情还没有结束。

现在发生的是代码检查它所等待的条件，也就是在缓冲区中是否有数据。如果没有数据存在，就调用 `schedule`，以使其它的进程运行并让当前进程真正地进入睡眠。一旦进程被唤醒，它就会再次检查条件，并且可能从循环中退出。

在循环之外，只有一点清理工作需要做。进程当前的状态被设置为 `TASK_RUNNING` 以反映该进程不再是睡眠的。这是必须的，因为如果我们没有睡眠就退出循环，我们可能还是处于 `TASK_INTERRUPTIBLE` 状态。然后，`remove_wait_queue` 被用来从等待队列中删除进程。

但是，为什么这段代码不会产生竞态？当新的数据来临时，中断处理程序会在 `short_queue` 上调用 `wake_up`，这将把所有睡眠在该队列上的进程状态设置为 `TASK_RUNNING`。如果 `wake_up` 调用在检查缓冲区之后发生，那么任务的状态会被改变并且 `schedule` 会使当前的进程继续运行——如果没有立即运行的话，会在一个很短的延迟之后。

这种“半睡眠状态时的检查”模式在内核源代码中是很普通的，2.1 内核开发期间加入了一对宏，以便实现这种策略更加容易：

```
wait_event(wq, condition);
wait_event_interruptible(wq, condition);
```

这两个宏实现刚才所讨论的代码，在“进入睡眠”过程的中间检查条件（因为这是一个宏，将在每次反复循环中被求值）。

9.9 向后兼容性

就像我们在本章的开始部分说明的一样，Linux 目前的中断处理与老内核的中断处理之间的兼容性问题相对较少。但有几个需要我们在这一章讨论。大多数的改变发生在内核的 2.0 版本与 2.2 版本之间，自从那时起，中断处理就已经非常稳定了。

9.9.1 与 2.2 内核的区别

自从 2.2 系列以来最大的变化是 `tasklets` 被增加到内核 2.3.43 中。在这个改变之前，BH 底半部机制是唯一的中断处理程序延期任务的方法。

在 Linux 2.2 中，`set_current_state` 函数已经不存在了（但是 `sysdep.h` 实现了它）。要操作当前进程的状态，它需要直接操作任务结构。例如：

```
current->state = TASK_INTERRUPTIBLE;
```

9.9.2 与 2.0 内核的更多区别

在 Linux2.0 中，快速和慢速处理程序之间有很多不同。慢速处理程序在它们开始执行之前也很慢，因为在内核中有额外的设置工作需要一些开支。快速处理程序不仅通过保持中断禁止来节省时间，而且在从中断返回之前也不检查底半部。这样，在 2.0 内核中，标记在中断处理程序中的底半部实际得到执行前的延迟就会更长一些。最后，当 2.0 内核的 IRQ 信号线被共享时，所有注册的处理程序必须都是快速中断或者慢速中断，两种模式不能混在一起。

大多数 SMP 问题在 2.0 内核中并不存在，当然，中断处理程序只能每次在一个 CPU 上执行，局部或者全局禁止中断是没有区别的。

`disable_irq_nosync` 函数在 2.0 的内核中并不存在，另外，不能嵌套调用 `disable_irq` 和 `enable_irq` 函数。

原子操作在 2.0 内核中是不同的。函数 `test_and_set_bit`、`test_and_clear_bit` 和 `test_and_change_bit` 并不存在，相反，`set_bit`、`clear_bit` 和 `change_bit` 返回一个值，就像现代的 `test_and_` 版本一样。对于整数操作，`atomic_t` 只是 `typedef` 为 `int`，并且 `atomic_t` 类型的变量能象 `int` 类型一样处理。`atomic_set` 和 `atomic_read` 函数也不存在。

`wait_event` 和 `wait_event_interruptible` 宏在 Linux 2.0 中也不存在。

9.10 快速参考

本章介绍了与中断管理相关的符号。

```
#include <linux/sched.h>
int request_irq(unsigned int irq, void (*handler)(),
unsigned long flags, const char *dev_name, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

上面这些调用用来注册和注销中断处理程序。

```
SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM"
```

`request_irq` 的标志。`SA_INTERRUPT` 要求安装一个快速的处理程序(相对于慢速的)，`SA_SHIRQ` 安装一个共享的处理程序，而第三个标志表明中断时间戳可用来产生系统熵。

```
/proc/interrupts
/proc/stat
```

这些文件系统节点用于汇报硬件中断和已安装处理程序的信息。

```
unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);
```

当驱动程序不得不探测设备，以确定该设备使用哪根中断信号线时，可以使用这些函数。在中断产生之后，`probe_irq_on` 的返回值必须传给 `probe_irq_off`，`probe_irq_off` 的返回值就是检测到的中

断号。

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

驱动程序可以启动和禁止中断报告。如果硬件试图在中断被禁止的时候产生中断，中断将永远丢失。使用共享处理程序的驱动程序不能使用这些函数。

```
DECLARE_TASKLET(name, function, arg);
tasklet_schedule(struct tasklet_struct *);
```

处理 **tasklet** 的工具。**DECLARE_TASKLET** 用给定的 **name** 声明一个 **tasklet**，运行时，将传递 **arg** 参数调用给定的 **function**，而使用 **tasklet_schedule** 调度一个要执行的 **tasklet**。

```
#include <linux/interrupt.h>
void mark_bh(int nr);
```

这个函数标记一个要执行的底半部。

```
#include <linux/spinlock.h>
spinlock_t my_lock = SPINLOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
spin_lock(spinlock_t *lock);
spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
spin_lock_irq(spinlock_t *lock);
spin_lock_bh(spinlock_t *lock);
spin_unlock(spinlock_t *lock);
spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
spin_unlock_irq(spinlock_t *lock);
spin_unlock_bh(spinlock_t *lock);
spin_is_locked(spinlock_t *lock);
spin_trylock(spinlock_t *lock);
spin_unlock_wait(spinlock_t *lock);
```

各种使用自旋锁的函数。

```
rwlock_t my_lock = RW_LOCK_UNLOCKED;
read_lock(rwlock_t *lock);
read_lock_irqsave(rwlock_t *lock, unsigned long flags);
read_lock_irq(rwlock_t *lock);
read_lock_bh(rwlock_t *lock);
read_unlock(rwlock_t *lock);
read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
read_unlock_irq(rwlock_t *lock);
read_unlock_bh(rwlock_t *lock);
write_lock(rwlock_t *lock);
write_lock_irqsave(rwlock_t *lock, unsigned long flags);
write_lock_irq(rwlock_t *lock);
write_lock_bh(rwlock_t *lock);
write_unlock(rwlock_t *lock);
write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
write_unlock_irq(rwlock_t *lock);
write_unlock_bh(rwlock_t *lock);
```

上述函数用于锁住和解开读者/写者自旋锁。

```
#include <asm/bitops.h>
void set_bit(nr, void *addr);
void clear_bit(nr, void *addr);
void change_bit(nr, void *addr);
test_bit(nr, void *addr);
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

这些函数原子地访问位的值，它们可以被用在标志或者锁变量上。使用这些函数可以防止任何与对位的并发访问有关的竞态。

```
#include <asm/atomic.h>
void atomic_add(atomic_t i, atomic_t *v);
void atomic_sub(atomic_t i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
```

这些函数原子地访问整型变量。如果不想在编译时出现警告，则只能通过这些函数来访问 `atomic_t` 变量。

```
#include <linux/sched.h>
TASK_RUNNING
TASK_INTERRUPTIBLE
TASK_UNINTERRUPTIBLE
```

这些是当前任务最常用的状态，这些状态可由 `schedule` 使用。

```
set_current_state(int state);
```

以给定的值设置当前任务的状态。

```
void add_wait_queue(struct wait_queue ** p, struct wait_queue * wait)
void remove_wait_queue(struct wait_queue ** p, struct wait_queue * wait)
void __add_wait_queue(struct wait_queue ** p, struct wait_queue * wait)
void __remove_wait_queue(struct wait_queue ** p, struct wait_queue * wait)
```

使用等待队列的最底层的函数。其中以下划线开头的函数是底层的，使用它们时，处理器上的中断报告必须已被禁止。

```
wait_event(wait_queue_head_t queue, condition);
wait_event_interruptible(wait_queue_head_t queue, condition);
```

这些宏在给定的队列上等待，直到给定的条件为真。

第 10 章 合理使用数据类型



在继续讨论更高级的主题之前，我们需要首先讨论一下可移植性问题。现代版本的 Linux 内核，能够非常容易地移植到若干具有很大差异的体系结构上运行。因为 Linux 的多平台特点，所以任何一个重要的驱动程序都应该是可移植的。

但是与内核代码相关的核心问题是，这些代码应该能够同时访问已知长度（例如，文件系统的数据结构或者设备板上的寄存器）的数据项，并充分利用不同处理器（32 位和 64 位体系结构，或者也可能是 16 位的）的能力。

在把 x86 上的代码移植到新的体系结构上时，内核开发人员遇到的若干问题都和 incorrect 的数据类型有关。坚持使用严格的数据类型，并且使用 `-Wall -Wstrict-prototypes` 选项编译可以防止大多数的 bug。

内核使用的数据类型主要被分成三大类：类似 `int` 这样的标准 C 语言类型，类似 `u32` 这样的有确定大小的类型，以及象 `pid_t` 这样的用于特定内核对象的类型。我们将讨论应该在什么情况下使用这三种典型类型，以及如何使用。当从 x86 平台向其它平台移植驱动程序代码时，读者可能遇到其它一些典型的问题，这些问题将在本章的最后一节讨论，还将介绍对新内核头文件所提供的对链表的通用支持。

如果读者遵循我们提供的指导方针，读者的驱动程序甚至可能在那些未经测试的平台上编译和运行。

10.1 使用标准 C 语言类型

尽管大多数程序员习惯于自由使用象 `int` 和 `long` 这样的标准类型，而编写设备驱动程序需要小心地避免类型冲突和潜在的 bug。

问题是，当我们需要“两个字节的填充单位”或者“用四个字节字符串表示的某个东西”时，我们不能使用标准类型，因为在不同的体系结构上，一般的 C 语言的数据类型所占空间大小并不相同。在 O'Reilly ftp 站点上的 `misc-procs` 目录下，提供的样例文件已经包含了 `datasize` 程序，它可以显示各种 C 语言数据类型的大小，这是 PC 上程序的运行样例（其中最后四个类型将在下节介绍）：

```
morgana% misc-progs/datasize
arch  Size:  char  shor  int   long  ptr  long-long  u8  u16  u32  u64
i686      1    2    4    4    4    8    1    2    4    8
```

这个程序也可以在 64 位平台上运行，其结果表明 long 整型和指针的大小和 32 位系统不同。下面的结果说明了该程序在不同平台上的运行结果：

```
arch  Size:  char  shor  int   long  ptr  long-long  u8  u16  u32  u64
i386      1    2    4    4    4    8    1    2    4    8
alpha      1    2    4    8    8    8    1    2    4    8
armv4l     1    2    4    4    4    8    1    2    4    8
ia64       1    2    4    8    8    8    1    2    4    8
m68k       1    2    4    4    4    8    1    2    4    8
mips       1    2    4    4    4    8    1    2    4    8
ppc        1    2    4    4    4    8    1    2    4    8
sparc      1    2    4    4    4    8    1    2    4    8
sparc64    1    2    4    4    4    8    1    2    4    8
```

值得注意的是，Linux-sparc64 的用户空间可以运行 32 位代码，所以在用户空间指针是 32 位宽的，而它们在内核空间是 64 位的，这可以通过装载 `kdatasize` 模块（可从 `misc-proc` 目录下的样例文件中得到）来验证。模块在装载时使用 `printk` 汇报大小信息并返回一个错误（所以不需要卸载这个模块）：

```
kernel: arch  Size:  char  short int   long  ptr  long-long  u8  u16  u32  u64
kernel: sparc64      1    2    4    8    8    8    1    2    4    8
```

尽管在混合使用不同数据类型时，我们必须小心谨慎，但有时还有一些其它的理由需要我们这样做。这样的一种情况是内存地址，一涉及到内核，内存地址就变得很特殊。虽然概念上地址是指针，但是通过使用无符号整数类型可以更好的实现内存管理。内核把物理内存看作一个巨型数组，一个内存地址就是该数组的索引。此外，我们可以很方便地使用指针指向的内容（译者注：使用 C 语言的“*”运算符。在 C 语言术语中，称为“反引用，**dereference**”，和“&”运算符相反，后者称“引用”），但在直接处理内存地址时，我们几乎从来不会以这种方式使用以整数表示的内存地址。使用一个整数类型可以防止类指针的使用方式，因而可避免出现 **bug**。因此，内核中的地址是 **unsigned long** 型数据，至少在当前 Linux 支持的所有平台上，指针和 long 整型的大小总是相同的。

C99 标准定义了 `intptr_t` 和 `uintptr_t` 类型，它们是能够保存指针值的整型变量。这些类型在 2.4 的内核中几乎没有用到，但是在将来的开发工作中，也许会经常用到。

10.2 为数据项分配确定的空间大小

有时内核代码需要指定大小的数据项，或者用来匹配预定义的二进制结构^{*}，或者通过在结构中插入“**filler**”成员（关于对齐的问题，请查阅本章后面的“数据对齐”一节）来对齐数据。

在读者需要知道自己的数据大小时，内核提供下列数据类型。所有类型在头文件 `<asm/types.h>` 中声明，这个文件又被头文件 `<linux/types.h>` 所包含：

```
u8; /* unsigned byte (8 bits) */
```

^{*} 这种情况一般发生在读分区表、执行二进制文件或者对网络数据包解码的时候。

```
u16; /* unsigned word (16 bits) */
u32; /* unsigned 32-bit value */
u64; /* unsigned 64-bit value */
```

这些数据类型只有内核代码（也就是说，必须在包含头文件<linux/types.h>之前定义 `__KERNEL__`）可以使用。相应的有符号类型也存在，但是很少使用，如果需要它们的话，只需要将名字中的 `u` 用 `s` 替换就可以了。

如果一个用户空间程序需要使用这些类型，它可以在名字前加上两个下划线作为前缀：`__u8` 和其它类型是独立于 `__KERNEL__` 定义的。例如，如果一个驱动程序需要通过 `ioctl` 系统调用与一个运行在用户空间的程序交换二进制结构的话，头文件应该用 `__u32` 来声明结构中的 32 位的成员。

重要的是要记住这些类型是 Linux 特有的，如果使用它们将阻碍软件向其它 Unix 变体的移植。使用新编译器的系统将支持 C99 标准类型，例如 `uint8_t` 和 `uint32_t`，可能的情况下，应使用这些类型以支持 Linux 相关变种。但是，如果代码用于 2.0 内核，就无法使用这些类型（因为 2.0 内核只能利用老的编译器来编译）。

读者可能也注意到有时内核使用传统的类型，例如 `unsigned int`，这通常用于大小独立于体系结构的数据项。这种做法通常是为了保持向后兼容性。当 `u32` 及其相关类型在版本 1.1.67 中引入时，开发者没有办法将现存的数据结构改变为新的类型，因为当结构成员和赋予的值之间类型不匹配时，编译器将发出警告*。

Linus 没有想到他自己编写的操作系统会用在多平台上，结果，旧的结构有时定义的不是很严格。

10.3 接口特有的类型

内核中最常用的数据类型由它们自己的 `typedef` 声明，这样防止出现任何移植性问题。例如，一个进程的标识符（`pid`）通常使用 `pid_t` 类型，而不是 `int`，使用 `pid_t` 屏蔽了在实际的数据类型中任何可能的差异。我们使用“接口特有”这一表达方式，是指由某个库定义的一种数据类型，以便为某个特定的数据结构提供接口。

即使没有定义接口特有的类型，也应该使用适当的数据类型，以便和内核其余部分保持一致。比如，一个 `jiffy` 计数总是属于 `unsigned long` 类型，而不管它的实际大小如何，因此，在使用 `jiffies` 的时候应该一直使用 `unsigned long` 类型。本节中我们主要讨论“`_t`”类型的用法。

完整的 `_t` 类型清单在头文件 `<linux/types.h>` 中定义，但是这个清单很少有用。在需要某个特定类型时，可在需要调用的函数原型，或者所使用的数据结构中找到这个类型。

只要驱动程序使用了需要这种“定制”类型的函数，又不遵守约定的时候，编译器会产生警告。如果使用 `-Wall` 编译选项并且细心地去除了所有警告，就可以自信代码是可移植的了。

`_t` 数据项的主要问题是当我们需要打印它们的时候，不太容易选择正确的 `printf` 或者 `printk` 的输

* 事实上，即使在两个类型只是同一对象的不同名字时，编译器还是会发出类型不一致的警告，就象 PC 上的 `unsigned long` 和 `u32`。

出格式，并且在一种体系结构上排除了警告，而在另一种体系结构上可能还会出现警告。例如，当 `size_t` 在一些平台上是 `unsigned long`，而在另一种平台上是 `int` 类型时，我们应该如何打印它呢？

在我们需要打印一些接口特定的数据类型时，最行之有效的办法，就是将其强制转换成最可能的类型（通常是 `long` 或者 `unsigned long`），然后用相应的格式打印出来。这种作法不会产生错误或者警告，因为格式和类型相匹配，而且也不会丢失数据位，因为强制类型转换要么是一个空操作，要么是将该数据项向更宽的数据类型扩展。

实际上，通常我们并不需要打印我们讨论的这些数据项，因此，只有在调试信息中才会出现这些问题。更经常的，除了将接口特有的数据类型作为参数传递给库函数或者内核函数之外，代码只须对它们进行储存和比较操作。

尽管 `_t` 类型在大多数情况下是正确的解决方案，但有时正确的类型并不存在。这发生在一些还没有被整理的旧接口上。

在内核头文件中我们已经发现一处疑点，`I/O` 函数的数据类型不是很严格（请看第 8 章的“平台相关性”一节），这种不严格的类型定义主要是出于历史原因，但是却可能在编写代码时产生问题。例如，经常在把参数交换给象 `outb` 这样的函数时遇到麻烦；如果有一种 `port_t` 类型，编译器就会发现这种错误类型。

10.4 其它有关移植性的问题

除了数据类型定义的问题之外，在编写一个能在不同的 Linux 平台间移植的驱动程序时，还必须注意其它一些软件上的问题。

一个通用的原则是对显式常量值持怀疑态度。通常，代码通过使用预处理的宏使之参数化。这一节列出了最重要的移植性问题，在遇到其它已经被参数化的值时，可以在头文件和正式内核发布的设备驱动程序中找到一些线索。

10.4.1 时间间隔

在处理时间周期时，不要假定每秒一定有 100 个 `jiffy`。尽管对于当前的 Linux-x86 这是正确的，但并不是每一种 Linux 平台都是以 100HZ（就象 2.4，你会发现这个值的范围是从 20 到 1200，尽管 20 只是用在 IA-64 模拟器里面）运行。即使在 x86 上这种假设也可能是错误的，因为 HZ 值可能已被改变，何况没有人知道未来的内核将发生什么改变。使用 `jiffy` 计算时间间隔的时候，应该用 HZ（每秒定时器中断的次数）来衡量。例如，为了检测半秒的超时，可以将消逝的时间与 $HZ/2$ 作比较。更常见的，与 `msec` 毫秒对应的 `jiffy` 数目总是 $msec * HZ / 1000$ 。很多网络驱动程序在移植到 Alpha 平台时上都必须修正该细节。它们中的一部分在 Alpha 平台上没有正常工作，就是因为它们假定了 HZ 是 100。

10.4.2 页大小

使用内存时，要记住内存页的大小为 `PAGE_SIZE` 字节，而不是 4KB。假定页大小就是 4KB，

并且硬编码该数值是 PC 程序员常犯的错误——相反，在已支持的平台上，页大小范围从 4KB 到 64KB，有时候它们在相同平台上的不同实现也是不一致的。这一问题涉及到的宏是 `PAGE_SIZE` 和 `PAGE_SHIFT`。后者是得到一个地址所在页的页号时，需要对该地址右移的位数。对于当前 4KB 和更大的页，这个数值通常是 12 或者更大。这些宏在头文件 `<asm/page.h>` 中定义。如果用户空间程序需要这些信息，则可以使用 `getpagesize` 来获得。让我们看看特殊情形，如果一个驱动程序需要 16KB 空间来储存临时数据，我们不应该指定传递给 `get_free_pages` 的参数为 2 的幂，而需要一个可移植的方案。使用大量的 `#ifdef` 条件编译可以很好地工作，但是它只能解决我们所知道的平台，而在其它的体系结构上可能出错，例如在某个未来支持的体系结构上。所以，我们建议使用下面的代码替代：

```
int order = (14 - PAGE_SHIFT > 0) ? 14 - PAGE_SHIFT : 0;
buf = get_free_pages(GFP_KERNEL, order);
```

解决方法利用了 16KB 等于 $1 \ll 14$ 这一常识。两个数的商就是它们对数的差，而 14 和 `PAGE_SIZE` 都是 2 的幂。幂是在编译时计算的，这种实现是分配任意 2 的幂次大小的内存空间的一种安全方法，而且不依赖于 `PAGE_SIZE`。

10.4.3 字节序

要小心的是不要做字节序的假设。尽管 PC 是按照低字节优先（低端优先，就是 `little-endian`）的方式存储多字节数值的，大多数高端平台是以另一种方式（`big-endian`）工作的。现代的处理器的操作两种模式，但是它们中的大部分更喜欢工作在 `big-endian` 模式下。某些现代处理器加入了对 `little-endian` 模式的内存访问的支持，以便和 PC 数据交互，而 Linux 通常更喜欢以处理器固有的字节序模式运行。只要可能，代码应该编写成不关心所操作数据的字节序的方式。可是，有时驱动程序需要从单字节建立整型数，或者相反。

例如，在填充网络数据包的头时，需要处理字节序的问题，或者在处理一个以特定字节序模式操作的外围设备时，也需要处理字节序额外难题。在这种情况下，代码应该包含头文件 `<asm/byteorder.h>` 并且应该检查头文件定义了 `__BIG_ENDIAN` 还是 `__LITTLE_ENDIAN`。

我们可以编写一组 `#ifdef __LITTLE_ENDIAN` 条件，但是有一个更好的方法。Linux 内核定义了一组宏，它可以在处理器字节序和特殊字节序数据之间进行转换。例如：

```
u32 __cpu_to_le32 (u32);
u32 __le32_to_cpu (u32);
```

这两个宏可以将一个 CPU 使用的值转换成一个无符号值的 32 位 `little-endian` 数，或者相反。它们可以正常工作而不管 CPU 是 `big-endian` 或 `little-endian`，也不管它是否是一个 32 位处理器。如果没有转换工作需要做，它们就返回未经修改的参数。使用这些宏可以使编写可移植代码的工作变得更加容易，从而无需使用很多条件编译。

类似例程有十几个之多，读者可以在头文件 `<linux/byteorder/big_endian.h>` 和 `<linux/byteorder/little_endian.h>` 中看到完整的列表。稍后能看到，这种模式很容易遵循。`__be64_to_cpu` 将一个无符号的 64 位 `big-endian` 的数值转换成 CPU 的内部表达。相应的，`__le16_to_cpus` 处理一个有符号的 16 位 `little-endian` 数值。在处理指针时，也可以使用类似 `__cpu_to_le32p` 这样的函数，它们使用指向数值的指针而不是数值本身。其它函数可参阅头文件。

并不是所有的 Linux 版本都定义了所有处理字节序的宏。特别要指出的是，linux/byteorder 目录出现在版本 2.1.72，用来重新整理各个 <asm/byteorder.h> 文件，并删除重复的定义。如果读者使用我们的 sysdep.h，在为 2.0 或者 2.2 内核编译代码时，则可以使用 Linux 2.4 所定义的所有宏。

10.4.4 数据对齐

最后值得关注的问题是如何访问未对齐的数据——例如，怎样读取一个存储在非四字节倍数的地址中的四字节值。PC 的用户常常访问未对齐的数据，但是只有很少的体系结构允许这样做，大部分现代的体系结构在每次程序试图除数未对齐的数据时，都会产生一个异常；这时，数据传输会被异常处理程序处理，因此会带来大量性能损失。如果需要访问未对齐的数据，则应该使用下面的宏：

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

这些宏是与类型无关的，对各种数据项，不管它是 1 字节、2 字节、4 字节还是 8 字节，这些宏都有效。所有版本的内核都定义了这些宏。

另一个关于数据对齐的问题是数据结构的跨平台可移植性。同样的数据结构（在 C 语言源文件中定义的）在不同的平台上可能会被编译成不同的样子，编译器排列数据结构的成员时，将根据平台的不同而进行不同的对齐。至少理论上，为了优化内存的使用，编译器甚至会重新排列数据结构的成员*。

为了编写含有可以在平台之间移动的数据项的数据结构，除了标准化特定的字节序，还应该始终坚持数据项的自然对齐。“自然对齐”意味着在数据项大小的整数倍（例如，8 字节数据项存入 8 的整数倍的地址）的地址处存储数据项。强制自然对齐可以防止编译器移动数据结构的成员，读者应该使用填充符（filler）成员以避免在数据结构中留下空洞。

为说明编译器是怎样强制对齐的，源代码的 misc-progs 目录中有个 dataalign 程序，对应模块是 kdataalign（在 misc-modules 目录中）。下面 dataalign 程序在若干平台上的输出，以及 kdataalign 模块在 SPARC64 体系结构上的输出：

```
arch Align: char short int long ptr long-long u8 u16 u32 u64
i386      1    2    4    4    4    4    1    2    4    4
i686      1    2    4    4    4    4    1    2    4    4
alpha     1    2    4    8    8    8    1    2    4    8
armv4l    1    2    4    4    4    4    1    2    4    4
ia64      1    2    4    8    8    8    1    2    4    8
mips      1    2    4    4    4    8    1    2    4    8
ppc       1    2    4    4    4    8    1    2    4    8
sparc     1    2    4    4    4    8    1    2    4    8
sparc64   1    2    4    4    4    8    1    2    4    8

kernel: arch Align: char short int long ptr long-long u8 u16 u32 u64
kernel: sparc64      1    2    4    8    8    8    1    2    4    8
```

* 在当前已支持的体系结构上，不会发生成员的重新排列，因为这会破坏与已有代码的协同工作能力，但是由于对齐的限制，新的体系结构可能为带有空洞的结构定义成员的重新排列规则。

值得注意的是，不是所有平台都在 64 位边界对齐 64 位数值，所以需要填充符成员来强制对齐并确保可移植性。

10.5 链表

就象其它很多程序一样，操作系统内核经常需要维护数据结构的列表。有时，Linux 内核中同时存在多个链表的实现代码。为了减少重复代码的数量，内核开发者已经建立了一套标准的循环链表、双向链表的实现。这套实现在版本 2.1.45 中引入，如果需要操作链表，则鼓励使用这一内核设施。

为了使用这个列表机制，驱动程序必须包含头文件 `<linux/list.h>`。该文件定义了一个简单的 `list_head` 类型的结构。

```
struct list_head {
    struct list_head *next, *prev;
};
```

用在实际代码中的链表几乎总是由某种结构类型构成，每个结构描述链表中的一个入口。为了在代码中使用 Linux 链表设施，只需要在构成链表的结构里面嵌入一个 `list_head`。如果驱动程序维护一个链表，则可声明如下：

```
struct todo_struct {
    struct list_head list;
    int priority; /* driver specific */
    /* ... add other driver-specific fields */
};
```

链表头必须是一个独立的 `list_head` 结构。在使用之前，必须用 `INIT_LIST_HEAD` 宏来初始化链表头。一个实际的链表头可如下声明并初始化：

```
struct list_head todo_list;
INIT_LIST_HEAD(&todo_list);
```

另外，可在编译阶段象下面这样初始化链表：

```
LIST_HEAD(todo_list);
```

头文件 `<linux/list.h>` 中声明了下面这些操作链表的函数：

```
list_add(struct list_head *new, struct list_head *head);
```

这个函数会立即在链表头后面添加新入口——通常是在链表的头部。这样，它可以用来建立栈。但需要注意的是，`head` 并不一定非得是链表的第一项，如果传递了一个恰巧位于链表中间某处的 `list_head` 结构，新入口会立即排在它的后面。因为 Linux 链表是循环的，链表头通常与其它入口没有本质上的区别。

```
list_add_tail(struct list_head *new, struct list_head *head);
```

在给定链表头的前面增加一个新的入口，即在链表的末尾添加。因此，可使用 `list_add_tail` 建立先入先出队列。

```
list_del(struct list_head *entry);
```

将给定的入口从链表中删除。

```
list_empty(struct list_head *head);
```

如果给定的链表是空的，就返回一个非零值。

```
list_splice(struct list_head *list, struct list_head *head);
```

这个函数通过在 **head** 的后面插入 **list** 来合并两个链表。

list_head 结构有利于实现具有相似结构的链表，但调用程序通常对建立链表的大结构更感兴趣。因此，可利用“**list_entry**”宏将一个 **list_head** 结构指针映射回一个指向大结构的指针。可如下调用这个宏：

```
list_entry(struct list_head *ptr, type_of_struct, field_name);
```

其中，**ptr** 是指向 **struct list_head** 结构的指针，**type_of_struct** 是包含 **ptr** 的结构类型，**field_name** 是结构中链表成员的名字。在我们之前的 **todo_struct** 结构中，链表成员只是简单地称为 **list**。这样，利用类似下面的代码行，我们可以将一个链表入口转换成包含它的结构：

```
struct todo_struct *todo_ptr =  
    list_entry(listptr, struct todo_struct, list);
```

宏“**list_entry**”需要稍微习惯一下，但还不是很难使用。

遍历链表很容易：只须跟随 **prev** 和 **next** 指针。作为例子，假设我们想让 **todo_struct** 链表中的项按照优先级（即 **priority** 成员）降序排列，则增加新入口的函数如下所示：

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next) {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_struct)
}
```

头文件 **<linux/list.h>** 也定义了宏“**list_for_each**”，在代码中它扩展为 **for** 循环使用。正如读者所怀疑的，我们在通过它修改链表时必须十分小心。

图 10-1 显示了怎样使用简单的 **struct list_head** 来维护数据结构链表。

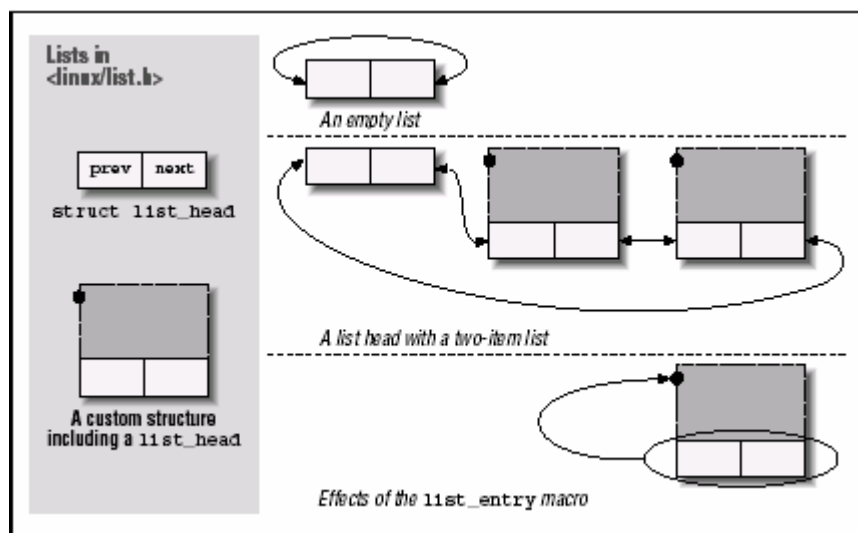


图 10-1: list_head 数据结构

老版本内核中缺少一些出现在 2.4 内核头文件“list.h”中的功能，但我们的头文件“sysdep.h”声明了所有可用于老版本内核中的宏和函数。

10.6 快速索引

本章引入了下列符号：

```
#include <linux/types.h>
typedef u8;
typedef u16;
typedef u32;
typedef u64;
```

这些类型保证是 8 位、16 位、32 位和 64 位的无符号整数值，对应的有符号类型同样存在。在用户空间，读者可以通过 __u8、__u16 等来使用这些类型。

```
#include <asm/page.h>
PAGE_SIZE
PAGE_SHIFT
```

这些符号定义了当前体系结构下每页包含的字节数和页偏移量的位数（12 对应 4KB 的页而 13 对应 8KB 的页）。

```
#include <asm/byteorder.h>
__LITTLE_ENDIAN
__BIG_ENDIAN
```

两个符号中只有一个被定义，这依赖于体系结构。

```
#include <asm/byteorder.h>
u32 __cpu_to_le32 (u32);
u32 __le32_to_cpu (u32);
```

在已知字节序和处理器字节序之间完成转换的函数。有多于 60 个这样的函数，完整的列表和它们的定义方式，可参阅目录 include/byteorder/ 下面的不同文件。

```
#include <asm/unaligned.h>
get_unaligned(ptr);
```

```
put_unaligned(val, ptr);
```

某些体系结构须使用这些宏来保护对未对齐数据的访问。在那些允许访问未对齐数据的体系结构上，这些宏扩展为取指针内容的通常操作。

```
#include <linux/list.h>
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head *head);
list_del(struct list_head *entry);
list_empty(struct list_head *head);
list_entry(entry, type, member);
list_splice(struct list_head *list, struct list_head *head);
```

操作循环链表和双向链表的函数。

第 11 章 kmod 和高级模块化



在本书的第二部分，我们要讨论更为高级的内容。我们将再次从模块化讲起。

第 2 章对模块化的介绍只是其中的一部分，内核和 `modutils` 包支持一些更高级的特性，它们比前面所讨论的安装和运行一个基本的驱动程序所需的特性要更为复杂。本章将讨论 `kmod` 进程以及模块中的版本支持（一种设施，如果利用该设施，则在升级内核时不必重新编译各个模块）。我们还将讨论如何从内核代码中运行用户空间的辅助程序。

随着时间的推移，按需加载模块的实现部分发生了显著的变化。和前面章节一样，本章也将讨论 2.4 内核中的实现方法。示例程序也尽可能地能够在 2.0 和 2.2 内核上运行，在本章的末尾，我们会介绍 2.4 与 2.0、2.2 之间的不同之处。

11.1 按需加载模块

为了方便用户加载和卸载模块，并且避免把不再使用的模块继续保留在内核中浪费内核的存储空间，同时又使得内核可以广泛地支持各种各样的硬件，Linux 提供了对模块自动加载和卸载的支持。要利用这一特性，在编译内核前进行的配置中，必须打开对 `kmod` 的支持选项。大多数 Linux 发行版安装的内核都开启了对 `kmod` 特性的支持。这种可以在需要时请求加载额外模块的能力，对于使用堆叠式模块的驱动程序尤其有用。

隐藏在 `kmod` 背后的思想很简单，但却很有效。一旦内核试图访问某种资源并发现该资源不可用时，它会对 `kmod` 子系统进行一次特殊的调用而不仅仅是返回一个错误。`kmod` 会加载相关的模块以获取该资源，如果它成功则内核继续工作，否则将返回错误。实际上请求任何一种资源都可以使用这种办法：诸如字符设备和块设备、文件系统、线路规程（line discipline）和网络协议等等。

一个得益于按需加载的例子是 ALSA（Advanced Linux Sound Architecture）声卡驱动程序组，也许未来某一天它会取代目前内核中使用的 OSS（Open Sound System）实现^{*}。ALSA 被分割成许多片段。其中共用的部分会首先被加载，其余片段是否加载则取决于所配硬件型号（这里指声卡）以及是否需要相应的功能（例如，MIDI 音序器、合成器、混音器以及 OSS 兼容功能等等）。于是，一个大而复杂的系统可以被分解成许多小部件，只有那些必不可少的部分才被真正载入到运行

^{*} ALSA 驱动程序可以从 www.alsa-project.org 获得。

系统中。

自动加载模块的另外一个常见应用是发行版中所安装的“万能内核”。Linux 发行商们总是希望他们的内核能够支持尽可能多的硬件设备，然而，那种简单地将任何可能用到的驱动程序全部配置进内核的做法是不现实的。那样做的结果将导致内核因尺寸过大而无法加载，而且有如此之多的驱动程序去探测硬件将非常容易导致冲突和混乱。通过自动装载机制，在所安装的每一个独立的系统中，内核都将会根据它所找到的硬件配置加载相应的模块以适应之。

11.1.1 在内核中请求模块

任何内核空间的代码在需要时都可以通过调用 `kmod` 程序来请求加载模块。`kmod` 最初被实现为一个处理模块装载请求的独立内核进程，但是，很久以前该进程就被简化成不需要单独的进程上下文。要利用 `kmod`，必须在驱动程序中包含 `<linux/kmod.h>` 头文件。

要请求加载模块，调用 `request_module`：

```
int request_module(const char *module_name);
```

`module_name` 既可以是特定的模块文件名，也可以是更为通用的模块功能信息。该函数的返回值为 0，如果发生错误将返回常规的负的错误码。

注意 `request_module` 的调用是同步的——它将进入睡眠直到模块的加载动作完成。当然，这也意味着 `request_module` 不能够在中断上下文中调用。同样要注意 `request_module` 的成功返回，并不能保证模块提供的功能会立即可用。函数的返回值只是表明该函数成功调用了 `modprobe` 但是并不表示 `modprobe` 本身的状态是成功的。许多问题和配置上的错误都会导致 `request_module` 返回一个成功状态，但实际却没有真正加载你所需的模块。

因此，函数 `request_module` 的正确用法，通常是进行两次测试以确保所需的相应功能特性的确已经存在：

```
if ( (ptr = look_for_feature()) == NULL) {
    /* if feature is missing, create request string */
    sprintf(modname, "fmt-for-feature-%i\n", featureid);
    request_module(modname); /* and try to load it */
}
/* Check for existence of the feature again; error if missing */
if ( (ptr = look_for_feature()) == NULL)
    return -ENODEV;
```

第一次检查避免了对 `request_module` 的重复调用。如果内核中没有我们所需的功能特性，就生成一个请求字符串并通过 `request_module` 去加载它；最后一次检查用来确定所请求的功能特性是否已经可用。

11.1.2 用户空间方面

模块加载任务的真正完成需要用户空间程序的帮助，原因很简单：在用户空间上下文中，达到所需要的可配置性和灵活性要容易的多。当内核代码调用 `request_module` 时，一个新的“内核线程”进程会被创建，它会在用户上下文中运行一个辅助程序，这个程序就是我们在本书前面部分已经简

要介绍过的 `modprobe`。

`modprobe` 可以作非常多的事情。最简单的情况下，它会直接使用 `request_module` 传过来的模块名字作为参数去调用 `insmod`。然而，内核代码经常会使用一个更为抽象的、用来代表所需功能特性的名字，例如：`scsi_hostadapter`，这时，`modprobe` 会找到并且加载正确的模块。`modprobe` 也可以处理模块之间的依赖关系；如果所需加载的模块需要其他模块，`modprobe` 会将它们一并加载——前提是模块被安装之后已经运行了 `depmod -a` 命令。^{*}`modprobe` 通过文件 `/etc/modules.conf` 进行配置。^{*}读者可查阅 `modules.conf` 手册页获得该文件所支持入口项的完整清单。下面简要描述一些最常用的入口项：

```
path[misc]=directory
```

`path[misc]` 指令告诉 `modprobe` 各种杂项模块可在给定目录的 `misc` 子目录中找到。其它值得设置的路径包括 `boot`，它指示了在系统启动时应该加载的模块所在的目录；`toplevel`，指出模块子目录树的顶层目录。通常，我们还需要包含一个单独的 `keep` 指令。

```
keep
```

通常，路径指令将导致 `modprobe` 放弃其它所有的路径（包括默认路径），通过将 `keep` 放在所有其他的路径指令之前，可以让 `modprobe` 将路径添加到路径列表，而不是替换掉已有路径。

```
alias alias_name real_name
```

`alias` 会使 `modprobe` 在要求加载 `alias_name` 模块时加载 `real_name` 模块。通常，别名用来标识特定的功能特性：它可能是 `scsi_hostadapter`, `eth0` 或 `sound` 等等。通过这种方式，可让一般性请求（比如“用于第一个以太网卡的驱动程序”）映射到特定的模块。系统安装程序经常会创建 `alias` 行；一旦安装程序在特定系统中找到了某一硬件，它就会为其创建适当的别名入口项以保证能够加载正确的驱动程序。

```
options [-k] module opts
```

`options` 提供了加载给定模块时的选项（`opts`）。当设置 `-k` 标志时，该模块不会在执行 `modprobe -r` 时自动卸载。

```
pre-install module command
post-install module command
pre-remove module command
post-remove module command"
```

前两个指令指定给定的模块被加载之前/之后要执行的命令，而后两个指令则指定模块被卸载之前/之后要执行的动作。使用这些指令可以使我们在加载或卸载模块时，方便地调用额外的用户进程或启动所需的守护进程。其中，`command` 应该给出完整的路径名，以避免因此而产生的问题。

注意，对于模块卸载时要运行的命令，只有在使用 `modprobe` 卸载的模块，才会执行相应的命令；它们不会由于模块被 `rmmod` 命令卸载，或者系统被关闭（不管是正常还是异常关机）而得到执行。

`modprobe` 所支持的指令远不只上边列举的几个，但其它指令通常只用在非常复杂的场合。

^{*} 大多数发行版会在启动时自动运行 `depmod -a`，所以不必对此担心，除非你在重新启动之后安装了新的模块。请参照 `modprobe` 的文档了解详细信息。

^{*} 在以前的版本中，相应的文件是 `/etc/conf.modules`，出于兼容性考虑，目前仍然支持这种文件名，但并不提倡使用它。

一个典型的 `/etc/modules.conf` 文件通常是这样的：

```
alias scsi_hostadapter aic7xxx
alias eth0 eeepro100
pre-install pcmcia_core /etc/rc.d/init.d/pcmcia start
options short irq=1
alias sound es1370
```

该文件告诉 `modprobe`，要想要使 SCSI 系统、以太网卡和声卡正常工作需要加载哪些驱动程序。它同时确保在加载 PCMCIA 驱动程序之前，首先先调用一个启动脚本以启动 PC 卡服务守护进程。最后，为驱动程序 `short` 提供了一个命令选项。

11.1.3 模块加载和安全性

由于被加载的模块代码会在最高的权限级别运行，很显然模块的加载会涉及到一些安全性的问题。正因为这样，在面对一个可加载模块的系统时应该格外小心。

当编辑文件 `modules.conf` 时，我们应该时刻记住，任何可以加载模块的人对整个系统有着完全的控制权。因此，任何被添加到模块加载路径列表的目录，以及 `modules.conf` 文件本身都应该仔细加以保护。

值得注意的是，`insmod` 通常会拒绝加载非 `root` 帐号所拥有的模块；这样做是在尽量防范取得模块加载路径写权限的攻击者。可以通过给 `insmod` 传一个选项（或者在 `modules.conf` 文件中添加一行）来强制取消这种检查，不过这样做会降低系统的安全性。

另一点需要注意的是，作为参数传递给 `request_module` 的模块名最终会成为 `modprobe` 的命令行参数。如果模块名是某个用户空间程序提供的，则必须在传递给 `request_module` 之前进行仔细的验证。例如，考虑对网络接口进行配置的系统调用。在响应 `ifconfig` 的调用时，这个系统调用会告诉 `request_module` 为（用户指定的）接口加载驱动程序。一个怀有敌意的用户可以精心挑选一个虚构的接口名使得 `modprobe` 做出一些不适当的操作，这实在是一个安全性隐患，而且直到在 `2.4.0-test` 开发周期的后期才被发现。最严重的问题已经被清除，但是系统还是容易受到通过某些恶意模块名进行的攻击。

11.1.4 模块加载实例

现在，让我们实际地使用按需加载模块的功能。在这里，我们将会使用两个模块：`master` 和 `slave`。读者可在 O'Reilly FTP 站点的 `misc-modules` 目录下找到它们的源代码。

为了无需将模块安装到默认搜索路径之下也可以运行这段测试代码，可以在 `/etc/modules.conf` 文件中添加如下几行：

```
keep
path[misc]=~/rubini/driverBook/src/misc-modules
```

`slave` 模块并不实现任何功能，而 `master` 模块的代码如下所示：

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



```
#include "sysdep.h"

int master_init_module(void)
{
    int r[2]; /* results */

    r[0]=request_module("slave");
    r[1]=request_module("nonexistent");
    printk(KERN_INFO "master: loading results are %i, %i\n", r[0],r[1]);
    return 0; /* success */
}

void master_cleanup_module(void)
{ }
```

在加载时，**master** 试着加载两个模块：**slave** 模块和一个并不存在的模块。**printk** 会将调试信息加到系统日志中，而且，如果使用默认的日志等级，调试信息会出现在控制台终端上。下面是当系统被配置成支持 **kmod** 并且该守护进程已经激活的时候在控制台下执行一下命令时的结果：

```
morgana.root# depmod -a
morgana.root# insmod ./master.o
master: loading results are 0, 0
morgana.root# cat /proc/modules
slave          248    0  (autoclean)
master         740    0  (unused)
es1370        34832   1
```

request_module 的返回值和 **/proc/modules** 文件（在第 2 章的“初始化和终止”一节中描述过）均显示 **slave** 模块已经被正确加载。然而，请注意加载不存在模块时的返回值也是成功的，这是因为 **request_module** 只要成功调用了 **modprobe**，它就会返回成功标志，而不去理会 **modprobe** 执行的情况如何。

我们看看在卸载 **master** 时会发生什么：

```
morgana.root# rmmod master
morgana.root# cat /proc/modules
slave          248    0  (autoclean)
es1370        34832   1
```

结果显示，**slave** 留在内核中。它将一直留在内核中，直到下一次模块清除过程结束（通常在现代操作系统中不会发生）。

11.1.5 运行用户态辅助程序

正如我们所看到的，**request_module** 程序运行了一个用户态程序（作为单独的进程，以非特权模式在用户空间内运行）来帮助它完成任务。在 2.3 系列的开发系列中，内核开发人员加入了“运行用户辅助程序”的机制。如果你的驱动程序需要一个用户态程序的支持其操作，则可以利用这个机制。由于它是 **kmod** 实现的一部分，我们将在这里讨论它。如果读者对这一机制感兴趣，推荐你看一看 **kernel/kmod.c**；它的代码不多而且对如何使用用户辅助程序做了很好的阐述。

运行辅助程序的接口函数非常简单。在内核 2.4.0-test9 中，有这样一个函数：**call_usermodehelper**，它主要用于热插拔子系统（比如 **USB** 设备等）中，以便在新设备连接到系统时，能够执行模块加载和配置任务。它的函数原型如下：

```
int call_usermodehelper(char *path, char **argv, char **envp);
```

它的参数形式并不陌生，分别是：所要执行的程序名，要传递给它的参数（依照惯例，`argv[0]` 是程序本身的名字），以及指向环境字符串指针数组的指针。这两个指针数组都要以 `NULL` 结尾，就象 `execve` 系统调用的那样。`call_usermodehelper` 将会睡眠直到辅助程序启动，然后返回操作的状态。

以这种方式运行的辅助程序实际上是作为一个叫做 `keventd` 的内核线程的子进程来运行的。这种设计意味着一个很重要的事实：你将无法知道什么时候辅助程序将会结束，以及他的返回状态如何。运行辅助程序的行为包含着对该程序的一种信任。

值得指出的是，真正使用用户辅助程序的场合是很少见的。在大多数情况下，较之于在内核代码中调用用户辅助程序，建立一个脚本以便在模块加载时进行所有必要工作的做法要好的多。

11.2 模块间通讯

在内核 `pre-2.4.0` 开发系列的很晚阶段，内核开发者提供了一个新的可以提供模块间简单通讯的接口。这一机制允许模块注册若干指向所关注数据的字符串，其它模块可检索这些字符串取得相关数据。我们接下来使用稍为变形的 `master` 和 `slave` 模块来简单讨论这个接口。

我们使用相同的 `master` 模块，但引入了一个新的称为 `inter` 的 `slave` 模块。`inter` 提供了与 `ime_string` 字符串和 `ime_function` 函数（其中的 `ime` 意指“intermodule example”）。它的代码如下面所示：

```
static char *string = "inter says 'Hello World'";

void ime_function(const char *who)
{
    printk(KERN_INFO "inter: ime_function called by %s\n", who);
}

int ime_init(void)
{
    inter_module_register("ime_string", THIS_MODULE, string);
    inter_module_register("ime_function", THIS_MODULE, ime_function);
    return 0;
}

void ime_cleanup(void)
{
    inter_module_unregister("ime_string");
    inter_module_unregister("ime_function");
}
```

这段代码使用了函数 `inter_module_register`，它的原型如下：

```
void inter_module_register(const char *string, struct module *module,
                          const void *data);
```

`string` 是其他模块用来找到数据的字符串；`module` 是指向 `data` 所有者的指针，它的值通常取

`THIS_MODULE`; `data` 可以指向任何要共享的数据;。注意, `data` 被声明成 `const` 指针, 这意味着它以只读方式导出。如果给定的 `string` 已经被注册过了, `inter_module_register` 会 (通过 `printk`) 表明错误。

在数据不再需要共享时, 模块应该调用 `inter_module_unregister` 清除共享数据:

```
void inter_module_unregister(const char *string);
```

下面两个函数用来访问通过 `inter_module_register` 共享的数据:

```
const void *inter_module_get(const char *string);
```

该函数查找给定的 `string` 并返回与之关联的 `data` 指针, 如果 `string` 没有注册, 将会返回 `NULL`。

```
const void *inter_module_get_request(const char *string, const char *module);
```

该函数与 `inter_module_get` 相似, 但增加了如下特性: 如果没有找到给定的 `string`, 它将使用给定的模块名去调用 `request_module`, 之后会再尝试用 `string` 查找一次。

这两个函数都会增加注册数据的模块的使用计数。因此通过 `inter_module_get` 或 `inter_module_get_request` 得到的指针将会一直保持有效, 直至被显式释放。在此期间, 建立该指针的模块至少不会被卸载; 但存在这样的可能性, 即这个模块本身可以进行一些操作从而使该指针无效。

在完成与该指针相关的操作后, 必须释放它以使得产生该指针的模块的使用计数被适当减少。调用函数

```
void inter_module_put(const char *string);
```

将释放该指针, 在此之后不应再次使用该指针。

在我们例子中, 模块 `master` 调用 `inter_module_get_request`, 使得 `inter` 模块被加载从而取得字符串指针和函数指针。字符串仅仅用来打印, 而函数指针用来实现从 `master` 模块对 `inter` 模块内函数的调用。`master` 模块其余的代码如下所示:

```
static const char *ime_string = NULL;
static void master_test_inter();

void master_test_inter()
{
    void (*ime_func)();
    ime_string = inter_module_get_request("ime_string", "inter");
    if (ime_string)
        printk(KERN_INFO "master: got ime_string '%s'\n", ime_string);
    else
        printk(KERN_INFO "master: inter_module_get failed");
    ime_func = inter_module_get("ime_function");
    if (ime_func) {
        (*ime_func)("master");
        inter_module_put("ime_function");
    }
}

void master_cleanup_module(void)
{

```

```

if (ime_string)
    inter_module_put("ime_string");
}

```

注意,其中一次对 `inter_module_put` 的调用在模块 `master` 清除时才进行,这会导致模块 `inter` 的使用计数在模块 `master` 被卸载之前始终保持(至少)为 1。

在使用模块间通讯函数时,还有一些值得紧记的细节。首先,即使在配置成不支持可加载模块的内核中,它们仍然是可用的,因此没有必要增加针对它们的 `#ifdef` 分支。其次,模块间通讯函数的名字空间是全局的,在选择名字时应该格外小心,否则将会导致冲突。最后,模块间的共享数据被简单地存储在链表中,大量的查找或过多的字符串存储将会导致性能上的损失。这一设施被设计为面向少量使用的,而绝非一个象字典一样的子系统。

11.3 模块中的版本控制

模块机制的主要问题之一是版本依赖性,在第 2 章我们曾经介绍过这方面的内容。在我们运行若干定制模块时,如果针对每一个要使用的内核版本,都要重新编译每个模块,将是件非常痛苦的事情。如果运行的是以二进制形式发布的商业模块时,甚至连编译也是不可能的。

幸运的是,内核开发者们找到了一个灵活的办法来处理版本问题。其思想是,只有内核提供的软件接口发生改变时,才会出现与新内核版本不兼容的问题。软件接口可以由函数原型以及函数调用所涉及的所有数据结构的确切定义来表示。最后,可以使用一个 **CRC** 算法^{*}把所有关于软件接口的信息映射到一个单一的 32 位数值上去。

这样,版本依赖性问题可通过在每个由内核导出的符号名中,包含与该符号相关的所有信息的校验和来得到处理,这些相关信息通过解析头文件来获得。这一设施是可选的,并可在编译阶段打。各种 Linux 发行版自带的内核一般都起用了版本化支持。

例如,在提供版本化支持时,符号 `printk` 是以类似 `printk_R12345678` 的形式向模块导出的,其中 `12345678` 是该函数使用的软件接口的校验和(16 进制表示)。要加载模块到内核时,仅当每个加到模块内符号上的校验和都与加到内核中相同符号上的校验和相匹配时, `insmod` (或 `modprobe`) 才可以完成它的任务。

上述做法有一些局限性。常见的问题在将一个针对 **SMP** 的模块加载到单处理器的系统(或者相反)时出现。因为许多内联函数(例如,自旋锁操作)和符号在 **SMP** 内核具有不同的定义,因此,保持模块和内核在 **SMP** 支持上一致性是很重要的。2.4 版本和近期推出的 2.2 版本的内核在编译支持 **SMP** 的系统时会给每一个符号前都额外地加一个 `smp_` 字符串以处理这一特殊情况。然而,还存在着一些潜在的问题。模块和内核会由于编译时所采用的编译器、它们所采用的内存布局,以及所支持的处理器版本等等的差异而不同。版本支持方案可以解决大多数常见问题,但是仍然要小心。

让我们来看看内核和模块均开启了版本支持的时候,会发生些什么:

^{*} CRC, 即循环冗余校验 (cyclic redundancy check), 一种根据任意数量的数据生成唯一数值的方法。

- 内核本身并不修改符号。连接进程以通常的方式工作，并且 `vmlinux` 文件的符号表看起来也和以前一样。
- 公共符号表使用版本化的名字创建，如 `/proc/ksyms` 文件所显示的那样。
- 模块必须使用合并后的名字编译，这些名字在目标文件中是以未定义符号的形式出现的。
- 装载程序 (`insmod`) 用模块中未定义的符号匹配内核中的公共符号，因此要使用版本信息。

注意，内核和模块必须就是否支持版本化达成一致，否则 `insmod` 将拒绝加载模块。

11.3.1 在模块中使用版本支持

如果希望模块支持版本化，驱动程序编写者就必须在代码中显式地加入支持。可以在两处之一加入版本控制：在 `makefile` 中或在源代码本身。由于 `modutils` 包的文档描述了如何在 `makefile` 添加版本支持，因此，我们在这里说明如何在 C 源代码中加入版本支持。用于演示 `kmod` 工作机制的 `master` 模块可支持版本化的符号。如果用于编译模块的内核使用了版本化支持的话，这种功能就会自动启动。

用于合并符号名字的主要设施定义在文件 `<linux/modversions.h>` 中，它包含了所有公共内核符号的预处理定义。该文件作为编译内核过程一部分而（确切的说是“`make depend`”）创建，如果你的内核从来没有编译过，或者没有编译成版本化支持的，那么该文件中就不会有我们所感兴趣的东西了。`<linux/modversions.h>` 一定要在包含其它任何头文件之前包含。然而，通常的做法是通过一个编译命令告诉 `gcc` 来做这件事。

```
gcc -DMODVERSIONS -include /usr/src/linux/include/linux/modversions.h...
```

包含头文件之后，无论何时模块使用内核符号，编译器都将看到合并之后的符号。

如果内核已经启用了版本支持，为了在模块中启用，则必须确保在 `<linux/config.h>` 中已定义过 `CONFIG_MODVERSIONS`。该头文件（在编译时）控制着在当前内核中启用了哪些特性。每个 `CONFIG_` 宏定义说明相应选项已被激活^{*}。

于是，`master.c` 的初始化部分包含如下代码：

```
#include <linux/config.h> /* retrieve the CONFIG_* macros */
#if defined(CONFIG_MODVERSIONS) && !defined(MODVERSIONS)
# define MODVERSIONS /* force it on */
#endif

#ifdef MODVERSIONS
# include <linux/modversions.h>
#endif
```

在针对版本化的内核编译这个文件时，目标文件的符号表会引用版本化的符号，这些符号与内核本身导出的符号相匹配。下面的屏幕快照显示了 `master.o` 中储存的符号名称。在 `nm` 的输出中，“T”代表“文本 (text)”，“D”代表“数据(data)”，“U”代表“未定义 (undefined)”。“未定

^{*} `CONFIG_` 宏定义在文件 `<linux/autoconf.h>` 中定义。然而，读者应该包含 `<linux/config.h>` 而不是 `<linux/autoconf.h>`，因为前者可避免自己被多次包含，而后者仅用于内部使用。而且内容源自 `<linux/autoconf.h>`。

义”表示目标文件引用了但没有被声明的符号。

```
00000034 T cleanup_module
00000000 t gcc2_compiled.
00000000 T init_module
00000034 T master_cleanup_module
00000000 T master_init_module
        U printk_Rsmp_1b7d4074
        U request_module_Rsmp_27e4dc04
morgana% fgrep 'printk' /proc/ksyms
c011b8b0 printk_Rsmp_1b7d4074
```

因为添加到 `master.o` 中的符号名上的校验和来自 `printk` 和 `request_module` 的完整原型，因此，该模块可与大部分的内核版本兼容。然而，如果与其中任一函数有关的数据结构发生了变化，`insmod` 将会因为模块与内核的不兼容而拒绝加载它。

11.3.2 导出版本化符号

前面的讨论中未涉及的情况是，当其它模块要使用另一个模块导出出的符号时，将会出现什么情况。如果依赖版本信息获得模块的可移植性，我们也希望把 **CRC** 校验码加到我们自己的符号上去。这个问题比仅仅连接到内核的技巧性要高一些，因为我们需要将合并后的符号名导出给其它模块，为此，我们需要一种办法来生成校验和。

分析头文件和生成校验和的任务是由随 `modutils` 包一起发行的一个工具 `genksyms` 完成的。该程序在自身的标准输入接收 `C` 预编译器的输出，并在标准输出上打印一个新的头文件。这个输出文件中定义了原始源文件中导出的每个符号的校验和版本。`genksyms` 的输出通常以 `.ver` 为后缀保存，以下我们将遵循同样的惯例。

为了说明如何导出符号，我们编写了两个名为 `export.c` 和 `import.c` 的模块文件。`export` 将导出一个叫做 `export_function` 的简单函数，它将由第二个模块 `import.c` 使用。该函数接收两个整形变量并返回它们的和——我们感兴趣的不是它的功能，而是连接过程。

在 `misc-modules` 目录中的 `Makefile` 有一条从 `export.c` 生成 `export.ver` 文件的规则，这样，`export_function` 的校验和符号可以被 `import` 模块使用：

```
ifndef CONFIG_MODVERSIONS
export.o import.o: export.ver
endif

export.ver: export.c
$(CC) -I$(INCLUDEDIR) $(CFLAGS) -E -D_ _GENKSYMS_ _ $^ | \
$(GENKSYMS) -k 2.4.0 > $@
```

这几行代码演示的如何生成 `export.ver`，并且只有定义了 `MODVERSIONS` 之后，才会把它加入到两个目标文件的依赖关系中去。如果内核启用了版本支持，还要添加几行到 `Makefile` 中处理 `MODVERSIONS`，但并不值得在这里展示它们。必须使用 `-k` 选项以通知 `genksyms` 为哪一内核版本进行工作，这样做的目的是要决定输出文件的格式。`genksyms` 并不需要匹配当前系统中运行着的内核。

另外一些值得说明的是 `GKSMP` 符号的定义。如前面提到的，如果内核被创建为支持 `SMP` 系统，

则会在每个校验和前加一个前缀 (`-p smp_`)。除非 `genksyms` 工具被明确告知，否则并不会自动添加前缀，`Makefile` 中如下的代码可保证适当设置这个前缀。

```
ifndef CONFIG_SMP
GENKSYMS += -p smp_
endif
```

然后，源文件必须为每个可能的预处理器步骤声明正确的预处理符号：不论是给 `genksyms` 的输入还是真正的编译过程，在启用或关闭版本支持的情况下都要声明适当的符号。进而，`export.c` 应该能够像 `master.c` 那样自动检测内核中的版本支持。下面几行说明了如何成功地做到这一点：

```
#include <linux/config.h> /* retrieve the CONFIG_* macros */
#if defined(CONFIG_MODVERSIONS) && !defined(MODVERSIONS)
#   define MODVERSIONS
#endif

/*
 * Include the versioned definitions for both kernel symbols and our
 * symbol, *unless* we are generating checksums (_ _GENKSYMS_ _
 * defined) */
#if defined(MODVERSIONS) && !defined(_ _GENKSYMS_ _)
#   include <linux/modversions.h>
#   include "export.ver" /* redefine "export_function" to include CRC */
#endif
```

这些代码虽然有些杂乱，但好处是可以让 `Makefile` 处于一个干净的状态。另一方面，由 `make` 来传递正确的标志，涉及到为各种情况编写冗长的命令行，因此，在这里我们没有这样做。

`import` 模块很简单，它传递两个数字（均为 2）作为参数调用 `export_function`，其结果当然是 4。下面的例子说明 `import` 确实连接到了 `export` 中的版本化符号，并调用了函数。版本化符号出现在 `/proc/ksyms` 文件中。

```
morgana.root# insmod ./export.o
morgana.root# grep export /proc/ksyms
c883605c export_function_Rsmp_888cb211 [export]
morgana.root# insmod ./import.o
import: my mate tells that 2+2 = 4
morgana.root# cat /proc/modules
import          312    0 (unused)
export          620    0 [import]
```

11.4 向后兼容性

在 2.1 系列的开发中，按需加载功能被完整重新实现。幸运的是，很少有模块需要注意这些改变。然而，出于完整性的考虑，我们在这里对旧的实现方式进行一下描述。

在 2.0 的时候，按需加载是被一个称为 `kerneld` 的独立的、用户空间的守护进程处理的。这个守护进程通过一个特殊的接口连接到内核，并在内核代码生成模块加载（卸载）请求时接收这些请求。这样的处理方式存在很多缺点，其中之一就是这样一个事实：在系统初始化进行到相当程度而启动 `kerneld` 之前，任何模块都不可能加载。

然而，在模块看来，`request_module` 函数保持不变，但是需要包含 `<linux/kerneld.h>` 取代对 `<linux/kmod.h>` 的包含。

2.0 版本内核中面向 SMP 系统的符号没有使用 `smp_` 前缀，这将会导致下面的结果：`insmod` 将一个面向 SMP 的模块加载到单处理器的内核中，反之亦然。通常这种不匹配将导致严重的混乱。

运行用户态辅助程序的功能以及模块间的通讯机制，直到 Linux 2.4 才出现。

11.5 快速索引

本章介绍了以下一些内核符号：

```
/etc/modules.conf
```

`modprobe` 和 `depmod` 的配置文件，它用于配置按需加载模块。在这两个程序的手册页中有描述。

```
#include <linux/kmod.h>
int request_module(const char *name);
```

该函数执行模块的按需加载。

```
void inter_module_register(const char *string, struct module *module, const void *data);
void inter_module_unregister(const char *);
```

`inter_module_register` 通过模块间通讯系统使数据可以为其他模块所用，取消对该数据的共享由 `inter_module_unregister` 函数完成。

```
const void *inter_module_get(const char *string);
const void *inter_module_get_request(const char *string, const char *module);
void inter_module_put(const char *string);
```

前两个函数在模块间通讯系统中查找字符串 `string`；当没有找到 `string` 时，`inter_module_get_request` 还会尝试着用给定的名字加载模块。两个函数都会增加导出 `string` 的模块的使用计数，`inter_module_put` 在不需要数据指针时，减少该使用计数。

```
#include <linux/config.h>
CONFIG_MODVERSIONS
```

只有当前内核被编译成支持版本化符号时，这个宏才会被定义。

```
#ifdef MODVERSIONS
#include <linux/modversions.h>
```

这个头文件只有在 `CONFIG_MODVERSIONS` 有效时才存在，它包含了内核开放的所有符号的版本化名字。

```
_ _GENKSYMS_ _
```

当 `gensyms` 读入预处理文件并生成新的版本代码时，`make` 定义了这个宏。在生成新的校验和时，该宏用于防止包含 `<linux/modversions.h>` 头文件。

```
int call_usermodehelper(char *path, char *argv[], char *envp[]);
```

该函数在 `keventd` 进程上下文中运行一个用户态辅助程序。

第 12 章 装载块设备驱动程序



到目前为止，我们的讨论焦点一直是字符驱动程序。我们曾提到，字符驱动程序并不是 Linux 系统所使用的唯一一种驱动程序，本章我们将会把注意力转向块驱动程序。块驱动程序提供了对面向块的设备的访问，这种设备以随机访问的方式传输数据，并且数据总是具有固定大小的块。典型的块设备是磁盘驱动器，当然也有其它类型的块设备。

字符驱动程序的接口相对清晰而且易于使用，但相反的是，块驱动程序的接口要稍微复杂一些，内核开发人员为此经常心生抱怨。出现这种情况的原因有两个：其一是因为其简单的历史——块驱动程序接口从 Linux 第一个版本开始就一直存在于每个版本的核心，并且已经证明很难修改或改进；其二是因为性能，一个慢设备驱动程序虽然不受欢迎，但仍可以接受，但一个慢的块驱动程序将影响整个系统的性能。因此，块驱动程序的接口设计经常受到速度要求的影响。

在 Linux 内核开发过程中，块驱动程序接口发生过重大的演变。和本书其余部分一样，本章将主要讲述 2.4 内核版本中的接口，而在最后讨论与其它早期版本之间的区别。但需要说明的是，本章的示例驱动程序能够在 2.0 和 2.4 之间的任意一个内核上运行。

本章利用两个新的示例驱动程序讲述块驱动程序的创建。第一个称为 **sbull** (Simple Block Utility for Loading Localities)，该驱动程序实现了一个使用系统内存的块设备，从本质上讲，属于一种 RAM 磁盘驱动程序。随后，我们将介绍该驱动程序的变种，称为 **spull**，该驱动程序说明了如何处理分区表。

上述示例驱动程序避免了许多实际的块驱动程序会遇到的问题，其目的主要是为了演示这类驱动程序必须处理的接口。实际的驱动程序需要处理复杂的硬件，因此，第 8 章和第 9 章中的内容会对读者有所帮助。

这里，我们需要做一点技术上的说明：本书所使用的“块”这一术语，指的是由内核决定的一个数据块。通常来讲，块的大小是 2 的幂，但不同的磁盘可能具有不同的块大小。而“扇区”则是由底层硬件决定的一个固定大小的数据单位，一个扇区通常都是 512 字节长。

12.1 注册驱动程序

和字符驱动程序一样，内核使用主设备号来标识块驱动程序，但块主设备号和字符主设备号是互不

相干的。一个主设备号为 32 的块设备可以和具有相同主设备号的字符设备同时存在，因为它们具有各自独立的主设备号分配空间。

用来注册和注销块设备驱动程序的函数，与用于字符设备的函数看起来很类似，如下所示：

```
#include <linux/fs.h>
int register_blkdev(unsigned int major, const char *name,
    struct block_device_operations *bdops);
int unregister_blkdev(unsigned int major, const char *name);
```

上述函数中的参数意义和字符设备相同，而且可以通过一样的方式动态赋予主设备号。因此，注册 **sbull** 设备时所使用的方法几乎和 **scull** 设备一模一样：

```
result = register_blkdev(sbull_major, "sbull", &sbull_bdops);
if (result < 0) {
    printk(KERN_WARNING "sbull: can't get major %d\n", sbull_major);
    return result;
}
if (sbull_major == 0) sbull_major = result; /* dynamic */
major = sbull_major; /* Use `major' later on to save typing */
```

然而，类似之处到此为止。我们已经看到了一个明显的不同：**register_chrdev** 使用一个指向 **file_operations** 结构的指针，而 **register_blkdev** 则使用 **block_device_operations** 结构的指针——这个变化从 2.3.38 版本就有了。在一些块驱动程序中，该接口有时仍然被称为 **fops**，但我们将称之为 **dbops**，以便更加贴近该结构本身的含义，并遵循推荐的命名方式。该结构的定义如下：

```
struct block_device_operations {
    int (*open) (struct inode *inode, struct file *filp);
    int (*release) (struct inode *inode, struct file *filp);
    int (*ioctl) (struct inode *inode, struct file *filp,
        unsigned command, unsigned long argument);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
};
```

这里列出的 **open**、**release** 和 **ioctl** 方法和字符设备的对应方法相同。其它两个方法是块设备所特有的，将在本章后面讨论。需要注意的是，该结构中没有 **owner**（所有者）成员，就算在 2.4 内核当中，块设备驱动程序仍然要手工维护其使用计数。

sbull 使用的 **bdops** 接口定义如下：

```
struct block_device_operations sbull_bdops = {
    open:          sbull_open,
    release:       sbull_release,
    ioctl:         sbull_ioctl,
    check_media_change: sbull_check_change,
    revalidate:    sbull_revalidate,
};
```

请读者注意，**block_device_operations** 接口中没有 **read** 或者 **write** 操作。所有涉及到块设备的 I/O 通常由系统缓冲（唯一的例外是下一章要讲到的“**raw**（裸）”设备），用户进程不会对这些设备执行直接的 I/O 操作。在用户模式下对块设备的访问，通常隐含在对文件系统的操作当中，而

这些操作能够从 I/O 缓冲当中获得明显的好处。但是，对块设备的“直接”I/O 访问，比如在创建文件系统时的 I/O 操作，也一样要通过 Linux 的缓冲区缓存*。为此，内核为块设备提供了一组单独的读写函数，驱动程序不必理会这些函数。

显然，块驱动程序最终必须提供完成实际块 I/O 操作的机制。在 Linux 当中，用于这些 I/O 操作的方法称为“request（请求）”，它和其它许多 Unix 系统当中的 strategy 函数等价。request 方法同时处理读取和写入操作，因此要复杂一些。我们稍后将详细讲述 request。

但在块设备的注册过程中，我们必须告诉内核实际的 request 方法。然而，该方法并不在 block_device_operations 结构中指定（这出于历史和性能两方面的考虑），相反，该方法和用于该设备的挂起 I/O 操作队列关联在一起。默认情况下，对每个主设备号并没有这样一个对应的队列。块驱动程序必须通过 blk_init_queue 初始化这一队列。队列的初始化和清除接口定义如下：

```
#include <linux/blkdev.h>
blk_init_queue(request_queue_t *queue, request_fn_proc *request);
blk_cleanup_queue(request_queue_t *queue);
```

init 函数建立队列，并将该驱动程序的 request 函数（通过第二个参数传递）关联到队列。在模块的清除阶段，应调用 blk_cleanup_queue 函数。sbull 驱动程序使用下面的代码行初始化它的队列：

```
blk_init_queue(BLK_DEFAULT_QUEUE(major), sbull_request);
```

每个设备有一个默认使用的请求队列，必要时，可使用 BLK_DEFAULT_QUEUE(major) 宏得到该默认队列。这个宏在 blk_dev_struct 结构形成的全局数组（该数组名为 blk_dev）中搜索得到对应的默认队列。blk_dev 数组由内核维护，并可通过主设备号索引。blk_dev_struct 接口定义如下：

```
struct blk_dev_struct {
    request_queue_t request_queue;
    queue_proc      *queue;
    void            *data;
};
```

request_queue 成员包含了初始化之后的 I/O 请求队列，我们将很快看到队列的成员。data 成员可由驱动程序使用，以便保存一些私有数据，但很少有驱动程序使用该成员。

图 12-1 说明了注册和注销一个驱动程序模块时所执行的主要步骤。如果图 2-1 相比较，将清楚地看到两者之间的相同点和不同点。

* 实际上，2.3 开发系列增加了裸的 I/O 能力，以允许用户进程能够在不通过缓冲区缓存的情况下将数据写入块设备。但块驱动程序全然不知裸 I/O 的存在，因此，我们将在下一章讨论这种机制。

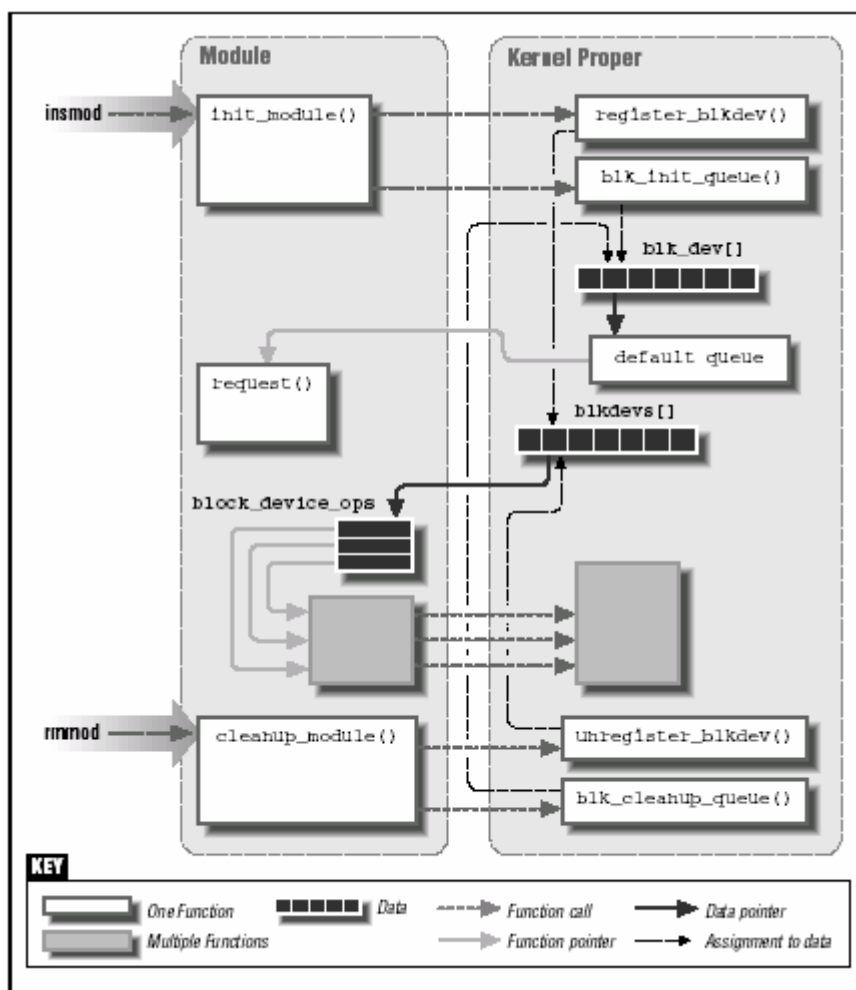


图 12-1: 注册块设备驱动程序

除了 `blk_dev` 之外，还有一些全局数组保存了块设备驱动程序的信息，这些数组通过主设备号索引，有时也通过次设备号索引。这些数组在 `drivers/block/ll_rw_block.c` 中声明和描述。

```
int blk_size[][];
```

该数组通过主设备号和次设备号索引。它描述了每个设备的大小，以千字节为单位。如果 `blk_size[major]` 为 `NULL`，则不会检查该设备的大小（也就是说，内核可以访问超过设备尾部的数据）。

```
int blksize_size[][];
```

该数组包含了每个设备所使用的块大小，以字节为单位。和前述数组一样，该二维数组也通过主设备号和次设备号索引。如果 `blksize_size[major]` 为空指针，则假定块的尺寸为 `BLOCK_SIZE`（当前定义为 1 KB）。设备的块大小必须是 2 的幂，这是因为内核使用位移操作将偏移量转换成块编号。

```
int hardsect_size[][];
```

和上述数组一样，该数组也通过主设备号和次设备号索引。默认的硬件扇区大小为 512 字节，2.2 和 2.4 内核也支持不同的扇区大小，但必须始终是一个大于或者等于 512 字节的 2 的幂。

```
int read_ahead[];
```

```
int max_readahead[][];
```

这两个数组定义了顺序读取一个文件时，内核要预先读入的扇区数目。`read_ahead` 应用于某个给定类型的所有设备，并由主设备号索引；`max_readahead` 应用于单独的设备，并由主设备号和次设备号索引。

在进程真正读取某个数据之前预先读入该数据，有助于提高系统性能和整体的吞吐率。在比较慢的设备上，应该指定一个较大的 `read-ahead` 值，而在较快的设备上应该指定一个较小的值。`read-ahead` 值越大，缓冲区缓存所使用的内存就越多。

这两个数组之间的主要不同在于：`read_ahead` 应用于块 I/O 级，并控制在当前请求之前，应该从磁盘上顺序读入多少数据块；`max_readahead` 工作在文件系统级，指的是文件中的块，而这些块在磁盘上并不一定是顺序存放的。内核开发正在从块 I/O 级的预读转向文件系统级的预读。但在 2.4 内核中，预读仍然在两个级别完成，因此要同时使用这两个数组。

每个主设备号有一个对应的 `read_ahead[]` 值，并应用于所有的次设备号，而 `max_readahead` 对应每个设备只有一个值，这些值均可通过设备驱动程序的 `ioctl` 方法改变。硬盘驱动程序通常将 `read_ahead` 设置为 8 个扇区，对应于 4 KB。相反，`max_readahead` 值则很少由驱动程序设置，它默认设置为 `MAX_READAHEAD`，当前为 31 页。

```
int max_sectors[][];
```

该数组限制单个请求的最大尺寸。它通常应该设置为硬件所能处理的最大传输尺寸。

```
int max_segments[];
```

该数组控制一个集群请求中能够出现的段的数量，但该数组在 2.4 内核发布之前已被删除。（有关集群请求的详细信息，参阅本章“集群请求”一节）。

`sbull` 设备允许我们在装载期间设置这些值，而且这些值将应用于该示例驱动程序的所有次设备号。`sbull` 所使用的变量名及其默认值定义如下：

```
size=2048 (kilobytes)
```

`sbull` 所建立的每个 RAM 磁盘使用两兆字节的 RAM。

```
blksize=1024 (bytes)
```

该模块所使用的软件“块”大小为一千字节，和系统默认值一样。

```
hardsect=512 (bytes)
```

`sbull` 的扇区大小为通常的 512 字节。

```
rahead=2 (sectors)
```

因为 RAM 磁盘是一个快速设备，默认的 `read-ahead` 值很小。

`sbull` 设备也允许我们选择要安装的设备个数。`devs`，即设备个数，默认设置为 2，这样，默认的内存使用为 4 兆字节，因为每个磁盘使用 2 兆字节。

在 `sbull` 中，上述数组的初始化过程如下：

```
read_ahead[major] = sbull_rahead;
```

```

result = -ENOMEM; /* for the possible errors */

sbull_sizes = kmalloc(sbull_devs * sizeof(int), GFP_KERNEL);
if (!sbull_sizes)
    goto fail_malloc;
for (i=0; i < sbull_devs; i++) /* all the same size */
    sbull_sizes[i] = sbull_size;
blk_size[major]=sbull_sizes;

sbull_blksizes = kmalloc(sbull_devs * sizeof(int), GFP_KERNEL);
if (!sbull_blksizes)
    goto fail_malloc;
for (i=0; i < sbull_devs; i++) /* all the same blocksize */
    sbull_blksizes[i] = sbull_blksize;
blksize_size[major]=sbull_blksizes;

sbull_hardsects = kmalloc(sbull_devs * sizeof(int), GFP_KERNEL);
if (!sbull_hardsects)
    goto fail_malloc;
for (i=0; i < sbull_devs; i++) /* all the same hardsect */
    sbull_hardsects[i] = sbull_hardsect;
hardsect_size[major]=sbull_hardsects;

```

出于简化，错误处理代码（即 `goto` 语句的 `fail_malloc` 目标）被忽略了，这段代码其实释放了所有已经成功分配的内存，然后注销设备，并返回一个失败状态。

最后一件事情就是要注册该驱动程序所提供的所有“磁盘”设备。`sbull` 如下调用 `register_disk` 函数：

```

for (i = 0; i < sbull_devs; i++)
    register_disk(NULL, MKDEV(major, i), 1, &sbull_bdops,
        sbull_size << 1);

```

在 2.4.0 内核当中，`register_disk` 函数在上述调用方式下不做任何事情。`register_disk` 的真正目的是用来设置分区表，但 `sbull` 并不支持分区。但是，所有的块设备驱动程序都需要调用这个函数，不管它们是否支持分区，这说明将来有可能所有的块设备都必须有分区。在 2.4.0 当中，没有分区的块驱动程序不需要调用这个函数就能工作，但调用该函数可以更加安全一些。在本章后面讲到分区时，我们将详细讲述 `register_disk` 函数，

`sbull` 所使用的清除函数定义如下：

```

for (i=0; i<sbull_devs; i++)
    fsync_dev(MKDEV(sbull_major, i)); /* flush the devices */
unregister_blkdev(major, "sbull");
/*
 * Fix up the request queue(s)
 */
blk_cleanup_queue(BLK_DEFAULT_QUEUE(major));

/* Clean up the global arrays */
read_ahead[major] = 0;
kfree(blk_size[major]);
blk_size[major] = NULL;
kfree(blksize_size[major]);
blksize_size[major] = NULL;
kfree(hardsect_size[major]);
hardsect_size[major] = NULL;

```

这里，对 `fsync_dev` 函数的调用是必须的，它将释放由内核保存在各种缓存当中的对该设备的引

用。fsync_dev 是 block_fsync 的实现，而 block_fsync 则是用于块设备的 fsync 方法。

12.2 头文件 blk.h

所有的块设备驱动程序都应该包含头文件 `<linux/blk.h>`。该文件定义了许多可由块驱动程序使用的常用代码，并提供了用来处理 I/O 请求队列的函数。

实际上，blk.h 头文件有点与众不同，因为它在符号 MAJOR_NR 的基础上定义了若干符号，而 MAJOR_NR 必须在包含该头文件之前由驱动程序声明。这一约定出现在早期的 Linux 当中，而那时所有的块设备必须具有预先确定的主设备号，而不支持模块化的块驱动程序。

如果阅读 blk.h 头文件，将看到许多设备相关的符号是根据 MAJOR_NR 的值声明的，因此，需要预先知道 MAJOR_NR 的值。但是，如果主设备号被动态赋予，驱动程序就无法在编译时知道被赋予的主设备号，因此就不能正确定义 MAJOR_NR。如果 MAJOR_NR 没有被定义，blk.h 就不能正确建立操作请求队列的某些宏。幸运的是，MAJOR_NR 可以被定义为一个整型变量，这样，动态的块设备驱动程序就能正常工作了。

blk.h 还使用了其它一些预先定义的、驱动程序相关的符号。下面描述了包含在 `<linux/blk.h>` 中必须预先定义的符号，并在最后给出了 sbull 定义这些符号的代码。

MAJOR_NR

该符号用来访问几个数组，尤其是 blk_dev 和 blksize_size。类似 sbull 这样的定制驱动程序，不能赋予该符号一个固定值，而必须将其 #define 为保存主设备号的变量。对 sbull 而言，该变量为 sbull_major。

DEVICE_NAME

将要创建的设备名称。该字符串用于打印错误信息。

DEVICE_NR(kdev_t device)

该符号用于从 kdev_t 设备编号中获得物理设备的顺序号。该符号还被用来声明 CURRENT_DEV，后者可在 request 函数中使用，用来确定哪个硬件设备拥有与某个数据传输请求相关联的次设备号。

这个宏的值可以是 MINOR(device) 或者其它表达式，这随着赋予设备和分区以次设备号的方式的不同而不同。对同一物理设备上的所有分区，这个宏应该返回相同的设备编号，也就是说，DEVICE_NR 代表的是磁盘编号，而不是分区编号。可分区设备将在本章后面介绍。

DEVICE_INTR

该符号用来声明一个指向当前底半处理程序的指针变量。可使用 SET_INTR(intr) 和 CLEAR_INTR 宏来对该变量赋值。当设备需要处理具有不同含义的中断时，使用多个处理程序是很方便的。

DEVICE_ON(kdev_t device)

DEVICE_OFF(kdev_t device)

这两个宏用来帮助设备在执行一组数据传输之前或之后执行其它附加处理。比如，软盘驱动程序可利用这两个宏在执行 I/O 之前启动驱动电机，或者在执行 I/O 之后关闭电机。现代的驱动程序不

再使用这两个宏，而且根本就没有机会去调用 `DEVICE_ON`。但是，可移植的驱动程序应该定义这两个宏（作为空符号），否则，在 2.0 和 2.2 内核上将出现编译错误。

DEVICE_NO_RANDOM

默认情况下，`end_request` 函数对系统熵（收集到的“随机性”总和）起作用，而系统熵将被 `/dev/random` 用来产生随机数。如果设备不能对随机设备贡献足够多的熵，则应该定义 `DEVICE_NO_RANDOM`。`/dev/random` 在第 9 章“安装中断处理程序”中介绍，并解释了 `SA_SAMPLE_RANDOM`。

DEVICE_REQUEST

用来指定驱动程序所使用的 `request` 函数名称。定义 `DEVICE_REQUEST` 之后，将立即声明一个 `request` 函数，除此之外，没有其它效果。这算是一个历史遗留问题，大多数（或者所有）的驱动程序无需考虑这个符号。

`sblock` 驱动程序如下声明这些符号：

```
#define MAJOR_NR sblock_major /* force definitions on in blk.h */
static int sblock_major; /* must be declared before including blk.h */

#define DEVICE_NR(device) MINOR(device) /* has no partition bits */
#define DEVICE_NAME "sblock" /* name for messaging */
#define DEVICE_INTR sblock_intrptr /* pointer to bottom half */
#define DEVICE_NO_RANDOM /* no entropy to contribute */
#define DEVICE_REQUEST sblock_request
#define DEVICE_OFF(d) /* do-nothing */

#include <linux/blk.h>

#include "sblock.h" /* local definitions */
```

`blk.h` 头文件使用上面列出的宏定义了驱动程序所使用的其它一些宏，在下面的章节当中，我们将描述这些宏。

12.3 请求处理简介

块驱动程序中最重要的函数就是 `request` 函数，该函数执行数据读写相关的低层操作。这一小节我们将讨论 `request` 函数的基本设计方法。

12.3.1 请求队列

在内核安排一次数据传输时，它首先在一个表中对该请求排队，并以最大化系统性能为原则进行排序。然后，请求队列被传递到驱动程序的 `request` 函数，该函数的原型如下：

```
void request_fn(request_queue_t *queue);
```

`request` 函数就队列中的每个请求执行如下任务：

1. 测试请求的有效性。该测试通过定义在 `blk.h` 中的 `INIT_REQUEST` 完成，用来检查系统的请求队列处理当中是否出现问题。
2. 执行实际的数据传输。`CURRENT` 变量（实际是一个宏）可用来检索当前请求的细节信息。

CURRENT 是指向 **struct request** 结构的指针，我们将在下一小节当中描述该结构的成员。

- 清除已经处理过的请求。该操作由 **end_request** 函数执行，该函数是一个静态函数，代码位于 **blk.h** 文件中。**end_request** 管理请求队列并唤醒等待 I/O 操作的进程。该函数同时管理 **CURRENT** 变量，确保它指向下一个未处理的请求。驱动程序只给该函数传递一个参数，成功时为 1，失败时为 0。当 **end_request** 在参数为 0 时调用，则会向系统日志（使用 **printk** 函数）递交一条“I/O error”消息。
- 返回开头，开始处理下一条请求。

根据前面的描述，一个并不进行实际数据传输的最小 **request** 函数，应该如下定义：

```
void sbull_request(request_queue_t *q)
{
    while(1) {
        INIT_REQUEST;
        printk("<l>request %p: cmd %i sec %li (nr. %li)\n", CURRENT,
            CURRENT->cmd,
            CURRENT->sector,
            CURRENT->current_nr_sectors);
        end_request(1); /* success */
    }
}
```

尽管上面的代码除了打印信息之外不做任何事情，但我们能够从这个函数当中看到数据传输代码的基本结构。上述代码还演示了 **<linux/blk.h>** 中所定义的宏的两个特点。首先，尽管代码中的 **while** 循环看似永不终止，但实际上，**INIT_REQUEST** 宏将在请求队列为空时返回。这样，循环将跌代请求队列中未处理的请求并最终从 **request** 函数中返回。其次，**CURRENT** 宏始终指向将要处理的请求。下一小节当中，我们将具体讲述 **CURRENT** 宏。

使用前述 **request** 函数的块驱动程序马上就能真正工作了。我们可以在该设备上建立一个文件系统，只要数据被保留在系统缓冲区缓存中，我们就可以访问该设备上的数据。

通过在编译阶段定义 **SBULL_EMPTY_REQUEST** 符号，我们仍可以在 **sbull** 中运行这个空的（但却罗嗦的）**request** 函数。如果读者想理解内核处理不同块大小的方法，可以在 **insmod** 的命令行尝试使用 **blksize=** 这个参数。空的 **request** 函数打印了每个请求的详细信息，借此可以看到内核的内部工作情况。

request 函数有一个非常重要的限制：它必须是原子的。通常，**request** 并不在响应用户请求时直接调用，并且也不会任何特定进程的上下文中运行。它可能在处理中断时被调用，也可能从 **tasklet** 中，或者其它许多地方被调用。这样，在执行其任务时，该函数不能进入睡眠状态。

12.3.2 执行实际的数据传输

为了理解如何为 **sbull** 建立一个能工作的 **request** 函数，首先我们要分析内核是如何在 **struct request** 结构中描述一个请求的。该结构在 **<linux/blkdev.h>** 中定义。通常，驱动程序通过 **CURRENT** 访问请求结构中的成员，通过这些成员，驱动程序可以了解到在缓冲区缓存和物理块设备之间进行数据传输所需的所有信息*。

* 实际上，并不是传递到块驱动程序的所有块都必须通过缓冲区缓存，但我们不会在本章中讨论这一特殊情况。

CURRENT 其实是一个指向 `blk_dev[MAJOR_NR].request_queue` 的指针。下面描述的这些结构成员保存有 **request** 函数经常用到的一些信息：

```
kdev_t rq_dev;
```

请求所访问的设备。默认情况下，某个特定驱动程序所管理的所有设备会使用相同的 **request** 函数。也就是说，单个 **request** 函数将处理所有的次设备号，这时，`rq_dev` 可用来表示实际操作的次设备。**CURRENT_DEV** 宏被简单定义为 `DEVICE_NR(CURRENT->rq_dev)`。

```
int cmd;
```

该成员描述了要执行的操作，它可以是 **READ**（从设备中读取），或者 **WRITE**（向设备写入）。

```
unsigned long sector;
```

表示本次请求要传输的第一个扇区编号。

```
unsigned long current_nr_sectors;
```

```
unsigned long nr_sectors;
```

表示当前请求要传输的扇区数目。驱动程序应该使用 `current_nr_sectors` 而忽略 `nr_sectors`（该变量只是为了完整性才列在这里）。有关 `nr_sectors` 的详细描述，可参阅本章后面的“集群请求”一节。

```
char *buffer;
```

数据要被写入（`cmd==READ`），或者要被读出（`cmd==WRITE`）的缓冲区缓存区域。

```
struct buffer_head *bh;
```

该结构描述了本次请求对应缓冲区链表的第一个缓冲区，即缓冲区头。缓冲区头在进行缓冲区缓存管理时使用，我们稍后将在“请求结构和缓冲区缓存”中详细描述。

该结构中还有其它一些成员，但绝大部分由内核内部使用，驱动程序没有必要使用这些成员。

sbull 设备中能够完成实际工作的 **request** 函数列在下面。在下面的代码中，**Sbull_Dev** 和第 3 章“**scull** 的内存使用”中介绍的 **Scull_Dev** 的功能相同。

```
void sbull_request(request_queue_t *q)
{
    Sbull_Dev *device;
    int status;

    while(1) {
        INIT_REQUEST; /* returns when queue is empty */

        /* Which "device" are we using? */
        device = sbull_locate_device (CURRENT);
        if (device == NULL) {
            end_request(0);
            continue;
        }

        /* Perform the transfer and clean up. */
        spin_lock(&device->lock);
        status = sbull_transfer(device, CURRENT);
        spin_unlock(&device->lock);
        end_request(status);
    }
}
```

```
}

```

上面的代码和前面给出的空 `request` 函数几乎没有什么不同，该函数本身集中于请求队列的管理上，而将实际的工作交给其它函数完成。第一个函数是 `sbull_locate_device`，检索请求当中的设备编号，并找出正确的 `Sbull_Dev` 结构：

```
static Sbull_Dev *sbull_locate_device(const struct request *req)
{
    int devno;
    Sbull_Dev *device;

    /* Check if the minor number is in range */
    devno = DEVICE_NR(req->rq_dev);
    if (devno >= sbull_devs) {
        static int count = 0;
        if (count++ < 5) /* print the message at most five times */
            printk(KERN_WARNING "sbull: request for unknown device\n");
        return NULL;
    }
    device = sbull_devices + devno; /* Pick it out of device array */
    return device;
}
```

该函数唯一“陌生”的功能是限制打印五次错误的条件语句。这是为了避免在系统日志当中生成太多的消息，因为 `end_request(0)` 会在请求失败时打印一条“I/O error”消息。静态的计数器（变量 `count`）是内核中经常用到的用来限制消息打印的一个标准方法。

请求的实际 I/O 由 `sbull_transfer` 函数完成：

```
static int sbull_transfer(Sbull_Dev *device, const struct request *req)
{
    int size;
    u8 *ptr;

    ptr = device->data + req->sector * sbull_hardsect;
    size = req->current_nr_sectors * sbull_hardsect;

    /* Make sure that the transfer fits within the device. */
    if (ptr + size > device->data + sbull_blksize*sbull_size) {
        static int count = 0;
        if (count++ < 5)
            printk(KERN_WARNING "sbull: request past end of device\n");
        return 0;
    }

    /* Looks good, do the transfer. */
    switch(req->cmd) {
        case READ:
            memcpy(req->buffer, ptr, size); /* from sbull to buffer */
            return 1;
        case WRITE:
            memcpy(ptr, req->buffer, size); /* from buffer to sbull */
            return 1;
        default:
            /* can't happen */
            return 0;
    }
}
```

因为 `sbull` 只是一个 RAM 磁盘，因此，该设备的“数据传输”只是一个 `memcpy` 调用而已。

12.4 请求处理详解

先前讲述的 `sbull` 驱动程序能够很好地工作。在类似 `sbull` 这样的简单情形下，可使用 `<linux/blk.h>` 中的宏方便地建立一个 `request` 函数，并获得一个能够工作的驱动程序。但是，我们曾提到，块驱动程序通常是内核中的性能关键部分。类似前面那样简单的驱动程序在许多情况下不能很好地执行，甚至可能导致系统整体性能的降低。在本小节中，我们将通过编写一个更快、更高效的驱动程序而讲解 I/O 请求队列的工作细节。

12.4.1 I/O 请求队列

每个块驱动程序至少拥有一个 I/O 请求队列。在任意给定时刻，该队列包含了内核想在该驱动程序的设备上完成的所有 I/O 操作。该队列的管理是复杂的，因此，系统性能依赖于队列的管理方式。

该队列是根据物理磁盘驱动器设计的。在磁盘中，传输一个数据块所需要的时间总量通常相对较短，但定位磁头到达传输位置（`seek`）的操作所需的时间量却往往很长。这样，Linux 内核要试图最小化设备定位的次数和长度。

为了达到这个目标，需要完成两件事情。其一，需要将请求集群到相邻的磁盘扇区上。大部分现代的文件系统会试图将文件保存在连续的扇区上，这样，位于磁盘相邻部分的请求将会很多。其二，内核在处理请求时使用“电梯”算法。摩天大楼中的电梯不是升就是降，而且在满足所有“请求”（乘客上下）之前，不会改变移动的方向。和电梯一样，内核也试图尽可能在一个方向移动磁头，这个方法在保证所有的请求最终被满足的同时，趋向于最小化定位时间。

Linux 的每个 I/O 请求队列由一个 `request_queue` 类型的结构表示，该结构在 `<linux/blkdev.h>` 中声明。`request_queue` 结构看起来类似 `file_operations` 或其它对象，其中包含有一组操作该队列的函数指针，比如说，驱动程序的 `request` 函数就保存在这里。其中还包含有一个队列头（使用第 10 章“链表”中曾描述过的 `<linux/list.h>` 中的函数），该队列头指向该设备的未处理请求。

这些请求具有 `request` 结构类型，前面我们已经提到过该结构中的一些成员。`request` 结构实际上要更加复杂一些，但是，要理解这个结构，首先要理解 Linux 的缓冲区缓存结构。

`request` 结构和缓冲区缓存

`request` 结构的设计和 Linux 内存管理方法有关。类似大部分的 Unix 类系统，Linux 维护一个缓冲区缓存，它是一个内存区域，保存有磁盘数据块的复本。在内核的更高级别，会执行大量的“磁盘”操作（比如在文件系统部分代码中），但这些操作只在缓冲区缓存上进行，却不会生成任何实际的 I/O 操作。通过主动缓存，内核能够避免许多读取操作，而且多个写入操作也经常可以合并为单个物理的磁盘写操作。

但是，缓冲区缓存不能避免的方面是，磁盘上相邻的数据块，在内存中肯定不会是相邻的。缓冲区缓存是一个动态的东西，数据块在内存最终是大大分散的。为了跟踪所有的事情，内核通过 `buffer_head` 结构管理缓冲区缓存，每个数据缓冲区关联有一个 `buffer_head`，该结构包含有大量的成员，但大部分成员和驱动程序编写者无关。但是，其中还是有一些重要的成员，如下所示：

```
char *b_data;
```

与该缓冲区头相关联的实际数据块。

```
unsigned long b_size;
```

`b_data` 所指向的数据块大小。

```
kdev_t b_rdev;
```

该缓冲区头所代表的数据块所在的设备。

```
unsigned long b_rsector;
```

该数据块在磁盘上的扇区编号。

```
struct buffer_head *b_reqnext;
```

指向请求队列中缓冲区头结构链表的指针，

```
void (*b_end_io)(struct buffer_head *bh, int uptodate);
```

指向一个函数的指针，当该缓冲区上的 I/O 操作结束时将调用这个函数。`bh` 是缓冲区头本身，而 `uptodate` 在 I/O 成功时取非零值。

传递到驱动程序 `request` 函数中的每个数据块，要么保存在缓冲区缓存中，要么在极少情况下保存在其它地方，但是却要使其看起来保存在缓冲区缓存中*。这样，传递到驱动程序的每个请求处理一个或更多的 `buffer_head` 结构。`request` 结构包含有一个称为 `bh` 的成员，该成员指向由这些结构组成的一个链表。为满足该请求，需要在该链表的每个缓冲区上执行指定的 I/O 操作。图 12-2 描述了请求队列和 `buffer_head` 结构之间的关系。

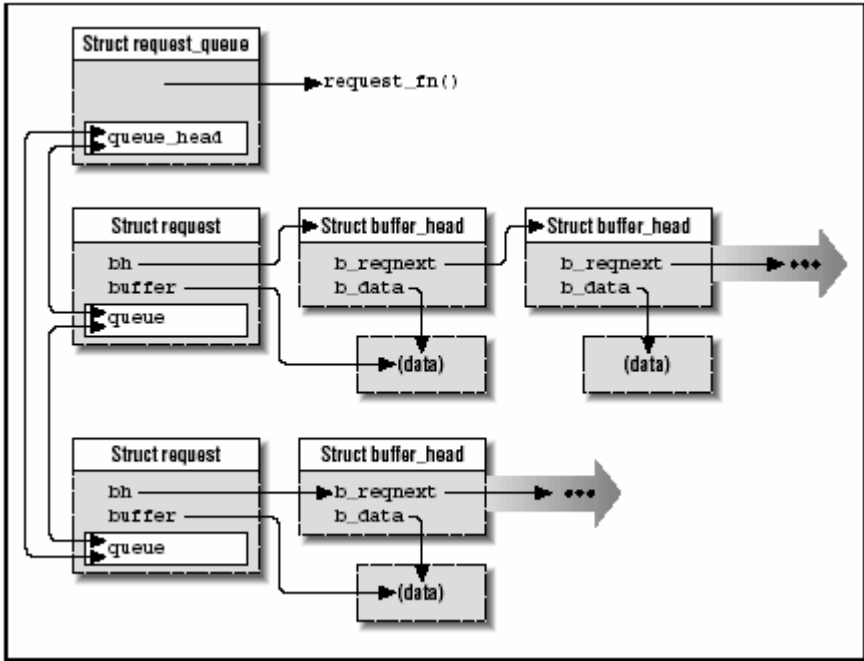


图 12-2: I/O 请求队列当中的缓冲区

* 例如，在我们 RAM 磁盘驱动程序中，需要使内存看起来位于缓冲区缓存中。因为该“磁盘”缓冲区已经存在于系统 RAM 了，因此就没有必要在缓冲区缓存中保留副本。这样，我们的示例代码比起一个正确实现的 RAM 磁盘来讲效率要低一些，因为根本没有考虑到 RAM 磁盘特有的性能问题。

请求并不是由随机的缓冲区链表组成的，相反，所有关联到某单个请求的缓冲区头属于磁盘上一系列相邻的数据块。这样，在某种意义上，一个请求将是针对磁盘上一组（也许很长）数据块的单个操作。分组之后的数据块称为“集群”，在结束请求链表的讨论之后，我们将详细讨论集群请求。

操作请求队列

在头文件 `<linux/blkdev.h>` 中定义了一些用来操作请求队列的函数，大部分实现为预处理程序的宏。并不是所有的驱动程序需要在这个级别上操作请求队列，但熟悉它的工作方式对我们来讲非常有帮助。大部分请求队列函数将在我们需要的时候讲述，但这里要介绍如下几个重要的函数：

返回请求链表中的下一个入口。通常，`head` 参数是 `request_queue` 结构的 `queue_head` 成员，这种情况下，该函数返回队列中的第一个入口。该函数使用 `list_entry` 宏在链表中执行检索。

```
struct request *blkdev_next_request(struct request *req);
struct request *blkdev_prev_request(struct request *req);
```

给定一个 `request` 结构，返回请求队列中的下一个或者前一个结构。

```
blkdev_dequeue_request(struct request *req);
```

从请求队列中删除一个请求。

```
blkdev_release_request(struct request *req);
```

在一个请求被完整执行后，将该 `request` 结构释放给内核。每个请求队列维护有它自己的空闲 `request` 结构链表（实际上有两个：一个用于读取，一个用于写入），该函数把要释放的 `request` 结构放回对应的空闲链表。`blkdev_release_request` 同时会唤醒任何等待在空闲请求结构上的进程。

上述所有函数均需要拥有 `io_request_lock`，下面我们将讨论这个请求锁。

I/O 请求锁

I/O 请求队列是一个复杂的数据结构，内核许多地方都要访问该结构。当你的驱动程序正在删除一个请求的同时，内核完全有可能正要往队列中添加更多的请求。因此，该队列成为通常所说的竞态，为此，必须采取适当的保护措施。

在 Linux 2.2 和 2.4 中，所有的请求队列通过一个单独的全局自旋锁 `io_request_lock` 来保护。所有需要操作请求队列的代码，都必须拥有该锁并禁止中断，只有一个小的例外：请求队列当中的第一个入口（默认情况下）被认为是由驱动程序所拥有。在操作请求队列之前未获取 `io_request_lock`，将导致该队列被破坏，随之而来的将是系统的崩溃。

前面那个简单的 `request` 函数不需要考虑这个锁，因为内核会在调用 `request` 函数的时候已经获取了 `io_request_lock`。这样，驱动程序就不会破坏请求队列，同时，也避免了对 `request` 函数的重入调用。这种方法保证了未考虑 SMP 的驱动程序能够在多处理器系统上正常工作。

然而，我们需要注意的是，`io_request_lock` 是一个昂贵的资源。在驱动程序拥有这个锁的同时，其它任何请求都不能排队到系统中的任何块设备上，也不会有其它任何 `request` 函数被调用。长时间拥有这个锁的驱动程序将最终降低整个系统的运行速度。

因此，好的块驱动程序经常在尽可能短的时间内释放这个锁，我们马上将会看到这种操作的示例。但是，主动释放 `io_request_lock` 的块驱动程序必须要处理两个重要的事情。首先，在 `request` 函数返回之前，必须重新获得该锁，因为调用 `request` 的代码期望 `request` 仍然拥有这个锁。另外一个需要注意的是，一旦 `io_request_lock` 被释放，对该 `request` 函数的重入调用就可能发生，因此，该函数必须能够处理这种可能性。

后面这种情况也可能在另外一种情形下发生，即当某个 I/O 请求仍然活动（正在被处理）的情况下，`request` 函数返回。许多针对实际硬件的驱动程序会启动一个 I/O 操作，然后返回，而该操作将会在驱动程序的中断处理程序中完成。在本章后面我们将详细讨论中断驱动的块 I/O，但此处仍然要提醒读者，`request` 函数可能在这些操作正在进行时被调用。

许多驱动程序通过维护一个内部的请求队列来处理 `request` 函数的重入性。`request` 函数只是简单地将新请求从 I/O 请求队列当中删除，并将这些请求添加到内部的队列，然后，通过组合 `tasklet` 和中断处理程序来处理这个内部队列。

blk.h 中的宏和函数是如何工作的

在我们前面那个简单的 `request` 函数中，我们没有考虑 `buffer_head` 结构或者链表。`<linux/blk.h>` 中的宏和函数隐藏了 I/O 请求队列的结构，以便简化块驱动程序的编写工作。这一小节，我们将讨论操作请求队列时涉及到的实际步骤，其后的小节将讨论编写块 `request` 函数的一些更为高级的技术。

我们早先看到的 `request` 结构的几个成员，即 `sector`、`current_nr_sectors` 以及 `buffer`，实际是保存在该链表第一个 `buffer_head` 结构中类似信息的复本。这样，一个通过 `CURRENT` 指针使用这些信息的 `request` 函数，实际处理的是该请求当中可能存在的许多缓冲区的第一个。将多个缓冲区请求分离成表面上独立的单个缓冲区请求的任务，由 `<linux/blk.h>` 中的两个重要的定义完成：`INIT_REQUEST` 宏和 `end_request` 函数。

其中，`INIT_REQUEST` 更为简单一些，它所做的一切工作，实际就是在请求队列上完成几个一致性检查，并且在队列为空时从 `request` 函数中返回。它仅仅确保还有其它的工作要做。

大量的队列管理工作由 `end_request` 完成。需要记住的是，该函数在驱动程序处理完单个“请求”（实际是一个缓冲区）时调用，它要执行如下几个任务：

1. 完成当前缓冲区上的 I/O 处理。它传递当前操作的状态并调用 `b_end_io` 函数，该函数将唤醒任何睡眠在该缓冲区上的进程。
2. 从请求的链表中删除该缓冲区。如果还有其它缓冲区需要处理，`request` 结构中的 `sector`、`current_nr_sectors` 和 `buffer` 成员将被更新，以便反映出链表中的下一个 `buffer_head` 结构。在这种情况下（即还有其它缓冲区需要传输），`end_request` 将结束本次迭代而不会执行第 3 步和第 5 步。
3. 调用 `add_blkdev_randomness` 更新熵池，除非 `DEVICE_NO_RANDOM` 已被定义（`sbull` 驱动程序就定义了这个宏）。
4. 调用 `blkdev_dequeue_request` 函数，从请求队列当中删除已完成的请求。这个步骤将修改请求队列，因此，一定要在拥有 `io_request_lock` 的情况下执行。

5. 将已完成的请求释放给系统，这里，`io_request_lock` 也必须被获得。

内核为 `end_request` 定义了两个辅助函数，它们可完成大部分的工作。第一个称为 `end_that_request_first`，它处理上面描述的前两个步骤。其原型为：

```
int end_that_request_first(struct request *req, int status, char *name);
```

`status` 是传递给 `end_request` 的请求状态，`name` 参数是设备名称，用于打印错误消息。如果当前请求中没有其它缓冲区需要处理，该函数返回非零值，这时，整个工作结束。否则，要调用 `end_that_request_last` 解除排队，并释放请求。`end_that_request_last` 的原型如下：

```
void end_that_request_last(struct request *req);
```

在 `end_request` 中，这个步骤由下面的代码完成：

```
struct request *req = CURRENT;
blkdev_dequeue_request(req);
end_that_request_last(req);
```

这就是 `end_request` 函数的所有信息了。

12.4.2 集群请求

现在起，我们将讨论如何应用上面这些背景知识以编写更好的块驱动程序，首先要讨论的是集群请求的处理。先前曾提到，集群是将磁盘上相邻数据块的操作请求合并起来的一种方法。集群能带给我们两个好处，首先，集群能够加速数据传输；其次，通过避免分配冗余的请求结构，集群可以节省内核中内存的使用。

我们知道，块驱动程序根本不用关心集群，`<linux/blk.h>` 已经透明地将每个集群请求划分成对应的多个组成片段。但是，在许多情况下，驱动程序能够通过显式操作集群而获得好处。我们经常可以同时为多个连续的数据块安排 I/O 操作，从而提高吞吐率。例如，Linux 软盘驱动程序努力做到在单个操作中将整个磁道写入软盘。大部分高性能的磁盘控制器也可以完成“分散/聚集”I/O，从而将获得更大的性能提高。

为了获得集群带来的好处，块驱动程序必须直接检查附加到某个请求上的 `buffer_head` 结构链表。`CURRENT->bh` 指向该链表，而随后的缓冲区可通过每个 `buffer_head` 结构中的 `b_reqnext` 指针找到。执行集群 I/O 的驱动程序应该大体采用下列顺序来操作集群中的每个缓冲区：

1. 安排传输地址 `bn->b_data` 处的数据块，该数据块大小为 `bh->b_size` 字节，数据传输方向为 `CURRENT->cmd`（要么为 `READ`，要么为 `WRITE`）。
2. 检索链表中的下一个缓冲区头：`bn->b_reqnext`，然后，从链表中解除已传输的缓冲区，可通过对其 `b_reqnext`（刚刚检索到的指向新缓冲区的指针）指针的清零而实现。
3. 更新 `request` 结构，以反映出被删除缓冲区上的 I/O 结束。`CURRENT->hard_nr_sectors` 和 `CURRENT->nr_sectors` 应该减去从该缓冲区中传输的扇区数（不是块数目）；而 `CURRENT->hard_sector` 和 `CURRENT->sector` 则应该加上相同的扇区数。执行上述操作可保持 `request` 结构的一致性。

4. 返回开头以传输下一个相邻数据块。

每个缓冲区上的 I/O 完成后，驱动程序应该调用该缓冲区的 I/O 结束例程来通知内核：

```
bh->b_end_io(bh, status);
```

在操作成功时，应传递 `status` 为非零值。当然，我们还得从队列当中将已完成操作的请求结构删掉。上述的步骤可在未获取 `io_request_lock` 的情况下完成，但对队列本身的修改，必须在获得该锁的情况下进行。

在 I/O 操作结束时，驱动程序仍然可以使用 `end_request`（和直接操作队列相反），只要注意正确设置 `CURRENT->bh` 指针即可。该指针应该取 `NULL`，或者指向最后一个被传输的 `buffer_head` 结构。在后面这种情况下，不应该在最后的缓冲区上调用 `b_end_io` 函数，因为 `end_request` 将对该缓冲区调用这个函数。

功能完整的集群实现可见 `drivers/block/floppy.c`，而所需的操作总结可见 `blk.h` 中的 `end_request` 函数。`floppy.c` 和 `blk.h` 都不太容易理解，但后者较为容易一些。

活动的队列头

另外一个关于 I/O 请求队列的细节涉及到处理集群的块驱动程序。它必须处理队列头，也就是队列中的第一个请求。考虑到历史遗留的兼容性原因，内核（几乎）总是假定块驱动程序会处理请求队列当中的第一个入口。为避免因为冲突而导致破坏性的结果，一旦内核获得某个请求队列的头，就不会修改这个请求。在该请求上，不会发生其它的集群，而电梯算法代码也不会将其它请求放到它的前面。

许多块驱动程序在开始处理某个队列当中的请求之前，首先要删除这些请求。如果读者的驱动程序恰好以这种方式工作，则应该特殊处理位于队列头部的请求。这种情况下，驱动程序应该调用 `blk_queue_headactive` 通知内核队列头不是活动的：

```
blk_queue_headactive(request_queue_t *queue, int active);
```

如果 `active` 是 0，内核可以对请求队列头进行修改。

12.4.2 多队列的块驱动程序

前面提到，内核默认时为每个主设备号维护一个单独的 I/O 请求队列。单个的队列对类似 `sbull` 这样设备来讲是足够了，但在实际情况下，单个队列总不是最优的。

考虑一个处理实际磁盘设备的驱动程序，每个磁盘能够独立操作，如果驱动程序能够并行工作，则系统性能肯定会更好一些。基于单个队列的简单驱动程序不可能实现这一目标，因为它每次只能在单个设备上执行操作。

对驱动程序而言，遍历请求队列，并找出独立驱动器的请求并不困难。但 2.4 内核通过允许驱动程序为每个设备建立独立的队列而使得实现这个目标更加容易。许多高性能的驱动程序利用了这个多队列功能。实现多队列并不困难，但仅仅 `<linux/blk.h>` 中的那些简单定义还不够。

如果在编译时定义了 `SBULL_MULTIQUEUE` 符号，则 `sbull` 驱动程序将在多队列模式下操作。它不使用 `<linux/blk.h>` 中的宏，并且演示了本小节中描述过的许多特性。

要在多队列模式下工作，块驱动程序必须定义它自己的请求队列。为此，`sbull` 在 `Sbull_Dev` 结构中添加了一个队列成员：

```
request_queue_t queue;
int busy;
```

`busy` 标志用来保护 `request` 函数的重入，具体将在下面讲到。

当然，请求队列必须被初始化。`sbull` 以下面的方式初始化它的设备相关队列：

```
for (i = 0; i < sbull_devs; i++) {
    blk_init_queue(&sbull_devices[i].queue, sbull_request);
    blk_queue_headactive(&sbull_devices[i].queue, 0);
}
blk_dev[major].queue = sbull_find_queue;
```

对 `blk_init_queue` 的调用和我们先前见过的一样，只是现在传递了设备相关的队列，而不是用于我们这个主设备编号的默认队列。上面的代码同时将队列标记为没有活动的队列头。

读者也许想知道内核是如何找到这些请求队列的，因为它们看起来隐藏在设备相关的私有结构当中。这里的关键在于上述代码中的最后一行，它设置了全局 `blk_dev` 结构当中的 `queue` 成员，该成员指向一个函数，这个函数为给定的设备编号找出正确的请求队列。使用默认队列的设备没有这样的函数，但多队列设备必须实现这个函数：

```
request_queue_t *sbull_find_queue(kdev_t device)
{
    int devno = DEVICE_NR(device);

    if (devno >= sbull_devs) {
        static int count = 0;
        if (count++ < 5) /* print the message at most five times */
            printk(KERN_WARNING "sbull: request for unknown device\n");
        return NULL;
    }
    return &sbull_devices[devno].queue;
}
```

和 `request` 函数类似，`sbull_find_queue` 必须是原子的（不允许睡眠）。

每个队列有自己的 `request` 函数，尽管通常驱动程序会对所有的队列使用相同的函数。内核会将实际的请求队列作为参数传递到 `request` 函数中，这样，`request` 函数就能够知道它正在操作的设备是哪一个。`sbull` 当中使用的多队列的 `request` 函数，看起来和我们先前看到的不太一样，这是因为这个 `request` 函数直接操作请求队列，同时，它还在执行传输时解除 `io_request_lock` 锁，以便内核能够执行其它的块操作。最后，代码还要注意避免两个独立的危险：对 `request` 函数的多次调用以及对设备本身的冲突访问。

```
void sbull_request(request_queue_t *q)
{
```

```

Sbull_Dev *device;
struct request *req;
int status;

/* Find our device */
device = sbull_locate_device (blkdev_entry_next_request(&q->queue_head));
if (device->busy) /* no race here - io_request_lock held */
    return;
device->busy = 1;

/* Process requests in the queue */
while(! list_empty(&q->queue_head)) {

/* Pull the next request off the list. */
    req = blkdev_entry_next_request(&q->queue_head);
    blkdev_dequeue_request(req);
    spin_unlock_irq (&io_request_lock);
    spin_lock(&device->lock);

/* Process all of the buffers in this (possibly clustered) request. */
    do {
        status = sbull_transfer(device, req);
    } while (end_that_request_first(req, status, DEVICE_NAME));
    spin_unlock(&device->lock);
    spin_lock_irq (&io_request_lock);
    end_that_request_last(req);
}
device->busy = 0;
}

```

上面 `request` 函数使用链表函数 `list_empty` 来测试特定的请求队列，而没有使用 `INIT_REQUEST`。只要有请求存在，它使用 `blkdev_dequeue_request` 函数从队列中删除每个请求，紧接着，一旦删除过程结束，就可以解除 `io_request_lock` 并获得设备相关的锁。实际的数据传输通过 `sbull_transfer` 完成，我们在前面讨论过这个函数。

每次对 `sbull_transfer` 的调用将处理附加到该请求上的一个 `buffer_head` 结构。然后，这个函数调用 `end_that_request_first` 处置已传输的缓冲区，如果该请求全部完成，则调用 `end_that_request_last` 函数整个清除该请求。

有必要了解一下这里的并行管理。`busy` 标志用于避免对 `sbull_request` 的多次调用。因为 `sbull_request` 始终在拥有 `io_request_lock` 的时候被调用，因此，不需要其它额外的保护就可以测试并设置 `busy` 标志。（否则，就需要使用 `atomic_t`）。在获得设备相关的锁之前，`io_request_lock` 被解除。这里，我们可以获得多个锁，而不需要冒死锁的风险，但实现起来很困难，所以，在条件允许的情况下，最好还是在获得另外一个锁之前释放当前的锁。

`end_that_request_first` 在不拥有 `io_request_lock` 的情况下被调用。因为该函数仅仅在给定的 `request` 结构上进行操作，因此，这样的调用是安全的——只要该请求不在队列上。但是，对 `end_that_request_last` 的调用，却需要获得 `io_request_lock` 锁，因为它会把该请求释放到请求队列的空闲链表中。同时，该函数始终从外层循环中退出，这时，驱动程序拥有 `io_request_lock`，但释放了设备锁。

当然，多队列驱动程序必须在删除模块时清除所有的队列：

```

for (i = 0; i < sbull_devs; i++)
    blk_cleanup_queue(&sbull_devices[i].queue);

```

```
blk_dev[major].queue = NULL;
```

但是，这段代码可以编写得更加有效一些。现在，代码在初始化阶段就分配所有的请求队列，不管其中一些是否会在将来用到。请求队列是非常大的数据结构，因为在队列初始化的时候，可能会分配许多（也许是上千个）`request` 结构。更加巧妙的做法是在 `open` 方法或者 `queue` 函数中，只有在必要的时候才分配请求队列。为了避免使代码复杂化，对 `sbull` 驱动程序，我们选择了一种更为简单的实现方法。

本小节讲述了多队列驱动程序机制。当然，处理实际硬件的驱动程序可能还有其它需要处理的问题，比如，对控制器的串行化访问等等。但是，多队列驱动程序的基本结构就是这样的。

12.4.4 没有请求队列的情况

到目前为止，我们一直在围绕 I/O 请求队列进行讨论。请求队列的目的，是为了通过让驱动器进行异步操作，或者更关键的，通过合并（磁盘上）相邻的操作而提高性能。对于通常的磁盘设备，连续块上的操作很常见，因此，这种优化是必要的。

但是，并不是所有的块设备都能从请求队列中获得好处。比如 `sbull`，进程同步请求而且在寻址时间上没有任何问题。其实，对 `sbull` 来讲，请求队列实际上降低了数据传输的速度。其它类型的块设备也可以在有请求队列的情况下工作得更好。例如，由多个磁盘组成的 RAID 设备，经常将“连续”的块分布在多个物理设备上。通过逻辑卷管理器（`logical volume manager`, LVM）功能（首次出现在 2.4 中）实现的块设备，比起提供给内核其它部分的块结构来讲，也具有更加复杂的实现。

在 2.4 内核中，块 I/O 请求由函数 `_make_request` 放在队列当中，该函数还负责调用驱动程序的 `request` 函数。但是，需要对请求的排队过程进行更多控制的块驱动程序，可以利用自己的“make request”函数替代这个函数。RAID 和 LVM 驱动程序就是这样做的，这样，它们最终就可以将每个 I/O 请求（根据不同的块设备号）重新排队到适当的低层设备上。而一个 RAM 磁盘驱动程序，则可以直接执行 I/O 操作。

在 2.4 系统上使用 `noqueue=1` 选项装载 `sbull` 时，它将提供自己的“make request”函数，并在没有请求队列的情况下工作。这种情况下，第一步首先要替换 `_make_request` 函数，“make request”函数指针保存在请求队列当中，可通过 `blk_queue_make_request` 函数改变：

```
void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);
```

其中，`make_request_fn` 类型的定义如下：

```
typedef int (make_request_fn) (request_queue_t *q, int rw,
                               struct buffer_head *bh);
```

“make request”函数必须安排传输给定的数据块，并在传输完成时调用 `b_end_io` 函数。在调用 `make_request_fn` 函数的时候，内核并不获得 `io_request_lock` 锁，因此，如果该函数要自己操作请求队列，则必须获取这个锁。如果传输已经建立（不一定完成），则该函数应该返回 0。

这里所说的“安排传输”这一短语，是经过仔细斟酌的。通常，驱动程序自己的“make request”

函数不会真正传输数据。考虑 RAID 设备的驱动程序，它的“make request”函数需要做的是将 I/O 操作映射到它的组成设备上，然后调用那个设备的驱动程序完成实际的工作。这个映射通过将 `buffer_head` 结构中的 `b_rdev` 成员设置成完成传输的“真实”设备的编号而实现，然后，通过返回一个非零值来表示该数据块仍然需要被写入。

当内核从“make request”函数中获得一个非零值时，它判断该工作尚未完成并且需要重试。但是，它首先要查找 `b_rdev` 成员所代表的设备的“make request”函数。这样，对 RAID 设备来讲，RAID 驱动程序的“make request”函数不会再次被调用，相反，内核将把这个数据块传递到低层设备的对应函数。

`sbull` 在初始化阶段，用下面的代码设置自己的“make request”函数：

```
if (noqueue)
    blk_queue_make_request(BLK_DEFAULT_QUEUE(major), sbull_make_request);
```

在这种模式下，`sbull` 并没有调用 `blk_init_queue`，因为我们不会使用请求队列。

当内核产生一个对 `sbull` 设备的请求时，它将调用 `sbull_make_request`，该函数的定义如下：

```
int sbull_make_request(request_queue_t *queue, int rw,
                      struct buffer_head *bh)
{
    u8 *ptr;

    /* Figure out what we are doing */
    Sbull_Dev *device = sbull_devices + MINOR(bh->b_rdev);
    ptr = device->data + bh->b_rsector * sbull_hardsect;

    /* Paranoid check; this apparently can really happen */
    if (ptr + bh->b_size > device->data + sbull_blksize*sbull_size) {
        static int count = 0;
        if (count++ < 5)
            printk(KERN_WARNING "sbull: request past end of device\n");
        bh->b_end_io(bh, 0);
        return 0;
    }

    /* This could be a high-memory buffer; shift it down */
#ifdef CONFIG_HIGHMEM
    bh = create_bounce(rw, bh);
#endif

    /* Do the transfer */
    switch(rw) {
    case READ:
    case READA: /* Read ahead */
        memcpy(bh->b_data, ptr, bh->b_size); /* from sbull to buffer */
        bh->b_end_io(bh, 1);
        break;
    case WRITE:
        refile_buffer(bh);
        memcpy(ptr, bh->b_data, bh->b_size); /* from buffer to sbull */
        mark_buffer_uptodate(bh, 1);
        bh->b_end_io(bh, 1);
        break;
    default:
        /* can't happen */
        bh->b_end_io(bh, 0);
        break;
    }
```

```

    }

    /* Nonzero return means we're done */
    return 0;
}

```

上面的大部分代码一定看似熟悉。它包含了通常的计算以确定块在 `sbull` 中的位置，并使用 `memcpy` 执行操作。因为该操作将立即结束，所以它调用 `bh->b_end_io` 函数表示操作已完成，然后给内核返回 0。

但是，这里有一个“make request”函数必须要注意的细节。要传输的缓冲区可能位于内存不能直接访问的高端内存。高端内存的具体内容将在第 13 章中讲述，这里不会重复。读者只要知道处理这个问题的一个办法就是，将高端内存中的缓冲区用一个可访问内存中的缓冲区替代。`create_bounce` 函数就是用来完成这个工作的，并且对驱动程序来讲是透明的。内核通常在将缓冲区放到驱动程序的请求队列之前使用 `create_bounce`，但是，如果驱动程序实现了自己的 `make_request_fn` 函数，则必须由自己完成这个工作。

12.5 挂装和卸载是如何工作的

块设备和字符设备以及普通文件之间有着明显的不同——块设备可以被挂装到系统的文件系统上。挂装提供了对字符设备来讲不可见的间接方法????，后者通常通过由特定进程所拥有的 `struct file` 指针来访问，当文件系统被挂装时，没有任何进程拥有这个 `file` 结构。

在内核挂装文件系统时的某个设备时，它调用标准的 `open` 方法访问驱动程序。但在这种情况下，用以调用 `open` 的两个参数 `filp` 和 `inode` 均为哑变量。在 `file` 结构中，只有 `f_mode` 和 `f_flags` 成员保存着有意义的值，而 `inode` 结构中，也只会用到 `i_rdev`。其余的成员含有随机值，因此不应该使用这些成员。`f_mode` 的值告诉驱动程序，以只读方式 (`f_mode == FMODE_READ`) 还是以读/写方式 (`f_mode == (FMODE_READ|FMODE_WRITE)`) 挂装设备。

这样，`open` 接口看起来有点奇怪，但有两个原因促使内核这样做。首先，进程可以以标准方式调用 `open` 来直接访问设备，比如 `mkfs` 工具。另外一个原因源于历史遗留问题：块驱动程序使用了与字符驱动程序一样的 `file_operations` 结构，因此，不得不遵循相同的接口。

除了传递给 `open` 方法的有限参数以外，驱动程序看不到任何挂装文件系统期间所发生的其它东西。设备被打开之后，内核就会调用 `request` 方法传输数据块，驱动程序其实无法了解发生在各种操作之间的区别，到底是在响应独立的进程（比如 `fsck`），还是在处理源于内核文件系统层的操作。

对 `umount` 来讲，它只是刷新缓冲区缓存然后调用驱动程序的 `release` 方法。因为没有任何有具体含义的 `filp` 可传递给 `release` 方法，所以，内核使用 `NULL`。因此，块驱动程序的 `release` 实现，不能使用 `filp->private_data` 来访问设备信息，而只能使用 `inode->i_rdev` 来区别不同的设备。这样，`sbull` 的 `release` 方法如下实现：

```

int sbull_release (struct inode *inode, struct file *filp)
{
    Sbull_Dev *dev = sbull_devices + MINOR(inode->i_rdev);

```

```

    spin_lock(&dev->lock);
    dev->usage--;
    MOD_DEC_USE_COUNT;
    spin_unlock(&dev->lock);
    return 0;
}

```

其它的驱动程序函数不会受到“不存在的 `filp`”问题的影响，因为它们根本不会涉及到文件系统的挂装和卸载。例如，`ioctl` 只会被显式调用 `open` 方法打开设备的进程调用。

12.6 `ioctl` 方法

和字符设备类似，我们也可以通过 `ioctl` 系统调用来操作块设备。块驱动程序和字符驱动程序在 `ioctl` 实现上的唯一不同，就是块设备驱动程序共享了大量常见的 `ioctl` 命令，大多数驱动程序都会支持这些命令。

块驱动程序通常要处理的命令如下所示（在 `<linux/fs.h>` 中声明）：

BLKGETSIZE

检索当前设备的大小，以扇区数表示。系统调用传递的 `arg` 参数是一个指向长整数的指针，用来将设备大小值复制到用户空间变量中。`mkfs` 可利用该 `ioctl` 命令了解将要创建的文件系统大小。

BLKFLSBUF

从字面上看，该命令的含义是“刷新缓冲区。”该命令的实现对所有设备来讲都是一样的，其代码可在下面的全局 `ioctl` 命令示例代码中找到。

BLKRRPART

重新读取分区表。该命令仅对可分区设备有效，将在本章后面介绍。

BLKRASET

BLKRASET

用来获取或者修改设备当前的块级预读值（即保存在 `read_ahead` 数组中的值）。对 `GET`，应该使用传递到 `ioctl` 的 `arg` 参数的指针，将当前值写入用户空间的长整型变量；而 `SET`，新的值作为参数传递。

BLKFRASET

BLKFRASET

获取或设置设备的文件系统级预读值（保存在 `max_readahead` 数组中的值）。

BLKROSET

BLKROGET

上述命令用来修改或者检查设备的只读标志。

BLKSECTGET

BLKSECTSET

上述命令检索或设置每个请求的最大扇区数（保存在 `max_sectors`）。

BLKSSZGET

通过指向调用者的整型变量指针，返回当前块设备的扇区大小，该大小值直接从 `hardsect_size` 数组中获得。

BLKPG

BLKPG 命令允许用户模式的程序添加或者删除分区。它由 **blk_ioctl**（很快就会讲到）实现，内核中的驱动程序不需要提供它们自己的实现。

BLKELVGET**BLKELVSET**

通过这些命令可控制电梯请求排序算法的工作方式。和 **BLKPG** 类似，驱动程序不需要直接实现该命令。

HDIO_GETGEO

定义在 `<linux/hdreg.h>` 中，用来检索磁盘的几何参数。应该通过一个 **hd_geometry** 结构将几何参数写入用户空间，该结构也定义在 `hdreg.h` 中。**sbull** 给出了该命令的通常实现。

HDIO_GETGEO 命令是 `<linux/hdreg.h>` 中定义的一系列 **HDIO_** 命令中最为常用的。感兴趣的读者可以阅读 `ide.c` 和 `hd.c` 了解这些命令的相应信息。

对所有的块驱动程序，上述 **ioctl** 命令几乎以相同的方式实现。2.4 内核提供了一个函数，即 **blk_ioctl**，可调用该函数实现常见命令，该函数在 `<linux/blkpg.h>` 中声明。通常来说，需要由驱动程序自己实现的命令是 **BLKGETSIZE** 和 **HDIO_GETGEO**，而其它命令，都可以传递给 **blk_ioctl** 处理。

sbull 设备只支持刚刚列出的常用命令，因为设备特有命令的实现方法，和字符驱动程序的实现方法没有任何区别。**sbull** 的 **ioctl** 实现如下：

```
int sbull_ioctl (struct inode *inode, struct file *filp,
                unsigned int cmd, unsigned long arg)
{
    int err;
    long size;
    struct hd_geometry geo;

    PDEBUG("ioctl 0x%x 0x%lx\n", cmd, arg);
    switch(cmd) {

        case BLKGETSIZE:
            /* Return the device size, expressed in sectors */
            if (!arg) return -EINVAL; /* NULL pointer: not valid */
            err = ! access_ok (VERIFY_WRITE, arg, sizeof(long));
            if (err) return -EFAULT;
            size = blksize*sbull_sizes[MINOR(inode->i_rdev)]
                / sbull_hardsects[MINOR(inode->i_rdev)];
            if (copy_to_user((long *) arg, &size, sizeof (long)))
                return -EFAULT;
            return 0;

        case BLKRRPART: /* reread partition table: can't do it */
            return -ENOTTY;

        case HDIO_GETGEO:
            /*
             * Get geometry: since we are a virtual device, we have to make
             * up something plausible. So we claim 16 sectors, four heads,
             * and calculate the corresponding number of cylinders. We set
             * the start of data at sector four.
             */
            err = ! access_ok(VERIFY_WRITE, arg, sizeof(geo));
```

```

        if (err) return -EFAULT;
        size = sbull_size * blksize / sbull_hardsect;
        geo.cylinders = (size & ~0x3f) >> 6;
        geo.heads = 4;
        geo.sectors = 16;
        geo.start = 4;
        if (copy_to_user((void *) arg, &geo, sizeof(geo)))
            return -EFAULT;
        return 0;

    default:
        /*
         * For ioctls we don't understand, let the block layer
         * handle them.
         */
        return blk_ioctl(inode->i_rdev, cmd, arg);
    }

    return -ENOTTY; /* unknown command */
}

```

该函数开头的 **PDEBUG** 语句被保留，这样，在编译该模块时，可打开调试选项，从而能够看到该设备上发生的 **ioctl** 命令。

12.7 可移动设备

到目前为止，我们一直忽略了 **block_device_operations** 结构中的两个文件操作，它们用于支持移动介质设备，现在我们介绍这两个操作。**sbull** 其实根本不是一种移动设备，但它伪装成了移动设备，因此，需要实现这两个操作。

这两个操作是 **check_media_change** 和 **revalidate**。前者用于检查自从上次访问以来，设备是否发生过变化，而后者在磁盘变化之后，重新初始化驱动程序状态。

对 **sbull** 而言，在其使用计数减小为零后，稍后就会释放与某个设备相关联的数据区域，这样，就可以通过保持设备被卸载（或关闭）足够长的时间来模拟磁盘变化，下一次对设备的访问，将分配一块新的内存区域。

这种类型的“适时过期”方法使用内核定时器实现。

check_media_change

这个检查函数只有一个 **kdev_t** 型参数，用来标识设备。返回值为 **1** 表明介质变化，反之返回 **0**。不支持移动设备的块驱动程序可将 **bdops->check_media_change** 设置为 **NULL**，从而无需声明该函数。

值得注意的是，当设备是可移动的，但又无法知道是否发生变化时，返回 **1** 是一种安全的选择。这正是 **IDE** 驱动程序处理可移动磁盘的方法。

sbull 的实现是在设备因为定时器到期而从内存中删除时返回 **1**，而在数据仍然有效时返回 **0**。如果打开调试选项，则会向系统日志打印一条消息，这样，用户可以验证该方法是由内核调用的。

```

int sbull_check_change(kdev_t i_rdev)
{

```

```

int minor = MINOR(i_rdev);
Sbull_Dev *dev = sbull_devices + minor;

PDEBUG("check_change for dev %i\n",minor);
if (dev->data)
    return 0; /* still valid */
return 1; /* expired */
}

```

12.7.1 revalidation

revalidation 函数在检测到磁盘变化时调用。内核 2.1 版本中实现的各种 **stat** 系统调用也会调用这个函数。该函数的返回值目前还没有被使用，为了安全起见，应该返回 0 以表示成功，返回负的错误值以表示错误。

由 **revalidation** 执行的操作是设备特有的，但 **revalidation** 通常用来更新内部的状态信息以便反映出新的设备。

sbull 的 **revalidation** 方法在没有合法内存区域的情况下，将试着分配一个新的数据区。

```

int sbull_revalidate(kdev_t i_rdev)
{
    Sbull_Dev *dev = sbull_devices + MINOR(i_rdev);

    PDEBUG("revalidate for dev %i\n",MINOR(i_rdev));
    if (dev->data)
        return 0;

    dev->data = vmalloc(dev->size);
    if (!dev->data)
        return -ENOMEM;
    return 0;
}

```

12.7.2 需要特别注意的事项

移动设备的驱动程序应该在设备被打开时检查磁盘变化情况。内核提供了一个函数，可致使检查的发生：

```
int check_disk_change(kdev_t dev);
```

如果检测到磁盘变化，则返回非零值。内核在挂装期间会自动调用 **check_disk_change**，但不会在 **open** 期间自动调用。

但是，某些程序会直接访问磁盘数据，而不会首先挂装设备，比如：**fsck**、**mcopy** 和 **fdisk** 等等。如果驱动程序在内存中保留了移动设备的状态信息，则应该在第一次打开设备时调用内核的 **check_disk_change** 函数。该函数使用驱动程序的 **check_media_change** 和 **revalidation** 方法，因此，没有必要在 **open** 本身中实现特殊的代码。

下面是 **sbull** 的 **open** 方法实现，该方法处理了磁盘的变化情况：

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

```

{
    Sbull_Dev *dev; /* device information */
    int num = MINOR(inode->i_rdev);

    if (num >= sbull_devs) return -ENODEV;
    dev = sbull_devices + num;

    spin_lock(&dev->lock);
    /* revalidate on first open and fail if no data is there */
    if (!dev->usage) {
        check_disk_change(inode->i_rdev);
        if (!dev->data)
        {
            spin_unlock (&dev->lock);
            return -ENOMEM;
        }
    }
    dev->usage++;
    spin_unlock(&dev->lock);
    MOD_INC_USE_COUNT;
    return 0;          /* success */
}

```

驱动程序不需要为磁盘变化做其它额外的工作。如果在打开计数仍然大于零的情况下发生磁盘变化，则数据将会被破坏。驱动程序能够避免发生这个问题的唯一方法，是利用使用计数控制介质门的锁，当然，物理设备要支持介质门的锁定。这样，`open` 和 `close` 就能够适当地禁止或者打开这个物理锁。

12.8 可分区设备

大多数块设备不会以整块方式使用，相反，系统管理员通常希望对该设备进行分区，也就是说，将整个设备划分成若干独立的伪设备。如果读者试图在 `sbull` 设备上利用 `fdisk` 建立分区，就会遇到问题。`fdisk` 程序称这些分区为 `/dev/sbull01`、`/dev/sbull02` 等等，但这些名称根本就不存在。还要指出的是，目前还没有一种机制将这些名称和 `sbull` 设备当中的分区绑定在一起，因此，在一个块设备能够被分区之前，必须完成一些准备工作。

为了演示如何支持分区，我们引入一个新的设备，称为“`spull`”，表示“Simple Partitionable Utility。”这个设备比起 `sbull` 来更为简单，因为它缺少请求队列的管理以及其它一些灵活性（比如改变硬扇区大小的能力）。该设备保存在 `spull` 目录中，虽然它和 `sbull` 共享某些代码，但和 `sbull` 没有任何关系。

为了在某个设备上支持分区，我们必须赋予每个物理设备若干个次设备号。一个设备号用来访问整个设备（例如，`/dev/hda`），而另外一些用来访问不同的分区（比如 `/dev/hda1`、`/dev/hda2` 等）。因为 `fdisk` 通过在磁盘设备的整体名称后添加数字后缀来建立分区名称，因此，我们会在 `spull` 驱动程序中遵循同样的命名习惯。

由 `spull` 实现的设备结点称为 `pd`，表示“partitionable disk（可分区磁盘）。”四个整体设备（也称为“单元（unit）”）分别命名为 `/dev/pda` 到 `/dev/pdd`，每个设备至多支持 15 个分区。次设备号的含义如下：低四位代表分区编号（0 表示整个设备），高四位表示单元编号。这一约定在源文件中通过下面的宏表示：

```
#define MAJOR_NR spull_major /* force definitions on in blk.h */
```

```
int spull_major; /* must be declared before including blk.h */

#define SPULL_SHIFT 4 /* max 16 partitions */
#define SPULL_MAXNRDEV 4 /* max 4 device units */
#define DEVICE_NR(device) (MINOR(device)>>SPULL_SHIFT)
#define DEVICE_NAME "pd" /* name for messaging */
```

spull 驱动程序同时将硬扇区大小硬编码在代码中，以便简化编程：

```
#define SPULL_HARDSECT 512 /* 512-byte hardware sectors */
```

12.8.1 一般性硬盘

所有的可分区设备都需要知道具体的分区结果，该信息可从分区表中获得，其初始化过程的一部分包括对分区表的解码，并更新内部数据结构来反映出分区信息。

解码并不简单，但所幸的是内核提供了可被所有块设备使用的“一般性硬盘（generic hard disk）”支持。这种支持最终减少了驱动程序中用以处理分区的代码量。这种一般性支持的另外一个好处是，驱动程序编写者不需要理解具体的分区方法，而且还可以在无需修改驱动程序代码的情况下，在内核中添加新的分区方案。

支持分区的块驱动程序必须包含 `<linux/genhd.h>` 头文件，并声明一个 `struct gendisk` 结构，该结构描述了驱动程序所提供的磁盘之布局。内核维护这类结构的一个全局链表，这样，可查询该结构而获得系统中可用的磁盘和分区。

在继续我们的讨论之前，首先了解一下 `struct gendisk` 中的一些成员。在利用一般性设备支持之前，我们需要首先理解这些成员。

```
int major
```

该结构所指的主设备编号。

```
const char *major_name
```

属于该主设备号的设备基本名称（base name）。每个设备的名称通过在基本名称之后添加一个表示单元的字母，以及一个表示分区的编号而形成。例如，“hd”是用来建立 `/dev/hda1` 和 `/dev/hdb3` 的基本名称。在现代模块中，磁盘名称的总长度可达 32 个字符，但 2.0 内核要小一些。如果驱动程序希望能够移植到 2.0 内核，则应该将 `major_name` 成员限制在五个字符之内。spull 的基本名称是 `pd`（“partitionable disk”）。

```
int minor_shift
```

`minor_shift` 表示从次设备编号中得出驱动器编号时的位移数。在 spull 中，该数值为 4。该成员的值应该和宏 `DEVICE_NR(device)`（见“头文件 `blk.h`”一节）的定义保持一致。spull 中，这个宏将展开成 `device>>4`。

```
int max_p
```

分区的最大个数。在我们的例子中，`max_p` 是 16，或者更为一般些，即 `1 << minor_shift`。

```
struct hd_struct *part
```

该设备解码后的分区表。驱动程序可以利用这一成员确定通过每个次设备号能够访问的磁盘扇区范围。驱动程序负责分配和释放该数组，大多数驱动程序将其实现为静态的数组，数组中共有 `max_nr`

<< minor_shift 个结构。在内核解码分区表之前，驱动程序应该将该数组初始化为零。

```
int *sizes
```

是个整型数组，其中包含了与全局 blk_size 数组一样信息，实际上，它们经常是同一个数组。驱动程序负责分配和释放 sizes 数组。注意设备的分区检查代码将把该指针复制到 blk_size，因此，处理可分区设备的驱动程序不必分配后一个数组。

```
int nr_real
```

实际存在的设备（单元）个数。

```
void *real_devices
```

驱动程序可使用此成员保存任何附加的私有数据。

```
void struct gendisk *next
```

用来实现一般性硬盘结构链表的指针。

```
struct block_device_operations *fops;
```

指向设备块操作结构的指针。

许多 gendisk 结构中的成员在初始化阶段进行设置，因此，编译阶段的设置相对简单一些：

```
struct gendisk spull_gendisk = {
    major:          0,          /* Major number assigned later */
    major_name:      "pd",      /* Name of the major device */
    minor_shift:     SPULL_SHIFT, /* Shift to get device number */
    max_p:           1 << SPULL_SHIFT, /* Number of partitions */
    fops:            &spull_bdops, /* Block dev operations */
    /* everything else is dynamic */
};
```

12.8.2 分区检测

在模块初始化其本身时，它必须为分区检测进行适当的设置。首先，spull 为 gendisk 结构设置 spull_sizes 数组（该数组也将赋于 blk_size[MAJOR_NR] 以及 gendisk 结构的 sizes 成员）以及 spull_partitions 数组，该数组保存了实际的分区信息（也将赋于 gendisk 结构的 part 成员）。这两个数组在这个阶段被初始化为零，其代码如下：

```
spull_sizes = kmalloc( (spull_devs << SPULL_SHIFT) * sizeof(int),
                      GFP_KERNEL);
if (!spull_sizes)
    goto fail_malloc;

/* Start with zero-sized partitions, and correctly sized units */
memset(spull_sizes, 0, (spull_devs << SPULL_SHIFT) * sizeof(int));
for (i=0; i< spull_devs; i++)
    spull_sizes[i<<SPULL_SHIFT] = spull_size;
blk_size[MAJOR_NR] = spull_gendisk.sizes = spull_sizes;

/* Allocate the partitions array. */
spull_partitions = kmalloc( (spull_devs << SPULL_SHIFT) *
                           sizeof(struct hd_struct), GFP_KERNEL);
if (!spull_partitions)
    goto fail_malloc;

memset(spull_partitions, 0, (spull_devs << SPULL_SHIFT) *
      sizeof(struct hd_struct));
```

```
/* fill in whole-disk entries */
for (i=0; i < spull_devs; i++)
    spull_partitions[i << SPULL_SHIFT].nr_sects =
        spull_size*(blksize/SPULL_HARDSECT);
spull_gendisk.part = spull_partitions;
spull_gendisk.nr_real = spull_devs;
```

驱动程序也应该将其 `gendisk` 结构包含到全局链表中。因为没有内核提供的函数添加 `gendisk` 结构到该链表中，因此，必须手工完成：

```
spull_gendisk.next = gendisk_head;
gendisk_head = &spull_gendisk;
```

在实际系统中，系统仅仅利用该链表实现了 `/proc/partitions`。

我们在前面看到的 `register_disk` 函数，用来读取磁盘的分区表。

```
register_disk(struct gendisk *gd, int drive, unsigned minors,
             struct block_device_operations *ops, long size);
```

这里，`gd` 就是我们先前准备好的 `gendisk` 结构，`drive` 是设备编号，`minors` 是所支持的分区个数，`ops` 是驱动程序的 `block_device_operations` 结构，而 `size` 则是设备以扇区计的大小。

固定磁盘可在模块初始化阶段以及 `BLKRRPART` 被调用时读取分区表，而移动设备的驱动程序还需要在 `revalidate` 方法中调用该函数。不管是哪种方法，都需要注意 `register_disk` 将调用驱动程序的 `request` 函数读取分区表，因此，驱动程序都应该在这点上经过足够的初始化以处理请求。我们还需注意不能在这时拥有任何可能与 `request` 函数中获得的锁冲突的锁。`register_disk` 必须为系统中实际存在的每个磁盘调用一次。

`spull` 在 `revalidate` 方法中建立分区：

```
int spull_revalidate(kdev_t i_rdev)
{
    /* first partition, # of partitions */
    int part1 = (DEVICE_NR(i_rdev) << SPULL_SHIFT) + 1;
    int npart = (1 << SPULL_SHIFT) - 1;

    /* first clear old partition information */
    memset(spull_gendisk.sizes+part1, 0, npart*sizeof(int));
    memset(spull_gendisk.part +part1, 0, npart*sizeof(struct hd_struct));
    spull_gendisk.part[DEVICE_NR(i_rdev) << SPULL_SHIFT].nr_sects =
        spull_size << 1;

    /* then fill new info */
    printk(KERN_INFO "Spull partition check: (%d) ", DEVICE_NR(i_rdev));
    register_disk(&spull_gendisk, i_rdev, SPULL_MAXNRDEV, &spull_bdops,
        spull_size << 1);
    return 0;
}
```

值得注意的是，`register_disk` 通过重复调用 `printk` 函数来打印分区信息：

```
printk(" %s", disk_name(hd, minor, buf));
```

这就是为什么 `spull` 会打印一个前导字符串的原因，这些额外的上下文信息也许会填满系统日志。

在卸载可分区模块时，驱动程序应该对每对它所支持的主/次设备编号调用 `fsync_dev`，以便刷新所有的分区，当然，还应该释放所有相关的内存。`spull` 的清除函数定义如下：

```
for (i = 0; i < (spull_devs << SPULL_SHIFT); i++)
    fsync_dev(MKDEV(spull_major, i)); /* flush the devices */
blk_cleanup_queue(BLK_DEFAULT_QUEUE(major));
read_ahead[major] = 0;
kfree(blk_size[major]); /* which is gendisk->sizes as well */
blk_size[major] = NULL;
kfree(spull_gendisk.part);
kfree(blksize_size[major]);
blksize_size[major] = NULL;
```

还需要从全局链表中删除 `gendisk` 结构。因为没有提供函数来完成该工作，因此需要手工完成：

```
for (gdp = &gendisk_head; *gdp; gdp = &((*gdp)->next))
    if (*gdp == &spull_gendisk) {
        *gdp = (*gdp)->next;
        break;
    }
```

注意没有和 `register_disk` 函数相对应的 `unregister_disk` 函数。`register_disk` 得到的所有结果保存在驱动程序自己的数组中，所以在卸载阶段没有任何清除工作需要完成。

12.8.3 使用 `initrd` 完成分区检测

如果我们想从某个设备上挂装根文件系统，而该设备的驱动程序只以模块形式存在，这时，我们就必须使用现代 Linux 内核提供的 `initrd` 设施。我们不会在这里介绍 `initrd`，所以，这个小节是针对了解 `initrd`，并且想知道它是如何影响块驱动程序的读者的。`initrd` 的详细信息可在内核源代码的 `Documentation/initrd.txt` 中找到。

当我们使用 `initrd` 引导内核时，它会在挂装实际的根文件系统之前建立一个临时的运行环境。通常，我们从用作临时根文件系统的 RAM 磁盘上装载模块。

因为 `initrd` 过程在所有引导阶段的初始化完成之后（但在挂装实际的根文件系统之前）运行，因此，在装载一个通常的模块与装载一个存在于 `initrd` RAM 磁盘上的模块之间，没有任何的区别。如果能够正确装载并以模块的方式使用某个驱动程序，所有支持 `initrd` 的 Linux 发行版就会将该驱动程序包含在安装磁盘中，而不需要我们去 hack 内核源代码。

12.8.4 `spull` 的设备方法

我们已经看到如何初始化可分区设备，但还不知道如何访问分区中的数据。为此，我们需要使用由 `register_disk` 保存在 `gendisk->part` 数组中的分区信息。该数组由 `hd_struct` 结构组成，并由次设备号索引。`hd_struct` 有两个值得注意的成员：`start_sect` 告诉我们给定分区在该磁盘上的起始位置，而 `nr_sects` 给出了该分区的大小。

这里我们将描述 `spull` 如何使用这些信息。下面的代码仅仅包含了 `spull` 不同于 `sbul` 的那些代码，因为大部分代码其实是一样的。

首先，`open` 和 `close` 保持每个设备的使用计数。因为使用计数是针对物理设备（单元）的，因此，下面的声明和赋值用于 `dev` 变量：

```
Spull_Dev *dev = spull_devices + DEVICE_NR(inode->i_rdev);
```

这里使用的 `DEVICE_NR` 宏是必须在包含 `<linux/blk.h>` 之前声明的宏之一，它定义了物理的设备编号，而不需要考虑正在使用哪个分区。

尽管几乎每个设备方法都可以将物理设备作为一个整体而处理，但 `ioctl` 需要访问每个分区特有的信息。例如，当 `mkfs` 调用 `ioctl` 检索要建立文件系统的设备大小时，它应该告诉 `mkfs` 对应分区的大小，而不是整个设备的大小。下面的代码说明了 `ioctl` 的 `BLKGETSIZE` 命令，如何受到每设备一个次设备号到多个次设备号这一改变的影响的。读者可能会想到，`spull_gendisk->part` 将用来获得分区大小。

```
case BLKGETSIZE:
    /* Return the device size, expressed in sectors */
    err = ! access_ok (VERIFY_WRITE, arg, sizeof(long));
    if (err) return -EFAULT;
    size = spull_gendisk.part[MINOR(inode->i_rdev)].nr_sects;
    if (copy_to_user((long *) arg, &size, sizeof (long)))
        return -EFAULT;
    return 0;
```

另外一个类似的 `ioctl` 命令是 `BLKRRPART`。对可分区设备来讲，重新读取分区表非常有意义，并且等价于在磁盘发生变化时的重生成（`revalidate`）操作：

```
case BLKRRPART: /* re-read partition table */
    return spull_revalidate(inode->i_rdev);
```

然而，`sbul` 和 `spull` 之间的最大不同在于 `request` 函数。在 `spull` 中，`request` 函数要使用分区信息以便从不同的次设备中传输数据。传输的定位，只需在请求所提供的扇区上加上分区的起始扇区，分区的大小信息也可用来确保请求发生在分区内部。在上述工作完成之后，其余的实现和 `sbul` 是一样的。

下面是 `spull_request` 中的相关代码行：

```
ptr = device->data +
    (spull_partitions[minor].start_sect + req->sector)*SPULL_HARDSECT;
size = req->current_nr_sectors*SPULL_HARDSECT;
/*
 * Make sure that the transfer fits within the device.
 */
if (req->sector + req->current_nr_sectors >
    spull_partitions[minor].nr_sects) {
    static int count = 0;
    if (count++ < 5)
        printk(KERN_WARNING "spull: request past end of partition\n");
    return 0;
}
```

扇区数乘以硬件的扇区大小（`spull` 中，该数值是硬编码的）可获得分区以字节计的大小。

12.9 中断驱动的块驱动程序

在一个驱动程序控制真正的硬件设备时，其操作通常是由中断驱动的。使用中断，可以在执行 I/O 操作过程中释放处理器，从而帮助提高系统性能。为了 I/O 能够以中断驱动的方式工作，所控制的设备必须能够异步传输数据并产生中断。

如果驱动程序是中断驱动的，`request` 函数应该提交一次数据传输并立即返回，而无需调用 `end_request`。但是，在没有调用 `end_request`（或其组成部分）之前，不会认为请求已经完成。因此，在设备告诉驱动程序已完成数据传输时，顶半或底半中断处理程序需要调用 `end_request`。

`sbulk` 和 `spull` 在不使用系统微处理器的情况下，都无法传输数据，但是，如果用户在装载 `spull` 的时候指定 `irq=1` 选项，则 `spull` 可以模拟中断驱动的操作。当 `riq` 为非零值时，驱动程序使用内核定时器来延迟当前请求的满足，延迟的长度就是 `irq` 的值：其值越大，延迟越长。

块的传输始终在内核调用驱动程序的 `request` 函数时开始。中断驱动设备的 `request` 函数指示硬件执行传输，然后返回，而不会等待传输的完成。`spull` 的 `request` 函数执行通常的错误检查，然后调用 `spull_transfer` 传输数据（相当于驱动程序指示实际的硬件执行异步传输），然后，`spull` 延迟请求完成确认，直到发生中断的那一刻：

```
void spull_irqdriven_request(request_queue_t *q)
{
    Spull_Dev *device;
    int status;
    long flags;

    /* If we are already processing requests, don't do any more now. */
    if (spull_busy)
        return;

    while(1) {
        INIT_REQUEST; /* returns when queue is empty */

        /* Which "device" are we using? */
        device = spull_locate_device (CURRENT);
        if (device == NULL) {
            end_request(0);
            continue;
        }
        spin_lock_irqsave(&device->lock, flags);

        /* Perform the transfer and clean up. */
        status = spull_transfer(device, CURRENT);
        spin_unlock_irqrestore(&device->lock, flags);
        /* ... and wait for the timer to expire -- no end_request(1) */
        spull_timer.expires = jiffies + spull_irq;
        add_timer(&spull_timer);
        spull_busy = 1;
        return;
    }
}
```

在设备处理当前请求时，还可以累积新的请求。因为在这种情况下，几乎总会发生重入调用，因此，`request` 函数设置 `spull_busy` 标志，以确保给定时间内只发生一次传输。因为整个函数在拥有 `io_request_lock` 锁（内核在调用 `request` 函数前获取该锁）的情况下运行，因此无需对该忙标志使用测试并设置操作。否则，为了避免竞态的发生，我们必须使用 `atomic_t` 类型的变量，而不是

int 变量。

中断处理程序要执行许多任务。首先，它必须检查未完成传输的状态，并清除该请求。然后，如果还有其它需要处理的请求，中断处理程序就要负责获得下一个已启动的请求。为了避免代码的重复，处理程序通常会调用 `request` 函数来启动下一个传输。需要注意的是，`request` 函数希望调用者拥有 `io_request_lock` 锁，因此，中断处理程序必须获得该锁。当然，`end_request` 函数也需要获得该锁。

在我们的示例模块中，中断处理程序的角色由定时器到期时所调用的函数担当，该函数调用 `end_request` 并调用 `request` 函数安排下一个数据传输。在这段简单的代码中，`spull` 的中断处理程序在“中断”期间执行其所有的工作，一个实际的驱动程序几乎肯定会推迟这些工作，并在任务队列或者 `tasklet` 中执行。

```
/* this is invoked when the timer expires */
void spull_interrupt(unsigned long unused)
{
    unsigned long flags

    spin_lock_irqsave(&io_request_lock, flags);
    end_request(1); /* This request is done - we always succeed */

    spull_busy = 0; /* We have io_request_lock, no request conflict */
    if (!QUEUE_EMPTY) /* more of them? */
        spull_irqdriven_request(NULL); /* Start the next transfer */
    spin_unlock_irqrestore(&io_request_lock, flags);
}
```

如果读者要让 `spull` 模块以中断驱动方式运行，几乎不可能注意到我们添加的延迟。该设备几乎和先前的一样快，因为缓冲区缓存避免了内存和设备之间的大多数数据传输。如果读者想感受到慢设备的行为，则可以在装载 `spull` 时为 `irq=` 指定一个较大的值。

12.10 向后兼容性

块设备层已经发生了许多改变，大部分变化发生在 2.2 和 2.4 稳定版本之间。这一小节将总结前面版本的不同之处。读者可以阅读示例源代码中可运行在 2.0、2.2 和 2.4 上的驱动程序，这样能看到移植性是如何处理的。

Linux 2.2 中不存在 `block_device_operations` 结构，相反，块驱动程序使用的是和字符驱动程序一样的 `file_operations` 结构，`check_media_change` 和 `revalidate` 也是该结构的一部分。内核同时提供了一组一般性函数，包括 `block_read`、`block_write` 和 `block_fsync`，大部分驱动程序可以在其 `file_operations` 结构中使用这些函数。2.2 或 2.0 `file_operations` 结构的典型初始化代码如下：

```
struct file_operations sbull_bdops = {
    read:      block_read,
    write:     block_write,
    ioctl:     sbull_ioctl,
    open:      sbull_open,
    release:   sbull_release,
    fsync:     block_fsync,
    check_media_change: sbull_check_change,
```

```
revalidate: sbull_revalidate
};
```

需要注意的是，块驱动程序也一样经历了 2.0 和 2.2 版本之间的 `file_operations` 原型变化，这和字符驱动程序是一样的。

在 2.2 及其先前内核中，`request` 函数保存在 `blk_dev` 全局数组中，因此，初始化时需要下面代码行：

```
blk_dev[major].request_fn = sbull_request;
```

因为该方法仅仅允许每个主设备号拥有一个队列，因此，2.4 内核中的多队列能力在先前版本中并不存在。因为只有一个队列，`request` 函数不需要将队列作为一个参数，所以该函数没有任何参数，它的原型如下：

```
void (*request) (void);
```

同时，所有的队列都拥有活动头，因此 `blk_queue_headactive` 也不存在。

在 2.2 及其先前版本中没有 `blk_ioctl` 函数。但是，有个称为 `RO_IOCTL` 的宏可插入 `switch` 语句来实现 `BLKROSET` 和 `BLKROGET`。示例源代码中的 `sysdep.h` 包含了一个使用 `RO_IOCTL` 的 `blk_ioctl` 实现，并且实现了其它一些标准的 `ioctl` 命令：

```
#ifndef RO_IOCTL
static inline int blk_ioctl(kdev_t dev, unsigned int cmd,
                           unsigned long arg)
{
    int err;

    switch (cmd) {
        case BLKRGGET: /* return the read-ahead value */
            if (!arg) return -EINVAL;
            err = ! access_ok(VERIFY_WRITE, arg, sizeof(long));
            if (err) return -EFAULT;
            PUT_USER(read_ahead[MAJOR(dev)], (long *) arg);
            return 0;

        case BLKRASET: /* set the read-ahead value */
            if (!capable(CAP_SYS_ADMIN)) return -EACCES;
            if (arg > 0xff) return -EINVAL; /* limit it */
            read_ahead[MAJOR(dev)] = arg;
            return 0;

        case BLKFLSBUF: /* flush */
            if (! capable(CAP_SYS_ADMIN)) return -EACCES; /* only root */
            fsync_dev(dev);
            invalidate_buffers(dev);
            return 0;

        RO_IOCTL(dev, arg);
    }
    return -ENOTTY;
}
#endif /* RO_IOCTL */
```

`BLKFRAGET`、`BLKFRASET`、`BLKSECTGET`、`BLKSECTSET`、`BLKELVGET` 和 `BLKELVSET` 命

令是 Linux 2.2 添加的，BLKPG 是在 2.4 中添加的。

Linux 2.0 中没有 `max_readahead` 数组，而是有一个 `max_segments` 数组，并在 Linux 2.0 和 2.2 中使用该数组，但设备驱动程序通常不需要设置这个数组。

最后，`register_disk` 直到在 Linux 2.4 中才出现。前面版本中有一个称为 `resetup_one_dev` 的函数，可完成类似的功能：

```
resetup_one_dev(struct gendisk *gd, int drive);
```

`sysdep.h` 利用下面的代码来模拟 `register_disk` 函数：

```
static inline void register_disk(struct gendisk *gdev, kdev_t dev,
                                unsigned minors, struct file_operations *ops, long size)
{
    if (! gdev)
        return;
    resetup_one_dev(gdev, MINOR(dev) >> gdev->minor_shift);
}
```

当然，因为 Linux 2.0 中不存在任何类型的 SMP 支持，所以没有 `io_request_lock`，也不需要为 I/O 请求队列的并行访问而担心。

最后还有一点需要提醒：尽管还没有人知道 2.5 开发系列版本中会发生什么，但块设备处理出现一次大的整修则是肯定的。许多人不太喜欢这个层的设计，因此有许多压力迫使内核开发人员重新编写块设备处理层。

12.11 快速参考

这里将总结编写块驱动程序时要用到的最重要的函数和宏，但是，为了节省篇幅，我们并不会列出 `struct request`、`struct buffer_head` 以及 `struct genhd` 的成员，而且还略去了预定义的 `ioctl` 命令。

```
#include <linux/fs.h>
int register_blkdev(unsigned int major, const char *name, struct block_device_operations
    *bdops);
int unregister_blkdev(unsigned int major, const char *name);
```

这些函数负责设备注册（在模块的初始化函数中）和删除设备（在模块的清除函数中）。

```
#include <linux/blkdev.h>
blk_init_queue(request_queue_t *queue, request_fn_proc *request);
blk_cleanup_queue(request_queue_t *queue);
```

第一个函数初始化队列并建立 `request` 函数，第二个函数在清除阶段使用。

```
BLK_DEFAULT_QUEUE(major)
```

整个宏返回给定主设备号的默认 I/O 请求队列。

```
struct blk_dev_struct blk_dev[MAX_BLKDEV];
```

该数组由内核用来检索给定请求的适当队列。

```
int read_ahead[];
```

```
int max_readahead[];
```

read_ahead 包含每个主设备号的块级预读值。对硬盘这样的设备，取值为 8 是比较合理的；对比较慢的介质，该值应该取得较大。**max_readahead** 包含了每个主设备号和次设备号的文件系统级预读值，通常无需改变系统所设置的默认值。

```
int max_sectors[];
```

该数组由主设备号和次设备号索引，含有可合并到单个 I/O 请求中的最大扇区数。

```
int blksize_size[];
int blk_size[];
int hardsect_size[];
```

这些二维数组由主设备号和次设备号索引。驱动程序负责分配和释放矩阵中与其主设备号相关联的行。这些数组分别代表设备块以字节计的大小（通常为 1 KB）、每个次设备以千字节计的大小（不是块），以及硬件扇区以字节计的大小。

```
MAJOR_NR
DEVICE_NAME
DEVICE_NR(kdev_t device)
DEVICE_INTR
#include <linux/blk.h>
```

驱动程序必须在包含 `<linux/blk.h>` 之前定义这些宏，因为该头文件要使用这些宏。其中，**MAJOR_NR** 是设备的主设备号，**DEVICE_NAME** 是错误消息中使用的设备名称，**DEVICE_NR** 返回某设备号对应的“物理”设备的编号（对非可分区设备来讲就是次设备号），而 **DEVICE_INTR** 是一个较少使用的符号，它指向设备的底半中断处理程序。

```
spinlock_t io_request_lock;
```

操作 I/O 请求队列时必须获得的自旋锁。

```
struct request *CURRENT;
```

在使用默认队列时，这个宏指向当前请求。**request** 结构描述了要传输的数据块，并在驱动程序的 **request** 函数中使用。

```
INIT_REQUEST;
end_request(int status);
```

INIT_REQUEST 检查队列中的下一个请求，并在没有其它请求需要处理时返回。**end_request** 在块请求完成时调用。

```
spinlock_t io_request_lock;
```

在操作请求队列时，必须获得这个 I/O 请求锁。（译者：和前面重复）

```
struct request *blkdev_entry_next_request(struct list_head *head);
struct request *blkdev_next_request(struct request *req);
struct request *blkdev_prev_request(struct request *req);
blkdev_dequeue_request(struct request *req);
blkdev_release_request(struct request *req);
```

用来处理 I/O 请求队列的各种函数。

```
blk_queue_headactive(request_queue_t *queue, int active);
```

该函数指出队列中的第一个请求是否正在由驱动程序处理，即活动头。

```
void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);
```

该函数指定使用某个函数超越内核而直接处理块 I/O 请求。

```
end_that_request_first(struct request *req, int status, char *name);
end_that_request_last(struct request *req);
```

上述函数用于块 I/O 请求完成的阶段。`end_that_request_last` 在请求中的所有缓冲区被处理后调用，也就是当 `end_that_request_first` 返回 0 时。

```
bh->b_end_io(struct buffer_head *bh, int status);
```

通知内核给定缓冲区上的 I/O 操作已结束。

```
int blk_ioctl(kdev_t dev, unsigned int cmd, unsigned long arg);
```

实现大部分标准块设备 `ioctl` 命令的辅助函数。

```
int check_disk_change(kdev_t dev);
```

该函数检查给定设备上是否发生介质变化，在检测到变化时，将调用驱动程序的 `revalidation` 方法。

```
#include<linux/gendisk.h>
struct gendisk;
struct gendisk *gendisk_head;
```

一般性硬盘允许 Linux 轻松支持可分区设备。`gendisk` 结构描述一个一般性磁盘，`gendisk_head` 是 `gendisk` 结构形成的链表，用来描述系统中所有的一般性磁盘。

```
void register_disk(struct gendisk *gd, int drive, unsigned minors, struct
    block_device_operations *ops, long size);
```

该函数扫描磁盘的分区表并重写 `genhd->part` 以反映出新的分区情况。

第 13 章 mmap 和 DMA



本章将深入探讨 Linux 内存管理部分，并强调了对设备驱动程序编写者非常有帮助的技术重点。这一章内容属于高级主题，不需要所有人都掌握它，虽然如此，很多任务只能通过更深入地研究内存管理子系统而做到，同时本章也能帮助读者了解内核重要组成部分的工作方式。

本章内容分为三节。第一节讲述了 mmap 系统调用的实现，mmap 允许直接将设备内存映射到用户进程的地址空间中。然后我们讨论内核 kiobuf 机制，它能提供从内核空间对用户内存的直接访问，kiobuf 系统能用来为某些种类的设备实现“裸 (raw) I/O”。最后一节讲述直接内存访问 (DMA) I/O 操作，它本质上提供了外围设备直接访问系统内存的能力。

当然，所有的这些技术都需要先了解 Linux 的内存管理是如何工作的，所以我们从内存子系统来开始本章的讨论。

13.1 Linux 的内存管理

这一节不是描述操作系统中内存管理的理论，而是关注于该理论在 Linux 实现中的主要特点。尽管为了实现 mmap，你无需成为 Linux 虚拟内存方面的专家，但是了解它们工作的基本概况还有很有帮助的。然后我们将用比较长的篇幅描述 Linux 用于内存管理的数据结构。一旦具备了必要的背景知识，我们就可以利用这些结构来实现 mmap。

13.1.1 地址类型

Linux 是一个使用虚拟内存的系统，这意味着用户程序看到的地址不是直接对应于硬件使用的物理地址。虚拟内存提出了一个间接的层，这对许多事情都是有利的。如果有虚拟内存，运行在系统上的程序就可以分配到比可用物理内存更多的内存。甚至一个单独的进程都可以拥有比系统的物理内存更大的虚拟地址空间，虚拟内存也能在进程地址空间上使用很多技巧，包括映射设备的内存。

迄今，我们已经讨论了虚拟地址和物理地址，但是很多细节被掩盖而没有提及。Linux 系统使用几种类型的地址，每种都有自己的语义。不幸地是，内核代码中关于哪种类型的地址应该在何种情形下使用的问题一直不是十分清晰，所以程序员必需谨慎地使用。

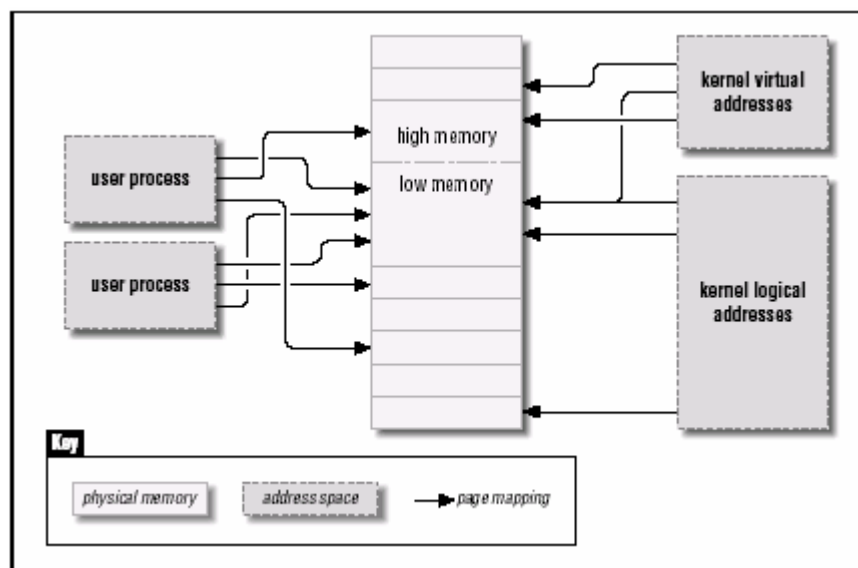


图 13-1: Linux 中使用的地址类型

下面列出了 Linux 用到的地址类型。图 13-1 描述了这些地址类型和物理内存之间的关系。

用户虚拟地址

该地址是用户空间的程序所能看到的常规地址。根据低层硬件体系结构的不同,用户地址可以是 32 位或者 64 位长,并且每个进程拥有自己独立的虚拟地址空间。

物理地址

该地址在处理器和系统内存之间使用。物理地址也是 32 或者 64 位长,在某些情况下,32 位系统也可以使用 64 位的物理地址。

总线地址

该地址在外设总线和内存之间使用。通常情况下,该地址和处理器所使用的物理地址是一样的,但并不总是这样。显然,总线地址非常依赖于体系结构。

内核逻辑地址

内核逻辑地址组成了常规的内核地址空间,这些地址映射了大部分乃至所有的主内存,并被视为物理内存使用。在大多数的体系结构中,逻辑地址及其所关联的物理地址之间的区别,仅仅在于一个常数的偏移量。逻辑地址使用硬件特有的指针大小,所以,在配置有大量内存的 32 位系统上,仅通过逻辑地址可能无法寻址所有的物理内存。在内核中,逻辑地址通常保存在 `unsigned long` 或者 `void *` 这样的变量中。由 `kmalloc` 返回的内存就是逻辑地址。

内核虚拟地址

这种地址和逻辑地址之间的区别在于,前者不一定能够直接映射到物理地址。所有的逻辑地址可看成是内核虚拟地址;由函数 `vmalloc` 分配的内存具有虚拟地址,这种地址却不一定能直接映射到物理内存。本章后面要讲到的 `kmap` 函数也返回虚拟地址。虚拟地址通常保存在指针变量中。

如果我们拥有一个逻辑地址，可通过定义在 `<asm/page.h>` 中的宏 `__pa()` 返回与其关联的物理地址。我们也可以使用 `__va()` 宏将物理地址映射回逻辑地址，但只能用于低端内存页。

不同的内核函数要求不同类型的地址。如果已经存在有定义好的 C 数据类型来明确表达我们所要求的地址类型，代码将变得更为清晰可读，但事实并没有我们想象的那么好。通过本章的学习，我们将会了解到哪种情况下应该使用哪种地址类型。

13.1.2 高端与低端内存

逻辑地址和内核虚拟地址的区别在装配了大量内存的 32 位系统上比较突出。使用 32 位来表示地址，就可以寻址 4GB 大小的内存。到最近为止，在 32 位的系统上 Linux 一直被限制使用少于 4GB 的内存，这是由于设置虚拟地址空间的方式导致的。系统不能处理比设置的逻辑地址更多的内存，因为它需要为全部内存直接映射内核地址。

近来的开发工作已经将这个内存上的限制排除掉了，32 位系统现在能够在超过 4GB 的系统内存（当然需要假定处理器自己能够寻址这些内存）上很好的工作了。但是关于多少内存可以以逻辑地址的形式直接映射的限制还是保留下来了，只有内存的最低一部分（一直到 1 或 2GB，依赖于硬件和内核配置）有逻辑地址，剩下部分没有。高端内存应该需要 64 位物理地址，并且内核必须明确地设置映射的虚拟地址来操作高端内存。这样，就限制很多内核函数只能使用低端内存，高端内存常常是保留给用户空间的进程页。

术语“高端内存”可能造成混淆，特别是它在 PC 方面还具有其他的含义。所以，为了解释清楚，我们将在这里定义这些术语：

低端内存

代表存在于内核空间的逻辑地址的内存。几乎在每种系统上读者都可能遇到，所有的内存都是低端内存。

高端内存

那些不存在逻辑地址的内存，因为相对于能够用 32 位来寻址的内存，系统通常有更多的物理内存。

在 i386 系统上，低端内存和高端内存的之间的界限通常设置为 1GB。这个界限与最初的 PC 上老的 640KB 限制没有任何关系，相反，它是内核本身设置的限制，用于将 32 位地址空间分割为内核空间 and 用户空间。

我们将指出本章中我们遇到的高端内存的限制。

13.1.3 内存映射和页结构

历史上，内核在提到内存页时都是使用逻辑地址。另外的高端内存的支持方法已经暴露了明显的问题——逻辑地址不能用于高端内存。这样，处理内存的内核函数趋向于使用指向 `struct page` 的指针。这个数据结构用于保存物理内存的所有信息，系统上的每一个物理内存页都有一个 `struct page`，该结构的部分成员如下：

```
atomic_t count;
```

对该页的访问计数。当计数下降到零，该页就返还给空闲链表。

```
wait_queue_head_t wait;
```

等待这个页的进程链表。尽管进程能够在内核函数出于某种原因锁住该页时等待，但通常驱动程序不需要考虑等待页。

```
void *virtual;
```

如果页面被映射，该成员就是页的内核虚拟地址，否则就是 **NULL**。低端内存页总是被映射的，高端内存页通常不是。

```
unsigned long flags;
```

描述页状态的一组位标志。其中包括表明内存中的页已经锁住的 **PG_locked**，以及完全阻止内存管理系统处理该页的 **PG_reserved**。

struct page 中还有更多的信息，但它们只是技巧性很强的内存管理的一部分，与驱动程序编写者关系不大。

内核维护了一个或者更多由 **struct page** 项构成的数组，它们跟踪系统上所有的物理内存。在大多数系统上，只有一个叫做 **mem_map** 的数组。然而在某些系统上，情况更为复杂，非一致性内存访问（Nonuniform Memory Access, NUMA）系统与具有普遍的不连续的物理内存的系统，都可以有多于一个的内存映射数组，所以可移植代码无论如何都应该避免直接访问数组。令人高兴的是，只是使用 **struct page** 指针而不用关心它们是从哪里获得，通常很容易。

一些函数和宏可用来在 **struct page** 和虚拟地址之间进行转换：

```
struct page *virt_to_page(void *kaddr);
```

这个宏在头文件 **<asm/page.h>** 中定义，它接受一个内核逻辑地址，并返回与其关联的 **struct page** 指针。因为它需要一个逻辑地址，它对 **vmalloc** 返回的内存和高端内存无效。

```
void *page_address(struct page *page);
```

如果这个地址存在的话，返回该页的内核虚拟地址。对于高端内存，仅在该页已经被映射的情况下，其地址才存在。

```
#include <linux/highmem.h>
void *kmap(struct page *page);
void kunmap(struct page *page);
```

对于系统中任意一个页，**kmap** 都返回一个内核虚拟地址。对于低端内存页，它只是返回页的逻辑地址；对于高端内存页，**kmap** 建立一个特殊的映射，**kmap** 建立的映射应该总是使用 **kunmap** 来释放。有限数量的这种映射是有用的，所以最好不要长时间地持有它们。**kmap** 调用是附加式的，所以如果两个或者更多的函数对同一页面调用 **kmap** 也是正确的。注意，如果没有映射可用，**kmap** 会进入睡眠。

在本章后面我们研究样例代码时，我们将会看到这些函数的一些用法。

13.1.4 页表

当有一个程序对虚拟地址进行查询时，CPU 必须把该虚拟地址转换成物理地址，这样才能对物理内存进行访问。这可以通过把地址分割成位字段（bitfield）的方法来实现。每一个位字段是一个数组的索引，该数组被称为“页表（page table）”。通过这些位字段可以获得下一个页表的地址或保存虚拟地址的物理页的地址。

为了将虚拟地址映射为物理地址，Linux 内核对三级页表进行管理。这种多级管理的方式可使内存范围得到稀疏利用；现代系统将会把进程扩展到一个大范围的虚拟内存上。这种方法很有意义，因为它考虑了内存页处理的运行时灵活性。

需要指出，在只支持两级页表的硬件中，或使用其他不同方法把虚拟地址映射成物理地址的硬件中，都可以使用三级系统。在不依赖于处理器的实现中使用三级系统，可以使 Linux 能够同时支持两级和三级页表的处理器，而不必使用大量的 #ifdef 语句进行编码。当内核在两级处理器中运行时，这种保守的编码并不会导致额外的开销，因为编译器实际上已经对不使用的级进行了优化。

现在来看一下实现内存分页系统所使用的数据结构。下面的列表概述了在 Linux 中三级管理的实现，图表 13-2 对此进行了描述。

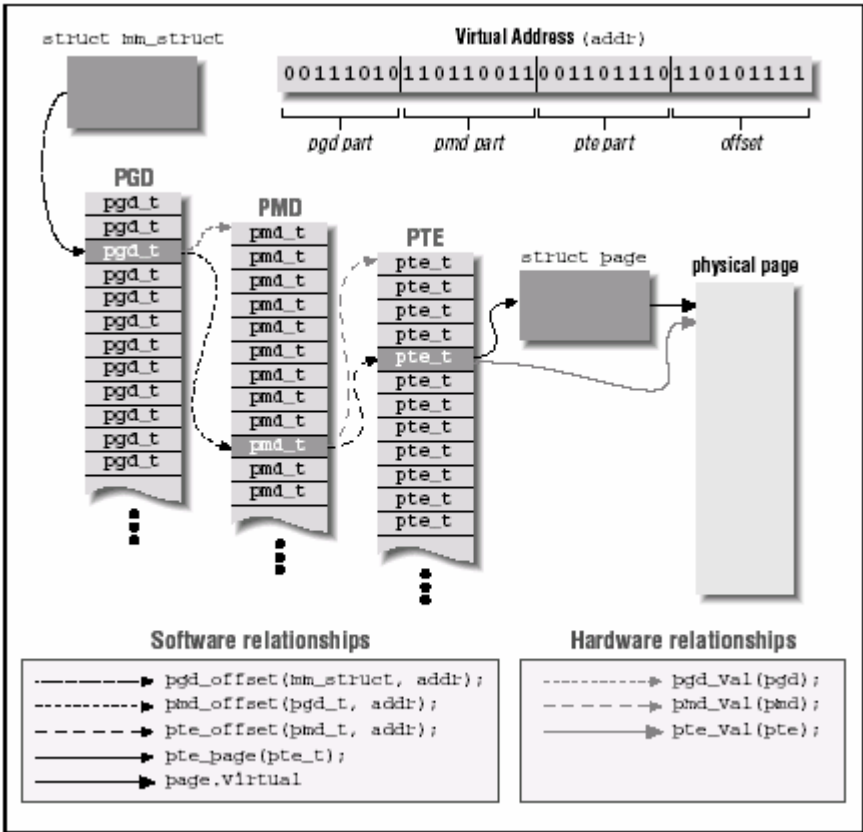


图 13-2: Linux 的三级页表

页目录 (PGD)

顶级页表。PGD 是由 `pgd_t` 项组成的数组，其中每一项指向一个二级页表。每一个进程有自己的页目录，内核空间也有一个自己的页目录。可以把页目录看作一个页对齐的 `pgd_t` 数组。

中级页目录 (PMD)

第二级表，PMD 是页对齐的 `pmd_t` 数组。一个 `pmd_t` 项是指向第三级页表的一个指针，两级处理器没有物理的 PMD，它们将自己的 PMD 作为一个单元素的数组，其值是 PMD 本身。在下面的部分将会看到在 C 语言中是如何解决这个问题，以及编译器怎样对该级进行优化。

页表

一个页对齐项的数组，每一项称为一个页表项，内核为这些项使用 `pte_t` 类型。一个 `pte_t` 包含了数据页的物理地址。

这里所介绍的类型在头文件 `<asm/page.h>` 中定义，每一个有关页处理的源文件必须包含它。

在正常的程序执行过程中，内核不必对页表进行查询，因为这可以由硬件来完成。但是，内核必须合理安排工作，这样硬件才能完成自己的工作。一旦处理器报告了一个页故障，也就是说，处理器所需要的与虚拟地址关联的页当前不在内存中，内核就必须建立页表，并且对它们进行查询。在实现 `mmap` 时，设备驱动程序必须建立页表并处理页故障。

值得注意的是，软件内存管理是如何利用 CPU 本身所使用的同一页表的。如果 CPU 不实现页表，这种区别只会隐藏在最底层的体系结构特有的代码中。所以，在 Linux 内存管理中，我们总是讨论三级页表，而忽略页表和硬件之间的关系。不使用页表的 CPU 家族的一个例子就是 PowerPC。PowerPC 设计者通过实现一个哈希算法，可以将虚拟地址映射为一个一级页表。在访问一个已位于内存，但其物理地址已经从 CPU 的高速缓冲存储器中去除的页时，CPU 只需读一次内存即可，这与在一个多级页表要进行两次或三次的访问是相反的。与多级表相似，在将虚拟地址映射为物理地址的过程中，哈希算法减少了所需要使用的内存数量。

不考虑 CPU 所使用的机制，Linux 软件实现是建立在三级页表基础上的，可用使用下面的符号来访问它们。必须包含头文件 `<asm/page.h>` 和 `<asm/pgtable.h>`，以便对所有的符号进行访问。

```
PTRS_PER_PGD
PTRS_PER_PMD
PTRS_PER_PTE
```

代表每一个表的大小。两级处理器将 `PTRS_PER_PMD` 设置为 1，以避免对中级页目录进行处理。

```
unsigned pgd_val(pgd_t pgd)
unsigned pmd_val(pmd_t pmd)
unsigned pte_val(pte_t pte)
```

这三个宏用来从类型化数据项中获得 `unsigned` 值。由于依赖于底层的体系结构和内核配置选项，实际使用的类型也是多样化的；通常可以是 `unsigned long`，也可以是支持高端内存的 32 位处理器中的 `unsigned long long`。SPARC64 处理器使用 `unsigned int`。这些宏有助于在源代码中使用严格的数据类型，而不会引入计算开销。

```
pgd_t * pgd_offset(struct mm_struct * mm, unsigned long address)
pmd_t * pmd_offset(pgd_t * dir, unsigned long address)
```

```
pte_t * pte_offset(pmd_t * dir, unsigned long address)
```

这些内联函数*用来获得与给定 `address` 相关的 `pgd`、`pmd` 和 `pte` 项。页表查询以指向 `struct mm_struct` 的指针为起点，与当前进程的内存映射相关联的指针是 `current->mm`，而指向内核空间的指针使用 `&init_mm` 来描述。两级处理器将 `pmd_offset(dir,add)` 定义为 `(pmd_t *)dir`，这样可以将 `pmd` 与 `pgd` 合并起来。扫描页表的函数总是声明为内联的，这样，编译器就能针对 `pmd` 查询进行优化。*

```
struct page *pte_page(pte_t pte)
```

该函数会为页表项中的页返回一个指向 `struct page` 项的指针。处理页表的代码通常使用 `pte_page`，而不是 `pte_val`，这是因为 `pte_page` 处理依赖于处理器的页表项格式，并且返回 `struct page` 指针，而这通常是必须的。

```
pte_present(pte_t pte)
```

该宏返回一个布尔值，该布尔值用来表明数据页当前是否在内存中。在访问 `pte` 低位（这些位被 `pte_page` 丢弃）的几个函数当中，这是最常用的函数。当然这些页也许并不存在，如果内核已经把它们交换到磁盘，或者它们根本没有被装载的话。然而，页表本身总是出现在当前的 Linux 实现中。把页表保存在内存中简化了内核代码，因为 `pgd_offset` 和其它相关项从来都不会失效。另一方面，一个“驻留存储大小”为零的进程也会在 RAM 中保存它的页表，这样，虽然浪费了一些内存，但总比用在其它方面要好些。

系统中的每个进程都有一个 `struct mm_struct` 结构体，在结构体中包含了进程的页表和许多其它的大量信息。其中也包含一个叫做 `page_table_lock` 的自旋锁，在移动或修改页表时，应该持有该自旋锁。

只了解这些函数还不足以让读者精通 Linux 的内存管理算法。真正的内存管理要复杂的多，而且必须处理其它的复杂情况，比如说高速缓存的一致性。但是前面列出的函数足以给读者一个关于页管理实现的初步印象。作为要经常与页表打交道的设备驱动程序作者来说，这些是必须要了解的。从内核源代码的 `include/asm` 和 `include/mm` 子树中可以获得更多的信息。

13.1.5 虚拟内存区域

尽管内存分页位于内存管理的最底层，在能有效的使用计算机资源之前，还是需要了解更多的东西。内核需要一个更高级的机制来处理进程自己的内存，在 Linux 中，这种机制是通过虚拟内存区域的方法来实现的，它们被称为区域或 VMA。

一个区域是在进程虚拟内存中的一个同构区间，一个具有相同许可标志的地址的连续范围。它和“段”的概念有点相当，尽管将后者描述为“具有自有属性的内存对象”更为贴切些。进程的内存映射由下面几个区域构成：

- 程序的执行代码区域（通常称作 `text` 段）。
- 每种类型的数据对应一个区域，其中包括初始化数据（在执行之初已经明确赋值的数据）、未初始化的数据（`BSS`）*、程序栈。

* 在 32 位的 SPARC 处理器中，这些函数不是内联函数，而是真正的外部函数，但这些函数不能输出到模块化的代码中。所以不能在 SPARC 上运行的模块中使用这些函数，但是通常也不需要这样做。

* 使用 `BSS` 这个名字有其历史原因，它源于以前的一个名为“Block started by symbol”的汇编运算符。可执行文件的 `BSS` 段并不会存储在磁盘中，而是由内核将零页映射到 `BSS` 地址区间中。

■ 每一个有效的内存映射区域。

一个进程的内存区域可以从 `/proc/pid/maps` 中看到（这里的 `pid` 也可以用进程的 ID 来替换）。`/proc/self` 是 `/proc/pid` 的特殊情况，因为它总是指向当前的进程。下面是一组内存映射的例子，在#号后面添加了一些短注释：

```
morgana.root# cat /proc/1/maps # look at init
08048000-0804e000 r-xp 00000000 08:01 51297 /sbin/init # text
0804e000-08050000 rw-p 00005000 08:01 51297 /sbin/init # data
08050000-08054000 rwxp 00000000 00:00 0 # zero-mapped bss
40000000-40013000 r-xp 00000000 08:01 39003 /lib/ld-2.1.3.so # text
40013000-40014000 rw-p 00012000 08:01 39003 /lib/ld-2.1.3.so # data
40014000-40015000 rw-p 00000000 00:00 0 # bss for ld.so
4001b000-40108000 r-xp 00000000 08:01 39006 /lib/libc-2.1.3.so # text
40108000-4010c000 rw-p 000ec000 08:01 39006 /lib/libc-2.1.3.so # data
4010c000-40110000 rw-p 00000000 00:00 0 # bss for libc.so
bffffe000-c0000000 rwxp fffff000 00:00 0 # zero-mapped stack

morgana.root# rsh wolf head /proc/self/maps ##### alpha-axp: static ecoff
000000011fffe000-0000000120000000 rwxp 0000000000000000 00:00 0 # stack
0000000120000000-0000000120014000 r-xp 0000000000000000 08:03 2844 # text
0000000140000000-0000000140002000 rwxp 0000000000014000 08:03 2844 # data
0000000140002000-0000000140008000 rwxp 0000000000000000 00:00 0 # bss
```

每行中的字段如下：

```
start—end perm offset major minor inode image
```

`/proc/*/maps`（映像名字本身除外）中的每一个字段都与 `struct vm_area_struct` 中的一个成员相对应，下面用一个列表对每个字段进行描述。

```
start
end
```

该内存区域的起始和结束虚拟地址。

```
perm
```

内存区域的读、写和执行许可的位掩码。该字段描述了允许进程对属于该区域的页所能进行的操作。字段中的最后一个字符既可以是 **p**（代表私有），也可以是 **s**（代表共享）。

```
offset
```

这里是内存区域在被映射文件中的起始位置。零偏移量表示内存区域的第一页与文件的第一页相对应。

```
major
minor
```

对应于被映射文件所在设备的主设备号和次设备号。主设备号和次设备号是由用户打开的设备特殊文件所在的磁盘分区来决定的，而不是由设备本身所决定，这一点很容易混淆。

```
inode
```

被映射文件的索引节点号。

```
image
```

已被映射的文件（通常是一个可执行映像）的名字。

实现 `mmap` 方法的驱动程序，要在映射其设备的进程的地址空间中填充一个 `VMA` 结构体。所以，驱动程序作者应该对 `VMA` 有一点了解，这样才能使用它们。

下面介绍一下在结构 `vm_area_struct`（在头文件 `<linux/mm.h>` 中定义）中最重要的几个成员。这些成员可能会在 `mmap` 实现中被设备驱动程序用到。需要指出，内核维护 `VMA` 链表和树以便优化对区域的查询，`vm_area_struct` 的几个成员则用来维护这种组织形式。因此，驱动程序不能随便地生成 `VMA`，否则结构体会遭到破坏。`VMA` 的主要成员如下（注意这些成员和刚看到的 `/proc` 输出之间的类似性）：

```
unsigned long vm_start
unsigned long vm_end
```

`VMA` 所覆盖的虚拟地址区间。这些成员是在 `/proc/*/maps` 中最先显示的两个字段。

```
struct file *vm_file;
```

指向与该区域（如果有的话）相关联的 `struct file` 结构体的一个指针。

```
unsigned long vm_pgoff
```

文件或页中的区域偏移量。在映射一个文件或设备时，这是在该区域中被映射文件的第一页的位置。

```
unsigned long vm_flags
```

一组描述该区域的标记。对设备驱动程序作者来说，最有意思的标志是 `VM_IO` 和 `VM_RESERVED`。`VM_IO` 将一个 `VMA` 标记为一个内存映射的 I/O 区域。`VM_IO` 会阻止系统将该区域包含在进程的 core dump 中。`VM_RESERVED` 会告诉内存管理系统不要试图把该 `VMA` 交换出去，在大多数的设备映射中都应该对它进行设置。

```
struct vm_operations_struct *vm_ops
```

内核可能调用的一组函数，用来对内存区域进行操作。它的存在说明内存区域是一个内核“对象”，就像本书中一直使用的 `structure file` 一样。

```
void *vm_private_data
```

可以被驱动程序用来存储自身信息的成员。

与结构 `vm_area_struct` 一样，`vm_operations_struct` 也是在头文件 `<linux/mm.h>` 中定义的，它包含了下面所列出的操作，这些操作是处理进程内存所必须的。这里按声明顺序将它们列在下面。本章后面将实现其中一些函数，并对它们进行详细的描述。

```
void (*open)(struct vm_area_struct *vma)
```

内核调用 `open` 方法以允许子系统实现 `VMA` 对区域的初始化，调整引用计数等等。在每次产生一个 `VMA` 的新引用（例如进程分叉）时，该方法就被调用。一个例外就是用 `mmap` 首次生成 `VMA` 的时候，在这种情况下，会调用驱动程序的 `mmap` 方法。

```
void (*close)(struct vm_area_struct *vma)
```

当一个区域被销毁时，内核会调用 `close` 操作。需要注意的是并没有与 `VMA` 相关联的使用计数；区域由使用它的每个进程正确地打开和关闭。


```
void (*unmap)(struct vm_area_struct *vma, unsigned long addr, size_t len)
```

内核调用该方法来撤销一个区域的部分或全部映射。如果整个区域的映射被撤销，内核在 `vm_ops->unmap` 返回时调用 `vm_ops->close`。

```
void (*protect)(struct vm_area_struct *vma, unsigned long, size_t, unsigned int newprot);
```

这种方法的目的是改变内存区域的保护权限，但是当前并未被使用。页表负责内存保护，而内核则分别创建各个页表项。

```
int (*sync)(struct vm_area_struct *vma, unsigned long, size_t, unsigned int flags);
```

系统调用 `msync` 会调用该方法将一个脏的内存区域保存到存储介质中。如果返回值为 0，则表示该方法成功；如果为负，则表示有错误产生。

```
struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int write_access);
```

当一个进程试图访问属于另一个当前并不在内存中的有效 VMA 页时，`nopage` 方法（如果它被定义的话）就会被调用以处理相关区域。这个方法返回物理页的 `struct page` 指针，然后就可能从辅助存储器中将其读入。如果没有为该区域定义 `nopage` 方法，内核就会分配一个空页。第三个参数 `write_access` 被当作“非共享”：一个非零值意味着该页必须为当前进程所有，为零则表示共享是可能的。

```
struct page *(*wppage)(struct vm_area_struct *vma, unsigned long address, struct page *page);
```

该方法处理写保护的页故障，但是当前并未使用。内核不需要调用区域特有的回调函数，就可以处理向一个被保护页面上写入的企图。写保护故障用来实现写时复制。一个私有页可以在进程之间共享，直到其中一个进程对该页进行写操作为止。当这种情况发生时，该页被复制，进程会向自己的复制页上写入。如果整个区域被标志为只读，则会向进程发送 `-SIGSEGV` 信号，并且不执行任何写时复制操作。

```
int (*swapout)(struct page *page, struct file *file);
```

当内核选择一个页交换出去时，就会调用该方法。如果返回零值，则表示调用成功，而其他任何的返回值都表示出现错误。在出现错误的情况下，内核会向拥有该页的进程发送一个 `SIGBUS` 信号。对驱动程序来说，没有多少必要去实现 `swapout` 方法；设备映射并不是内核能写入磁盘的东西。

我们总结了 Linux 内存管理的数据结构的概要，现在我们可以继续讨论“`mmap`”系统调用的实现了。

13.2 mmap 设备操作

内存映射是现代 Unix 系统最有趣的特征之一。对于驱动程序来说，内存映射可以提供给用户程序直接访问设备内存的能力。

看一下 X Window 系统服务器的虚拟内存区域，就可以看到 `mmap` 用法的一个明显的例子：

```
cat /proc/731/maps
08048000-08327000 r-xp 00000000 08:01 55505 /usr/X11R6/bin/XF86_SVGA
08327000-08369000 rw-p 002de000 08:01 55505 /usr/X11R6/bin/XF86_SVGA
40015000-40019000 rw-s fe2fc000 08:01 10778 /dev/mem
40131000-40141000 rw-s 000a0000 08:01 10778 /dev/mem
40141000-40941000 rw-s f4000000 08:01 10778 /dev/mem
```

...

X 服务器的 VMA 的整个列表是很长的，但是这里我们对大部分的项都不感兴趣。然而，确实可以看到 `/dev/mem` 的三个独立的映射，它可以使我们对 X 服务器怎样与显示卡协同工作有一些了解。第一个映射显示了映射到 `fe2fc000` 的一个 16KB 区域，这个地址远远高于系统上最高的 RAM 地址，它是 PCI 外围设备（显示卡）的一段内存区域，它是该卡的控制区域。中间的映射位于 `a0000`，它是在 640KB ISA 空洞中的标准位置。最后的 `/dev/mem` 映射是位于 `f4000000` 位置的一个相当大的区域，而且是显示内存本身。这些区域也可以在 `/proc/iomem` 中看到：

```
000a0000-000bffff : Video RAM area
f4000000-f4ffffff : Matrox Graphics, Inc. MGA G200 AGP
fe2fc000-fe2fffff : Matrox Graphics, Inc. MGA G200 AGP
```

映射一个设备，意味着使用用户空间的一段地址关联到设备内存上。无论何时，只要程序在分配的地址范围内进行读取或者写入，实际上就是对设备的访问。在 X 服务器的例子中，使用 `mmap` 可以既快速又简单地访问显示卡的内存。对于象这样的性能要求比较严格的应用来说，直接访问能给我们提供很大不同。

正如读者所怀疑的，并不是所有的设备都能进行 `mmap` 抽象；例如，象串口设备和其它面向流的设备，就无法实现这种抽象。`mmap` 的另一个限制是映射都是以 `PAGE_SIZE` 为单位的。内核只能在页表一级上处理虚拟地址；因此，被映射的区域必须是 `PAGE_SIZE` 的整数倍，而且必须位于起始于 `PAGE_SIZE` 整数倍地址的物理内存内。如果区域的大小不是页大小的整数倍，内核可通过生成一个稍微大一些的区域来调节页面大小粒度。

这些限制对于驱动程序来说并不是很大的问题，因为程序访问设备的动作总是依赖于设备的，它需要知道如何使得被映射的内存区域有意义，所以 `PAGE_SIZE` 对齐不是一个问题。在 ISA 设备用于某些非 x86 平台时存在一个比较大的限制，因为它们对于 ISA 硬件的开发并不一样。例如，某些 Alpha 计算机将 ISA 内存看成是不能直接映射的、离散的 8 位、16 位或者 32 位项的集合。这种情况下，根本不能使用 `mmap`。不能直接将 ISA 地址映射到 Alpha 地址的原因，是由于两种系统间不兼容的数据传输规范导致的。早期的 Alpha 处理器只能进行 32 位和 64 位的内存访问，而 ISA 只能进行 8 位和 16 位的数据传输，并且没有透明地将一个协议映射到另一个之上的方法。

在能够使用 `mmap` 的情况下，使用 `mmap` 还有另外一个优势。例如，我们已经讨论了 X 服务器，它可以与显示内存进行大量的数据交换。相对于 `lseek/write` 实现来说，将图形显示映射到用户空间可以显著提高吞吐量，另一个典型的例子是受程序控制的 PCI 设备。大多数 PCI 外围设备都将它们自己的控制寄存器映射到内存地址上，而苛刻的应用程序可能更喜欢直接访问寄存器，而不是重复调用 `ioctl` 来完成它的工作。

`mmap` 方法是 `file_operations` 结构中的一员，并且在执行 `mmap` 系统调用时就会调用该方法。在调用实际方法之前，内核会完成很多工作，而且该方法的原型与其系统调用的原型具有很大区别。这与其它系统调用如 `ioctl` 和 `poll` 不同，在调用它们之前内核不需要做太多的工作。

系统调用声明如下（就像在 `mmap (2)` 手册页中描述的一样）：

```
mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)
```

另一方面，文件操作却声明为

```
int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

该方法中的参数 `filp` 与在第 3 章中介绍的一样，而 `vma` 包含了用于访问设备虚拟地址区间的信息。大部分工作已经由内核完成了；要实现 `mmap`，驱动程序只要为这一地址范围构造合适的页表，如果需要的话，用一个新的操作集替换 `vma->vm_ops`。

有两种建立页表的方法：使用 `remap_page_range` 函数可一次建立所有的页表，或者通过 `nopage VMA` 方法每次建立一个页表。这两种方法有它各自的优势，我们从“一次建立所有”的方法开始谈起，因为这个方法很简单。接下来我们会针对实际的实现增加复杂性。

13.2.1 使用 `remap_page_range`

构造用于映射一段物理地址的新页表的工作，是由 `remap_page_range` 完成的，它的原型如下：

```
int remap_page_range(unsigned long virt_add, unsigned long phys_add, unsigned long size,
pgprot_t prot);
```

函数返回的值通常是 0 或者一个负的错误码。让我们看看该函数参数的确切含义：

virt_add

重映射起始处的用户虚拟地址。函数为虚拟地址 `virt_add` 和 `virt_add + size` 之间的区间构造页表。

phys_add

虚拟地址所映射的物理地址。函数影响从 `phys_add` 到 `phys_add + size` 的物理地址。

size

被重映射的区域的大小，以字节为单位。

prot

新 VMA 的“保护（protection）”。驱动程序可以（或者应该）使用 `vma->vm_page_prot` 中的值。

`remap_page_range` 的参数还算是比较容易理解的，并且在你的 `mmap` 方法被调用时，它们中的大部分已经在 VMA 中提供给你了。一种复杂一些的情形涉及到高速缓存：通常，对设备内存的引用不应该被处理器缓存。系统的 BIOS 会正确的设置它，但是也可以通过 `protection` 成员来禁止指定 VMA 的高速缓存。不幸的是，在这一级上禁止高速缓存是高度依赖于处理器的。感兴趣的读者可查看 `drivers/char/mem.c` 中的函数 `pgprot_noncached` 来了解这个过程所涉及到的内容。我们在这里不会进一步讨论这个话题。

13.2.2 一个简单的实现

如果读者的驱动程序需要实现一个简单的、设备内存的线性映射到用户地址空间中，调用 `remap_page_range` 几乎就是需要做的所有工作了。下面的代码是从文件 `drivers/char/mem.c` 中摘取的，并说明了在一个典型的名为“simple”（Simple Implementation Mapping Pages with Little

Enthusiasm) 的模块中这个任务是如何执行的:

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

int simple_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (offset >= _pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;

    if (remap_page_range(vma->vm_start, offset,
                        vma->vm_end-vma->vm_start, vma->vm_page_prot))
        return -EAGAIN;
    return 0;
}
```

/dev/mem 的代码进行了检查, 以查看所请求的偏移量是否超出了物理内存; 如果是的话, 则设置 VMA 的 VM_IO 标志, 以标记该区域为 I/O 内存。VM_RESERVED 标志总是被设置以防止系统将该区域交换出去, 然后它必然调用 remap_page_range 来构造必需的页表。

13.2.3 增加 VMA 操作

正如我们已经看到的, 结构 vm_area_struct 包含了一系列可以应用于 VMA 的操作。现在我们会着眼于以一种简单的方法来提供那些操作, 更详细的例子会在后面给出。

这里, 我们会为我们的 VMA 提供 open 和 close 操作。这些操作会在进程打开或关闭 VMA 的任何时候被调用, 特别地, open 方法会在进程分叉并创建该 VMA 的新引用时被调用。VMA 的 open 和 close 方法在内核执行的处理之外调用, 因此不必在这里重新实现内核完成的这些工作。它们的存在, 只是提供给驱动程序程序一个途径, 以便完成一些额外的、必需的处理。

我们将使用这些方法, 在 VMA 被打开时增加模块的使用计数, 而在被关闭时减少使用计数。在现代的内核中, 这个工作并不是严格必需的; 只要 VMA 保持打开状态, 内核就不会调用驱动程序的 release 方法, 因此, 直到对 VMA 的所有引用都被关闭之后, 使用计数才会下降到零。但 2.0 内核中没有执行该跟踪, 所以可移植代码仍会希望维护使用计数。

所以, 我们会用跟踪使用计数的操作来覆盖默认的 vma->vm_ops。代码相当简单——对模块化 /dev/mem 的一个完整 mmap 实现, 如下所示:

```
void simple_vma_open(struct vm_area_struct *vma)
{ MOD_INC_USE_COUNT; }

void simple_vma_close(struct vm_area_struct *vma)
{ MOD_DEC_USE_COUNT; }

static struct vm_operations_struct simple_remap_vm_ops = {
    open: simple_vma_open,
    close: simple_vma_close,
};

int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = VMA_OFFSET(vma);
```

```

if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
    vma->vm_flags |= VM_IO;
vma->vm_flags |= VM_RESERVED;

if (remap_page_range(vma->vm_start, offset, vma->vm_end-vma->vm_start,
                    vma->vm_page_prot))
    return -EAGAIN;

vma->vm_ops = &simple_remap_vm_ops;
simple_vma_open(vma);
return 0;
}

```

这段代码依赖于这一事实：在调用 `f_op->mmap` 之前，内核将最近创建的区域中的 `vm_ops` 成员初始化为 `NULL`。为安全起见，上述代码检查了指针的当前值，这在将来的内核中可能需要做一些改动。

出现在这段代码中的 `VMA_OFFSET` 宏，用来隐藏 `vma` 结构在不同内核版本之间的差异。因为偏移量在 2.4 中是以页为单位的，而在 2.2 和更早的内核中是以字节为单位的，为此，头文件 `<sysdep.h>` 声明了这个宏以便这个差异能够被透明处理（其结果以字节表达）。

13.2.4 使用 `nopage` 映射内存

尽管 `remap_page_range` 能够在多数情况下工作良好，但并不是能够适合所有的情况。驱动程序的 `mmap` 实现有时必须具有更好的灵活性。在这种情形下，提倡使用 VMA 的 `nopage` 方法实现内存映射。

`nopage` 方法具有如下原型：

```

struct page (*nopage)(struct vm_area_struct *vma,
                     unsigned long address, int write_access);

```

当一个用户进程试图访问当前不在内存中的 VMA 页面时，就会调用关联的 `nopage` 函数。参数 `address` 包含导致失效的虚拟地址，该地址向下圆整到所在页的起始地址。函数 `nopage` 必须定位并返回指向用户所期望的页的 `struct page` 指针。这个函数还要调用 `get_page` 宏，增加它返回的页面的使用计数：

```

get_page(struct page *pageptr);

```

这一步骤是必要的，以保证被映射页面上的正确引用计数。内核为每个页维护这个计数；当这个计数降为 0 时，内核知道该页应该被置入空闲链表。在一个 VMA 被取消映射时，内核会为该区域中的每一页减少使用计数。如果在向区域中添加一页时，驱动程序没有增加计数，那么使用计数就可能过早地变为零，从而危及到系统的完整性。

`nopage` 方法在 `mremap` 系统调用中非常有用。应用程序使用 `mremap` 来改变一个映射区域的边界地址。如果驱动程序希望处理 `mremap`，先前的实现就不能正确工作，这是因为驱动程序没有办法知道被映射的区域已经改变了。

Linux 的 `mremap` 实现没有通知驱动程序被映射区域的变化。事实上，当区域减小时，它会通过 `unmap` 方法通知驱动程序；但如果区域变大，却没有相应的回调函数可以利用。

将区域减少的变化通知驱动程序，这种做法背后的基本思想是，驱动程序（或者将一个常规文件映射到内存的文件系统）需要知道何时区域被撤销映射，以便采取适当的动作，例如将页面刷新到磁盘上等等。另一方面，映射区域的增长，在程序调用 `mremap` 访问新的虚拟地址之前，对于驱动程序来说却没有实际意义。在实际情况中，经常会出现映射区域从来不会被用到的情况（例如，一段无用的程序代码）。因此，在映射区域增长时，Linux 内核并不会通知驱动程序，因为在实际访问这样的页面时，`nopage` 方法会处理这种情况。

换句话说，在映射区域增长时驱动程序不会得到通知，因为 `nopage` 方法会在将来完成相应的工作，从而不必在真正需要之前使用内存。这个优化主要针对常规文件的，因为它们使用真正的 RAM 进行映射。

因此，如果我们要支持 `mremap` 系统调用，就必须实现 `nopage` 方法。但是，一旦拥有 `nopage` 方法，我们就可以选择广泛使用该方法，当然会有一些限制（后面描述）。这个方法在下面的代码段中给出。在这个 `mmap` 实现中，设备方法仅仅替换了 `vma->vm_ops`。而 `nopage` 方法每次重映射一个页面，并返回它的 `struct page` 结构的地址。因为这里我们只是实现了物理内存之上的一个窗口，因此，重映射步骤非常简单——我们仅需要查找并返回一个指向预期 `struct page` 地址的指针。

使用 `nopage` 的 `/dev/mem` 实现如下所示：

```
struct page *simple_vma_nopage(struct vm_area_struct *vma,
                             unsigned long address, int write_access)
{
    struct page *pageptr;
    unsigned long physaddr = address - vma->vm_start + VMA_OFFSET(vma);
    pageptr = virt_to_page(_va(physaddr));
    get_page(pageptr);
    return pageptr;
}

int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = VMA_OFFSET(vma);

    if (offset >= _pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;

    vma->vm_ops = &simple_nopage_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

这里，我们再次简单地映射主内存，因此，`nopage` 函数只要找到对应于失效地址的正确 `struct page`，并增加它的引用计数。这样，必要的处理顺序如下：首先计算想得到的物理地址，然后用 `_va` 将它转换成逻辑地址，最后用 `virt_to_page` 将逻辑地址转成一个 `struct page`。一般而言，直接从物理地址获得 `struct page` 是可能的，但这样的代码很难在不同的体系结构之间移植。如果有人试图映射高端内存，那么这样的代码或许还是需要的，因为高端内存没有逻辑地址。“simple”很简单，因此不需要考虑此种情况。

如果 `nopage` 方法被置为 `NULL`，处理页故障的内核代码就会将零页映射到失效的虚拟地址。零页

是一个读取时为零的写时复制页，可以用来映射 BSS 段。因此，如果一个进程通过调用 `mremap` 来扩展一个已被映射的区域，并且驱动程序还没有实现 `nopage` 方法，它就会得到零页而不是段错误。

`nopage` 方法通常会返回一个指向 `struct page` 的指针。如果由于某种原因，不能返回正常的页（例如，请求的地址超出了设备的内存区域），就可以返回 `NOPAGE_SIGBUS` 以报告错误。`nopage` 也可以返回 `NOPAGE_OOM`，来指出由于资源限制造成的失败。

注意，对于 ISA 内存区域，这个实现会正常工作，而在 PCI 总线上却不行。PCI 内存被映射到系统内存最高端之上，因此在系统内存映射中没有这些地址的入口。因为无法返回一个指向 `struct page` 的指针，所以 `nopage` 不能用于此种情形；这种情况下，读者必须使用 `remap_page_range`。

13.2.5 重映射特定的 I/O 区域

迄今为止，我们看到的所有例子都是 `/dev/mem` 的再次实现，它们将物理地址重映射到用户空间。然而，典型的驱动程序只想映射对应外围设备的小地址区间，而不是所有的内存。为了向用户空间映射整个内存区间的一个子集，驱动程序仅仅需要处理偏移量。下面几行为映射起始于物理地址 `simple_region_start`、大小为 `simple_region_size` 字节的区域的驱动程序完成了这项工作。

```
unsigned long off = vma->vm_pgoff << PAGE_SHIFT;
unsigned long physical = simple_region_start + off;
unsigned long vsize = vma->vm_end - vma->vm_start;
unsigned long psize = simple_region_size - off;

if (vsize > psize)
    return -EINVAL; /* spans too high */
remap_page_range(vma->vm_start, physical, vsize, vma->vm_page_prot);
```

除了计算偏移量，这段代码还引入了检测，可以在程序试图映射多于目标设备 I/O 区域可用内存时报告一个错误。在本段代码中，`psize` 是指定偏移之后剩余的物理 I/O 大小，而 `vsize` 是请求的虚拟内存大小，该函数拒绝映射超出允许内存范围的地址。

注意，用户程序总是能够使用 `mremap` 来扩展它的映射，从而可能超越物理设备区域的末端。如果驱动程序没有 `nopage` 方法，就永远不会获得关于这个扩展的通知，而且附加的区域会映射到零页。作为驱动程序作者，读者可能希望阻止这种行为；将零页映射到区域的末端并不是一个很糟糕的事情，但是程序员也不希望这种情况出现。

阻止扩展映射的最简单办法是实现一个简单的 `nopage` 方法，它总是向错误进程发送一个总线错误信号。这个简单的 `nopage` 方法如下所示：

```
struct page *simple_nopage(struct vm_area_struct *vma,
                          unsigned long address, int write_access);
{ return NOPAGE_SIGBUS; /* send a SIGBUS */ }
```

13.2.6 重映射 RAM

当然，一个更彻底的实现应该检查出错的地址是否位于设备区域中，如果是，则执行重映射。然而需要再次说明，`nopage` 不会处理 PCI 内存区域，所以 PCI 映射的扩充是不可能的。

在 Linux 中，如果内存映像中的一页物理地址被标记为“reserved（保留的）”，就表明该页对内存管理来说不可用。例如在 PC 上，640 KB 到 1 MB 之间的部分被标记为保留的，因为这个范围是位于内核自身代码的页。

`remap_page_range` 的一个有意思的限制是，它只能对保留页和物理内存之上的物理地址给予访问。保留页被锁在内存中，是仅有的能安全映射到用户空间的页，这个限制是系统稳定性的一个基本要求。

因此，`remap_page_range` 不会允许重映射常规地址——包括通过调用 `get_free_page` 获得的页面。它会改为映射到零页，虽然如此，该函数还是做了大多数硬件驱动程序需要它做的事情，因为它能够重映射高端 PCI 缓冲区和 ISA 内存。

`remap_page_range` 的限制能够通过运行 `mapper` 看到，`mapper` 是 O'Reilly FTP 站点上提供的 `misc-progs` 目录下的一个样例程序。`mapper` 是一个可以用来快速检验 `mmap` 系统调用的简单工具，它根据命令行选项映射一个文件中的只读部分，并把映射区域的内容列在标准输出上。例如，下面这个会话过程表明，`/dev/mem` 没有映射位于 64 KB 地址处的物理页——而我们看到的是全是零的页（这个例子中的主机是 PC，但在其它平台上结果应该是一样的）：

```
morgana.root# ./mapper /dev/mem 0x10000 0x1000 | od -Ax -t x1
mapped "/dev/mem" from 65536 to 69632
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

`remap_page_range` 对处理 RAM 的无能为力表明，类似 `scullp` 这样的设备不能简单地实现 `mmap`，因为它的设备内存是常规 RAM，而不是 I/O 内存。幸运的是，有一种简单的方法可以帮助那些需要映射 RAM 到用户空间的驱动程序，就是使用我们先前看到的 `nopage` 方法。

使用 `nopage` 方法重映射 RAM

将实际的 RAM 映射到用户空间的方法是使用 `vm_ops->nopage` 来处理每个页故障。作为 `scullp` 模块的一部分，示例实现已经在第 7 章介绍过了。

`scullp` 是面向页的字符设备。正因为它是面向页的，所以能够在它的内存中实现 `mmap`。实现内存映射的代码使用一些先前在“Linux 中的内存管理”一节中介绍过的概念。

在查看代码之前，让我们看一下影响 `scullp` 中 `mmap` 实现的设计选择。

只要设备是被映射的，`scullp` 就不会释放设备内存。这与其说是需求，不如说是策略，而且这与 `scull` 及类似设备的行为不同，因为它们会在写打开时截为零。拒绝释放被映射的 `scullp` 设备这一行为，能够允许一个进程改写正在被另一个进程映射的区域，这样读者就能够测试并看到进程与设备内存之间是如何交互的。为了避免释放已映射的设备，驱动程序必须保存一个活动映射的计数，这可以使用设备结构中的 `vmas` 成员实现。

只有在 `scullp` 的 `order` 参数为 0 时，才执行映射内存。该参数控制如何调用 `get_free_pages`（参见第 7 章“`get_free_page` 及相关函数”），而这种选择是由 `get_free_pages` 的内部实现决定的，

而这个函数是 `scullp` 使用的分配引擎。为了取得最佳的分配性能，Linux 内核为每一个分配幂次维护一个空闲页的列表，而且只有簇中的第一个页的页计数可以由 `get_free_pages` 增加，并由 `free_pages` 减少。如果分配幂次大于 0，那么对于 `scullp` 设备来说 `mmap` 方法是关闭的，因为 `nopage` 只处理单页而不处理一簇页面（如果读者需要复习一下 `scullp` 和内存分配幂次的值，可以返回到第 7 章的“使用一整页的 `scull: scullp`”一节）。

最后一个选择主要是保证代码简洁。通过处理页的使用计数，也有可能为多页分配正确地实现“`mmap`”，但那样只能增加例子的复杂性，而不能带来任何我们感兴趣的内容。

如果代码想要按照上面描述的规则来映射 RAM，就需要实现 `open`、`close` 和 `nopage` 等方法，它也需要访问内存映像来调整页的使用计数。

这个 `scullp_mmap` 的实现是很简洁的，因为它依赖 `nopage` 函数来完成所有的工作：

```
int scullp_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct inode *inode = INODE_FROM_F(filp);

    /* refuse to map if order is not 0 */
    if (scullp_devices[MINOR(inode->i_rdev)].order)
        return -ENODEV;

    /* don't do anything here: "nopage" will fill the holes */
    vma->vm_ops = &scullp_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = scullp_devices + MINOR(inode->i_rdev);
    scullp_vma_open(vma);
    return 0;
}
```

开头的条件语句的目的，是为了避免映射分配幂次不为 0 的设备。`scullp` 的操作被存储在 `vm_ops` 成员中，而且一个指向设备结构的指针被存储在 `vm_private_data` 成员中。最后，`vm_ops->open` 被调用以更新模块的使用计数和设备的活动映射计数。

`open` 和 `close` 只是简单地跟踪这些计数，这些方法定义如下：

```
void scullp_vma_open(struct vm_area_struct *vma)
{
    ScullP_Dev *dev = scullp_vma_to_dev(vma);

    dev->vmas++;
    MOD_INC_USE_COUNT;
}

void scullp_vma_close(struct vm_area_struct *vma)
{
    ScullP_Dev *dev = scullp_vma_to_dev(vma);

    dev->vmas--;
    MOD_DEC_USE_COUNT;
}
```

函数 `scullp_vma_to_dev` 简单地返回成员 `vm_private_data` 的内容。因为在 2.4 之前的内核版本中没有 `vm_private_data` 成员，所以 `scullp_vma_to_dev` 以一个单独的函数的形式提供，实际上它只是用来获得成员 `vm_private_data` 的指针。更详细的内容请看本章最后部分的“向后兼容

性”小节。

大部分工作由 `nopage` 完成。在 `scullp` 的实现中，`nopage` 的参数 `address` 用来计算设备里的偏移量，然后该偏移量可在 `scullp` 的内存树中查找正确的页。

```
struct page *scullp_vma_nopage(struct vm_area_struct *vma,
                               unsigned long address, int write)
{
    unsigned long offset;
    ScullP_Dev *ptr, *dev = scullp_vma_to_dev(vma);
    struct page *page = NOPAGE_SIGBUS;
    void *pageptr = NULL; /* default to "missing" */

    down(&dev->sem);
    offset = (address - vma->vm_start) + VMA_OFFSET(vma);
    if (offset >= dev->size) goto out; /* out of range */

    /*
     * Now retrieve the scullp device from the list, then the page.
     * If the device has holes, the process receives a SIGBUS when
     * accessing the hole.
     */
    offset >>= PAGE_SHIFT; /* offset is a number of pages */
    for (ptr = dev; ptr && offset >= dev->qset;) {
        ptr = ptr->next;
        offset -= dev->qset;
    }
    if (ptr && ptr->data) pageptr = ptr->data[offset];
    if (!pageptr) goto out; /* hole or end-of-file */
    page = virt_to_page(pageptr);

    /* got it, now increment the count */
    get_page(page);
out:
    up(&dev->sem);
    return page;
}
```

`scullp` 使用由 `get_free_pages` 获得的内存。该内存使用逻辑地址寻址，所以所有的 `scullp_nopage` 不得不调用 `virt_to_page` 来获得一个 `struct page` 指针。

现在 `scullp` 设备可以按预期工作了，正如读者在工具 `mapper` 的如下示例输出中所看到的。这里，我们发送一个 `/dev`（很长的）目录清单给 `scullp` 设备，然后使用工具 `mapper` 来查看 `mmap` 生成的清单片断。

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140
mapped "/dev/scullp" from 0 to 140
total 77
-rwxr-xr-x    1 root    root      26689 Mar  2  2000 MAKEDEV
crw-rw-rw-    1 root    root       14, 14 Aug 10 20:55 admmidi0
morgana% ./mapper /dev/scullp 8192 200
mapped "/dev/scullp" from 8192 to 8392
0
crw-----    1 root    root      113,  1 Mar 26  1999 cum1
crw-----    1 root    root      113,  2 Mar 26  1999 cum2
crw-----    1 root    root      113,  3 Mar 26  1999 cum3
```

13.2.7 重映射虚拟地址

尽管很少需要重映射虚拟地址，但看看驱动程序是如何使用 `mmap` 将虚拟地址映射到用户空间还是很有意义的。记住，一个真正的虚拟地址是由函数 `vmalloc` 或者 `kmap` 等返回的地址——也就是已经被映射到内核页表的虚拟地址。本节中的代码是取自 `scullv`，这个模块完成与 `scullp` 类似的工作，只是它是通过 `vmalloc` 来分配它的存储空间。

`scullv` 的大部分实现与我们刚刚看到的 `scullp` 基本类似，除了不需要检查控制内存分配的 `order` 参数之外。原因是 `vmalloc` 每次只分配一页，因为单页分配比多页分配容易成功的多。因此，分配的幂次问题在 `vmalloc` 分配的空间中不存在。

`vmalloc` 的大部分工作是构造页表，从而可以象连续地址空间一样访问分配的页。为了向调用者返回一个 `struct page` 指针，`nopage` 方法必须将页表打散。因此，`scullv` 的 `nopage` 实现必须扫描页表以取得与页相关联的页映像入口。

除了结尾部分，这个函数与我们在 `scullp` 中看到的一样。这个代码的节选只包括了 `nopage` 中与 `scullp` 不同的部分：

```
pgd_t *pgd; pmd_t *pmd; pte_t *pte;
unsigned long lpage;

/*
 * After scullv lookup, "page" is now the address of the page
 * needed by the current process. Since it's a vmalloc address,
 * first retrieve the unsigned long value to be looked up
 * in page tables.
 */
lpage = VMALLOC_VMADDR(pageptr);
spin_lock(&init_mm.page_table_lock);
pgd = pgd_offset(&init_mm, lpage);
pmd = pmd_offset(pgd, lpage);
pte = pte_offset(pmd, lpage);
page = pte_page(*pte);
spin_unlock(&init_mm.page_table_lock);

/* got it, now increment the count */
get_page(page);
out:
up(&dev->sem);
return page;
```

页表由本章开头介绍的函数来查询。为这一目的而使用的页目录储存在内核空间的内存结构 `init_mm` 中。注意，`scullv` 要在访问页表之前获得 `page_table_lock`，如果没有持有该锁，在 `scullv` 查找过程进行的中途，其它的处理器可能会改变页表，从而导致错误的结果。

宏 `VMALLOC_VMADDR(pageptr)` 返回正确的 `unsigned long` 值，用于 `vmalloc` 地址的页表上查询。由于一个内存管理的小问题，对该值的强制类型转换在早于 2.1 的 x86 内核上不能正常工作。2.1.1 版本的 x86 内存管理做了改动，和其它平台一样，现在的 `VMALLOC_VMADDR` 被定义为一个实体函数。然而，为了编写可移植性代码，我们仍然建议使用这个宏。

基于上述讨论，读者可能也希望将 `ioremap` 返回的地址映射到用户空间。这种映射很容易实现，因为读者可以直接使用 `remap_page_range`，而不用实现虚拟内存区域的方法。换句话说，

`remap_page_range` 已经可用于构造将 I/O 内存映射到用户空间的新页表，并不需要象我们在 `scullv` 中那样查看由 `vremap` 构造的内核页表。

13.3 kiobuf 接口

从版本 2.3.12 开始，Linux 内核支持一种叫做内核 I/O 缓冲区或者 `kiobuf` 的 I/O 抽象对象。`kiobuf` 接口用来从设备驱动程序（以及系统的其它 I/O 部分）中隐藏虚拟内存系统的复杂性。开发人员打算为 `kiobuf` 实现很多特性，但是它们最初用于 2.4 内核是为了便于将用户空间的缓冲区映射到内核。

13.3.1 kiobuf 结构

任何使用 `kiobuf` 的代码必须包含头文件 `<linux/iobuf.h>`。该文件定义了 `struct kiobuf` 结构，它是 `kiobuf` 接口的核心部分。这个结构描述了构成 I/O 操作的一个页面数组，它的成员有：

```
int nr_pages;
```

该 `kiobuf` 中页的数量

```
int length;
```

在缓冲区中数据的字节数量

```
int offset;
```

相对于缓冲区中第一个有效字节的偏移量

```
struct page **maplist;
```

一个 `struct page` 结构数组，每一项代表 `kiobuf` 中的一个数据页。

`kiobuf` 接口的关键之处就是 `maplist` 数组。那些用来操作存储在 `kiobuf` 中的页的函数，直接处理 `page` 结构，这样，就可以跳过所有的虚拟内存系统开支。这种实现允许驱动程序独立于复杂的内存管理，并且以非常简单的方式运行。

在使用之前，必须对 `kiobuf` 进行初始化。很少会初始化单个的 `kiobuf`，但如果需要，这种初始化可通过 `kiobuf_init` 来执行：

```
void kiobuf_init(struct kiobuf *iobuf);
```

通常，`kiobuf` 以组的方式被分配为一个内核 I/O 向量（即 `kiovec`）的一部分。通过调用 `alloc_kiovec`，就可以分配和初始化一个 `kiovec`。

```
int alloc_kiovec(int nr, struct kiobuf **kiovec);
```

照常，返回值是 0 或一个错误码。在代码结束 `kiovec` 结构的使用时，就应该将它返还给系统：

```
void free_kiovec(int nr, struct kiobuf **);
```

内核提供一对函数来锁住和解锁 `kiovec` 中被映射的页：

```
int lock_kiobuf(int nr, struct kiobuf *iovec[], int wait);
int unlock_kiobuf(int nr, struct kiobuf *iovec[]);
```

然而，对于在设备驱动程序中看到的大多数 `kiobuf` 应用来说，以这种方式锁住一个 `kiobuf` 是多余的。

13.3.2 映射用户空间缓冲区以及裸 I/O

Unix 系统已经对某些设备提供了“裸”接口——特别是块设备——它直接从用户空间缓冲区执行 I/O，从而避免了经由内核复制数据。在某些情况下可以通过这种方式大大地提高性能，尤其在被传输的数据近期不会被再次使用的情况下。例如，典型的磁盘备份只是从磁盘一次性的读入大量数据，然后就不会处理这些数据了，通过裸接口运行备份会避免无用的数据占用系统缓冲区的高速缓存。

出于很多原因，早期的 Linux 内核没有提供裸接口。然而，由于系统获得了普及，并且更多的希望能处理裸 I/O 的应用程序（例如大型数据库管理系统）被移植过来。所以 2.3 开发系列最终还是增加了裸 I/O；为了增加这种裸 I/O 能力，才有了 `kiobuf` 接口。

裸 I/O 不是象某些人认为的那样总是高性能的推进措施，驱动程序作者不应该只因为它能够增进性能而总是采用。设置裸传输的开支也是很明显的，而且内核缓冲数据的优势也就丢失了。例如，裸 I/O 操作几乎总是必须同步的——`write` 系统调用直到操作结束后才能返回。当前，Linux 还缺少可以让用户程序能够在用户缓冲区上安全地执行异步裸 I/O 的机制。

本节中，我们为块设备驱动程序范例 `sbull` 增加了裸 I/O 能力。在 `kiobuf` 可用时，`sbull` 实际上注册了两个设备。我们已经在第 12 章中详细讨论了块的 `sbull` 设备。我们上一章没有看到的是第二个字符设备（叫做“`sbullr`”），它提供了对 RAM 磁盘设备的裸访问。这样，`/dev/sbull0` 和 `/dev/sbullr0` 访问同一块内存，前面的样例使用传统的缓冲模式，而第二个样例通过 `kiobuf` 机制提供了裸访问。

在 Linux 系统中值得注意的是，对于块设备驱动程序来说，我们不需要提供这种接口。在源文件 `drivers/char/raw.c` 中，`raw` 设备为所有块设备以精美而通用的方法提供了这种能力。块驱动程序甚至不需要知道它们是否正在处理裸 I/O。出于示范的目的，`sbull` 中的裸 I/O 代码本质上是 `raw` 设备代码的简化版本。

块设备的裸 I/O 必须始终是扇区对齐的，而且它的长度必须是扇区大小的整数倍。其它类型的设备，例如磁带驱动程序可以没有这样的限制。`sbullr` 象块设备一样运转，而且必须符合对齐和长度的要求，为此，它定义了几个符号：

```
# define SBULLR_SECTOR 512 /* insist on this */
# define SBULLR_SECTOR_MASK (SBULLR_SECTOR - 1)
# define SBULLR_SECTOR_SHIFT 9
```

裸设备 `sbullr` 只有在硬扇区尺寸等于 `SBULLR_SECTOR` 时才会被注册。没有真正的原因为什么不支持大的硬扇区尺寸，只是因为这样会导致示例代码复杂化。

`sbullr` 实现只是对现存的 `sbull` 增加了少量代码。特别地，`sbull` 中的 `open` 和 `close` 方法未经修

改而直接使用。因为 **sbullr** 是一个字符设备，所以它需要 **read** 和 **write** 方法。它们都被定义为使用一个单独的传输函数，就像下面这样：

```
ssize_t sbullr_read(struct file *filp, char *buf, size_t size,
                   loff_t *off)
{
    Sbull_Dev *dev = sbull_devices +
        MINOR(filp->f_dentry->d_inode->i_rdev);
    return sbullr_transfer(dev, buf, size, off, READ);
}

ssize_t sbullr_write(struct file *filp, const char *buf, size_t size,
                    loff_t *off)
{
    Sbull_Dev *dev = sbull_devices +
        MINOR(filp->f_dentry->d_inode->i_rdev);
    return sbullr_transfer(dev, (char *) buf, size, off, WRITE);
}
```

在将实际的数据传送给另一个函数时，函数 **sbullr_transfer** 处理所有的组装和拆卸任务。该函数实现如下：

```
static int sbullr_transfer (Sbull_Dev *dev, char *buf, size_t count,
                           loff_t *offset, int rw)
{
    struct kiobuf *iobuf;
    int result;

    /* Only block alignment and size allowed */
    if ((*offset & SBULLR_SECTOR_MASK) || (count & SBULLR_SECTOR_MASK))
        return -EINVAL;
    if ((unsigned long) buf & SBULLR_SECTOR_MASK)
        return -EINVAL;

    /* Allocate an I/O vector */
    result = alloc_kiobuf(1, &iobuf);
    if (result)
        return result;

    /* Map the user I/O buffer and do the I/O. */
    result = map_user_kiobuf(rw, iobuf, (unsigned long) buf, count);
    if (result) {
        free_kiobuf(1, &iobuf);
        return result;
    }
    spin_lock(&dev->lock);
    result = sbullr_rw_iobuf(dev, iobuf, rw,
                            *offset >> SBULLR_SECTOR_SHIFT,
                            count >> SBULLR_SECTOR_SHIFT);
    spin_unlock(&dev->lock);

    /* Clean up and return. */
    unmap_kiobuf(iobuf);
    free_kiobuf(1, &iobuf);
    if (result > 0)
        *offset += result << SBULLR_SECTOR_SHIFT;
    return result << SBULLR_SECTOR_SHIFT;
}
```

在作两个常规检查之后，代码使用 **alloc_kiobuf** 创建了一个 **kiobuf**（包含单个 **kiobuf**）。然后它调用 **map_user_kiobuf** 将用户缓冲区映射到该 **kiobuf**：

```
int map_user_kiobuf(int rw, struct kiobuf *iobuf,
                    unsigned long address, size_t len);
```

如果所有工作正常进行，这个调用的结果是将给定（用户虚拟） **address** 且长度为 **len** 的缓冲区映射到给定的 **iobuf**。该操作可能进入睡眠，因为用户缓冲区很可能会需要经过页故障处理以装入内存。

当然，通过这种方式映射的 **kiobuf** 最终必须被撤销映射以保持页引用计数的连贯性。在代码中可以看到，这个撤销映射的过程通过将 **kiobuf** 传递给 **unmap_kiobuf** 而实现。

迄今为止，我们已经看到如何为 I/O 准备 **kiobuf**，但没有看到如何去实际执行这个 I/O。最后一个步骤涉及到 **kiobuf** 中的每一页，并完成所请求的传送；在 **sbullr** 中，该任务通过 **sbullr_rw_iovec** 处理。本质上，这个函数遍历每一页，将它拆分成扇区大小的块，并通过一个伪请求结构将这些块传递给 **sbull_transfer**：

```
static int sbullr_rw_iovec(Sbull_Dev *dev, struct kiobuf *iobuf, int rw,
                           int sector, int nsectors)
{
    struct request fakereq;
    struct page *page;
    int offset = iobuf->offset, ndone = 0, pageno, result;

    /* Perform I/O on each sector */
    fakereq.sector = sector;
    fakereq.current_nr_sectors = 1;
    fakereq.cmd = rw;

    for (pageno = 0; pageno < iobuf->nr_pages; pageno++) {
        page = iobuf->maplist[pageno];
        while (ndone < nsectors) {
            /* Fake up a request structure for the operation */
            fakereq.buffer = (void *) (kmap(page) + offset);
            result = sbull_transfer(dev, &fakereq);
            kunmap(page);
            if (result == 0)
                return ndone;
            /* Move on to the next one */
            ndone++;
            fakereq.sector++;
            offset += SBULLR_SECTOR;
            if (offset >= PAGE_SIZE) {
                offset = 0;
                break;
            }
        }
    }
    return ndone;
}
```

这里，**kiobuf** 结构的 **nr_pages** 成员告诉我们要多少页需要传送，而 **maplist** 数组可以让我们访问每一页。这样，我们就能够方便地遍历这些页了。但要注意，**kmap** 被用于为每一页获得内核的虚拟地址，这样，即使用户缓冲区处在高端内存，函数也能正常工作。

一些对于复制数据的快速测试表明：一次与 **sbullr** 设备之间的数据复制相比于与 **sbull** 块设备的同样复制只会大约花费后者三分之二的系统时间。这个时间上的节省是通过避免了额外的经由缓冲区高速缓存的复制而获得的。注意，如果同样的数据被几次重复读取的话，那么这种节省就没有

了——特别是对于一个真正的硬件设备。裸设备访问常常不是最好的方法，但是对于某些应用来说，它能够提供很大的性能改进。

尽管 `kiobuf` 在内核开发团体中存有争议，在很多情况下使用它们还是很有意思的。例如，有一个用 `kiobuf` 实现 Unix 管道的补丁——数据被直接从一个进程的地址空间复制到另一个进程的地址空间而根本没有经过内核的缓冲。还有一个补丁，它能够方便地将内核虚拟地址映射到进程的地址空间，这样，就消除了使用前述 `nopage` 实现的需求。

13.4 直接内存访问和总线控制

直接内存访问，或者 DMA，是我们最后要讨论的高级主题。DMA 是一种硬件机制，它允许外围设备和主内存之间直接传输它们的 I/O 数据，而不需要在传输中使用系统处理器。使用这种机制可以大大提高与设备通讯的吞吐量，因为免除了大量的计算开支。

为了利用硬件的 DMA 能力，设备驱动程序需要能够正确地设置 DMA 传输并能够与硬件同步。不幸的是，由于硬件本身的性质，DMA 是高度依赖于系统的。每一种体系结构都有它自己的管理 DMA 传输的技术，而且彼此的编程接口也是不同的。内核不能提供统一的接口，因为驱动程序很难将底层的硬件机制适当地抽象。然而在最近的内核中，某些步骤已经被向这个方向发展。

本章主要集中在 PCI 总线上，因为它是当前可用的外围总线中最流行的一种，但很多概念是普遍适用的。我们也会简单谈到其它总线处理 DMA 的方式，例如 ISA 和 Sbus。

13.4.1 DMA 数据传输概览

在介绍编程细节之前，让我们回顾一下 DMA 传输是如何进行的。为简化讨论，只考虑输入传输。

数据传输可以以两种方式触发：或者软件请求数据（例如通过函数 `read`）或者由硬件将数据异步地推向系统。

在第一种情况下，调用的步骤可以概括如下：

1. 在进程调用 `read` 时，驱动程序的方法分配一个 DMA 缓冲区，随后指示硬件传送它的数据。进程进入睡眠。
2. 硬件将数据写入 DMA 缓冲区并在完成时产生一个中断。
3. 中断处理程序获得输入数据，应答中断，最后唤醒进程，该进程现在可以读取数据了。

第二种情形是在 DMA 被异步使用时发生的。例如，数据采集设备持续地推入数据，即使没有进程读取它。这种情况下，驱动程序应该维护一个缓冲区，使得接下来的 `read` 调用可以将所有累积的数据取回到用户空间。这种传送的调用步骤稍有不同：

1. 硬件发出中断来通知新的数据已经到达。
2. 中断处理程序分配一个缓冲区并且通知硬件将数据传往何处。
3. 外围设备将数据写入缓冲区，然后在完成时发出另一个中断。
4. 处理程序分发新的数据，唤醒任何相关进程，然后处理一些杂务。

不同的异步方法常常可以在网卡的代码中看到。这些网卡经常期望能有一个循环缓冲区（通常叫做 **DMA 环形缓冲区**）建立在与处理器共享的内存中。每一个输入数据包被放置在环形缓冲区中下一个可用缓冲区，并且发出中断。然后驱动程序将网络数据包传给内核的其它部分处理，并在环形缓冲区中放置一个新的 **DMA 缓冲区**。

上面这两种情况下的处理步骤都强调：高效的 **DMA** 处理依赖于中断报告。尽管可以用一个轮询驱动程序来实现 **DMA**，但这样做没有什么意义，因为一个轮询驱动程序会将 **DMA** 相对于简单的处理器驱动 I/O 获得的性能优势抵消掉。

这里介绍的另一个相关问题是 **DMA 缓冲区**。为了利用直接内存访问，设备驱动程序必须能够分配一个或者更多的适合 **DMA** 的特殊缓冲区。注意，很多驱动程序在初始化时分配它们的缓冲区，并使用它们直到停止运行——因此在前面涉及到的“分配”一词意味着“获取一个先前分配的缓冲区”。

13.4.2 分配 DMA 缓冲区

本节主要讨论在较低层的 **DMA 缓冲区** 分配方法，很快我们会介绍一个较高层的接口，但是正确理解这里介绍的内容还是很重要的。

DMA 缓冲区 的主要问题是，当它大于一页时，它必须占据物理内存中的连续页，因为设备使用 **ISA** 或者 **PCI** 系统总线传送数据，它们都使用的是物理地址。值得注意的是，这个限制对于 **SBus** 并不适用（见第 15 章中的“**SBus**”小节），它在外围总线上使用虚拟地址。一些体系结构也能够 **PCI** 总线上使用虚拟地址，但是是一个可移植的驱动程序不能依靠这种能力。

尽管 **DMA 缓冲区** 可以在系统引导或者运行时分配，但模块只能在运行时分配它们的缓冲区。第 7 章介绍了这些技术：“系统启动时的内存分配”一节讲述了系统引导时的分配，而“**kmalloc** 函数的内幕”和“**get_free_page** 和相关函数”描述了运行时分配。驱动程序作者必须小心分配可以应用于 **DMA** 操作的正确内存类型——不是所有的内存区段都适合。特别地，高端内存存在大多数系统上不能用于 **DMA**——外围设备不能使用高端地址工作。

现代总线上的大部分设备都能够处理 32 位地址，这就意味着普通的内存分配就会很好地为其工作。然而某些 **PCI** 设备未能实现完整的 **PCI** 标准，因而不能使用 32 位地址工作。而 **ISA** 设备却只能限制在 16 位地址上。

对于具有该限制的设备，通过给调用 **kmalloc** 和 **get_free_pages** 增加 **GFP_DMA** 标志就可以从 **DMA** 区段中分配内存。当该标志存在时，只会分配可使用 16 位寻址的内存。

DIY 分配

我们已经明白为什么 **get_free_pages**（所以 **kmalloc**）不能返回多于 128 KB 的连续内存空间（或者更普遍而言，32 页）。即使在分配小于 128 KB 的缓冲区时，这个请求也很容易失败，因为随着时间的推移，系统内存会成为一些碎片。^{*}

^{*} “碎片”这个词一般用于磁盘，表达文件在磁性介质上不连续地存放。这个概念同样适用于内存，即当每个虚拟地址空间都散布在整个物理 **RAM** 中时，就很难为 **DMA** 的缓冲区请求分配连续的空闲页面。

在内核不能返回要求数量的内存时，或者在我们需要多于 128 KB 的内存时（例如，PCI 帧捕获卡的普遍请求），相对于返回 `-ENOMEM`，另外一个可选的方法是在引导时分配内存或者为缓冲区保留物理 RAM 的顶部。我们已经在第 7 章的“系统启动时的内存分配”小节描述了引导时的分配，但是这种方法对于模块是不可用的。通过在引导时给内核传递一个“`mem=`”参数可以保留 RAM 的顶部。例如，如果系统有 32MB 内存，参数“`mem=31M`”阻止内核使用最顶部的一兆字节。稍后，模块可以使用下面的代码来访问这些保留的内存：

```
dmabuf = ioremap( 0x1F00000 /* 31M */, 0x100000 /* 1M */);
```

实际上，还有另一种分配 DMA 空间的方法：不断地执行分配，直到能够获得足够的连续页面来构造缓冲区。如果有任意其它方法可以实现这一目的，则不应该使用这种分配技术。不断地分配会导致很高的系统负荷，如果这种作法没有被正确地调整，也可能导致系统锁住。但另一方面，有时确实没有别的方法可以利用。

在实践中，代码调用 `kmalloc (GFP_ATOMIC)` 直到失败为止，然后它等待内核释放若干页面，接下来再一次进行分配。如果密切注意已分配的页面池，迟早会发现由连续页面组成的 DMA 缓冲区已经出现；这时，我们可以释放除了被选中的缓冲区之外的所有页面。这种行为是相当危险的，因为它会导致死锁。我们建议使用内核定时器来释放每一页，以防在给定时间内分配还不能成功。

这里，我们准备给出代码，但是读者会在 `misc-modules/allocator.c` 中找到；代码被注释得很详细，而且被设计为可以被其他模块调用。不同于本书中给出的其他源程序，`allocator` 适用于 GPL 条款。我们决定将源程序置于 GPL 条款之下的理由既不是由于它特别优美也不是由于它特别有技巧，而是如果有人想要使用它，我们希望代码同模块一起发行。

13.4.3 总线地址

一个使用 DMA 的设备驱动程序通常会与连接到接口总线上的硬件通讯，这些硬件使用物理地址，而程序代码使用虚拟地址。

事实上，情况还要更复杂些。基于 DMA 的硬件使用总线地址而不是物理地址，尽管在 PC 上，ISA 和 PCI 地址与物理地址一样，但并不是所有的平台都这样。有时，接口总线是通过将 I/O 地址映射到不同物理地址的桥接电路连接的。甚至某些系统有一个页面映射方案，能够使任意页面在外围总线上表现为连续的。

在最低层（相对，我们很快就会看到一个高层的解决方案），通过导出下列在头文件 `<asm/io.h>` 中定义的函数，Linux 内核提供了一个可移植的解决方案：

```
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```

当驱动程序需要向一个 I/O 设备（例如扩展板或者 DMA 控制器）发送地址信息时，必须使用 `virt_to_bus` 转换，在接受到来自连接到总线上硬件的地址信息时，必须使用 `bus_to_virt` 了。

13.4.4 PCI 总线上的 DMA

2.4 内核包含一个支持 PCI DMA 的灵活机制（也称作“总线控制”）。它处理缓冲区分配的细节，也能够为支持多页传送的硬件进行总线硬件设置。这些代码也能处理缓冲区位于不具有 DMA 能力的内存区域的情形，尽管这只在某些平台上实现，并且还有一些额外的计算开支（稍后会看到）。

本节中的函数需要一个用于我们的设备的 `struct pci_dev` 结构。设置 PCI 设备的细节会在第 15 章中讲述。注意，这里描述的例程也能够用于 ISA 设备，这种情况下，只需将 `struct pci_dev` 指针赋值为 `NULL`。

使用下面这些函数的驱动程序应该包含头文件 `<linux/pci.h>`。

处理不同硬件

在执行 DMA 之前，第一个必须回答的问题是：是否给定的设备在当前主机上具备执行这些操作的能力。很多 PCI 设备不能实现完整的 32 位总线地址空间，常常是因为它们其实是老式 ISA 硬件的修订版本。Linux 内核会试图与这些设备协同工作，但不总是能成功的。

函数 `pci_dma_supported` 应该为任何具有地址限制的设备所调用：

```
int pci_dma_supported(struct pci_dev *pdev, dma_addr_t mask);
```

这里，`mask` 仅仅是描述哪些地址位可以被设备使用的位掩码。如果返回值非零，表示 DMA 可用，我们的驱动程序应该将 PCI 设备结构中的 `dma_mask` 成员设置为该掩码值（即 `mask`）。对于只能处理 16 位地址的设备，我们应该使用类似如下的调用：

```
if (pci_dma_supported (pdev, 0xffff))
    pdev->dma_mask = 0xffff;
else {
    card->use_dma = 0;    /* We'll have to live without DMA */
    printk (KERN_WARN, "mydev: DMA not supported\n");
}
```

内核 2.4.3 中提供了一个新的函数 `pci_set_dma_mask`。这个函数具有如下原型：

```
int pci_set_dma_mask(struct pci_dev *pdev, dma_addr_t mask);
```

如果使用给定的掩码能够支持 DMA，这个函数返回 0 并且设置 `dma_mask` 成员；否则，返回 `-EIO`。

对于能够处理 32 位地址的设备，就没有调用 `pci_dma_supported` 函数的必要了。

DMA 映射

一个 DMA 映射就是分配一个 DMA 缓冲区并为该缓冲区生成一个能够被设备访问的地址的组合操作。很多情况下，简单地调用 `virt_to_bus` 就可以获得需要的地址，然而有些硬件要求映射寄存器也被设置在总线硬件中。映射寄存器（mapping register）是一个类似于外围设备的虚拟内存等价物。在使用这些寄存器的系统上，外围设备有一个相对较小的、专用的地址区段，可以在此区段执行 DMA。通过映射寄存器，这些地址被重映射到系统 RAM。映射寄存器具有一些好的特性，包

括使分散的页面在设备地址空间看起来是连续的。但不是所有的体系结构都有映射寄存器，特别地，PC 平台没有映射寄存器。

在某些情况下，为设备设置有用的地址也意味着需要构造一个反弹（bounce）缓冲区。例如，当驱动程序试图在一个不能被外围设备访问的地址（一个高端内存地址）上执行 DMA 时，反弹缓冲区被创建。然后，按照需要，数据被复制到反弹缓冲区，或者从反弹缓冲区复制。如果想要代码在反弹缓冲区上正常工作，就应该符合某些规则，正如我们很快会看到的。

DMA 映射提出了一个新的类型（`dma_addr_t`）来表示总线地址。`dma_addr_t` 类型的变量应该被驱动程序作为不透明物来对待；只有允许的操作会被传给 DMA 支持例程或者传给驱动程序自己。

根据 DMA 缓冲区期望保留的时间长短，PCI 代码区分两种类型的 DMA 映射：

- 一致 DMA 映射 它们存在于驱动程序的生命周期内。一个被一致映射的缓冲区必须同时可被 CPU 和外围设备访问（正如我们稍后会看到的，其他类型的映射在给定的时间只能用于一个或另一个）。如果可能，缓冲区也应该没有高速缓存问题——即能够造成一个（如 CPU）不会看到另一个（如外设）所作的更新的问题。
- 流式 DMA 映射 流式 DMA 映射是为单个操作进行的设置。在使用流式映射时，某些体系结构允许重要优化，但是，正如我们会看到的，这些映射也要服从一组更加严格的访问规则。内核开发者推荐应尽可能使用流式映射，而不是一致映射。这个推荐是基于两个原因。首先，在支持一致映射的系统上，每个 DMA 映射会使用总线上一个或多个映射寄存器。具有较长生命周期的一致映射，会独占这些寄存器很长时间——即使它们没有被使用。其次，在某些硬件上，流式映射能够以某种方式优化，而一致映射却不能。

两种映射类型必须以不同的方法操作，现在让我们看一下细节。

建立一致 DMA 映射

驱动程序可调用 `pci_alloc_consistent` 设置一致映射：

```
void *pci_alloc_consistent(struct pci_dev *pdev, size_t size,
                          dma_addr_t *bus_addr);
```

这个函数能够处理缓冲区的分配和映射。前两个参数是我们的 PCI 设备结构以及所需缓冲区大小，函数在两处返回 DMA 映射的结果，返回值是缓冲区的内核虚拟地址，它可以被驱动程序使用；而相关的总线地址在 `bus_addr` 中返回。该函数对分配的缓冲区做了一些处理，从而缓冲区可用于 DMA；通常只是通过 `get_free_pages` 分配内存（但是要注意，大小以字节计算而不是幂次的值）。

大多数支持 PCI 的体系结构以 `GFP_ATOMIC` 优先级执行分配，而且这样不会睡眠。但是，内核的 ARM 移植是个例外。

当不再需要缓冲区时（通常在模块卸载时），应该调用 `pci_free_consistent` 将它返还给系统：

```
void pci_free_consistent(struct pci_dev *pdev, size_t size,
                        void *cpu_addr, dma_handle_t bus_addr);
```

注意这个函数需要提供 CPU 地址和总线地址。

建立流式 DMA 映射

由于多种原因，流式映射具有比一致映射更复杂的接口。这些映射希望能与已经由驱动程序分配的缓冲区协同工作，因而不得不处理它们没有选择的地址。在某些体系结构上，流式映射也能够由多个不连续的页和多个“分散/集中”缓冲区。

在设置流式映射时，我们必须通知内核数据将向哪个方向传送。已经为此定义了如下符号：

```
PCI_DMA_TODEVICE
PCI_DMA_FROMDEVICE
```

这两个符号无需多做说明。如果数据被发送到设备（也许为响应系统调用 `write`），应该使用 `PCI_DMA_TODEVICE`；相反，如果数据将发送到 CPU，则应标记 `PCI_DMA_FROMDEVICE`。

```
PCI_DMA_BIDIRECTIONAL
```

如果数据能够进行两个方向的移动，就使用 `PCI_DMA_BIDIRECTIONAL`。

```
PCI_DMA_NONE
```

这个符号只是为帮助调试而提供。试图以这个“方向”使用缓冲区会造成内核 `panic`。

出于许多我们很快就会遇到的原因，为流式 DMA 映射选取正确的方向值是很重要的。虽然任何时候都选取 `PCI_DMA_BIDIRECTIONAL` 是很诱人的，但在某些体系结构上，会因这种选择而损失性能。

在只有单个用于传送的缓冲区时，应该使用 `pci_map_single` 来映射它：

```
dma_addr_t pci_map_single(struct pci_dev *pdev, void *buffer,
                          size_t size, int direction);
```

返回值是可以传递给设备的总线地址，如果出错的话就为 `NULL`。

一旦传送完成，应该使用 `pci_unmap_single` 删除映射：

```
void pci_unmap_single(struct pci_dev *pdev, dma_addr_t bus_addr,
                      size_t size, int direction);
```

这里，`size` 和 `direction` 参数必须匹配于它们用来映射缓冲区时的值。

下面是一些应用于流式 DMA 映射的重要规则：

- 缓冲区只能用于这样的传送，即其传送方向匹配于映射时给定的方向值。
- 一旦缓冲区已经被映射，它就属于设备而不再属于处理器了。在缓冲区被撤销映射之前，驱动程序不应该以任何方式触及其内容。只有在 `pci_unmap_single` 被调用之后，对驱动程序来说，访问缓冲区内容才是安全的（但我们将很快看到一个例外）。尤其要说明的是，这条规则意味着要写入设备的缓冲区在包含所有要写入的数据之前，不能映射该缓冲区。

■ 在 DMA 仍然进行时，缓冲区不能被撤销映射，否则会造成严重的系统不稳定性。

读者可能会觉得奇怪，为什么一旦缓冲区被映射驱动程序就不能够再使用它。实际上有两个原因使得出现这个规则。第一，在缓冲区为 DMA 映射时，内核必须确保缓冲区中所有的数据已经被实际写到内存。可能有些数据还会保留在处理器的高速缓冲存储器中，因此必须显式刷新。在刷新之后，由处理器写入缓冲区的数据对设备来说也许是不可见的。

第二，如果欲映射的缓冲区位于设备不能访问的内存区段时，我们考虑会产生什么结果。这种情况下，某些体系结构仅仅会操作失败，而其它的体系结构会创建一个反弹缓冲区。反弹缓冲区只是一个可被设备访问的独立内存区域。如果一个缓冲区使用 `PCI_DMA_TODEVICE` 方向映射，并且需要一个反弹缓冲区，则原始缓冲区的内容作为映射操作的一部分被复制。很明显，原始缓冲区在复制之后的变化对设备来说是不可见的。同样地，`PCI_DMA_FROMDEVICE` 反弹缓冲区通过 `pci_unmap_single` 被复制回原始缓冲区；直到复制完成后，来自设备的数据才可用。

顺便提及，反弹缓冲的存在，是“为什么获得正确的方向很重要的”一个理由。`PCI_DMA_BIDIRECTIONAL` 反弹缓冲区会在操作的前后被复制，而这常常是一种不必要的 CPU 时钟周期的浪费。

有时候，驱动程序需要不经过撤销映射就访问流式 DMA 缓冲区的内容，为此，内核提供了如下调用：

```
void pci_sync_single(struct pci_dev *pdev, dma_handle_t bus_addr,
                    size_t size, int direction);
```

该函数应该在处理器访问 `PCI_DMA_FROMDEVICE` 缓冲区之前，或者在访问 `PCI_DMA_TODEVICE` 缓冲区之后调用。

分散/集中映射

分散/集中映射是流式 DMA 映射的一种特例。假设你有几个缓冲区，而它们需要传送到设备或者从设备传送回来。这种情形可能以几种途径产生，包括从 `readv` 或者 `writv` 系统调用产生，从集群的磁盘 I/O 请求产生，或者从映射的内核 I/O 缓冲区中的页面表产生。我们可以简单地依次映射每一个缓冲区并且执行请求的操作，但是一次映射整个缓冲区表还是很有利的。

一个原因是一些设计巧妙的设备能够接受由数组指针和长度组成的“分散表（scatterlist）”并在一个 DMA 操作中传送所有数据；例如，如果数据包能够组装成多块，那么“零拷贝”网络是很容易的实现的。Linux 将来很可能会很好地利用这些设备。另一个整个映射分散表的原因是，可以利用总线硬件上具有映射寄存器的系统。在这样的系统上，物理上不连续的页面能够被装配成从设备角度看是单个的连续数组。这种技术只能用在分散表中的项在长度上等于页面大小的时候（除了第一个和最后一个之外），但是在其工作时，它能够将多个操作转化成单个 DMA 操作，因而能够加速处理工作。

最后，如果必须使用反弹缓冲区，将整个表接合成一个单个缓冲区是很有意义的（因为无论如何它也会被复制）。

所以现在你可以确信在某些情况下分散表的映射是值得做的。映射分散表的第一步是建立并填充一个描述被传送缓冲区的 `struct scatterlist` 数组。该结构是体系结构相关的，并且在头文件 `<linux/scatterlist.h>` 中描述。然而，该结构会始终包含两个成员：

```
char *address;
```

用在分散/集中操作中的缓冲区地址

```
unsigned int length;
```

该缓冲区的长度

为了映射一个分散/集中的 DMA 操作，驱动程序应该为每个欲传送的缓冲区准备的 `struct scatterlist` 项中设置 `address` 和 `length` 成员。然后调用：

```
int pci_map_sg(struct pci_dev *pdev, struct scatterlist *list,
               int nents, int direction);
```

返回值是要传送的 DMA 缓冲区数；它可能会小于 `nents`，也就是传入的分散表项的数量。

驱动程序应该传送每一个 `pci_map_sg` 返回的缓冲区。每一个缓冲区的总线地址和长度会被存储在 `struct scatterlist` 项中，但是它们在结构中的位置在不同的体系结构中是不同的。已经定义的两个宏使得编写可移植代码成为可能：

```
dma_addr_t sg_dma_address(struct scatterlist *sg);
```

从该分散表项中返回总线地址

```
unsigned int sg_dma_len(struct scatterlist *sg);
```

返回该缓冲区的长度

此外，记住准备传送的缓冲区的地址和长度可能会不同于传入 `pci_map_sg` 的值。

一旦传输完成，分散/集中映射通过调用 `pci_unmap_sg` 来撤销映射：

```
void pci_unmap_sg(struct pci_dev *pdev, struct scatterlist *list,
                  int nents, int direction);
```

注意，`nents` 必须是原先传给 `pci_map_sg` 的项的数量，而不是函数返回给我们的 DMA 缓冲区的数量。

分散/集中映射是流式 DMA 映射，关于单个种类，同样的访问规则适用于它们。如果读者必须访问一个已映射的分散/集中链表，就必须首先同步它：

```
void pci_dma_sync_sg(struct pci_dev *pdev, struct scatterlist *sg,
                     int nents, int direction);
```

支持 PCI DMA 的不同体系结构

正如我们在本节开始是说明的，DMA 是硬件特有的操作。我们刚才描述的 PCI DMA 接口试图将很多硬件依赖性尽可能地抽象出来，然而还有一些问题还没有解决。

M68K
S/390
Super-H

到 2.4.0 版本为止，这些体系结构不支持 PCI 总线。

IA-32 (x86)
MIPS
PowerPC
ARM

这些平台支持 PCI DMA 接口，但是其接口主要是骗人的外表。总线接口中没有映射寄存器，所以分散表不能被组合而且不能使用虚拟地址。也没有反弹缓冲区支持，所以不能完成高端地址的映射。ARM 体系结构上的映射函数能够睡眠，而在其它平台上这些函数不能睡眠。

IA-64

Itanium 体系结构也缺少映射寄存器。这个 64 位体系结构能够容易地生成 PCI 外围设备不能使用的地址，因而在此平台上的 PCI 接口实现了反弹缓冲区，允许任意地址被 DMA 操作所使用（表面上）。

Alpha
MIPS64
SPARC

这些体系结构支持 I/O 内存管理单元。自 2.4.0 起，MIPS64 内核实际不再利用这个功能，所以它的 PCI DMA 实现看起来就象 IA-32 的实现。尽管 Alpha 和 SPARC 内核能够利用正确的分散/集中支持来实现完整的缓冲区映射。

这里列出的区别对于大多数驱动程序作者都不是问题，只要遵从接口的一些规则即可。

一个简单的 PCI DMA 例子

在 PCI 总线上 DMA 操作的实际形式非常依赖于被驱动的设备。这样，这个例子不能应用于任何真实设备；它只是一个叫做“dad (DMA Acquisition Device)”的假想设备的一部分。该设备的驱动程序定义了一个这样的传输函数：

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer,
                 size_t count)
{
    dma_addr_t bus_addr;
    unsigned long flags;

    /* Map the buffer for DMA */
    dev->dma_dir = (write ? PCI_DMA_TODEVICE : PCI_DMA_FROMDEVICE);
    dev->dma_size = count;
    bus_addr = pci_map_single(dev->pci_dev, buffer, count,
                              dev->dma_dir);
    dev->dma_addr = bus_addr;

    /* Set up the device */
    writew(dev->registers.command, DAD_CMD_DISABLEDMA);
    writew(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
    writel(dev->registers.addr, cpu_to_le32(bus_addr));
    writel(dev->registers.len, cpu_to_le32(count));

    /* Start the operation */
    writew(dev->registers.command, DAD_CMD_ENABLEDMA);
    return 0;
}
```



```
}

```

该函数映射了准备进行传输的缓冲区并且开始设备操作。另一半工作必须在中断服务例程中完成，它看起来有点类似这样：

```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct dad_dev *dev = (struct dad_dev *) dev_id;

    /* Make sure it's really our device interrupting */

    /* Unmap the DMA buffer */
    pci_unmap_single(dev->pci_dev, dev->dma_addr, dev->dma_size,
                     dev->dma_dir);

    /* Only now is it safe to access the buffer, copy to user, etc. */
    ...
}
```

显而易见地，在这个例子中已经忽略了大量细节，包括用来阻止同时开始多个 DMA 操作的必要步骤。

简单看看 Sbus 上的情况

传统上，基于 SPARC 的系统包括一个由 Sun 公司设计的叫做 SBus 的总线。该总线超出了本章的讨论范围，但是简单的讨论还是值得的。有一组在 Sbus 总线上执行 DMA 映射的函数（在头文件 `<asm/sbus.h>` 中描述）；它们的名称类似 `sbus_alloc_consistent` 和 `sbus_map_sg`。换句话说，SBus 总线的 DMA API 看起来很象 PCI 接口。在使用 SBus 总线上的 DMA 之前，还是需要详细地看看这些函数的定义，但是概念会与先前对 PCI 总线的讨论相似。

13.4.5 ISA 设备的 DMA

ISA 总线允许两种 DMA 传输：本地（native）DMA 和 ISA 总线控制（bus-master）DMA。本地 DMA 使用主板上的标准 DMA 控制器电路驱动 ISA 总线上的信号线；另一方面，ISA 总线控制 DMA 则完全由外围设备处理。后一种 DMA 类型很少使用，所以就不在这里不讨论了，因为它类似于 PCI 设备的 DMA，至少从驱动程序的角度看是这样的。ISA 总线控制的一个例子是 1542 SCSI 控制器，它的驱动程序是内核源代码中的 `drivers/scsi/aha1542.c`。

至于本地 DMA，有三种实体涉及到 ISA 总线上的 DMA 数据传输：

- **8237 DMA 控制器（DMAC）** 控制器存有有关 DMA 传送的信息，例如传送方向、内存地址和传送大小。它也包含一个跟踪传送状态的计数器。在控制器收到一个 DMA 请求信号时，它获得总线的控制权并且驱动信号线以使设备能够读写数据。
- **外围设备** 设备在准备好传送数据时，必须激活 DMA 请求信号。实际的传输由 DMAC 负责管理，当控制器选通设备后，硬件设备就可以顺序地读/写总线上的数据。传输结束时，设备通常会发出中断。
- **设备驱动程序** 驱动程序只需做好如下几点：它向 DMA 控制器提供方向、总线地址和传送大小。它还告诉外围设备准备好传送数据，并在 DMA 结束时响应中断。

原先在 PC 中使用的 DMA 控制器能够管理 4 个通道，每一个通道与一组 DMA 寄存器关联。

因此 4 个设备能够同时在控制器中保存它们的 DMA 信息。新的 PC 有两套相当于 DMAC 的设备：^{*}第二个（主）控制器被连接到系统处理器上，第一个（从）控制器被连接到第二个控制器的通道 0 上。^{*}

通道编号为 0 到 7。因为通道 4 是内部用于将从控制器级联到主控制器上的，所以它对于 ISA 外围设备不可用。这样，可用的通道是从控制器上的 0 到 3（8 位通道）和主控制器上的 5 到 7（16 位通道）。每次 DMA 传送的大小保存在控制器中，是一个 16 位的数值，表示总线周期数。因此，从控制器的最大传输大小为 64 KB，主控制器的最大传输大小为 128 KB。

因为 DMA 控制器是一个系统级的资源，所以内核协助处理这一资源。内核使用 DMA 注册表为 DMA 通道提供了请求/释放机制，并且提供了一组函数在 DMA 控制器中配置通道信息。

注册 DMA 的方法

读者应该对内核注册表（registry）很熟悉了——我们已经在 I/O 端口和中断信号线部分接触过它们，DMA 通道的注册与它们很相似。在包含头文件 <asm/dma.h> 之后，就可以使用下面的函数来获取和释放 DMA 通道的所有权：

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

参数 channel 是一个 0 到 7 的数值，确切的说，是一个小于 MAX_DMA_CHANNELS 的正数。在 PC 上，为了和硬件相匹配，MAX_DMA_CHANNELS 被定义为 8。参数 name 是一个用来识别设备的字符串，它所标识的名字出现在文件 /proc/dma 中，该文件可以被用户程序读取。

request_dma 函数的返回值可能是：0 表示成功，-EINVAL 或者 -EBUSY 表示失败。返回 -EINVAL 表示请求的通道超出范围，返回 -EBUSY 表示该通道正在被其它设备所使用。

我们建议读者象对待 I/O 端口和中断信号线一样地小心处理 DMA 通道。在 open 时请求 DMA 通道要比在模块初始化时请求更为有利，推迟请求会为驱动程序间共享 DMA 通道创造条件，例如，声卡可以和相类似的 I/O 接口共享同一 DMA 通道，只要它们不在同一时间使用该通道。

我们同样建议读者在请求了中断线之后请求 DMA 通道，并且在释放中断线之前释放它。这是请求两种资源的通常顺序，依照惯例是为了避免可能的死锁。注意，每一个使用 DMA 的设备同时也需要中断信号线，否则就无法发出数据传输完成的通知。

典型的情况下的 open 代码如下所示。这段代码引用了我们假想的 dad 模块，dad 设备使用了一个快速中断处理并且不支持共享 IRQ 信号线。

```
int dad_open (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;

    /* ... */
}
```

^{*} 现在，这些电路是主板芯片的一部分，但是几年前它们还是两个独立的 8237 芯片。

^{*} 最初的 PC 只有一个控制器，第二个控制器在基于 286 的平台上加入。然而，第二个控制器被作为主控制器连接是因为它能够处理 16 位传输。而第一个控制器一次只能传送 8 位，它的存在是为了向后兼容。

```

if ( (error = request_irq(my_device.irq, dad_interrupt,
                        SA_INTERRUPT, "dad", NULL)) )
    return error; /* or implement blocking open */

if ( (error = request_dma(my_device.dma, "dad")) ) {
    free_irq(my_device.irq, NULL);
    return error; /* or implement blocking open */
}
/* ... */
return 0;
}

```

与 `open` 相对应的 `close` 实现如下所示：

```

void dad_close (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;

    /* ... */
    free_dma(my_device.dma);
    free_irq(my_device.irq, NULL);
    /* ... */
}

```

关注一下 `/proc/dma` 文件，这是安装了声卡的系统上该文件的内容：

```

merlino% cat /proc/dma
1: Sound Blaster8
4: cascade

```

值得注意的是，默认的声卡驱动程序在系统启动时就获取了 **DMA** 通道，并且一直没有释放它。正如前面解释过的那样，显示的 **cascade** 项是一个占位符，表示通道 **4** 不能被其他设备所使用。

与 DMA 控制器通讯

在注册之后，驱动程序的主要工作包括配置 **DMA** 控制器以使其正常工作。这一工作非常重要，但幸运的是内核导出了典型驱动程序需要的所有函数。

在调用 `read` 或 `write` 时，或者准备进行异步传输时，驱动程序都需要配置 **DMA** 控制器。依驱动程序和其所实现的策略的不同，这一工作可以在打开设备时，或者在应答 `ioctl` 命令时进行。这里给出的代码是驱动程序的 `read` 和 `write` 方法所调用的典型代码。

这一小节提供了 **DMA** 控制器内部的概貌，这样，读者就能理解这里所给出的代码。如果读者想学习关于这部分的更多知识，我们强烈推荐读者去阅读头文件 `<asm/dma.h>` 以及一些描述 **PC** 体系结构的硬件手册。尤其是，我们不会在这里处理与 **16** 位数据传输相对的 **8** 位传输问题，如果你正在为 **ISA** 设备板卡编写设备驱动程序，则应该在该设备的硬件手册中查找相关的信息。

DMA 控制器是一个共享资源，如果多个处理器同时试图对其进行操作，将会引起混乱。有鉴于此，**DMA** 控制器被一个叫做 `dma_spin_lock` 的自旋锁所保护。然而，驱动程序不应直接对该锁进行操作，内核提供了两个对其操作的函数：

```

unsigned long claim_dma_lock();

```

获取 **DMA** 自旋锁，该函数会阻塞本地处理器上的中断，因此，其返回值是“标志”值，在重新

打开中断时必须使用该值。

```
void release_dma_lock(unsigned long flags);
```

释放 DMA 自旋锁，并且恢复以前的中断状态。

在使用接下来描述的那些函数时，应该持有自旋锁。然而，在驱动程序做真正的 I/O 操作时，不应该持有自旋锁。驱动程序在持有自旋锁时绝对不能进入睡眠。

必须装载到 DMA 控制器的信息由三部分组成：RAM 地址、必须传输的原子项个数（以字节或字为单位），以及传输的方向。最后，头文件 `<asm/dma.h>` 导出了下面几个函数：

```
void set_dma_mode(unsigned int channel, char mode);
```

指出通道要从设备读出（DMA_MODE_WRITE）数据，还是向设备写入数据。还存在有第三种模式，即 DMA_MODE_CASCADE，用于释放对总线的控制。级联就是将第一个控制器连接到第二个控制器的顶端的方式，但是也能够被真正的 ISA 总线控制设备使用。这里，我们不讨论总线控制。

```
void set_dma_addr(unsigned int channel, unsigned int addr);
```

该函数给 DMA 缓冲区的地址赋值。该函数将 `addr` 的最低 24 位存储到控制器中。参数 `addr` 必须是总线地址（参照本章前面“总线地址”小节）。

```
void set_dma_count(unsigned int channel, unsigned int count);
```

该函数对传输的字节数赋值。参数 `count` 也代表 16 位通道的字节数，在此情况下，这个数字必须是偶数。

除这些函数之外，在处理 DMA 设备时，还必须使用很多处理杂项的工具函数：

```
void disable_dma(unsigned int channel);
```

DMA 通道可以在控制器内被禁止掉。通道应该在配置 DMA 控制器之前被禁止，以防止进行不正确的操作（控制器是通过 8 位数据传送进行编程的，这样，前面所有的函数都不会原子地执行）。

```
void enable_dma(unsigned int channel);
```

该函数通知 DMA 控制器 DMA 通道中包含了合法的数据。

```
int get_dma_residue(unsigned int channel);
```

有时，驱动程序需要知道一个 DMA 传输是否已经完成。该函数返回尚未传送的字节数。函数在传输成功时的返回值是 0，当控制器正在工作时的返回值是不可预知的（但不是 0）。返回值的不可预测表明这样一个事实，即剩余量是一个 16 位数，它是通过两个 8 位输入操作获取的。

```
void clear_dma_ff(unsigned int channel)
```

该函数清除 DMA 触发器（flip-flop），该触发器用来控制对 16 位寄存器的访问。可以通过两个连续的 8 位操作来访问这些寄存器，触发器被清除时用来选择低字节，触发器被置位时用来选择高字节。在传输 8 位后，触发器会自动反转；在访问 DMA 寄存器之前，程序员必须清除触发器（将它设置为某个已知状态）。

使用这些函数，驱动程序可以实现一个类似下面代码所示的函数为 DMA 传输作准备：

```
int dad_dma_prepare(int channel, int mode, unsigned int buf,
                    unsigned int count)
{
    unsigned long flags;

    flags = claim_dma_lock();
    disable_dma(channel);
    clear_dma_ff(channel);
    set_dma_mode(channel, mode);
    set_dma_addr(channel, virt_to_bus(buf));
    set_dma_count(channel, count);
    enable_dma(channel);
    release_dma_lock(flags);

    return 0;
}
```

然后，类似下面的函数会被用来检查 DMA 传输是否成功结束。

```
int dad_dma_isdone(int channel)
{
    int residue;
    unsigned long flags = claim_dma_lock ();
    residue = get_dma_residue(channel);
    release_dma_lock(flags);
    return (residue == 0);
}
```

剩下的唯一要做的事情就是配置设备板卡了。这种设备特有的任务通常仅仅是对少数 I/O 端口的读写操作，但不同的设备之间存在很大的差别。例如，某些设备需要程序员通知硬件 DMA 的缓冲区有多大，而有时，驱动程序不得不读出固化在设备里的值。配置板卡时，硬件手册是我们唯一的朋友。

13.5 向后兼容性

随着时间的推移，象内核中的其它部分一样，内存映射和 DMA 也发生了很多改变。本节将描述一些驱动程序作者在编写可移植代码时需要注意的事项。

13.5.1 内存管理部分的改变

2.3 开发系列的主要变化发生在内存管理部分。2.2 内核很大程度上受限于它能使用的内存数量，尤其在 32 位处理器上这种情况尤为突出。对于 2.4 版本，这种限制被减轻了；现在，Linux 能够管理处理器所能够寻址的所有内存，而且某些事情不得不为此改变以使其实现这种能力；但是，API 层的改动比例是很小的。

正如我们所看到的，2.4 版内核广泛使用了指向 page 结构的指针来在内存中查阅特定的页面。这个结构已经存在于 Linux 中很长时间了，但是先前并没有用这个结构来指代页面本身，相反，内核使用的是逻辑地址。

例如，pte_page 返回一个 unsigned long 值而不是 struct page *。宏 virt_to_page 根本就不存在了，如果需要找到一个 struct page 项，就不得不直接从内存映射中查找。宏 MAP_NR 会将逻辑地址变成 mem_map 中的索引；这样，当前的 virt_to_page 宏可以被如下定义（在示例代码

的 `sysdep.h` 头文件中):

```
#ifndef MAP_NR
#define virt_to_page(page) (mem_map + MAP_NR(page))
#endif
```

在 `virt_to_page` 被引入时, 宏 `MAP_NR` 就没有用了。宏 `get_page` 在 2.4 内核之前也不存在, 所以 `sysdep.h` 如下定义它:

```
#ifndef get_page
# define get_page(p) atomic_inc(&(p)->count)
#endif
```

`struct page` 也已经随时间而改变, 特别, `virtual` 成员仅出现在 2.4 内核的 Linux 中。

`page_table_lock` 在 2.3.10 版本中引入。先前的代码在穿越页表之前应该获得“大的内核锁”(在遍历页表前后调用 `lock_kernel` 和 `unlock_kernel`)。

结构 `vm_area_struct` 在 2.3 开发系列中发生了很多变化, 在 2.1 系列中变化更多。这些变化包括:

- 在 2.2 和之前的版本中, 成员 `vm_pgoff` 叫做 `vm_offset`。它是字节的偏移量而不是页面的偏移量。
- 成员 `vm_private_data` 在 2.2 版本的 linux 中并不存在, 所以驱动程序并没有在 VMA 中保存自身信息的方法。许多驱动程序使用 `vm_pte` 成员。但是, 从 `vm_file` 中获得次设备号, 并使用它来获取需要的信息, 这种方法更安全些。
2.4 版本内核在调用 `mmap` 方法之前初始化 `vm_file` 指针。在 2.2 版本的内核中, 驱动程序不得不自己赋值, 并使用 `file` 结构作为参数传递进入。
- 在 2.0 版本的内核中, `vm_file` 指针更本不存在; 替代地, 有一个 `vm_inode` 指针指向 `inode` 结构。该成员需要由驱动程序赋值, 它也必须在 `mmap` 方法中增加成员 `inode->i_count` 的值。
- 标志 `VM_RESERVED` 在内核版本 2.4.0-test10 中加入。

对于存储在 VMA 中的各个 `vm_ops` 方法来说, 发生的改变如下:

- 2.2 和之前版本的内核有一个叫做 `advise` 的方法, 但实际上内核从来没有使用它。还有一个 `swpin` 方法, 它用来将数据从备用存储器上读入内存, 通常, 这对于驱动程序作者来说没有什么意义。
- 在 2.2 版本的内核中, `nopage` 和 `wppage` 方法返回 `unsigned long`, 而不是 `struct page *`。
- `NOPAGE_SIGBUS` 和 `NOPAGE_OOM` 返回代码表示 `nopage` 并不存在。`nopage` 简单地返回 0 来指出问题并给受影响的进程发送一个总线信号。

因为 `nopage` 以前返回 `unsigned long`, 它的工作是返回页的逻辑地址而不是它的 `mem_map` 项。

当然，在老内核中没有高端内存支持。所有内存都有逻辑地址，而且 `kmap` 和 `kunmap` 函数也不存在。

在 2.0 版本内核中，结构 `init_mm` 没有为模块导出。这样，想要访问 `init_mm` 的模块为了找到它不得不在任务表中搜索（作为“init”进程的一部分）。在 2.0 内核上运行时，`scullp` 通过下面这段代码找到 `init_mm`：

```
static struct mm_struct *init_mm_ptr;
#define init_mm (*init_mm_ptr) /* to avoid ifdefs later */

static void retrieve_init_mm_ptr(void)
{
    struct task_struct *p;

    for (p = current ; (p = p->next_task) != current ; )
        if (p->pid == 0)
            break;

    init_mm_ptr = p->mm;
}
```

2.0 版本内核在逻辑地址和物理地址之间也缺少明显的区别，所以宏 `__va` 和 `__pa` 也不存在。那时也不需要它们。

在 2.0 版本的内核中不存在的另一个处理就是，在映射内存区域中并没有维护模块的使用计数。在低于 2.0 版本的内核中，实现了 `mmap` 的驱动程序需要提供 `open` 和 `close` VMA 操作并调整使用计数。实现了 `mmap` 的示例模块提供了这些操作。

最后，就像大多数其它的方法，驱动程序的 `mmap` 方法的 2.0 版本有一个 `struct inode` 参数，方法的原型是：

```
int (*mmap)(struct inode *inode, struct file *filp,
            struct vm_area_struct *vma);
```

13.5.2 DMA 的变化

正如先前描述的，PCI 的 DMA 接口在 2.3.41 版本之前并不存在。然而之前，DMA 以更直接的（而且依赖于系统的）方式处理。缓冲区通过调用 `virt_to_bus` 来映射，而且没有通用接口来处理总线映射寄存器。

对于那些想要写可移植的 PCI 驱动程序的人来说，样例代码中文件 `sysdep.h` 包括有 2.4 版本的 DMA 接口的一个简单实现，该实现可用于老的内核。

另一方面，ISA 接口自从 2.0 版本的 Linux 以来几乎没有改变过。ISA 是一种老的体系结构，毕竟，有许许多多的变化没有跟上。在 2.2 版本中唯一增加的是 DMA 自旋锁，在该内核之前，不需要对 DMA 控制器的冲突访问进行保护。这些函数的版本已经在文件 `sysdep.h` 中定义，它们禁止和恢复中断，而不执行其他功能。

13.6 快速参考

本章介绍了与内存处理有关的下列符号。因为第一节本身是一个大的列表并且它们的符号对于设备驱动程序来说很少使用，所以在第一节中介绍的符号没有在下面列出。

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

所有与内存管理有关的函数和结构在这个头文件中定义并给出原型。

```
int remap_page_range(unsigned long virt_addr, unsigned long phys_addr, unsigned long size,
    pgprot_t prot);
```

这些函数是 **mmap** 的核心。它将大小为 **size** 字节、起始地址为 **phys_addr** 的物理内存映射到虚拟地址 **virt_addr**。与虚拟空间相联系的保护位在 **prot** 中指定。

```
struct page *virt_to_page(void *kaddr);
void *page_address(struct page *page);
```

这些宏在内核逻辑地址以及它们相关联的内存映射入口之间进行转换。**page_address** 只能处理低端内存页或者已经被显式映射的高端内存页。

```
void *_va(unsigned long physaddr);
unsigned long __pa(void *kaddr);
```

这些宏在内核逻辑地址和物理地址之间进行转换。

```
unsigned long kmap(struct page *page);
void kunmap(struct page *page);
```

kmap 返回一个被映射到给定页的内核虚拟地址，如果需要的话，建立该映射。**kunmap** 删除给定页的映射。

```
#include <linux/kiobuf.h>
void kiobuf_init(struct kiobuf *iobuf);
int alloc_kiovec(int number, struct kiobuf **iobuf);
void free_kiovec(int number, struct kiobuf **iobuf);
```

这些函数处理内核 I/O 缓冲区的分配、初始化和释放。**kiobuf_init** 初始化单个 **kiobuf**，但是很少使用。替代地，可使用 **alloc_kiovec** 来分配并初始化一个 **kiobuf** 向量，并使用 **free_kiovec** 释放 **kiobuf** 向量。

```
int lock_kiovec(int nr, struct kiobuf *iovec[], int wait);
int unlock_kiovec(int nr, struct kiobuf *iovec[]);
```

这两个函数分别锁住并释放内存中的 **kiovec**。在使用 **kiobuf** 进行用户空间内存的 I/O 时，不必使用这两个函数。

```
int map_user_kiobuf(int rw, struct kiobuf *iobuf, unsigned long address, size_t len);
void unmap_kiobuf(struct kiobuf *iobuf);
```

map_user_kiobuf 将一个用户空间的缓冲区映射成给定的内核 I/O 缓冲区，**unmap_kiobuf** 撤销该映射。

```
#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```

这些函数完成内核虚拟地址和总线地址之间的转换。必须使用总线地址来和外围设备对话。

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

使用下面这些函数时，必须包含该头文件。


```
int pci_dma_supported(struct pci_dev *pdev, dma_addr_t mask);
```

对于那些不能对全部的 32 位地址空间寻址的外围设备来说,这个函数决定了主机系统上是否支持 DMA。

```
void *pci_alloc_consistent(struct pci_dev *pdev, size_t size, dma_addr_t *bus_addr)
void pci_free_consistent(struct pci_dev *pdev, size_t size, void *cpuaddr, dma_handle_t
    bus_addr);
```

这些函数为驱动程序生命期中一直有效的缓冲区分配和释放一致 DMA 映射。

```
PCI_DMA_TODEVICE
PCI_DMA_FROMDEVICE
PCI_DMA_BIDIRECTIONAL
PCI_DMA_NONE
```

这些符号用来告诉流式映射函数数据从缓冲区中移入或者移出的方向。

```
dma_addr_t pci_map_single(struct pci_dev *pdev, void *buffer, size_t size, int
    direction);
void pci_unmap_single(struct pci_dev *pdev, dma_addr_t bus_addr, size_t size, int
    direction);
```

建立和销毁一个独家使用的流式 DMA 映射。

```
void pci_sync_single(struct pci_dev *pdev, dma_handle_t bus_addr, size_t size, int
    direction)
```

同步一个具有流式映射的缓冲区。如果处理器必须访问一个正在使用流式映射的缓冲区（亦即，设备拥有该缓冲区），则必须使用这个函数。

```
struct scatterlist { /* ... */ };
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```

scatterlist 结构描述了调用多于一个缓冲区的 I/O 操作。在实现分散/集中操作时，宏 **sg_dma_address** 和 **sg_dma_len** 可以用来获得总线地址和缓冲区长度，并传递给驱动程序。

```
pci_map_sg(struct pci_dev *pdev, struct scatterlist *list, int nents, int direction);
pci_unmap_sg(struct pci_dev *pdev, struct scatterlist *list, int nents, int direction);
pci_dma_sync_sg(struct pci_dev *pdev, struct scatterlist *sg, int nents, int direction)
```

pci_map_sg 映射一个分散/集中操作，并且由 **pci_unmap_sg** 来取消映射。如果需要在映射有效时访问缓冲区，**pci_map_sync_sg** 可用来同步上述操作。

```
/proc/dma
```

这个文件包含了 DMA 控制器中已分配通道的文本快照。由于每个 PCI 板卡独立工作并且不需要在 DMA 控制器中分配通道，所以基于 PCI 的 DMA 不会显示在这个文件中。

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

所有关于 DMA 的函数和宏，都在这个头文件定义或给出了原型。如果要使用下述符号，就必须包含这个头文件。

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

这些函数访问 DMA 注册表。注册必须在使用 ISA DMA 通道之前执行。

```
unsigned long claim_dma_lock();
void release_dma_lock(unsigned long flags);
```

这些函数获得和释放 DMA 自旋锁。在调用其他的 ISA DMA 函数（后面描述）期间必须持有该自旋锁。它们同时在本地处理器上禁止并重新打开中断。

```
void set_dma_mode(unsigned int channel, char mode);  
void set_dma_addr(unsigned int channel, unsigned int addr);  
void set_dma_count(unsigned int channel, unsigned int count);
```

这些函数用来在 DMA 控制器中设置 DMA 信息。addr 是总线地址。

```
void disable_dma(unsigned int channel);  
void enable_dma(unsigned int channel);
```

在配置期间，DMA 通道必须被禁止。这些函数用来改变 DMA 通道的状态。

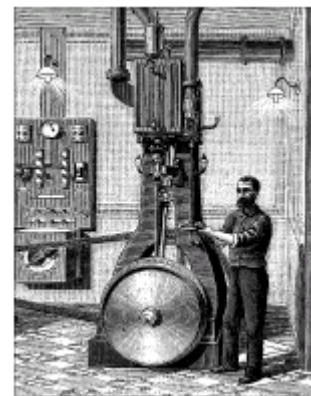
```
int get_dma_residue(unsigned int channel);
```

如果驱动程序需要了解 DMA 传输进行的当前状态，则可以调用这个函数。该函数返回尚未传输的数据的数量。在 DMA 成功完成之后，函数返回 0；在数据传输正在进行时，这个值是不可预知的。

```
void clear_dma_ff(unsigned int channel)
```

控制器用 DMA 触发器来传输 16 位值，这可以由两个 8 位的操作来进行。在给控制器发送任何数据之前必须将该触发器清零。

第 14 章 网络驱动程序



我们已经彻底讨论了字符驱动程序和块驱动程序，本章将把重点移向引人入胜的网络领域。网络接口是 Linux 的第三类标准设备，本章将描述它们和内核其余部分之间的互相作用。

系统中网络接口的角色，和一个已挂装的块设备类似。一个块设备将自己注册到 `blk_dev` 数组以及其它内核结构，然后通过自己的 `request` 函数“传输”和“接收”数据块。类似地，网络接口也必须在特定的数据结构中注册自己，以便在与外界交换数据包时被调用。

在已挂装磁盘和数据包发送接口之间，存在着一些非常重要的不同。首先，磁盘在 `/dev` 目录中作为一个特殊文件而存在，而网络接口并没有类似的 `/dev` 目录入口点。普通的文件操作（读取、写入等等）对网络接口来讲没有任何意义，因此，无法将 Unix 的“所有东西都是文件”这一思想应用于网络接口。这样，网络接口存在于它们自己的名字空间中，同时导出一组不同的操作。

读者也许会指出，应用程序在使用套接字时，使用的就是 `read` 和 `write` 系统调用，但其实这些系统调用作用于与网络接口完全不同的软件对象上面。在同一物理网络接口之上，可同时存在几百个多工的套接字。

但这两者之间最重要的差别在于，块驱动程序只对来自内核的请求做出响应，而网络驱动程序却异步地接收来自外界的数据包。这样，块驱动程序“被请求”向内核发送一个缓冲区，而网络设备却“请求”将引入的数据包推向内核。用于网络驱动程序的内核接口，就是为这种不同的操作模式设计的。

网络驱动程序同时必须支持大量的管理任务，比如设置地址，修改传输参数，以及维护流量和错误统计等等。网络驱动程序的 API 反映出了这种需求，因此，在某种程度上与我们已经看到过的接口大为不同。

Linux 内核的网络子系统是完全与协议无关的——不管是网络协议（IP、IPX 或其它协议），还是硬件协议（以太网、令牌环等等）。网络驱动程序和内核其余部分之间的交互，每次处理的是一个网络数据包。这样，驱动程序无需关心协议问题，而协议也不必关心数据的物理传输。

本章将描述网络接口是如何服务于内核其余部分的，同时讲述一个基于内存的模块化网络接口，我

们称之为“**snull**”。为了简化我们的讨论，该接口使用以太网硬件协议，并传输 IP 数据包。从学习 **snull** 中获得的知识，可非常容易地应用到其它非 IP 协议，而非以太网驱动程序的编写，也仅仅在涉及到实际网络协议的微小细节上存在差异。

本章不会论及 IP 地址编号方式、网络协议或其它一般性的网络概念。这些内容（通常）不是驱动程序编写者需要考虑的，而且，想在不到一百页的篇幅中提供令人满意的网络技术综述，简直是不可能的。感兴趣的读者，可以参考其它描述网络技术的书籍。

在内核开发人员为提供更好的网络性能而做出的多年努力中，网络子系统发生了许多变化。本章的大部分内容描述 2.4 内核中的网络驱动程序的实现，但我们的示例代码可在 2.0 和 2.2 内核上工作。本章最后将给出早期内核和 2.4 内核之间的不同。

在讲述网络设备之前，需要对一个术语做出解释。在网络领域，我们使用“**octet**”来表示一个八位位组，这是网络设备和协议能够理解的最小单位。而“**byte**（字节）”这个词，几乎不会在本章出现。为了遵循标准用法，我们在论及网络设备时使用“**octet**（八位位组）”一词。

14.1 snull 的设计

这一小节讨论 **snull** 网络接口在设计上的一些概念。尽管这些内容适合在页边上做注脚用，但如果不能正确理解这个驱动程序，就有可能无法正确利用示例代码。

第一个，也是最重要的设计决策，就是示例接口应该不依赖于任何实际硬件，而一点类似本书其它的示例代码。这一限制使得我们的接口看上去类似回环（**loopback**）接口。但是，**snull** 并不是一个回环设备，它模拟了和实际远程主机之间的会话，以便更好地演示网络驱动程序的编写。**Linux** 回环驱动程序实际上相当简单，读者可在 `drivers/net/loopback.c` 中找到该驱动程序。

snull 的另一个特点是，它支持 IP 流量。这是该接口的内部工作方式所决定的——为了正确模拟一对硬件接口，**snull** 必须观察并解释数据包。实际的接口不会依赖于被传输的协议，而 **snull** 的这一限制也不会影响本章所描述的代码片段。

14.1.1 赋予 IP 号

snull 模块建立两个接口，这两个接口和简单的回环设备不同。我们通过其中一个接口传输的数据，将在另外一个接口上出现，而不是它本身。看起来，我们似乎有两个外部链路，但实际上计算机是在应答它本身。

不幸的是，这一效果不能通过单个 IP 号获得，因为内核不会把通过接口 A 直接发送到自身接口 B 的数据包发送出去，相反，内核会使用回环通道而不会通过 **snull** 设备。为了在 **snull** 的接口之间建立通讯连接，我们必须在数据传输过程中修改源地址和目标地址。换句话说，通过其中一个接口发送的数据应该被另外一个接口接收到，但不能将外发数据的接收者认作本地主机，同样的规则也应该应用于已接收数据包的源地址。

为了实现这种“隐藏的回环”设备，**snull** 接口切换源地址和目标地址的第三个 **octet** 的最低位；也就是说，它修改了 C 类 IP 号的网络编号和主机编号。其效果是，发送到网络 A（连接到 **sn0**，

即第一个接口)的数据包,将在属于网络 B 的 sn1 接口上出现。

为避免涉及太多的数字,我们赋予相关的 IP 号一些符号名:

- snullnet0 是连接到 sn0 接口的 C 类网络。类似地,snullnet1 是连接到 sn1 的网络。上述网络地址仅仅在第三个 octet 的最低位有差别。
- local0 是赋予 sn0 接口的 IP 地址,它属于 snullnet0。和 sn1 关联的地址是 local1。local0 和 local1 必须在第三和第四个 octet 的最低位上不同。
- remote0 是 snullnet0 网络中的一个主机,它的第四个 octet 和 local1 相同。发送到 remote0 的任意数据包将在接口代码修改了其 C 类地址之后,到达 local1。remote1 属于 snullnet1,它的第四个 octet 和 local0 一样。

snull 接口的操作在图 14-1 中描述,其中与每个接口关联的主机名打印在该接口名的旁边。

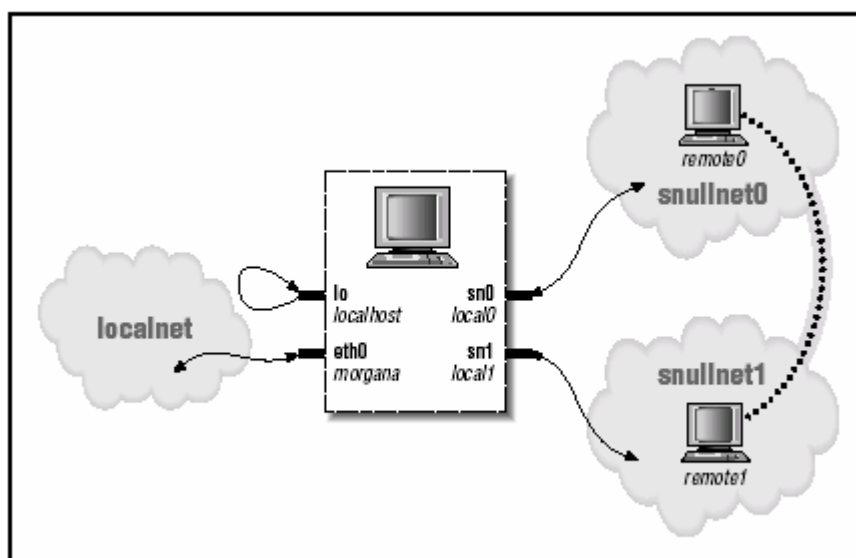


图 14-1: 主机和接口的关系

下面是一些满足上述要求的可能网络编号。将这两行放入 `/etc/networks` 文件之后,就可以用名字来指代网络。这些网络编号值选择自非正式使用的 IP 号范围。

```
snullnet0    192.168.0.0
snullnet1    192.168.1.0
```

下面是可加入到 `/etc/hosts` 的可能主机 IP 号:

```
192.168.0.1  local0
192.168.0.2  remote0
192.168.1.2  local1
192.168.1.1  remote1
```

上述编号一个重要特点是,local0 的主机部分和 remote1 的主机部分一样,而 local1 的主机部分,和 remote0 的主机部分一样。只要满足上述关系,读者就可以选择一组完全不同的网络号和

主机号。

但是需要小心的是，如果计算机已经连入一个实际的网络，则你所选择的编号，可能是实际 Internet 或 intranet 号，将这些编号赋予自己的接口，将导致无法和实际主机通讯。例如，尽管上述编号并不是可路由的 Internet 编号，但可能已经在防火墙之后的内部私有网络当中使用。

不管选择什么地址编号，可通过如下命令设置接口：

```
ifconfig sn0 local0
ifconfig sn1 local1
case "`uname -r`" in 2.0.*)
    route add -net snulnet0 dev sn0
    route add -net snulnet1 dev sn1
esac
```

不需要在 2.2 和其后内核上调用 route，因为路由会自动添加。同时，如果你选择的地址不是 C 类地址，则需要添加网络掩码参数，即 255.255.255.0。

至此，接口的“远程”端就可到达了。下面给出的屏幕输出说明了主机是如何通过 snul 接口到达 remote0 和 remote1 的。

```
morgana% ping -c 2 remote0
64 bytes from 192.168.0.99: icmp_seq=0 ttl=64 time=1.6 ms
64 bytes from 192.168.0.99: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss

morgana% ping -c 2 remote1
64 bytes from 192.168.1.88: icmp_seq=0 ttl=64 time=1.8 ms
64 bytes from 192.168.1.88: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss
```

注意，我们无法到达属于这两个网络的任意其它“主机”，这是因为当地址被修改且数据包被接收到时，计算机就会将该数据包丢弃。比如，离开 sn0 发送到 192.168.0.32 的数据包，将重新出现在 sn1 接口，但目标地址已修改成为 192.168.1.32，这并不是主机的本地地址。

14.1.2 数据包的物理传输

对数据传输而言，snul 接口属于以太网类型。

snul 模拟以太网，是因为大量已有的网络（至少一台工作站连接到的网段）基于以太网技术，比如 10baseT、100baseT，或者千兆以太网等等。另外，内核为以太网设备提供了一些通用支持，我们没有理由不利用这些通用的支持。成为一个以太网设备的优点如此出众，以至于 plip 接口（使用打印机端口的接口）也将自己声明为一个以太网设备。

对 snul 来说，使用以太网的最后一个优点是，我们可以在该接口上运行 tcpdump 看到数据包的传输情况。利用 tcpdump 观察接口，是一种了解这两个接口工作情况的便捷途径。（注意在 2.0 内核上，tcpdump 不能正常工作，除非 snul 接口显示为 ethx。在装载该驱动程序时，可传递 eth=1 选项，这样就可以使用通常的以太网名称，而不是 snx 名称。）

先前曾提到，snul 只能利用 IP 数据包。这一限制源于如下事实：为了我们的示例代码正常工作，

snnull 需要监听数据包，甚至修改数据包。代码要修改每个数据包 IP 头中的源、目标以及校验和，但不会检查数据包是否真正传送 IP 信息。这种“快速而恶劣的”数据修改，会破坏非 IP 数据包。如果读者希望通过 **snnull** 传送其它协议，则需要修改模块源代码。

14.2 连接到内核

下面我们剖析 **snnull** 的源代码，并开始分析网络驱动程序的结构。如果手头有若干个实际驱动程序的源代码，则能帮助读者跟上我们的讨论，并看到真实世界中，Linux 网络驱动程序的工作情况。为此，我们推荐读者首先阅读 **loopback.c**、**plip.c** 和 **3c509.c**，这些驱动程序的复杂性是逐渐递增的。另外，**skeleton.c** 也可以帮助我们理解网络驱动程序的结构，但该驱动程序只是个示例，不能真正运行。上述这些文件均保存在内核源代码树的 **drivers/net** 目录中。

14.2.1 模块的装载

当一个模块被装载到正在运行的内核中时，它要请求资源并提供一些功能设施，这点上，网络驱动程序也一样，而且在资源请求的方式上也没有任何不同。驱动程序要按照第 9 章“安装中断处理程序”中讲到的方法探测其设备和硬件位置（I/O 端口及 IRQ 线），但不需要进行注册。网络驱动程序在其模块初始化函数中的注册方法，和字符驱动程序及块驱动程序不同。因为对网络接口来讲，没有和主设备号及次设备号等价的东西，所以，网络驱动程序不必请求这种设备号。相反，驱动程序对每个新检测到的接口，向全局的网络设备链表中插入一个数据结构。

每个接口由一个 **struct net_device** 结构描述。**snnull** 的两个接口，即 **sn0** 和 **sn1**，它们的 **struct net_device** 定义如下：

```
struct net_device snnull_devs[2] = {
    { init: snnull_init, }, /* init, nothing more */
    { init: snnull_init, }
};
```

上面的初始化看起来相当简单——它只设置了一个成员。实际上，**net_device** 结构很是巨大，我们要在稍后填充其它的成员。但在这个阶段，没有必要讲解整个结构，相反，我们会在用到每个成员时进行解释。感兴趣的读者，可在 `<linux/netdevice.h>` 中找到这个结构的定义。

我们首先看 **struct net_device** 结构的 **name** 成员，其中保存了该接口名称（用来标识接口的字符串）。驱动程序可为接口定义硬编码的名称，也可以动态赋值。动态赋值的工作过程如下：如果名称中包含 **%d** 格式字符串，则会用某个小整数替换该格式字符串，并使用找出的第一个可用名称。这样，如果 **name** 被设定为 **eth%d**，将使用第一个可用的 **ethn** 名称——第一个接口称为 **eth0**，然后其它接口依次定义为 **eth1**、**eth2** 等等。**snnull** 的接口默认称为 **sn0** 和 **sn1**。但是，如果在装载时指定 **eth=1** 选项（导致整型变量 **snnull_eth** 设置为 1），**snnull_init** 就会使用动态赋值，如下所示：

```
if (!snnull_eth) { /* call them "sn0" and "sn1" */
    strcpy(snull_devs[0].name, "sn0");
    strcpy(snull_devs[1].name, "sn1");
} else { /* use automatic assignment */
    strcpy(snull_devs[0].name, "eth%d");
    strcpy(snull_devs[1].name, "eth%d");
}
```

另外一个成员是我们已初始化的成员，即 `init`，它是一个函数指针。不管何时注册设备，内核都会请求驱动程序对其本身进行初始化。初始化意味着要探测物理接口，并用正确的值填充 `net_device` 结构（下面的小节中讲述）。如果初始化失败，则结构不会被链接到网络设备的全局链表中。这种特殊的设置方式，在系统引导阶段非常有用——所有的驱动程序都试图注册自己的设备，但只有真正存在的设备才会被链接到设备链表中。

因为实际的初始化在别处执行，所以初始化函数本身要做的工作很少，而只有一条语句：

```
for (i=0; i<2; i++)
    if ( (result = register_netdev(snull_devs + i)) )
        printk("snull: error %i registering device \"%s\"\n",
            result, snull_devs[i].name);
    else device_present++;
```

14.2.2 初始化每个设备

设备的探测工作应该在接口的 `init` 函数中执行（经常称为“`probe`”函数）。`init` 函数接收的第一个参数是指向欲初始化设备的指针，其返回值要么是 0，要么是负的错误号，通常是 `-ENODEV`。

对 `snull` 接口来讲，没有真正要做的探测工作，因为它根本没有绑定到任何硬件。在我们为实际接口编写实际的驱动程序时，可使用通常的设备探测方法，这主要依赖于所使用的外设总线。同时，应该避免在这时注册 I/O 端口和中断线。硬件注册工作应该延迟到打开设备的时候，这在共享中断线的情况下尤其重要。我们并不希望每次在其它设备触发 `IRQ` 线时自己却被调用，而仅仅说一句“不，这不是我。”

初始化例程的主要功能是填充该设备的 `dev` 结构。注意对网络设备而言，该结构始终在运行时设置。因为网络接口的探测方式不同，所以无法采用和 `file_operations` 或 `block_device_operations` 结构一样的方式，即在编译阶段进行 `dev` 结构的设置。这样，在从 `dev->init` 中返回时，`dev` 结构中就应该填充了正确的值。幸运的是，内核提供了 `ether_setup` 函数，可处理某些以太网相关的默认设置，因此，调用这个函数，可填充 `struct net_device` 结构的一些成员。

`snull_init` 函数的核心代码如下：

```
ether_setup(dev); /* assign some of the fields */

dev->open          = snull_open;
dev->stop          = snull_release;
dev->set_config     = snull_config;
dev->hard_start_xmit = snull_tx;
dev->do_ioctl      = snull_ioctl;
dev->get_stats     = snull_stats;
dev->rebuild_header = snull_rebuild_header;
dev->hard_header    = snull_header;
#ifdef HAVE_TX_TIMEOUT
dev->tx_timeout     = snull_tx_timeout;
dev->watchdog_timeo = timeout;
#endif
/* keep the default flags, just add NOARP */
dev->flags          |= IFF_NOARP;
dev->hard_header_cache = NULL; /* Disable caching */
SET_MODULE_OWNER(dev);
```


上述代码中唯一不同寻常的地方是设置 `IFF_NOARP` 标志的代码行。这行代码指定该接口不使用 ARP，即地址解析协议。ARP 是一种底层的以太网协议，用来将 IP 地址转换成以太网介质访问控制（Ethernet Medium Access Control, MAC）地址。因为 `snull` 模拟的“远程”系统并不真正存在，因此没有人会应答这些远程系统的 ARP 请求。我们没有选择添加额外的 ARP 实现使得 `snull` 复杂化，而是简单地将接口标记为不能处理该协议。对 `hard_head_cache` 的赋值也出于同样的原因：赋值为 `NULL`，将禁止该接口上的 ARP 请求缓存。相关内容将在本章后面的“MAC 地址解析”一节中讨论。

初始化代码还设置了一些用来处理传输超时的成员（`tx_timeout` 和 `watchdog_timeo`）。在本章后面的“传输超时”一节中，我们将完整介绍相关内容。

最后，代码调用了 `SET_MODULE_OWNER`，这个调用将初始化 `net_device` 结构的 `owner` 成员，并设置为指向模块本身的指针。内核使用这一成员的方式，和它使用 `file_operations` 结构 `owner` 成员的方式一模一样——用来维护模块的使用计数。

这里需要对 `struct net_device` 的一个成员作进一步解释。该成员的作用和字符驱动程序中 `private_data` 指针的作用类似。但和 `fops->private_data` 不同，`priv` 指针是在初始化阶段分配的，而不是打开阶段，这是因为 `priv` 所指向的数据项通常包含了接口上的统计信息。因为用户希望在任何时刻（即使在接口停止工作时）都能够调用 `ifconfig` 获得统计数据，所以，统计信息应该始终可用。不在 `open` 阶段分配，而在初始化阶段分配 `priv` 所引起的内存浪费无关紧要，因为大多数已探测到的接口会立即开始工作并运行。`snull` 模块为 `priv` 成员声明了 `snull_priv` 数据结构：

```
struct snull_priv {
    struct net_device_stats stats;
    int status;
    int rx_packetlen;
    u8 *rx_packetdata;
    int tx_packetlen;
    u8 *tx_packetdata;
    struct sk_buff *skb;
    spinlock_t lock;
};
```

这个结构包含了一个 `struct net_device_stats` 实例，它是保存接口统计信息的标准地方。下面的代码行分配和初始化 `dev->priv`：

```
dev->priv = kmalloc(sizeof(struct snull_priv), GFP_KERNEL);
if (dev->priv == NULL)
    return -ENOMEM;
memset(dev->priv, 0, sizeof(struct snull_priv));
spin_lock_init(&((struct snull_priv *) dev->priv)->lock);
```

14.2.3 模块的卸载

在卸载 `snull` 模块时，没有什么特殊的事情需要完成。模块的清除函数在释放私有结构所使用的内存之后，将自己从全局设备链表中注销：

```
void snull_cleanup(void)
{
    int i;
```

```
for (i=0; i<2; i++) {
    kfree(snull_devs[i].priv);
    unregister_netdev(snull_devs + i);
}
return;
}
```

14.2.4 模块化和非模块化的驱动程序

尽管字符和块驱动程序都不在乎它们是模块还是连接到了内核，但对网络驱动程序来讲，情况就不一样了。

当驱动程序直接连接到 Linux 内核时，它不会声明自己的 `net_device` 结构，而要使用 `drivers/net/Space.c` 中声明的结构。`Space.c` 声明了一个所有网络设备的链表，既包含驱动程序相关的结构，比如 `plip1`，也包含通用的 `eth` 设备。以太网驱动程序根本不关心它们的 `net_device` 结构，因为它们使用的是通用的结构。这种一般性的 `eth` 设备结构声明 `ethif_probe` 作为它们的 `init` 函数。希望在主流内核当中插入新以太网接口的程序员，只需在 `ethif_probe` 中添加对驱动程序初始化函数的调用即可。另一方面，非以太网接口的驱动程序作者，需要将他们的 `net_device` 结构插入 `Space.c` 文件。如果驱动程序必须连接到内核，则不管是哪种情况，都只需修改 `Space.c` 源文件。

在系统引导时，网络初始化代码遍历所有的 `net_device` 结构，并调用它们的探测函数（即 `dev->init`），同时传递给指向设备本身的指针。如果探测函数成功了，内核会初始化下一个 `net_device` 结构。这种驱动程序的处理方式允许增量式地赋予设备以 `eth0`、`eth1` 等名称，而无需修改每个设备的 `name` 成员。

另一方面，在装载模块化的驱动程序时，即使它所控制的接口是以太网接口，也要声明自己的 `net_device` 结构（如同本章所描述的）。

感兴趣的读者可阅读 `Space.c` 和 `net_init.c` 获得接口初始化相关的详细信息。

14.3 `net_device` 结构的细节

`net_device` 结构位于网络驱动程序层的最核心地位，因此值得对它进行完整的描述。但是，第一次阅读时，读者可以跳过这个小节，因为你无需彻底理解这个数据结构。这个小节将描述所有的成员，但更倾向于提供一个参考而不是用来记忆。本章其余部分将在成员出现在示例程序时进行简要的描述，这样，读者就不必总要返回到这个小节来查阅。

`struct net_device` 可从概念上划分为两部分：可见部分和不可见部分。该结构的可见部分由可在静态 `net_device` 结构中进行显式赋值的成员组成。`drivers/net/Space.c` 中的所有结构都是这样初始化的，而没有采用标记化的结构初始化语法。其余的成员由网络代码内部使用，通常不在编译阶段初始化，也不用标记化的初始化方法。驱动程序可以访问某些成员（例如，初始化期间进行赋值的那些成员），但不能修改其它一些成员。

14.3.1 可见的成员

`struct net_device` 的第一部分由如下成员组成（顺序给出）：

```
char name[IFNAMSIZ];
```

设备名称。如果名称中包含 `%d` 格式化字符串，则使用给定基本名称上的第一个可用设备名，编号从零开始。

```
unsigned long rmem_end;
unsigned long rmem_start;
unsigned long mem_end;
unsigned long mem_start;
```

设备内存信息。这些成员保存了设备使用的共享内存之起始和终止地址。如果该设备具有不同的接收和传输内存，则 `mem` 成员用于传输内存，而 `rmem` 成员用于接收内存。`mem_start` 和 `mem_end` 可于系统引导期间在内核命令行指定，并由 `ifconfig` 命令查询。`rmem` 字段从来不会在驱动程序本身之外被引用。根据约定，`end` 成员的设置要保证 `end - start` 等于可用的板卡内存量。

```
unsigned long base_addr;
```

网络接口的 I/O 基地址。这个成员和前述成员类似，要在设备探测阶段赋值。`ifconfig` 命令可显示或修改当前值。`base_addr` 也可在系统引导期间，或在装载期间在命令行显式赋值。和前面的内存成员类似，内核不会使用该成员。

```
unsigned char irq;
```

被赋予的中断号。在列出接口时，`ifconfig` 命令将打印 `dev->irq` 的值。这个值通常在引导或装载阶段设置，其后可利用 `ifconfig` 修改。

```
unsigned char if_port;
```

指定在多端口设备上使用哪个端口。举例来说，如果设备同时支持同轴电缆（`IF_PORT_10BASE2`）和双绞线（`IF_PORT_10BASET`）以太网连接时，可使用该成员。完整的已知端口类型在 `<linux/netdevice.h>` 中定义。

```
unsigned char dma;
```

为设备分配的 DMA 通道。该成员只对某些外设总线有用，比如 ISA。除了用于显示信息（`ifconfig` 命令）之外，不会在设备驱动程序之外使用这个成员。

```
unsigned long state;
```

设备状态。这个成员包含有若干标志。驱动程序通常无需直接操作这些标志，相反，内核提供了一组工具函数。在讲述驱动程序操作时，我们将讨论这些函数。

```
struct net_device *next;
```

指向全局链表下一个设备的指针。驱动程序不应该修改这个成员。

```
int (*init)(struct net_device *dev);
```

先前描述过的初始化函数。

14.3.2 隐藏的成员

`net_device` 结构包含了许多附加成员，这些成员通常在设备初始化时赋值。其中一些成员含有接

口相关的信息，而另外一些仅仅用来为驱动程序提供便利（也就是说，内核不会使用这些成员）；其它成员，尤其是设备方法，则是内核/驱动程序接口的一部分。

我们将单独列出三组成员。列出的成员不再按照实际的顺序给出，因为顺序并不重要。

接口信息

大部分接口相关的信息可由 `ether_setup` 函数正确设置。以太网卡可利用这个通用函数设置大部分成员，但 `flags` 和 `dev_addr` 字段是设备特有的，因此必须在初始化期间显式赋值。

某些非以太网接口也可以使用类似 `ether_setup` 这样的辅助函数。`drivers/net/net_init.c` 导出了一些类似的函数，如下所示：

```
void ltalk_setup(struct net_device *dev);
```

设置 LocalTalk 设备的字段。

```
void fc_setup(struct net_device *dev);
```

初始化光纤通道设备。

```
void fddi_setup(struct net_device *dev);
```

配置光纤分布式数据接口（Fiber Distributed Data Interface, FDDI）网络的接口。

```
void hippi_setup(struct net_device *dev);
```

初始化高性能并行接口（High-Performance Parallel Interface, HIPPI）的高速互连驱动程序的成员。

```
void tr_configure(struct net_device *dev);
```

处理令牌环网络接口的设置。注意 2.4 内核也导出了一个 `tr_setup` 函数，可怜的是，这个函数什么也不做。

大部分设备可划分到上述类型当中。但是，如果你的驱动程序是崭新的、完全不同于上述这些接口类型，则需要对下面的字段进行手工赋值。

```
unsigned short hard_header_len;
```

硬件头的长度，即数据包中位于 IP 头，或者其它协议信息之前的 `octet` 数目。对以太网接口，`hard_header_len` 的值是 14（`ETH_HLEN`）。

```
unsigned mtu;
```

最大传输单元（MTU）。网络层使用该成员驱动数据包的传输。以太网的 MTU 是 1500 个 `octet`（`ETH_DATA_LEN`）。

```
unsigned long tx_queue_len;
```

可在设备的传输队列中排队的最大帧数目。`ether_setup` 将该成员设置为 100，但我们也可以修改它。例如，为避免浪费系统内存，`plip` 使用 10（比起实际的以太网接口，`plip` 的吞吐率要低些）。

```
unsigned short type;
```

接口的硬件类型。ARP 使用 `type` 成员判断接口所支持的硬件地址类型。以太网接口的正确值是 `ARPHRD_ETHER`，这也是 `ether_setup` 所设置的值。可识别的类型在 `<linux/if_arp.h>` 中定义。

```
unsigned char addr_len;
unsigned char broadcast[MAX_ADDR_LEN];
unsigned char dev_addr[MAX_ADDR_LEN];
```

硬件（MAC）地址长度以及设备的硬件地址。以太网地址长度是 6 个 octet（即接口板卡的硬件 ID），广播地址由 6 个 0xff octet 组成。ether_setup 会对上述值进行正确的设置。另一方面，设备地址必须从接口板卡中以设备特有的方式读取，因此，驱动程序要负责将该地址复制到 dev_addr。在数据包交给驱动程序传输之前，要利用硬件地址生成正确的以太网数据包头。snul 不使用物理接口，从而使用的是它自己设定的硬件地址。

```
unsigned short flags;
```

接口标志，下面详细介绍。

该标志成员是一个包含如下位值的位掩码。IFF_ 前缀表示“接口标志”。某些标志由内核维护，而其它一些则由接口在初始化期间设置，用来声明接口的各种能力及其它特性。有效的标志定义在 <linux/if.h> 中，解释如下：

IFF_UP

对驱动程序，该标志只读。当接口被激活并可以开始传输数据包时，内核设置该标志。

IFF_BROADCAST

该标志说明接口允许广播。以太网卡是可广播的。

IFF_DEBUG

表示调试模式。该标志可用来控制用于调试目的的详细 printk 调用。尽管目前还没有正式的驱动程序使用该标志，但用户程序可通过 ioctl 设置或清除该标志，因此，你的驱动程序可利用这个标志。mics-progs/netifdebug 程序可用来打开或关闭该标志。

IFF_LOOPBACK

该标志只能对回环设备进行设置。内核检查 IFF_LOOPBACK 标志以判断接口是否为回环设备，而不是将 lo 作为特殊的接口名称进行判断。

IFF_POINTOPOINT

该标志表明接口连接到点对点链路。这个标志由 ifconfig 设置。例如，plip 和 PPP 驱动程序将设置该标志。

IFF_NOARP

该标志表明接口不能执行 ARP。例如，点对点接口不需要运行 ARP，????。snul 缺少 ARP 功能，因此设置了这个标志。

IFF_PROMISC

设置该标志将激活混杂模式。默认情况下，以太网接口使用一个硬件过滤器来确保它只接收广播数据包，以及直接发送到接口硬件地址的数据包。象 tcpdump 这样的数据包侦听器（sniffer）会在接口上设置混杂模式，以便检索到通过传输介质的所有数据包。

IFF_MULTICAST

该标志由能够进行组播（multicast）的接口设置。ether_setup 默认时设置 IFF_MULTICAST，因此，如果你的驱动程序不支持组播，必须在初始化时清除该标志。

IFF_ALLMULTI

该标志告诉接口接收所有的组播数据包。内核在主机执行组播路由时设置该标志，但仅仅在 **IFF_MULTICAST** 被设置的情况下。**IFF_ALLMULTI** 对接口来讲是只读的。我们将在本章后面的“组播”一节中看到组播标志的使用。

IFF_MASTER**IFF_SLAVE**

该标志由负载均衡代码使用。接口驱动程序无需了解该标志。

IFF_PORTSEL**IFF_AUTOMEDIA**

该标志表明设备能够在多种介质类型之间切换，例如，在非屏蔽双绞线（**UTP**）和同轴以太网电缆之间。如果 **IFF_AUTOMEDIA** 被设置，设备会自动选择正确的介质类型。

IFF_DYNAMIC

该标志表示接口地址可改变，拨号设备使用该标志。

IFF_RUNNING

该标志表示接口正在运行。该标志主要用于 **BSD** 兼容性，内核很少使用该标志。大多数网络驱动程序不需要关心 **IFF_RUNNING** 标志。

IFF_NOTRAILERS

Linux 不使用该标志，只是为了 **BSD** 兼容性。

在程序改变 **IFF_UP** 时，会调用 **open** 或 **stop** 设备方法。当 **IFF_UP** 或其它任意一个标志被修改，**set_multicast_list** 方法会被调用。如果驱动程序需要在标志被修改时执行一些动作，则必须在 **set_multicast_list** 中完成这些动作。例如，当 **IFF_PROMISC** 被设置或清除，**set_multicast_list** 必须通知板卡上的硬件过滤器。“组播”一节中将概述该设备方法的职责。

设备方法

和字符及块设备类似，每个网络设备要声明作用其上的函数。本节将给出可在网络接口上执行的操作，某些操作可保留为 **NULL**，其它一些无需修改，因为 **ether_setup** 将赋予适当的方法。

网络接口的设备方法可划分为两个类型：基本的和可选的。基本方法包括使用接口必需的方法；可选方法实现了一些更为高级的功能，但并不严格要求有这些方法。下面是基本方法：

```
int (*open)(struct net_device *dev);
```

打开接口。在 **ifconfig** 激活接口时，接口将被打开。**open** 方法应该注册所有的系统资源（I/O 端口、IRQ、DMA 等等），打开硬件并增加模块使用计数。

```
int (*stop)(struct net_device *dev);
```

停止接口。应该在该方法中执行打开期间执行的操作的反操作。

```
int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev);
```

该方法初始化数据包的传输。完整的数据包（协议头和数据）包含在一个套接字缓冲区（**sk_buffer**）结构中。套接字缓冲区将在本章后面介绍。

```
int (*hard_header)(struct sk_buff *skb, struct net_device *dev, unsigned short type,
```

```
void *daddr, void *saddr, unsigned len);
```

该函数根据先前检索到的源和目标硬件地址建立硬件头。该函数的任务是将作为参数传递进入的信息，组织成设备特有的适当硬件头。**eth_header** 是以太网类型接口的默认函数，**ether_setup** 将该成员赋值成 **eth_header**。

```
int (*rebuild_header)(struct sk_buff *skb);
```

该函数用来在传输数据包之前重新建立硬件头。以太网设备使用的默认函数使用 **ARP** 填充数据包中缺少的信息。2.4 内核中很少用到 **rebuild_header** 方法，而主要使用 **hard_header**。

```
void (*tx_timeout)(struct net_device *dev);
```

当数据包的传输在合理的时间段内失败，则假定丢失了中断或接口被锁住，这时，将调用该方法。这个方法应解决问题并重新开始数据包的传输。

```
struct net_device_stats *(*get_stats)(struct net_device *dev);
```

应用程序需要获得接口的统计信息时，将调用该方法。例如，在运行 **ifconfig** 或 **netstat -i** 命令时将利用该方法。我们将在本章后面的“统计信息中”看到 **snull** 的样例实现。

```
int (*set_config)(struct net_device *dev, struct ifmap *map);
```

改变接口配置。该方法是配置驱动程序的入口点。利用 **set_config**，可在运行中改变设备的 I/O 地址和中断号。在探测不到接口时，系统管理员可利用该功能指定设备资源。现代硬件的驱动程序通常不需要实现该方法。

其余的设备操作可看作是可选的方法。

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

执行接口特有的 **ioctl** 命令。本章后面的“定制 **ioctl** 命令”中描述了这些命令的实现。如果接口不需要实现任何接口特有的命令，则 **net_device** 中对应的成员可保持为 **NULL**。

```
void (*set_multicast_list)(struct net_device *dev);
```

当设备的组播列表改变了，或者设备标志改变时，将调用该方法。“组播”一节将详细描述该方法，并给出一个样例实现。

```
int (*set_mac_address)(struct net_device *dev, void *addr);
```

如果接口支持硬件地址的改变，则可实现该方法。许多接口根本不支持这种功能。其它接口使用默认的 **eth_mac_addr** 实现（在 **drivers/net/net_init.c** 中定义）。**eth_mac_addr** 仅仅将新地址复制到 **dev->dev_addr** 中，而且只能在接口不工作时进行设置。使用 **eth_mac_addr** 的驱动程序应该在配置时从 **dev->dev_addr** 中取出地址并设置其硬件的 **MAC** 地址。

在接口的 **MTU**（最大传输单元）改变时，该函数负责采取相应的动作。如果驱动程序在 **MTU** 改变时需要完成某些特定工作，则应该声明自己的函数，否则，默认的函数可正确实现相关处理。**snull** 实现了该方法，可作为模板参考。

```
int (*header_cache) (struct neighbour *neigh, struct hh_cache *hh);
```

head_cache 将根据 **ARP** 查询的结果填充 **hh_cache** 结构。几乎所有的驱动程序都可以使用默认的 **eth_header_cache** 实现。

```
int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev, unsigned char *haddr);
```

在发生变化时，该方法更新 `hh_cache` 结构中的目标地址。以太网设备使用 `eth_header_cache_update`。

```
int (*hard_header_parse)(struct sk_buff *skb, unsigned char *haddr);
```

`hard_header_parse` 方法从 `skb` 中包含的数据包中获得源地址，并将其复制到位于 `haddr` 的缓冲区。该函数的返回值是地址的长度。以太网设备通常使用 `eth_header_parse`。

工具成员

其余的 `struct net_device` 数据成员由接口使用，保存一些有用的状态信息。某些成员由 `ifconfig` 和 `netstat` 使用，以便为用户提供当前的配置信息。因此，接口应该给这些成员赋予适当的值。

```
unsigned long trans_start;
unsigned long last_rx;
```

这些成员都保存一个 `jiffies` 值。驱动程序分别在传输开始及接收到数据包时负责更新这些值。网络子系统使用 `trans_stat` 值检测传输器是否被锁住。`last_rx` 当前未使用，但驱动程序应该维护这个成员，以准备为将来使用。

```
int watchdog_timeo;
```

在网络层确定传输已经超时，并且调用驱动程序的 `tx_timeout` 函数之前的最小时间（`jiffies` 为单位）。

```
void *priv;
```

和 `filp->private_data` 等价。驱动程序拥有该指针，可随意使用。通常，私有数据结构中含有一个 `struct net_device_stats` 项。本章后面的“初始化每个设备”一节中使用这个成员。

```
struct dev_mc_list *mc_list;
int mc_count;
```

上面这两个成员用来处理多点传输。`mc_count` 是 `mc_list` 所包含的项的数目。详细信息，可参阅“多点传输”一节。

```
spinlock_t xmit_lock;
int xmit_lock_owner;
```

`xmit_lock` 用来避免对驱动程序 `hard_start_xmit` 函数的多次并行调用。`xmit_lock_owner` 是获得 `xmit_lock` 的 CPU 编号。驱动程序不应改变这些成员。

```
struct module *owner;
```

“拥有”该设备结构的模块，用来维护模块的使用计数。

`struct net_device` 中还有其它一些成员，但网络驱动程序不使用这些成员。

14.4 打开和关闭

我们的驱动程序可在装载阶段或内核引导阶段探测接口。但是，在接口能够传送数据包之前，内核必须打开接口并赋予其地址。内核可在响应 `ifconfig` 命令时将打开或关闭一个接口。

在使用 `ifconfig` 向接口赋予地址时，要执行两个任务。首先，它通过 `ioctl(SIOCSIFADDR)` 赋予地址，其中 `SIOCSIFADDR` 表示套接字 I/O 控制集接口地址（Socket I/O Control Set Interface

Address)。然后，它通过 `ioctl(SIOCSIFFLAGS)` 设置 `dev->flag` 中的 `IFF_UP` 标志以打开接口，其中 `SIOCSIFFLAGS` 表示套接字 I/O 控制集接口标志（Socket I/O Control Set Interface Flags）。

对设备而言，无需对 `ioctl(SIOCSIFADDR)` 做任何工作。内核不会调用任何驱动程序函数，也即，该任务由内核来执行，是设备无关的。而后一个命令（`ioctl(SIOCSIFFLAGS)`）会调用设备的 `open` 方法。

类似地，在接口被关闭时，`ifconfig` 使用 `ioctl(SIOCSIFFLAGS)` 来清除 `IFF_UP` 标志，然后，`stop` 方法被调用。

这两个设备方法在成功时均返回 0，而在失败时和通常一样，返回负值。

对实际代码而言，驱动程序必须执行许多和字符及块设备相同的任务。`open` 请求必要的系统资源，并告诉接口开始工作；`stop` 关闭接口并释放系统资源。但是，除此之外，还要执行其它一些步骤。

首先，在接口能够和外界通讯之前，要将硬件地址从硬件设备复制到 `dev->dev_addr`。硬件地址可在探测期间或者打开期间赋值，这取决于驱动程序本身。`snul` 软件接口在 `open` 时赋予硬件地址——它其实使用了一个长度为 `ETH_ALEN` 的 ASCII 字符串作为假的硬件地址，其中 `ETH_ALEN` 是以太网硬件地址的长度。

一旦准备好开始发送数据后，`open` 方法还应该启动接口的传输队列（允许接口接受传输数据包）。内核提供的如下函数可启动该队列：

```
void netif_start_queue(struct net_device *dev);
```

`snul` 的 `open` 代码如下所示：

```
int snul_open(struct net_device *dev)
{
    MOD_INC_USE_COUNT;

    /* request_region(), request_irq(), .... (like fops->open) */

    /*
     * Assign the hardware address of the board: use "\0SNULx", where
     * x is 0 or 1. The first byte is '\0' to avoid being a multicast
     * address (the first byte of multicast adrs is odd).
     */
    memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
    dev->dev_addr[ETH_ALEN-1] += (dev - snul_devs); /* the number */

    netif_start_queue(dev);
    return 0;
}
```

读者已经看到，在缺少实际硬件的情况下，`open` 方法要做的事情很少。对 `stop` 方法而言，也是这样，它只是 `open` 的反操作。出于这个原因，实现 `stop` 的函数经常被称为 `close` 或 `release`。

```
int snul_release(struct net_device *dev)
{
    /* release ports, irq and such -- like fops->close */
```

```

netif_stop_queue(dev); /* can't transmit any more */
MOD_DEC_USE_COUNT;
return 0;
}

```

函数：

```
void netif_stop_queue(struct net_device *dev);
```

是 `netif_start_queue` 的对立面，它标记设备不能传输其它数据包。在接口被关闭时（在 `stop` 方法中），必须调用该函数，但是，该函数也可以用来临时停止传输，我们将在下一节讲述相关内容。

14.5 数据包传输

网络接口所执行的最重要任务是数据的传输和接收。我们首先讨论传输，因为数据的传输相对容易理解一些。

无论何时内核要传输一个数据包，它会调用 `hard_start_transmit` 方法将数据放入外发队列。内核处理后的每个数据包位于一个套接字缓冲区结构（`struct sk_buff`），该结构定义在 `<linux/skbuff.h>` 中。这个结构的名称来自于表示网络连接的 Unix 抽象物，即套接字（`socket`）。尽管接口无需处理套接字，但每个网络数据包属于更高网络层的某个套接字，而且所有套接字的输入/输出缓冲区都是 `struct sk_buff` 结构形成的链表。同一个 `sk_buff` 结构还用主机网络数据以及所有的 Linux 网络子系统，但是，对接口而言，套接字缓冲区仅仅是一个数据包而已。

指向 `sk_buff` 的指针通常称为 `skb`，因此，我们将在代码和正文中使用这个叫法。

套接字缓冲区是一个复杂的结构，内核提供了许多用来操作该结构的函数。我们将在“套接字缓冲区”一节中描述这些函数，现在，我们只需了解一些关于 `sk_buff` 的基本概念，就能编写一个可工作的驱动程序。

传递到 `hard_start_xmit` 的套接字缓冲区包含了物理数据包（以它在介质上的格式），并拥有完整的传输层数据包头。接口无需修改要传输的数据。`skb->data` 指向要传输的数据包，而 `skb->len` 是以 `octet` 为单位的长度。

下面是 `snull` 的数据包传输代码。实现传输的物理机制在另外一个单独的函数中实现，这是因为每个接口驱动程序都必须根据其驱动的特有硬件实现这段代码。

```

int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
    int len;
    char *data;
    struct snull_priv *priv = (struct snull_priv *) dev->priv;
    len = skb->len < ETH_ZLEN ? ETH_ZLEN : skb->len;
    data = skb->data;
    dev->trans_start = jiffies; /* save the timestamp */

    /* Remember the skb, so we can free it at interrupt time */
    priv->skb = skb;
}

```

```

/* actual delivery of data is device specific, and not shown here */
snnull_hw_tx(data, len, dev);

return 0; /* Our simple device cannot fail */
}

```

这样，该传输函数只执行了一些数据包的一致性检查，然后通过硬件相关的函数传输数据。我们在这里忽略 `snnull_hw_tx` 函数的原因，是因为其中尽是 `snnull` 设备的欺骗性代码（包括操作源和目标地址），从而对实际网络驱动程序编写者来讲意义不大。当然，如果读者对此感兴趣，也可以从示例代码中看到这个函数的完整实现。

14.5.1 控制并发传输

`hard_start_xmit` 函数通过 `net_device` 结构中的一个自旋锁（`xmit_lock`）获得并发调用时的保护。但是，在该函数返回后，有可能再次被调用。当软件指示硬件开始传输数据包之后，该函数返回，但是硬件传输可能尚未结束。这对 `snnull` 来说不是问题，因为它利用 CPU 完成所有的工作，因此，在传输函数返回时，数据包的传输已经结束。

另一方面，实际的硬件接口却是异步传输数据包的，而且可用来保存外发数据包的存储空间非常有限。在内存被耗尽时（对某些硬件，也许单个外发数据包的传输就会使内存耗尽），驱动程序需要告诉网络系统在硬件能够接受新数据之前，不能启动其它的数据包传输。

调用 `netif_stop_queue` 可完成这一通知。前面我们在停止队列时介绍过这个函数。在驱动程序停止队列之后，它必须在将来的某个时刻，当硬件能够再次接受数据包的传输时，重新启动该队列。为此，应调用：

```
void netif_wake_queue(struct net_device *dev);
```

这个函数除了通知网络系统可开始传输数据包以外，和 `netif_start_queue` 函数一样。

许多现代的网络接口在传输多个数据包时，维护一个内部的队列，这样可以获得最好的网络性能。这种设备的网络驱动程序在任意时刻都可支持多个外发传输数据包，但不管设备是否支持多个外发传输数据包，设备内存都会被填满。一旦设备内存填充到容不下最大可能的数据包的时候，驱动程序应该停止队列，直到空间再次可用为止。

14.5.2 传输超时

大部分处理实际硬件的驱动程序必须能够应付硬件偶尔不能正确响应的问题。接口也许会忘记它在做什么，或者系统有可能丢失中断。这种类型的问题在个人计算机上的某些设备中很常见。

许多驱动程序利用定时器处理这类问题；如果某个操作在定时器到期时还未完成，则认为出现了问题。网络系统，从本质上讲，是通过大量定时器控制的多个状态机的复杂组合。从这个角度上讲，网络代码能够自动检测传输超时。

这样，网络驱动程序无需自己检测这种问题。相反，驱动程序只需设置一个超时周期，并在 `net_device` 结构的 `watchdog_timeo` 成员中设置。这个周期以 `jiffies` 为单位，对通常的传输延迟（比如网络介质上因堵塞造成的冲突）来讲应该是足够长的。

如果当前的系统时间超过设备的 `trans_start` 时间至少一个超时周期，网络层将最终调用驱动程序的 `tx_timeout` 方法。这个方法的任务是完成解决超时问题而需要的任何工作，并确保正在进行的任何传输能够正常结束。驱动程序不能将网络代码已经提交的套接字缓冲区丢掉，这一点尤其重要。

`snull` 可模拟这种传输器被锁住的情形，可通过装载时的两个参数控制：

```
static int lockup = 0;
MODULE_PARM(lockup, "i");

#ifdef HAVE_TX_TIMEOUT
static int timeout = SNULL_TIMEOUT;
MODULE_PARM(timeout, "i");
#endif
```

如果装载驱动程序时指定 `lockup=n` 参数，则会在传输 `n` 个数据包之后模拟一次传输器硬件的锁住，并且 `watchdog_timeo` 成员将设置为给定的超时值。在模拟被锁住的情况时，`snull` 会调用 `netif_stop_queue` 以避免产生其它的传输请求。

`snull` 的传输超时处理器程序如下所示：

```
void snull_tx_timeout (struct net_device *dev)
{
    struct snull_priv *priv = (struct snull_priv *) dev->priv;

    PDEBUG("Transmit timeout at %ld, latency %ld\n", jiffies,
           jiffies - dev->trans_start);
    priv->status = SNULL_TX_INTR;
    snull_interrupt(0, dev, NULL);
    priv->stats.tx_errors++;
    netif_wake_queue(dev);
    return;
}
```

出现产生传输超时的情况下，驱动程序必须在接口统计信息中标记该错误，并要将设备重置为一个合理的状态，以便传输新的数据包。在 `snull` 中发生超时，驱动程序调用 `snull_interrupt` 填补“丢失”的中断，并调用 `netif_wake_queue` 重新启动传输队列。

14.6 数据包的接收

从网络上接收数据要比传输数据复杂一点，这是因为我们必须在中断处理程序中分配一个 `sk_buff` 并传递给上层处理。接收数据包的通常方法是通过中断，除非该接口是一个类似 `snull` 的纯软件接口或者回环接口。尽管有可能编写一个轮询的驱动程序，而且正式内核中也存在几个这样的驱动程序，但中断驱动的操作更好一些——不管是从吞吐率还是从计算需求上讲。因为大部分网络接口是中断驱动的，因此我们不会在这里讨论利用内核定时器实现的轮询驱动程序，

`snull` 的实现从设备无关的管理工作中将“硬件”细节隔离了出来。`snull_rx` 函数在硬件接收到数据包之后（数据包已经在计算机内存中了）被调用。`snull_rx` 接收一个指向数据的指针，以及数据包的长度。它还负责将数据包以及其它附加信息发送到上层的网络代码。这一代码和获得数据指针及其长度的途径无关。

```

void snull_rx(struct net_device *dev, int len, unsigned char *buf)
{
    struct sk_buff *skb;
    struct snull_priv *priv = (struct snull_priv *) dev->priv;

    /*
     * The packet has been retrieved from the transmission
     * medium. Build an skb around it, so upper layers can handle it
     */
    skb = dev_alloc_skb(len+2);
    if (!skb) {
        printk("snull rx: low on mem - packet dropped\n");
        priv->stats.rx_dropped++;
        return;
    }
    memcpy(skb_put(skb, len), buf, len);

    /* Write metadata, and then pass to the receive level */
    skb->dev = dev;
    skb->protocol = eth_type_trans(skb, dev);
    skb->ip_summed = CHECKSUM_UNNECESSARY; /* don't check it */
    priv->stats.rx_packets++;
    priv->stats.rx_bytes += len;
    netif_rx(skb);
    return;
}

```

该函数十分通用，从而可以作为真实网络驱动程序的一个模板，但是，在你使用这些代码段之前，需要一些必要的解释。

第一步是分配一个保存数据包的缓冲区。注意缓冲区的分配函数（`dev_alloc_skb`）需要知道数据长度。这个函数利用这一信息为缓冲区分配空间。`dev_alloc_skb` 以原子的优先权调用 `kmalloc`，因此可在中断期间安全使用。内核提供了分配套接字缓冲区的其它接口，但不值得在这里讨论。本章后面的“套接字缓冲区”一节中将详细解释套接字缓冲区。

一旦拥有一个合法的 `skb` 指针，上述代码调用 `memcpy` 将数据包数据复制到缓冲区中；`skb_put` 函数更新缓冲区中的数据尾（`end-of-data`）指针，并返回指向新建立空间的指针。

如果你正在为某个接口编写能够实现完整总线控制 I/O 的高性能驱动程序，则可以考虑一种优化可能性。某些驱动程序在传入数据包到达之前为他们分配套接字缓冲区，然后指示接口直接将数据包数据放入套接字缓冲区中。网络层与这种策略相配合，会在可进行 DMA 的空间中分配所有的套接字缓冲区。这样，可避免填充套接字缓冲区的额外复制操作，但是因为我们无法预先知道传入的数据包大小，所以必须谨慎处理缓冲区大小。在这种情况下，`chang_mtu` 方法的实现也很重要，因为它可以让驱动程序对最大数据包大小的改变做出适当的响应。

在能够处理数据包之前，网络层必须知道数据包的一些信息。为此，必须在将缓冲区传递到上层之前，对 `dev` 和 `protocol` 成员正确赋值。然后，我们需要指定如何求得校验和，或者已经在数据包上求得了校验和（`snull` 无需求得任何校验和）。`skb->ip_summed` 的可能策略如下所示：

CHECKSUM_HW

设备已经在硬件层求得了校验和。SPARC HME 接口是硬件校验和的一个例子。

CHECKSUM_NONE

校验和仍需验证，而且该任务必须由系统软件完成。对新分配的缓冲区，这是默认策略。

CHECKSUM_UNNECESSARY

不进行任何校验和的计算。这是 `snull` 和回环接口的策略。

最后，驱动程序更新其统计计数器，以记录已接收到一个数据包。统计结构中包含若干成员，最重要的有 `rx_packets`、`rx_bytes`、`tx_packets` 和 `tx_bytes`，其中包含了已接收和已发送的数据包个数，以及已传输的 `octet` 总量。本章后面的“统计信息”一节中将全面介绍所有的成员。

接收数据包过程中的最后一个步骤由 `netif_rx` 执行，它将套接字缓冲区传递给上层软件处理。

14.7 中断处理程序

大多数硬件接口通过中断处理程序来控制。接口在两种可能的事件下中断处理器：新数据包到达，或者外发数据包的传输已经完成。这种一般性的描述并不是永远有效，但的确说明了和异步数据包传输相关的所有问题。并行线路网际协议（**Parallel Line Internet Protocol, PLIP**）和点对点协议（**Point-to-Point Protocol, PPP**）是不适合上述一般性的两个特例。它们处理相同的事件，但底层中断处理有稍许不同。

通常的中断例程可检查物理设备上的一个状态寄存器，从而了解新数据包到达中断和传输完成中断之间的区别。`snull` 接口也以类似的方式工作，但是它的状态字是通过软件实现的，且保存在 `dev->priv` 中。一个网络接口的中断处理程序可如下所示：

```
void snull_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int statusword;
    struct snull_priv *priv;
    /*
     * As usual, check the "device" pointer for shared handlers.
     * Then assign "struct device *dev"
     */
    struct net_device *dev = (struct net_device *)dev_id;
    /* ... and check with hw if it's really ours */

    if (!dev /*paranoid*/ ) return;

    /* Lock the device */
    priv = (struct snull_priv *) dev->priv;
    spin_lock(&priv->lock);

    /* retrieve statusword: real netdevices use I/O instructions */
    statusword = priv->status;
    if (statusword & SNULL_RX_INTR) {
        /* send it to snull_rx for handling */
        snull_rx(dev, priv->rx_packetlen, priv->rx_packetdata);
    }
    if (statusword & SNULL_TX_INTR) {
        /* a transmission is over: free the skb */
        priv->stats.tx_packets++;
        priv->stats.tx_bytes += priv->tx_packetlen;
        dev_kfree_skb(priv->skb);
    }

    /* Unlock the device and we are done */
    spin_unlock(&priv->lock);
    return;
}
```

```
}

```

该处理程序的第一个任务是检索指向正确 `struct net_device` 的指针。该指针通常来自以参数形式接收到的 `dev_id` 指针。

该处理程序有意思的部分是对“传输结束”情形的处理。在这种情况下，统计信息要被更新，而且要调用 `dev_kfree_skb` 将（不再使用）的套接字缓冲区释放给系统。如果驱动程序临时停止了传输队列，就可以在这个时候调用 `netif_wake_queue` 重新启动队列。

另一方面，数据包的接收不需要其它任何特殊的中断处理。简单地调用 `snull_rx`（我们已经介绍过这个函数）就足够了。

14.8 链路状态的改变

网络连接要和本地系统之外的外界打交道，因此经常会受到外部事件的影响，而这些事件又可能是瞬时发生的。网络子系统需要了解网络链路是否正常，因而提供了驱动程序可以利用的几个函数。

大多数涉及实际的物理连接的网络技术提供载波状态信息；载波的存在意味着硬件功能是正常的。例如，以太网适配器能感知线路上的载波信号；当用户断开电缆时，载波消失，链路就不能正常工作了。默认情况下，网络设备假定存在载波信号。但是，利用下面的函数，驱动程序可显式改变这个状态：

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);

```

如果驱动程序检测出设备上不存在载波，则应该调用 `netif_carrier_off` 通知内核这一情况。当载波再次出现时，应调用 `netif_carrier_on`。某些驱动程序在发生重要的配置变化时（比如介质类型），也会调用 `net_carrier_off`；一旦适配器完成了本身的重置，就会检测到新的载波，而数据传输可以重新开始。

还存在一个返回整型的函数：

```
int netif_carrier_ok(struct net_device *dev);

```

该函数用来检测当前的载波状态（和设备结构中反映的状态一样）。

14.9 套接字缓冲区

我们已经讨论了大部分网络结构相关的问题，但尚未详细讲解 `sk_buff` 结构。该结构在 Linux 内核中处于网络子系统的核心地位，接下来我们介绍该结构的主要成员，以及用来操作这个结构的重要函数。

尽管没有严格要求读者理解 `sk_buff` 的内部，但是，在跟踪问题或者试图优化代码时，如果能够看懂该结构的内容，则会很有帮助。例如，在 `loopback.c` 中，你会发现作者在理解 `sk_buff` 内部细节的基础上对代码进行了优化。但也要注意如下警告：如果你编写的代码利用了 `sk_buff` 结构

的内部细节，则可能会在未来内核发布时出现问题。当然，有时性能上的好处和额外的维护成本是成正比的。

我们不会在这里描述整个结构，只会描述驱动程序可能会用到的那些成员。如果读者想了解更多的成员，可参考 `<linux/skbuff.h>` 文件，其中定义了这个结构以及操作该结构的函数原型。其它有关成员和函数用法的相关信息，也可通过 `grep` 内核源代码而获得。

14.9.1 重要成员

这里介绍的成员是驱动程序可能会访问到的成员（排列无特定顺序）。

```
struct net_device *rx_dev;
struct net_device *dev;
```

设备分别接收和发送该缓冲区。

```
union { /* ... */ } h;
union { /* ... */ } nh;
union { /*... */} mac;
```

指向数据包中各个层的报文头的指针。联合中的每个成员是指向不同数据结构类型的指针。`h` 中包含有传输层报文头（例如，`struct tcphdr *th`）的指针；`nh` 包含网络层报文头（比如 `struct iphdr *iph`）的指针；而 `mac` 中包含的是链路层报文头（例如 `struct ethdr *ethernet`）的指针。

如果驱动程序需要查询 TCP 数据包的源和目标地址，可在 `skb->h.th` 中找到这些地址。头文件中定义了可通过这种方式访问的完整报文头类型。

注意网络驱动程序要负责设置传入数据包的 `mac` 指针。这个任务通常由 `ether_type_trans` 函数处理，但是非以太网驱动程序需要直接设置 `skb->mac.raw`，我们将在“非以太网报文头”中说明。

```
unsigned char *head;
unsigned char *data;
unsigned char *tail;
unsigned char *end;
```

用来寻址数据包中数据的指针。`head` 指向已分配空间的开头，`data` 是有效 `octet`（通常要比 `head` 大一些）的开头，`tail` 是有效 `octet` 的结尾，而 `end` 指向 `tail` 可达到的最大地址。另外，我们还可以从这些指针得出可用缓冲区空间为 `skb->end - skb->head`，而当前已使用的数据空间为 `skb->tail - skb->data`。

```
unsigned long len;
```

数据本身的长度（`skb->tail - skb->data`）。

```
unsigned char ip_summed;
```

该数据包的校验和策略。该成员由驱动程序对传入数据包进行设置，我们在“数据包的接收”一节中有过讨论。

```
unsigned char pkt_type;
```

数据包类型，用于数据包的发送。驱动程序负责将其设置为 `PACKET_HOST`（该数据包是给我的）、`PACKET_BROADCAST`、`PACKET_MULTICAST`，或 `PACKET_OTHERHOST`（不，该数据包不是我的）。以太网驱动程序不必显式修改 `pkt_type`，因为 `eth_type_trans` 会完成这个工作。

该结构中的其余成员没有多少关注的意义。它们用来维护缓冲区链表、记录拥有缓冲区的套接字所占内存等等。

14.9.2 操作套接字缓冲区的函数

使用套接字缓冲区的网络设备通过一些正式的接口函数来操作该结构。操作套接字缓冲区的函数很多，下面是其中最重要的一些：

```
struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
```

分配一个缓冲区。`alloc_skb` 函数分配一个缓冲区并初始化 `skb->data` 和 `skb->tail` 为 `skb->head`。`dev_alloc_skb` 函数以 `GFP_ATOMIC` 优先级调用 `alloc_skb`，并在 `skb->head` 和 `skb->data` 之间保留一些空间，网络层使用这一数据空间进行优化工作，驱动程序不应修改这个空间。

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
```

释放一个缓冲区。`kfree_skb` 调用由内核内部使用。驱动程序应该使用 `dev_kfree_skb` 函数，在驱动程序上下文中调用该函数是安全的。

```
unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
```

上述内嵌函数更新 `sk_buff` 结构的 `tail` 和 `len` 成员。可用这些函数在缓冲区尾部添加数据。每个函数的返回值是 `skb->tail` 的先前值（换句话说，它指向刚刚建立的数据空间）。驱动程序可以使用该返回值，并通过调用 `ins(ioaddr, skb_put(...))` 或 `memcpy(skb_put(...), data, len)` 来复制数据。这两个函数之间的不同是，`skb_put` 会检查数据可安全放入缓冲区，而 `__skb_put` 忽略这个检查过程。

```
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
```

上述函数减少 `skb->data`，并增加 `skb->len`。它们类似 `skb_put`，除了数据添加在数据包的头部而不是尾部之外。返回值指向刚刚创建的数据空间。在传输数据包之前，可使用该函数添加硬件头。和前面一样，`__skb_push` 只在是否检查可用空间上和 `skb_push` 不同。

```
int skb_tailroom(struct sk_buff *skb);
```

该函数返回缓冲区中可用来“放入（put）”数据的可用空间总量。如果驱动程序在缓冲区中放入多于其能容纳的数据，则系统会出现 `panic`。尽管读者可能会反对说 `printk` 足够标记该错误了，但内存破坏对系统非常有害，因此内核开发人员决定在这种情况下采取最后的决定性动作，即 `panic`。实际情况下，如果正确分配了缓冲区，就不需要检查可用空间。因为驱动程序通常可在分配缓冲区之前获得数据包的大小，因此，只有被严重破坏的驱动程序才会向缓冲区中放入太多数据，这时，作为惩罚，内核就会出现 `panic`。

```
int skb_headroom(struct sk_buff *skb);
```

返回 `data` 之前可用的空间总量，也即，有多少 `octet` 能够“推给（push）”缓冲区。

```
void skb_reserve(struct sk_buff *skb, int len);
```

这个函数增加 `data` 和 `tail`。该函数可在填充缓冲区之前保留报文头空间（`headroom`）。大多数以太网接口在数据包之前保留 2 个字节，这样，IP 头可在 14 字节的以太网头之后，在 16 字节

边界上对齐。`snull` 也这样做了。我们在“数据包的接收”一节中，为了避免引入额外的概念，而没有提及这一点。

```
unsigned char *skb_pull(struct sk_buff *skb, int len);
```

从数据包头中拿出数据。驱动程序无需使用这个函数，包含该函数只是为了完整性。它减少 `skb->len` 并增加 `skb->data`；这是从传入数据包的头部剥离硬件头（以太网或等价硬件）所使用的方法。

内核定义了其它一些操作套接字缓冲区的函数，但这些函数主要用于上层网络代码，驱动程序不需要这些函数。

14.10 MAC 地址解析

以太网通讯中有一个有趣的问题，即如何将 IP 号和 MAC 地址（接口的唯一硬件 ID）关联起来。大部分协议也有类似的问题，但我们在这里重点讲述类以太网的情形。我们会提供有关该问题的完整描述，其中涉及到三种情形：ARP、无 ARP 的以太网头（类似 `plip`），以及非以太网头。

14.10.1 在以太网中使用 ARP

处理地址解析的通常方法是使用 ARP，即地址解析协议。幸运的是，ARP 由内核维护，而以太网接口不需要做任何特殊工作就能支持 ARP。只要在打开时正确设置 `dev->addr` 和 `dev->addr_len`，驱动程序就无需担心将 IP 号解析为物理地址这件事；`ether_setup` 会将正确的设备方法赋予 `dev->hard_header` 和 `dev->rebuild_header`。

尽管内核通常处理地址解析的细节（并缓存其结果），但是，它要调用接口的驱动程序来帮助建立数据包。毕竟驱动程序了解物理层数据包头的细节，而网络代码的作者试图将内核的其余部分和 ARP 隔离开来。为此，内核调用驱动程序的 `hard_header` 方法，将 ARP 查询的结果安排在数据包的适当位置。通常，以太网驱动程序编写者无需了解这个过程——通用的以太网代码会处理这一切。

14.10.2 重载 ARP

类似 `plip` 的简单点对点网络接口可从以太网头中获得一些好处，但可避免因来回发送 ARP 数据包而带来的开支。`snull` 中的示例代码也属于这种网络设备类型。`snull` 不能使用 ARP，这是因为驱动程序修改了正在传输的数据包中的 IP 地址，而同时 ARP 数据包也会调换 IP 地址。尽管我们可以实现一个简单的 ARP 应答生成器，但解释直接处理物理层数据包头的方法，要更为直观一些。

如果设备希望使用通常的硬件头，但不想运行 ARP，则需要重载默认的 `dev->hard_header` 方法。下面是 `snull` 对该方法的实现。

```
int snull_header(struct sk_buff *skb, struct net_device *dev,
                unsigned short type, void *daddr, void *saddr,
                unsigned int len)
{
    struct ethhdr *eth = (struct ethhdr *)skb_push(skb, ETH_HLEN);
```

```

eth->h_proto = htons(type);
memcpy(eth->h_source, saddr ? saddr : dev->dev_addr, dev->addr_len);
memcpy(eth->h_dest, daddr ? daddr : dev->dev_addr, dev->addr_len);
eth->h_dest[ETH_ALEN-1] ^= 0x01; /* dest is us xor 1 */
return (dev->hard_header_len);
}

```

这个函数根据内核提供的信息，然后将其格式化成标准的以太网头。它同时切换了目标以太网地址的一个位，其原因将在后面介绍。

在接口接收到一个数据包时，`eth_type_trans` 会以两种方式使用硬件头。我们已经在 `snull_rx` 中看到过这个函数：

```
skb->protocol = eth_type_trans(skb, dev);
```

该函数从以太网头中获得协议标识符（这里是 `ETH_P_IP`）；它还对 `skb->mac.raw` 进行赋值，并从数据包数据中删除硬件头（利用 `skb_pull`），并设置 `skb->pkt_type`。`skb->pkt_type` 在分配 `skb` 时赋予其默认值为 `PACKET_HOST`（表示这个数据包是发送到该主机的），而 `eth_type_trans` 会根据以太网的目标地址修改它。如果目标地址和接收它的接口地址不匹配，会将 `pkt_type` 设置为 `PACKET_OTHERHOST`。接下来，除非该接口处于混杂模式，`netif_rx` 会丢弃所有类型为 `PACKET_OTHERHOST` 的数据包。为此，`snull_header` 仔细地保证目标硬件地址和“接收”接口的地址是匹配的。

如果接口是点对点链路，则可能不希望接收非预期的组播数据包。为了避免这个问题，要知道第一个 `octet` 的最低位（LSB）为 0 的目标地址是发送到单个主机的（也就是说，它要么是 `PACKET_HOST`，要么是 `PACKET_OTHERHOST`）。`plip` 驱动程序使用 `0xfc` 作为其硬件地址的第一个 `octet`，而 `snull` 使用 `0x00`。这两个地址均可以在类以太网的点对点链路中工作。

14.10.3 非以太网头

我们刚刚看到硬件头中除目标地址之外，还包含其它一些信息，其中最重要的是通讯协议。现在我们介绍硬件头是如何封装相关信息的。如果读者需要了解细节，可从内核源代码或特定传输介质的技术文献中找到。大多数驱动程序编写者可忽略这个小节，而仅仅使用以太网的实现。

值得注意的是，并不是每个协议都必须提供所有的信息。类似 `plip` 的点对点链路或者 `snull` 可在不减少一般性的情况下避免传输整个以太网头。前面描述过的由 `snull_header` 实现的 `hard_header` 设备方法从内核接收发送信息——协议层和硬件地址。它还从 `type` 参数中接收 16 位的协议号；例如 IP 协议由 `ETH_P_IP` 标识。驱动程序希望能够向接收主机正确地递交数据包数据以及协议号。点对点链路可在硬件头中省略地址，而仅仅传输协议号，这是因为数据包的发送和源及目标地址无关。一个仅仅传输 IP 数据包的链路，甚至可以避免传输任何的硬件头。

当链路的另一端接收到数据包时，驱动程序的接收函数应该正确设置 `skb->protocol`、`skb->pkt_type` 和 `skb->mac.raw` 成员。

`skb->mac.raw` 是一个字符指针，由上层网络代码（例如，`net/ipv4/arp.c`）实现的地址解析机制使用。它必须指向与 `dev->type` 匹配的一个机器地址。设备类型的可能值定义在 `<linux/if_arp.h>` 中；以太网接口使用 `ARPHRD_ETHER`。作为示例，下面是 `eth_type_trans` 处理已接收数据包

的以太网头的代码：

```
skb->mac.raw = skb->data;
skb_pull(skb, dev->hard_header_len);
```

在最简单的情况下（没有头的点对点链路），`skb->mac.raw` 可指向一个静态的缓冲区，其中包含接口的硬件地址，而 `protocol` 可被设置位 `ETH_P_IP`，`packet_type` 可保留其默认值，即 `PACKET_HOST`。

因为每个硬件类型都是独特的，因此很难给出更多的介绍，但内核中尽是实例。例如，你可以参考 `AppleTalk` 驱动程序（`drivers/net/appletalk/cops.c`），红外驱动程序（`drivers/net/irda/smc_ircc.c`），或者 `PPP` 驱动程序（`drivers/net/ppp_generic.c`）。

14.11 定制 ioctl 命令

我们已经看到了在套接字上实现的 `ioctl` 系统调用；`SIOCSIFADDR` 和 `SIOCSIFMAP` 是“套接字 `ioctl`”的两个例子。现在我们讨论网络代码是如何使用该系统调用的第三个参数的。

当 `ioctl` 系统调用在某个套接字之上调用时，命令号是定义在 `<linux/sockios.h>` 中的某个符号，而函数 `sock_ioctl` 直接调用一个协议相关的函数（这里的“协议”指主要的网络协议，例如 `IP` 或 `AppleTalk`）。

任何协议层不能识别的 `ioctl` 命令会传递到设备层。这些设备相关的 `ioctl` 命令从用户空间接受第三个参数，即一个 `struct ifreq *` 指针。这个结构在 `<linux/if.h>` 中定义。`SIOCSIFADDR` 和 `SIOCSIFMAP` 命令实际利用了 `ifreq` 结构。`SIOCSIFMAP` 的其它参数，尽管被定义为 `ifmap`，但其实只是 `ifreq` 的一个成员。

除了使用标准化调用之外，每个接口可以定义它自己的 `ioctl` 命令。例如，`plip` 接口允许通过 `ioctl` 修改接口内部的超时值。套接字的 `ioctl` 实现能够识别 16 个接口私有的命令：从 `SIOCDEVPRIVATE` 到 `SIOCDEVPRIVATE+15`。

如果是上述这些命令，则会调用相关接口驱动程序的 `dev->do_ioctl`。该函数接收相同的 `struct ifreq *` 指针，其原型如下：

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

`ifr` 指针指向内核空间的地址，其中保存有用户传递的结构副本。在 `do_ioctl` 返回时，该结构将复制回用户空间；这样，驱动程序可使用私有命令来接收和返回数据。

设备特有的命令可选择使用 `struct ifreq` 中的成员，但是他们已经具有标准的含义，而且驱动程序也很难使这个结构适应自己的需求。`ifr_data` 成员是一个 `caddr_t` 型数据（一个指针），可用来满足设备特有的需求。驱动程序和调用 `ioctl` 的程序，要在 `ifr_data` 的使用上达成一致。例如，`pppstats` 使用设备特有的命令从 `ppp` 接口驱动程序中检索信息。

在这里说明 `do_ioctl` 的实现不太值得，但通过本章以及内核中的实例，读者应该能够编写一个满

足自己需求的 `do_ioctl` 函数。但要注意，`plip` 使用 `ifr_data` 的方法不正确，所以不应作为 `ioctl` 的实现样例。

14.12 统计信息

驱动程序需要的最后一个方法是 `get_stats`。这个方法返回设备统计结构的指针。它的实现非常简单；下面给出的这个实现甚至能够用于同一驱动程序管理多个接口的情况下，因为统计信息一般保存在设备的数据结构中。

```
struct net_device_stats *snull_stats(struct net_device *dev)
{
    struct snull_priv *priv = (struct snull_priv *) dev->priv;
    return &priv->stats;
}
```

用来返回实际统计信息的代码分布在的驱动程序中更新各种成员的许多地方。下面的清单给出了 `struct net_device_stats` 中最有意思的一些成员。

```
unsigned long rx_packets;
unsigned long tx_packets;
```

上述成员保存了接口成功传输的传入和传出数据包的总量。

```
unsigned long rx_bytes;
unsigned long tx_bytes;
```

接口接收和发送的字节总数。这两个成员在 2.2 内核中添加。

```
unsigned long rx_errors;
unsigned long tx_errors;
```

接收和发送错误的个数。在数据包传输过程中，可能出现无数错误，因此 `net_device_stats` 结构中包含有用于特定接收错误的 6 个计数器，以及用于发送错误的 5 个计数器。完整的清单可见 `<linux/netdevice.h>`。如果可能，驱动程序应该维护详细的错误统计，因为这对系统管理员跟踪问题非常有帮助。

```
unsigned long rx_dropped;
unsigned long tx_dropped;
```

在接收和发送过程中丢弃的数据包数目。在没有可用内存保存数据包数据时，会将数据包丢弃。`tx_dropped` 很少用到。

```
unsigned long collisions;
```

因介质拥堵而导致的冲突个数。

```
unsigned long multicast;
```

接收到的组播数据包个数。

这里需要重复的是，`get_stats` 方法可能会在任意时间调用——即使在接口被关闭时，因此，驱动程序不应在运行 `stop` 方法时释放统计信息。

14.13 组播

组播数据包是期望由多个主机、但不是所有主机接收的网络数据包。这一功能通过赋予针对一组主机的特殊硬件地址而完成。发送到这个特殊地址的数据包，应该由该组中的所有主机接收到。对以太网而言，组播地址的第一个 **octet** 的最低位设置为 1，而所有设备板卡将自己的硬件地址的相应位清零。

对主机分组和硬件地址的处理由应用程序和内核执行，而接口的驱动程序无需处理这些问题。

组播数据包的传输并不困难，因为它们看起来和其它数据包没有两样。接口在通讯介质上传输这些数据包，但并不关心它们的目标地址。内核负责赋予一个正确的硬件目标地址；如果定义了 **hard_header** 设备方法，则不必查看内核准备的数据。

内核在任意给定时刻均要跟踪组播地址。组播的主机清单可能会频繁改变，因为这是可在任意给定时间内运行的应用程序的功能，而且是用户的选择。驱动程序应该负责接受感兴趣的组播地址清单，然后将发送到这些地址的任意数据包交付给内核。驱动程序实现组播清单的方法，在某种程度上依赖于底层硬件的工作方式。通常来说，考虑组播时，硬件可划分为三类：

- 接口不能处理组播。这种接口要么接收直接发送到其硬件地址的数据包（包括广播数据包），要么接收所有数据包。它们只能通过接收所有数据包而接收组播的数据包，这样，就可能因为大量“不感兴趣”的数据包而使操作系统崩溃。我们通常不会将这类接口看成是能够进行组播的接口，因此，驱动程序不能在 **dev->flags** 中设置 **IFF_MULTICAST**。
点对点接口是一种特殊情况，因为它们不会执行任何硬件过滤而会接收所有数据包。
- 能够区分组播数据包和其它数据包（主机到主机或者广播）的接口。可指示这类接口接收每个组播数据包，并且让软件确定主机是否为有效的接收者。这种情况下引入的开支是能够接受的，因为典型网络中的组播数据包数目比较低。
- 能够为组播地址进行硬件检测的接口。可传递给这类接口一个可接收的组播地址清单，然后，接口将忽略其它的组播数据包。这对内核来讲是最好的情形，因为内核不需要在丢弃“不感兴趣的”数据包上浪费处理器时间。

因为第三种设备类型是最为常见的，所以，内核对这种类型的设备支持最好。为此，内核会在有效组播地址发生变化时通知驱动程序，并且将新的清单传递给驱动程序，这样，接口就可以根据新的信息更新其硬件过滤器。

14.13.1 对组播的内核支持

对组播数据包的支持由如下几项组成：一个设备方法、一个数据结构，以及若干设备标志。

```
void (*dev->set_multicast_list)(struct net_device *dev);
```

这个设备方法在设备相关的机器地址清单发生变化时调用。该方法还在 **dev->flags** 被修改时调用，因为某些标志（比如，**IFF_PROMISC**）也需要我们对硬件过滤器进行重新编程。这个方法接收一个指向 **struct net_device** 结构的指针并返回 **void**。如果驱动程序不想实现这个方法，可设置为 **NULL**。

```
struct dev_mc_list *dev->mc_list;
```

这是与设备关联的所有组播地址形成的一个链表。该结构的实际定义将在本节末尾讲述。

```
int dev->mc_count;
```

链表中的节点数目。这个信息有点冗余，但检查 `mc_count` 是否为零，是检查链表是否为空的一个便捷方法。

IFF_MULTICAST

除非驱动程序在 `dev->flags` 中设置这个标志，接口是不会请求处理组播数据包的。虽然如此，但当 `dev->flags` 发生变化时，`set_multicast_list` 方法会被调用，这是因为接口未被激活的时候，组播清单可能已经发生改变。

IFF_ALLMULTI

这个标志由网络软件在 `dev->flags` 中设置，用来告诉驱动程序检索来自网络的所有组播数据包。这在组播路由被使能时发生。如果设置了这个标志，`dev->mc_list` 不应该用来过滤组播数据包。

IFF_PROMISC

当接口被设置为混杂模式时在 `dev->flags` 中设置该标志。不管 `dev->mc_list` 中含有哪些主机地址，接口应该接收所有的数据包。

驱动程序开发人员需要的最后一点知识是 `struct mc_list` 结构的定义，该结构在 `<linux/netdevice.h>` 中定义。

```
struct dev_mc_list {
    struct dev_mc_list *next;      /* Next address in list */
    __u8 dmi_addr[MAX_ADDR_LEN]; /* Hardware address */
    unsigned char dmi_addrlen;    /* Address length */
    int dmi_users;                /* Number of users */
    int dmi_gusers;               /* Number of groups */
};
```

因为组播和硬件地址与实际的数据包传输是无关的，所以这个结构在各种网络实现上是可移植的，每个地址有一个 `octet` 的字符串组成，并定义了该地址的长度，这点和 `dev->dev_addr` 类似。

14.13.2 一个典型实现

描述 `set_multicast_list` 的最好方法是使用一些伪代码。

下面的函数是一个功能完整的（**full-featured**，故名为“**ff**”）驱动程序对该函数的一个典型实现。这个驱动程序所控制的接口具有一个复杂的硬件数据包过滤器，它可以保存主机能接收的组播地址表。表的最大尺寸是 `FF_TABLE_SIZE`。

具有 `ff_` 前缀的函数表示该函数是一个针对硬件的操作。

```
void ff_set_multicast_list(struct net_device *dev)
{
    struct dev_mc_list *mcptr;

    if (dev->flags & IFF_PROMISC) {
        ff_get_all_packets();
        return;
    }
    /* If there's more addresses than we handle, get all multicast
    packets and sort them out in software. */
}
```

```

if (dev->flags & IFF_ALLMULTI || dev->mc_count > FF_TABLE_SIZE) {
    ff_get_all_multicast_packets();
    return;
}
/* No multicast? Just get our own stuff */
if (dev->mc_count == 0) {
    ff_get_only_own_packets();
    return;
}
/* Store all of the multicast addresses in the hardware filter */
ff_clear_mc_list();
for (mc_ptr = dev->mc_list; mc_ptr; mc_ptr = mc_ptr->next)
    ff_store_mc_address(mc_ptr->dmi_addr);
ff_get_packets_in_multicast_list();
}

```

如果接口不能在硬件过滤器中保存针对传入数据包的组播表，则可以简化这个实现。这种情况下，`FF_TABLE_SIZE` 减小为 0，也不需要最后四行代码。

我们前面提到，不能处理组播数据包的接口也需要实现 `set_multicast_list` 方法，以便对 `dev->flags` 的变化做出响应。这种情况我们称之为“非完整”（nonfeatured，故称之为“nf”）实现，其实现非常简单，代码如下所示：

```

void nf_set_multicast_list(struct net_device *dev)
{
    if (dev->flags & IFF_PROMISC)
        nf_get_all_packets();
    else
        nf_get_only_own_packets();
}

```

`IFF_PROMISC` 的实现非常重要，因为不对该标志做出响应，用户就无法运行 `tcpdump` 或其它网络分析器。另一方面，如果接口运行在点对点链路上，就根本没有必要实现 `set_multicast_list`，因为用户总是会接收到所有的数据包。

14.14 向后兼容性

内核的 2.3.43 版本对网络子系统进行了重新设计。在性能和干净设计方面，新的“softnet”实现有了大大提高。当然，这也对网络驱动程序接口带来了一些变化——尽管这个变化比读者想象的要少。

14.14.1 Linux 2.2 中的不同

首先，Linux 2.3.14 将以前一直称为 `struct device` 的网络设备接口重命名为 `struct net_device`。新的名称显然要更加合适些，因为这个结构并不是用来描述通用设备的。

在版本 2.3.43 之前，不存在 `netif_start_queue`、`netif_stop_queue` 和 `netif_wake_queue` 函数。相反，数据包的传输由 `device` 结构中的三个成员控制，`sysdep.h` 利用这三个成员实现了上述三个函数，在 2.2 和 2.0 中编译我们的样例驱动程序时会用到。

```

unsigned char start;

```


这个变量指出接口准备好进行操作；通常在 `open` 方法中设置为 1。当前的实现是调用 `netif_start_queue` 函数。

```
unsigned long interrupt;
```

`interrupt` 用来指示设备正在服务一个中断，也就是说，在中断处理程序的开始，该变量设置为 1，而在返回时设置为 0。这个变量并不是用来替代正确的锁定的，现在已经由内部的自旋锁替代。

```
unsigned long tbusy;
```

如果非零，这个变量表示设备不能处理更多外发数据包。在 2.4 中，驱动程序应该在这种情况下调用 `netif_stop_queue`，老的驱动程序只需将 `tbusy` 设置为 1。重新启动队列时，要将 `tbusy` 设置回 0，并调用 `mark_bh(NET_BH)`。

通常，设置 `tbusy` 可足够确保驱动程序的 `hard_start_xmit` 方法不被调用。但是，如果网络系统确定传输器已经被锁住，则无论如何都要调用 `hard_start_xmit` 函数。在集成 `softnet` 之前，没有 `tx_timeout` 方法。这样，`softnet` 之前的驱动程序必须在 `tbusy` 被设置时，要明确检查对 `hard_start_xmit` 的调用并重新动作。

`struct device` 中 `name` 成员的类型在 2.2 中不同，为：

```
char *name;
```

这样，接口名称的存储空间必须单独分配，并将 `name` 赋值为指向该空间。通常，设备名称保存在驱动程序的静态变量中。用于动态赋予接口名的 `%d` 记号在 2.2 中并不存在，相反，如果名称由一个空字节或者空格开始，内核将分配下一个 `eth` 名称。2.4 仍然实现了这一处理行为，但很少用到。在 2.5 中，将只能识别 `%d` 格式。

`owner` 成员（以及 `SET_MODULE_OWNER` 宏）在内核 2.4.0-test11 中添加，也就是正式的稳定版发布之前。在此之前，网络驱动程序模块必须维护自己的使用计数。`sysdep.h` 为没有这个宏的内核定义了一个空的 `SET_MODULE_OWNER`；可移植代码仍应该继续手工维护自己的使用计数（除了让网络系统维护之外）。

链路状态函数（`netif_carrier_on` 和 `netif_carrier_off`）在 2.2 内核中不存在。那时的内核忽略了链路状态信息。

14.14.2 Linux 2.0 中其它不同

2.1 开发系列也有一些对网络驱动程序接口的修改。大部分的变化是函数原型上面的，而不是网络代码的整体改变。

接口统计信息在 `struct lenet_statistics` 结构中保存，该结构定义在 `<linux/if_ether.h>` 中。非以太网驱动程序也使用这些结构。

2.0 内核处理传输器锁住的方法和 2.2 一样。但是，有一个附加函数：

```
void dev_tint(struct device *dev);
```

当锁定被解除时，应该调用这个函数重新启动数据包传输。

有几个函数具有不同的原型。`dev_kfree_skb` 有第二个参数，是个整型值，可取 `FREE_READ`（用于传入数据包，也即，驱动程序分配 `skb`），或者取 `FREE_WRITE`（用于外发数据包，也即，上层网络代码分配 `skb`）。驱动程序中，几乎所有对 `dev_kfree_skb` 的调用都使用 `FREE_WRITE`。`skb` 函数的非检查版本（类似 `__skb_push`）也不存在；示例代码的 `sysdep.h` 提供了 2.0 下对这些函数的模拟。

`rebuild_header` 方法的参数形式不同：

```
int (*rebuild_header)(void *eth, struct device *dev, unsigned long raddr, struct sk_buff *skb);
```

这个版本的 Linux 内核大量使用了 `rebuild_header` 函数，而现在原先由该函数完成的大部分工作由 `hard_header` 完成。当 `snull` 在 Linux 2.0 下编译时，该驱动程序利用下面的代码建立硬件头：

```
int snull_rebuild_header(void *buff, struct net_device *dev, unsigned long dst,
                        struct sk_buff *skb)
{
    struct ethhdr *eth = (struct ethhdr *)buff;

    memcpy(eth->h_source, dev->dev_addr, dev->addr_len);
    memcpy(eth->h_dest, dev->dev_addr, dev->addr_len);
    eth->h_dest[ETH_ALEN-1] ^= 0x01; /* dest is us xor 1 */
    return 0;
}
```

2.0 内核用于数据包头缓存的设备方法也有很大的差别。如果驱动程序需要直接实现这些函数（很少会这样做），并且还希望能够在 2.0 内核下工作的话，可参阅 `<linux/netdevice.h>` 中的定义。

14.14.3 探测和 HAVE_DEVLIST

如果读者阅读内核中网络驱动程序的源代码，则会发现类似下面的这种样板文件：

```
#ifdef HAVE_DEVLIST
/*
 * Support for an alternate probe manager,
 * which will eliminate the boilerplate below.
 */
struct netdev_entry netcard_drv =
{cardname, netcard_probe1, NETCARD_IO_EXTENT, netcard_portlist};
#else
/* Regular probe routine defined here */
```

有意思的是，上述代码自从 1.1 开发系列就有了，但我们仍然在等待另外一个探测管理器的出现。也许我们不用为这种重大的变化担心，因为在将来的时间，探测管理器的实现思想还会发生变化。

14.15 快速参考

这个小节给出本章介绍过的概念的快速参考，同时解释了驱动程序应该包含的每个头文件。但是，

`net_device` 和 `sk_buff` 结构的成员不会在这里重复。

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

这个头文件保存有 `struct net_device` 和 `struct net_device_stats` 的定义，并包含了网络驱动程序需要的其它几个头文件。

```
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
```

注册和注销一个网络设备。

```
SET_MODULE_OWNER(struct net_device *dev);
```

这个宏将在设备结构（实际上，将在所有结构的 `owner` 成员）中保存一个指向当前模块的指针；它可让网络子系统管理模块的使用计数。

```
netif_start_queue(struct net_device *dev);
netif_stop_queue(struct net_device *dev);
netif_wake_queue(struct net_device *dev);
```

上述函数控制外发数据包向驱动程序的传递。在调用 `netif_start_queue` 之前，不会传输任何数据包。`netif_stop_queue` 暂停传输，而 `netif_wake_queue` 重新启动队列并通知网络层重新启动数据包的传输。

```
void netif_rx(struct sk_buff *skb);
```

调用（包括中断期间）这个函数可通知内核已经接收到一个数据包，并封装入一个套接字缓冲区。

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

`netdevice.h` 包含这个头文件，其中声明了接口标志（`IFF_` 宏）以及 `struct ifmap` 结构。这个结构在网络驱动程序的 `ioctl` 实现中扮演主要角色。

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);
int netif_carrier_ok(struct net_device *dev);
```

前两个函数可用来通知内核给定接口上是否存在载波信号。`netif_carrier_ok` 将根据设备结构中的信息检查载波状态并返回。

```
#include <linux/if_ether.h>
ETH_ALEN
ETH_P_IP
struct ethhdr;
```

由 `netdevice.h` 包含，`if_ether.h` 定义了所有的 `ETH_` 宏，这些宏表示 `octet` 长度（比如地址长度）以及网络协议（比如 `IP`）等。这个头文件还定义了 `ethhdr` 结构。

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

这个头文件包含了 `struct sk_buff` 以及相关结构的定义，还定义了若干操作缓冲区的内嵌函数。`netdevice.h` 包含该头文件。

```
struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
```

这些函数处理套接字缓冲区的分配和释放。驱动程序通常使用这些函数的 `dev_` 变种。

```
unsigned char *skb_put(struct sk_buff *skb, int len);
```

```
unsigned char *_skb_put(struct sk_buff *skb, int len);
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *_skb_push(struct sk_buff *skb, int len);
```

这些函数向 **skb** 中添加数据；**skb_put** 在 **skb** 的尾部放入数据，而 **skb_push** 在头部放入数据。这些函数的常规版本检查 **skb** 确保有足够的空间，而具有双下划线前缀的版本不进行空间检查。

```
int skb_headroom(struct sk_buff *skb);
int skb_tailroom(struct sk_buff *skb);
void skb_reserve(struct sk_buff *skb, int len);
```

上述函数执行 **skb** 中的空间管理。**skb_headroom** 和 **skb_tailroom** 分别返回 **skb** 头部和尾部的可用空间。**skb_reserve** 可用来在空的 **skb** 头部保留空间。

```
unsigned char *skb_pull(struct sk_buff *skb, int len);
```

skb_pull 通过调整内部指针从 **skb** 中“删除”数据。

```
#include <linux/etherdevice.h>
void ether_setup(struct net_device *dev);
```

这个函数为以太网驱动程序设置大部分通用的设备方法。它同时设置 **dev->flags**。如果设备名称的第一个字符为空，或者是空格的话，这个函数将把下一个可用的 **ethx** 名称赋予 **dev->name**。

```
unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev);
```

在以太网接口接收一个数据包时，可调用这个函数设置 **skb->pkt_type**。返回值是通常保存在 **skb->protocol** 中的协议编号。

```
#include <linux/sockios.h>
SIOCDEVPRIVATE
```

这是每个驱动程序可为自己的私有用途实现的 16 个 **ioctl** 命令中的第一个。所有的网络 **ioctl** 命令在 **sockios.h** 中定义。

第 15 章 外设总线综述



第 8 章介绍了最低层的硬件控制，本章将综述高级总线的结构。总线不仅构成了电气接口，同时还定义了编程接口。本章将重点讨论编程接口。

本章涉及到若干不同的总线结构，但讨论的重点是用于访问 PCI 外设的内核函数，这是因为 PCI 总线是当今普遍使用在桌面以及大型计算机上的外设总线，而且在内核中也得到了最好的支持。虽然 ISA 总线是一种“裸金属”类型的总线，但对某些电子爱好者来说，ISA 仍然很常用，因此将在本章后面讨论。当然，除了我们在第 8 章和第 9 章中所提到的内容以外，其实也没有多少需要对 ISA 总线进行讨论的内容。

15.1 PCI 接口

尽管许多计算机用户将 PCI（Peripheral Component Interconnect，外设组件互连接）看成是一种布置电子线路的方式，但实际上，它其实是一组完整的规范，定义了计算机的各种不同部分之间应该如何交互。

PCI 规范涵盖了计算机接口相关的大部分问题。我们不会在这里讲述所有的内容，这个小节将主要集中于 PCI 驱动程序寻找其硬件的方法，以及如何获得对 PCI 设备的访问。第 2 章“自动和手工配置”以及第 9 章中的“自动检测 IRQ 号”所提到的探测技术，也可用于 PCI 设备，但是 PCI 规范提供了一种更好的探测方法。

PCI 设计为 ISA 标准的替代品，它有三个目标：在计算机和外设之间传输数据时，能够有更好的性能；能够尽量独立于平台；简化向/从系统中添加/删除外设的工作。

通过使用比 ISA 更高的时钟频率，PCI 总线获得了更好的性能。它的时钟频率一般在 25 到 33 MHz 范围（实际的频率取决于系统时钟），最新的总线可达到了 66 MHz，甚至 133 MHz。另外，PCI 总线具有 32 位的数据总线，而且规范中还包括有 64 位扩展（当然只有 64 位平台才会实现 64 位的数据总线）。平台无关性通常也是计算机总线的一个设计目标，对 PCI 来讲，平台无关性尤其重要，这是因为 PC 世界以往总是由一些处理器特有的接口标准所控制。目前，PCI 总线广泛应用于 IA-32、Alpha、PowerPC、SPARC64 和 IA-64 系统中，其它一些平台也使用了 PCI 总线。

驱动程序编写者常常头疼的问题是接口板的自动检测。PCI 设备是无跳线设备（不象某些老式外

设)，可在引导阶段自动配置。这样，设备驱动程序必须能够访问设备中的配置信息以便完成初始化。对 PCI 设备，这些工作无需探测，就能完成。

15.1.1 PCI 寻址

每个 PCI 外设由一个总线编号、一个设备编号及一个功能编号标识。PCI 规范允许一个系统能够拥有高达 256 个总线，每个总线上可存在 32 个设备，而每个设备也可以是多功能板（比如音频设备外加 CD-ROM 驱动器），最多可达 8 种功能。每种功能在硬件级由一个 16 位的地址（或键）标识。为 Linux 编写的设备驱动程序，无需处理这些二进制的地址，因为它们可使用一种特殊的数据结构，称为 `pci_dev` 来访问设备。（我们在第 13 章看到过这个 `struct pci_dev` 结构。）

最近的工作站一般配置有至少两个 PCI 总线。在单个系统中插入多个总线，可通过“桥”来完成，桥是用来连接两个总线的特殊 PCI 外设。PCI 系统的整体布局组织为树型，其中每个总线连接到上一级总线，直到 0 号总线。CardBus PC 卡系统也通过桥连接到 PCI 系统上。典型的 PCI 系统可见图 15-1，其中标记出了各个不同的桥。

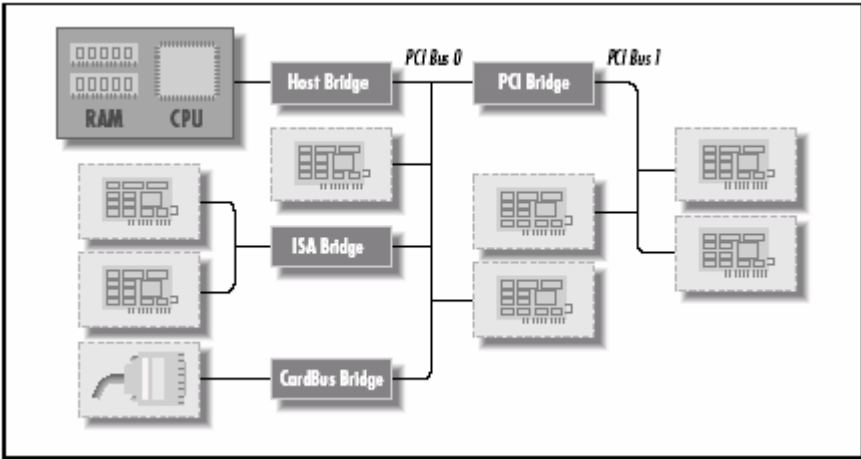


图 15-1: 典型 PCI 系统的布局

尽管和 PCI 外设关联的 16 位硬件地址通常隐藏在 `struct pci_dev` 对象中，但有时仍然可见，尤其在列出正在使用的设备时。比如，在 `lspci`（`pciutils` 包的一个组件，大多数发行版中含有这个包）的输出，以及 `/proc/pci` 和 `/proc/bus/pci` 的布局信息中*。

在显示硬件地址时，有时显示为一个 16 位的值，有时显示为两个值（一个 8 位的总线编号和一个 8 位的设备及功能编号），有时显示为三个值（总线、设备和功能）。所有的值通常都显示为 16 进制数。

例如，`/proc/bus/pci/devices` 使用单个 16 位字段（便于分析及排序），而 `/proc/bus/pci/busnumber` 将地址划分成了三个字段。下面说明了地址出现的方式，注意只列出了输出行的前面几行：

```
rudo% lspci | cut -d: -f1-2
00:00.0 Host bridge
```

* 注意这里的讨论是基于 2.4 内核版本的，向后兼容性问题在本章后面讨论。

```

00:01.0 PCI bridge
00:07.0 ISA bridge
00:07.1 IDE interface
00:07.3 Bridge
00:07.4 USB Controller
00:09.0 SCSI storage controller
00:0b.0 Multimedia video controller
01:05.0 VGA compatible controller
rudo% cat /proc/bus/pci/devices | cut -d\          -f1,3
0000      0
0008      0
0038      0
0039      0
003b      0
003c      b
0048      a
0058      b
0128      a

```

这两个设备清单以相同的顺序排列，因为 `lspci` 使用 `/proc` 文件作为其信息来源。拿 **VGA video controller**（VGA 视频控制器）作为例子，将 `0x128` 划分为总线（8 位）、设备（5 位）及功能（3 位）时，可表示成 `01:05.0`。上述清单中的前后第二个字段分别表示了设备类型和中断号。

每个外设板的硬件电路对如下三种地址空间的查询进行应答：内存位置、I/O 端口以及配置寄存器。前两类地址空间由同一 **PCI** 总线上的所有设备共享（也就是说，在访问内存位置时，所有的设备将在同一时间看到该总线周期）。另一方面，配置空间利用了“位置寻址（**geographical addressing**）”。配置事务（亦即，总线对配置空间的访问）每次只会对一个 **PCI** 槽寻址，这样，在配置访问期间，根本不会发生任何冲突。

对驱动程序而言，内存和 I/O 区域通过通常的方式，即 `inb` 和 `readb` 等等进行访问。另一方面，配置事务却通过调用特定的内核函数来访问配置寄存器。关于中断，每个 **PCI** 槽有四个中断引脚，每个设备功能可使用其中的一个，而不用考虑这些引脚如何连接到 **CPU**。到 **CPU** 的中断连接由计算机平台负责，一般在 **PCI** 总线之外实现。因为 **PCI** 规范要求中断线是可共享的，因此，尽管处理器可能会限制 **IRQ** 线的数量（比如 **x86**），但仍然可以安装许多 **PCI** 接口板（每个接口板均有四个中断引脚）。

PCI 总线中的 I/O 空间使用 32 位地址总线（因此可有 4GB 个端口），而内存空间可通过 32 位或 64 位地址访问。但是，64 位地址仅仅在几个平台上可用。通常假定地址对设备是唯一的，但是软件可能会错误地将两个设备配置成相同的地址，导致无法访问这两个设备。但是，如果驱动程序不去访问那些不应该访问的寄存器，这样的问题就不会发生。另外，由接口板提供的每个内存和 I/O 地址区域，都可通过配置事务进行重新映射。这样，固件在系统引导时初始化 **PCI** 硬件，并将每个区域映射到不同的地址以避免冲突。^{*}这些区域当前的映射情况，可从配置空间中读取，因此，Linux 驱动程序不需要探测就能访问其设备。在读取配置寄存器之后，驱动程序就可以安全访问其硬件。

PCI 配置空间由 256 个字节组成，每个设备功能有一个配置空间，而且配置寄存器的布局是标准化的。配置空间的 4 个字节含有唯一的功能 ID，因此，驱动程序可通过查询外设的特定 ID 来标

^{*} 实际上，配置过程并不限于系统引导阶段。比如热插拔设备，不在引导阶段出现，而会在后来出现。这里强调的是，设备驱动程序无需修改 I/O 或内存区域地址。

识其设备。*

总之，我们可以独立地检索每个设备板的配置寄存器，这些寄存器中的信息可用来执行通常的 I/O 访问，而无需其它的“位置寻址（geographic addressing）”。

到此应该清楚的是，PCI 接口标准在 ISA 之上的主要创新，在于配置地址空间。因此，除了通常的驱动程序代码之外，PCI 驱动程序还应该有能力访问配置空间，而无需冒险进行探测。

本章其余内容中，我们将使用“设备”一词来表示一种设备功能，因为我们可以将多功能板上的每个功能看成是一个独立的入口。我们谈到设备时，表示的是一组“总线编号、设备编号、功能编号”，它们可由一个 16 位数或者两个 8 位数（通常称为 bus 和 devfn）来表示。

15.1.2 引导阶段

了解 PCI 的工作原理，我们需要从系统引导开始讲起，因为这是配置设备的阶段。

当 PCI 设备上电时，硬件保持未激活状态。换句话说，该设备只会对配置事务做出响应。上电时，不会有内存和 I/O 端口映射到计算机的地址空间，其它设备相关功能，比如中断报告，也被禁止。

幸运的是，每个 PCI 主板均配备有能够处理 PCI 总线的固件，称为 BIOS、NVRAM 或 PROM（这取决于平台）。固件通过读写 PCI 控制器种中的寄存器，提供了对设备配置地址空间的访问。

系统引导时，固件（或者 Linux 内核，如果经过配置的话）在每个 PCI 外设上执行配置事务，以便为设备的每个地址区域分配一个安全的位置。在驱动程序访问设备的时候，设备的内存和 I/O 区域已经映射到了处理器的地址空间。驱动程序可以修改默认的配置，但几乎没有任何理由需要这样做。

我们讲到，用户可以读取 `/proc/bus/pci/devices` 和 `/proc/bus/pci/*/*` 来了解 PCI 设备清单以及设备的配置寄存器。前者是个文本文件，包含有十六进制的设备信息，而后者是若干二进制文件，包含了每个设备的配置寄存器信息，每个文件对应一个设备。

15.1.3 配置寄存器和初始化

先前提到，配置空间的布局是设备无关的。本节我们将看到用来标识外设的配置寄存器。

PCI 设备配备有一个 256 字节的地址空间。前 64 字节是标准化的，其后的字节是设备相关的。图 15-2 给出了设备无关的配置空间。

* 我们可从设备的硬件手册中查到其 ID。文件 `pci.ids` 中包含有一个清单，该文件是 `pciutils` 包以及内核源代码的一部分。该文件并不打算包含完整的清单，但其中已列出了大部分有名的生产商和设备。

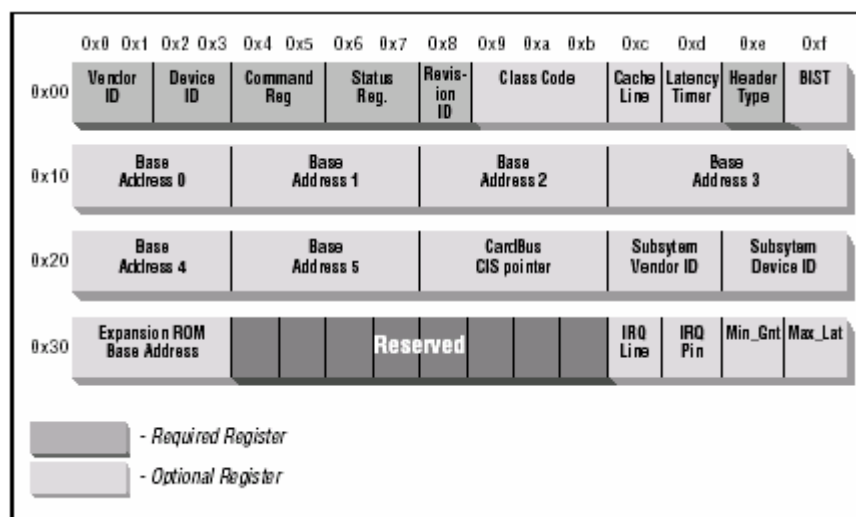


图 15-2: 标准化的 PCI 配置寄存器

如图所示，某些 PCI 配置寄存器是必需的，而某些是可选的。每个 PCI 设备要在必需的寄存器中包含有效值，而可选寄存器中的内容依赖于外设的实际功能。可选字段通常无用，除非必需字段表明它们是有效的。这样，必需的字段表明板子的功能，其中包括其它字段是否有用的信息。

值得注意的是，PCI 寄存器始终是 **little-endian** 的。尽管该标准设计为体系结构无关的，但 PCI 设计者仍然有点偏好 PC 环境。驱动程序编写者在访问多字节的配置寄存器时，要十分注意字节序。能够在 PC 上工作的代码到其它平台上，可能就无法工作。Linux 开发人员已经注意到了字节序问题（见下面一节“访问配置空间”），但是这个问题还是应该牢记心中。如果需要将系统固有字节序转换成 PCI 字节序，或者相反，则可以借助定义在 `<asm/byteorder.h>` 中的函数，这些函数在第 10 章中介绍，注意 PCI 字节序是 **little-endian**。

对这些配置项的描述已经超过了本书讨论的范围。通常，和设备一同发布的技术文档会详细描述已支持的寄存器。我们所关心的是，驱动程序如何查询设备，以及如何访问设备的配置空间。

用三个或五个 PCI 寄存器可标识一个设备：**vendorID**、**deviceID** 和 **class** 是常用的三个寄存器。每个 PCI 生产商会将正确的值赋于上述三个只读的寄存器，驱动程序可利用它们查询设备。另外，有时生产商会利用 **subsystem vendorID** 和 **subsystem deviceID** 两个字段进一步区分相似设备。

下面是这些寄存器的详细介绍。

vendorID

这是一个 16 位的寄存器，用于标识硬件制造商。例如，每个 Intel 设备被标识为同一个生产商编号，即 0x8086。PCI Special Interest Group 维护有一个全球的生产商编号注册表，制造商必须申请一个唯一编号并赋于该寄存器。

deviceID

这是另外一个 16 位寄存器，由制造商选择，而无需对设备 ID 进行官方注册。该 ID 通常和制造商 ID 配对生成一个唯一的 32 位硬件设备标识符。我们使用签名（**signature**）来表示一对制造商和设备 ID。设备驱动程序通常依靠该签名来标识其设备。我们可从硬件手册中找到目标设备的签名值。

```
class
```

每个外部设备属于某个“类（class）”。class 寄存器是一个 16 位的值，其中高 8 位标识了“基类（base class）”，或者组。例如，“ethernet（以太网）”和“token ring（令牌环）”是同属“network（网络）”组的两个类，而“serial（串行）”和“parallel（并行）”类同属“communication（通讯）”组。某些驱动程序可支持多个相似的设备，每个设备具有不同的签名，却属于同一个类。这种驱动程序可依靠 class 寄存器来标识它们的外设（如后所述）。

```
subsystem vendorID
subsystem deviceID
```

这两个字段可用来进一步标识设备。如果设备中的芯片是连接到本地（onboard）总线上一个通用接口芯片，则可能会用于完全不同的多种用途，这时，驱动程序必须标识它所关心的实际设备。子系统（subsystem）标识符就用在这种场合。

使用这些标识符，我们可以检测并访问设备。在 2.4 内核中，已经引入了 PCI 驱动程序的概念，以及专用的初始化接口。新的驱动程序应该优先选择这个接口，然而，老的内核版本却无法使用。另外，还可以使用下面的头文件、宏和函数作为 PCI 驱动程序的接口，PCI 模块可使用它们查询它们的硬件设备。我们首先介绍向后兼容的接口，是因为这些接口能够移植到本书提到的所有内核版本。另外，这个接口还具有更加贴近直接的硬件管理、而较少抽象的优点。

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

驱动程序需要了解内核是否具备 PCI 功能。包含该头文件后，驱动程序可访问 CONFIG_ 宏，包括下面要讲到的 CONFIG_PCI。但要注意的是，每个包含 <linux/module.h> 的源文件，都已经包含了这个头文件。

```
CONFIG_PCI
```

如果内核包含有对 PCI 调用的支持，则定义这个宏。并不是所有的计算机都有 PCI 总线，所以内核开发者选择 PCI 支持为一个编译选项，以便在非 PCI 计算机上运行 Linux 时节省内存。如果没有定义 CONFIG_PCI，每个 PCI 函数调用将返回一个错误状态，因此，驱动程序既可以使用预编译条件，也可以不使用预编译条件。如果驱动程序只能处理 PCI 设备（与同时具有 PCI 实现和非 PCI 实现相反），则应该在这个宏未定义时，产生一个编译错误。

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

这个头文件声明了本节介绍的所有原型，以及和 PCI 寄存器和位相关的符号名称。我们应该始终包含该头文件。这个头文件还包含有函数所返回的错误码的符号值。

```
int pci_present(void);
```

因为 PCI 相关函数不能在非 PCI 计算机上正常工作，因此，可利用 pci_present 检查 PCI 功能是否可用。在 2.4 中，不鼓励使用这个函数，因为它将检查是否存在某些 PCI 设备。但在 2.0 中，驱动程序必须调用这个函数，以避免在查询设备时出现错误。新近内核只会报告是否存在 PCI 设备。如果主机中没有 PCI 设备，则该函数返回布尔值真（非零）。

```
struct pci_dev;
```

该数据结构作为表示 PCI 设备的软件对象。它是系统每一个 PCI 操作的核心。

```
struct pci_dev *pci_find_device (unsigned int vendor, unsigned int device, const struct
pci_dev *from);
```

如果定义有 CONFIG_PCI，而且 pci_present 返回真，则可利用该函数扫描已安装的设备链表，

以查询具有特定签名的设备。**from** 参数用来得到具有相同签名的多个设备。该参数会指向已发现的最后一个设备，这样，下一个搜索就可以从这个位置开始，而无需从链表头开始。为了找到第一个设备，可将 **from** 指定为 **NULL**。如果找不到设备，返回 **NULL**。

```
struct pci_dev *pci_find_class (unsigned int class, const struct pci_dev *from);
```

该函数类似前一个函数，但它查询的是属于特定类（16 位的类，含有基类和子类值）的设备。除了非常低层的 **PCI** 驱动程序以外，现在该函数已经很少使用。**from** 参数和 **pci_find_device** 中的用法一样。

```
int pci_enable_device (struct pci_dev *dev);
```

该函数真正使能指定的设备。它激活该设备，在某些情况下，同时赋予其中断号和 **I/O** 区域。例如，在使能 **CardBus** 设备（在驱动程序级，它完全等价于 **PCI**）时将发生后面一种情况。

```
struct pci_dev *pci_find_slot (unsigned int bus, unsigned int devfn);
```

该函数根据一对总线/设备返回一个 **PCI** 设备结构。**devfn** 参数表示设备和功能两个项。该函数很少用到（驱动程序无需关心设备插入哪个 **PCI** 槽），在这里列出它，只是出于完整性考虑。

根据以上内容，处理单个设备类型的典型驱动程序，其初始化过程应该类似下面的代码。这段代码用于一个假设备 **jail**（Just Another Instruction List）：

```
#ifndef CONFIG_PCI
# error "This driver needs PCI support to be available"
#endif

int jail_find_all_devices(void)
{
    struct pci_dev *dev = NULL;
    int found;

    if (!pci_present())
        return -ENODEV;

    for (found=0; found < JAIL_MAX_DEV; ) {
        dev = pci_find_device(JAIL_VENDOR, JAIL_ID, dev);
        if (!dev) /* no more devices are there */
            break;
        /* do device-specific actions and count the device */
        found += jail_init_one(dev);
    }
    return (index == 0) ? -ENODEV : 0;
}
```

jail_init_one 函数是设备特有的，所以没有在这里列出来。尽管如此，在编写上述函数时，有若干需要特别注意的事项：

- 该函数需要执行一些额外的探测，以确保该设备是真正由驱动程序所支持的。某些 **PCI** 外设包含一个通用的 **PCI** 接口芯片以及设备特有的电路。所有使用相同接口芯片的外设板具有相同的签名。可通过读取子系统标识符，或设备特有的寄存器（在设备 **I/O** 区域中，将在后面介绍）执行进一步探测。
- 在访问任意设备资源（**I/O** 区域或者中断）之前，驱动程序必须调用 **pci_enable_device**。如果要执行上面所说的额外探测，则该函数必须在探测发生之前调用。
- 网络接口驱动程序应该将 **dev->driver_data** 指向与该接口关联的 **struct net_device** 结构。

上述摘录代码中的函数在拒绝该设备时返回 0，而在接受该设备时返回 1（可能还要根据进一步探测的结果决定返回值）。

上述代码在驱动程序只处理一种类型的 PCI 设备（由 JAIL_VENDRO 和 JAIL_ID 标识）时是正确的。如果支持更多 vendor/device 对，则最好使用后面“硬件抽象”一节中介绍的技术，除非需要支持 2.4 版本之前的内核——这种情况下，可使用 pci_find_class。

使用 pci_find_class 要求 jail_find_all_devices 执行其它一些工作，它应该检查匹配 vendor/device 对的新设备（使用 dev->vendor 和 dev->device）。代码类似如下：

```
struct devid {unsigned short vendor, device} devlist[] = {
    {JAIL_VENDOR1, JAIL_DEVICE1},
    {JAIL_VENDOR2, JAIL_DEVICE2},
    /* ... */
    { 0, 0 }
};

/* ... */

for (found=0; found < JAIL_MAX_DEV;) {
    struct devid *idptr;
    dev = pci_find_class(JAIL_CLASS, dev);
    if (!dev) /* no more devices are there */
        break;
    for (idptr = devlist; idptr->vendor; idptr++) {
        if (dev->vendor != idptr->vendor) continue;
        if (dev->device != idptr->device) continue;
        break;
    }
    if (!idptr->vendor) continue; /* not one of ours */
    jail_init_one(dev); /* device-specific initialization */
    found++;
}
```

15.1.4 访问配置空间

在驱动程序检测到设备之后，它通常需要读取或写入三个地址空间：内存、端口和配置。对驱动程序而言，对配置空间的访问尤其不可缺少，这是因为对配置空间的访问，是找到设备内存和 I/O 空间映射结果的唯一途径。

因为处理器没有任何直接访问配置空间的途径，因此，需要计算机生产商来提供这个途径。为了访问配置空间，CPU 必须读取或写入 PCI 控制器的寄存器，但具体的实现依赖于计算机生产商，和我们这里的讨论无关，这是因为 Linux 提供了访问配置空间的标准接口。

对驱动程序，可通过 8 位、16 位或 32 位的数据传输访问配置空间，相关函数的原型定义在 <linux/pci.h> 中：

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *ptr);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *ptr);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *ptr);
```

从 dev 标识的设备配置空间中读入一个、两个或四个字节。where 参数是从配置空间起始位置计算的字节偏移量。从配置空间获得的值，通过 ptr 返回，函数本身的返回值是错误码。word 和 dword 函数会将读取到的 little-endian 值转换成处理器固有的字节顺序，因此，我们自己无需处理字节序。

```
int pci_write_config_byte (struct pci_dev *dev, int where, u8 val);
int pci_write_config_word (struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword (struct pci_dev *dev, int where, u32 val);
```

向配置空间写入一个、两个或四个字节。和上面的函数一样，**dev** 标识设备，要写入的值通过 **val** 传递。**word** 和 **dword** 函数在把值写入外设之前，会将其转换成 **little-endian** 字节序。

读取配置变量的首选方法，是使用设备对应的 **struct pci_dev** 结构中的成员。虽然如此，如果需要写入并读取某个配置变量，我们仍然需要上述的函数。同时，如果需要和 2.4 版本之前的内核保持兼容，也需要使用 **pci_read_** 函数。^{*}在使用 **pci_read_** 函数时，定位配置变量的最好方法是利用定义在 `<linux/pci.h>` 中的符号名称。例如，下面的函数调用检索设备的修订（revision）ID，注意为 **pci_read_config_byte** 函数传递了符号名称：

```
unsigned char jail_get_revision(unsigned char bus, unsigned char fn)
{
    unsigned char *revision;

    pci_read_config_byte(bus, fn, PCI_REVISION_ID, &revision);
    return revision;
}
```

我们曾提到，以单个字节的形式访问多字节值时，我们必须正确处理字节序问题。

配置空间示例

如果要浏览系统中 PCI 设备的配置空间，则可以选择两种途径之一。比较简单的方法是利用 Linux 通过 `/proc/bus/pci` 提供的资源（2.0 版的内核不提供这种支持）。另外一个是我们这里介绍的方法，就是自己编写一些代码来完成该任务。该代码能够在已知所有的 2.x 内核版本上运行，并且也是分析 PCI 设备工作情况的一个好途径。源文件 `pci/pcidata.c` 包含在 O'Reilly FTP 站点提供的示例代码中。

该模块建立了一个动态的 `/proc/pcidata` 文件，其中包含了 PCI 设备配置空间的二进制快照。该快照在每次读取这个文件时更新。`/proc/pcidata` 的大小限制在 `PAGE_SIZE` 字节（为避免处理多页的 `/proc` 文件，相关内容可见第 4 章“使用 `/proc` 文件系统”）。因此，该文件只能列出前 `PAGE_SIZE/256` 个设备的配置空间内容，依赖于所运行的平台，这个数字可能是 16 或者 32。我们选择 `/proc/pcidata` 为二进制文件，而不是类似其它 `/proc` 文件那样的文本格式，只是为了让代码简单一些。需要注意的是，`/proc/bus/pci` 中的文件也是二进制的。

`pcidata` 的另一个限制是它仅仅扫描系统中的第一个 PCI 总线。如果计算机含有到其它 PCI 总线的桥，`pcidata` 将忽略这些总线。这对示例代码来讲，并不是个大问题。

出现在 `/proc/pcidata` 中的设备的顺序，和 `/proc/bus/pci/devices` 使用的顺序一样（但和版本 2.0 中的 `/proc/pci` 所使用的顺序相反）。

例如，我们的帧捕获器在 `/proc/pcidata` 中位于第五，其配置寄存器内容（当前的）如下：

^{*} `struct pci_dev` 中的成员名称在 2.2 和 2.4 之间发生了一些变化，因为 2.2 中的设计存在一些不足。对 2.0，根本没有 `pci_dev` 结构，我们可利用的是由 `pci-compat.h` 头文件提供的一种简单模拟。

```
morgana% dd bs=256 skip=4 count=1 if=/proc/pcidata | od -Ax -t x1
1+0 records in
1+0 records out
000000 86 80 23 12 06 00 00 02 00 00 00 04 00 20 00 00
000010 00 00 00 f1 00 00 00 00 00 00 00 00 00 00 00 00
000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000030 00 00 00 00 00 00 00 00 00 00 00 00 0a 01 00 00
000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
000100
```

上述转储结构中的数字表示 PCI 寄存器中的值。参考图 15-2，我们可以了解每个数字的含义。另外，我们也可以使用 `pcidump` 程序（也可从 FTP 站点获得），它可以将上述数字格式化并利用标记输出。

`pcidump` 的代码不值得在这里列出，因为其中包含有一个大表，以及扫描这个表的 10 多行代码。但是，我们给出这个程序的若干输出（节录）：

```
morgana% dd bs=256 skip=4 count=1 if=/proc/pcidata | ./pcidump
1+0 records in
1+0 records out
    Compulsory registers:
Vendor id: 8086
Device id: 1223
I/O space enabled: n
Memory enabled: y
Master enabled: y
Revision id (decimal): 0
Programmer Interface: 00
Class of device: 0400
Header type: 00
Multi function device: n
    Optional registers:
Base Address 0: f1000000
Base Address 0 Is I/O: n
Base Address 0 is 64-bits: n
Base Address 0 is below-1M: n
Base Address 0 is prefetchable: n
Does generate interrupts: y
Interrupt line (decimal): 10
Interrupt pin (decimal): 1
```

`pcidata`、`pcidump` 以及 `grep` 工具，可用来调试驱动程序的初始化代码。当然，这些工具完成的任务，部分可从 `pciutils` 包中获得，这个包包含在新近发布的所有 Linux 发行版中。还需注意的是，和本书其它的代码不同，`pcidata.c` 模块遵循 GPL，这是因为我们从内核源代码的 PCI 扫描循环中拿了一些代码。作为驱动程序编写者，这应该不是个问题，因为这个模块只是作为一个支持工具，而不是一个可重复利用的新驱动程序模板。

15.1.5 访问 I/O 和内存空间

一个 PCI 设备可实现多达 6 个 I/O 地址区域。每个区域可以是内存也可以是 I/O 位置区间。大多数设备在内存区域实现 I/O 寄存器，因为这是一个通用的明智选择（详细情况，可参阅第 8 章“I/O 端口和 I/O 内存”）。但是，不象通常的内存，I/O 寄存器不应该由 CPU 缓存，因为每次访问可能具有边界效应（side effect）。将 I/O 寄存器实现为内存区域的 PCI 设备，通过在配置寄存

器中设置“(内存是可预取的) **memory-is-prefetchable**”标志而标记这个不同。*

如果内存区域被标记为可预取(**prefetchable**)，则 CPU 可缓存其内容，并实现所有的优化。另一方面，非可预取的(**nonprefetchable**)内存访问不能被优化，因为每个访问可能具有边界效应，尤其是通常的 I/O 端口。PCI 设备通常将映射到内存地址区间的控制寄存器标记为非可预取的，而将 PCI 板上类似显示内存这样的东西，标记为可预取的。在本节，我们使用“区域”一词指代一般的 I/O 地址空间，不管是内存映射的，还是端口映射的。

一个接口板通过配置寄存器报告其区域的大小和当前位置——即图 15-2 中的 6 个 32 位寄存器，它的符号名称为 **PCI_BASE_ADDRESS_0** 到 **PCI_BASE_ADDRESS_5**。因为 PCI 定义的 I/O 空间是 32 位地址空间，因此，内存和 I/O 可使用相同的配置接口。如果设备使用 64 位的地址总线，则可为每个区域使用两个连续的 **PCI_BASE_ADDRESS** 寄存器(低位优先)，来声明 64 位内存空间中的区域。对一个设备来讲，既可以提供 32 位区域，也可以提供 64 位区域。

Linux 2.4 中的 PCI I/O 资源

在 Linux 2.4 中，PCI 设备的 I/O 区域已经集成到了一般的资源管理。出于该原因，我们无需访问配置变量来获得设备内存和 I/O 空间的映射情况。获得区域信息的首选接口，由如下函数组成：

```
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
```

该函数返回六个 PCI I/O 区域之一的第一个地址(内存地址或 I/O 端口编号)。该区域由整数的 **bar** (**base address register**，基地址寄存器)决定，可取 0 到 5 的值。

```
unsigned long pci_resource_end(struct pci_dev *dev, int bar);
```

该函数返回第 **bar** 个 I/O 区域的后一个地址。注意这是最后一个可用的地址，而不是该区域之后的第一个地址。

```
unsigned long pci_resource_flags(struct pci_dev *dev, int bar);
```

该函数返回资源关联的标志。

资源标志用来定义单个资源的某些特性。对与 PCI I/O 区域关联的 PCI 资源，该信息从基地址寄存器中获得，但对其它 PCI 设备无关的资源，它可能来自任何地方。

所有在资源标志定义在 `<linux/ioport.h>` 中，下面列出其中最重要的几个：

```
IORESOURCE_IO  
IORESOURCE_MEM
```

如果对应的 I/O 区域存在，将设置上面标志中的一个，而且仅一个。

```
IORESOURCE_PREFETCH  
IORESOURCE_READONLY
```

上述标志定义内存区域是可预取的，或者是写保护的。对 PCI 资源来讲，从来不会设置后面的那个标志。

通过使用 **pci_resource_** 函数，设备驱动程序可完全忽略底层的 PCI 寄存器，因为系统已经使用这些寄存器构建了资源信息。

* 该信息保存在 PCI 寄存器基地址的低位上，这些位在 `<linux/pci.h>` 中有定义。

基地址寄存器

通过避免对 PCI 寄存器的访问，我们可以获得更好的硬件抽象，以及向前的兼容性，但却不能获得向后兼容性。如果希望自己的设备驱动程序能够在 2.4 之前的内核上工作，就不能使用这些漂亮的资源接口，而必须直接访问 PCI 寄存器。

在这个小节，我们讨论基地址寄存器的工作方式，以及访问这些寄存器的方法。如果读者可以直接利用先前讲述过的资源管理函数，这里的这些内容明显是多余的。

我们不会在这里详细地介绍基地址寄存器，这是因为，如果我们正准备编写一个 PCI 驱动程序，则无论如何都会有设备的硬件手册可作参考。尤其是，我们不会使用寄存器的可预取位，以及两个“类型”位，而且我们的讨论会限制在 32 位外设上。但是，这些东西的实现方法，以及 Linux 驱动程序处理 PCI 内存的方法，仍然值得一看。

PCI 规范指出，制造商必须将每个有效区域映射到可配置地址。这意味着，设备必须为每个区域装备一个可编程的 32 位地址解码器，利用了 64 位 PCI 扩展的任何 PCI 板，都应该有 64 位的可编程解码器。

实际的实现，以及可编程解码器的使用，因如下事实而简化：一个区域中的字节数，通常是 2 的幂，例如 32 字节、4 KB 或者 2MB 等等。另外，也无需考虑将区域映射到不对齐地址的情况，1MB 的区域会对齐在 1MB 倍数的地址处，而 32 字节的区域会在 32 的倍数处对齐。PCI 规范利用了这种对齐，它指出地址解码器只需看到地址总线上的高位，而且仅仅高位是可编程的。这种约定也意味着任意区域的大小都必须是 2 的幂。

将一个 PCI 区域映射到物理地址空间的工作，通过在配置寄存器高位中设置适当的值而实现。例如，对 1 MB 区域，它有 20 位的地址空间，通过设置寄存器的高 12 位来进行重新映射，这样，为了让 PCI 板具有 64 MB 到 65MB 的地址范围，可向寄存器写入 0x040xxxxx 范围的任意一个地址。实际情况下，只有很高的地址才用来映射 PCI 区域。

“部分解码”技术还有一个附加的有点，软件可以检查配置寄存器中的非可编程位数来判断 PCI 区域的大小。为此，PCI 标准指出不用的位必须在读取时始终为 0。标准还要求 I/O 区域的最小大小是 8 字节，而内存区域的最小范围是 16 字节，这样，就可以在基地址寄存器的低位中保存一些额外的信息：

- 位 0 位“空间 (space)”位。如果区域映射到内存地址空间，则设置为 0；如果映射到 I/O 地址空间，则设置为 1。
- 位 1 和 2 是“类型 (type)”位：内存区域可标记为 32 为区域、64 位区域或者“必须映射到 1 MB 以下的 32 位区域”（已废弃的 x86 特有类型，现在已不再使用）。
- 位 3 是“可预取 (prefetchable)”位，用于内存区域。

读到这里，读者应该知道资源标志的来源了。

检测 PCI 区域的大小，可利用 `<linux/pci.h>` 中定义的若干位掩码来简化：如果是内存区域，则 `PCI_BASE_ADDRESS_SPACE` 位掩码被设置为 `PCI_BASE_ADDRESS_SPACE_MEMORY`，

是 I/O 区域时，设置为 `PCI_BASE_ADDRESS_IO`。若要知道映射后内存区域的实际地址，可将 PCI 寄存器和 `PCI_BASE_ADDRESS_MEM_MASK` 进行“与”操作，以便丢弃前面那些低位值。对 I/O 寄存器，应使用 `PCI_BASE_ADDRESS_IO_MASK`。需要注意的是，设备制造商可能以任意顺序使用 PCI 区域。使用第一个和第三个区域，却留下第二个区域不用的设备，还是很常见的。

下面将给出报告 PCI 区域当前位置和大小的典型代码。这段代码是 `pciregions` 模块的一个部分，和 `pcidata` 在同一目录中发布。该模块建立一个 `/proc/pciregions` 文件，并使用先前给出的代码生成数据。该程序将一个全 1 的值写入配置寄存器，然后读取这些值，以便了解寄存器中哪些位可被编程。注意在这个程序探测配置寄存器时，设备实际被映射到了物理地址空间的顶端，这就是探测过程中禁止中断报告的原因（这样可在将区域映射到错误的地点时，避免驱动程序访问该区域）。

尽管 PCI 规范指出 I/O 地址空间是 32 位宽，但某些制造商明显倾向于 x86 平台，假设它为 64 KB，而不会实现基地址寄存器的所有 32 位。这就是下面的代码（以及内核）忽略 I/O 区域地址掩码高位的原因。

```
static u32 addresses[] = {
    PCI_BASE_ADDRESS_0,
    PCI_BASE_ADDRESS_1,
    PCI_BASE_ADDRESS_2,
    PCI_BASE_ADDRESS_3,
    PCI_BASE_ADDRESS_4,
    PCI_BASE_ADDRESS_5,
    0
};

int pciregions_read_proc(char *buf, char **start, off_t offset,
                        int len, int *eof, void *data)
{
    /* this macro helps in keeping the following lines short */
#define PRINTF(fmt, args...) sprintf(buf+len, fmt, ## args)
    len=0;

    /* Loop through the devices (code not printed in the book) */

    /* Print the address regions of this device */
    for (i=0; addresses[i]; i++) {
        u32 curr, mask, size;
        char *type;

        pci_read_config_dword(dev, addresses[i], &curr);
        cli();
        pci_write_config_dword(dev, addresses[i], ~0);
        pci_read_config_dword(dev, addresses[i], &mask);
        pci_write_config_dword(dev, addresses[i], curr);
        sti();

        if (!mask)
            continue; /* there may be other regions */

        /*
         * apply the I/O or memory mask to current position.
         * note that I/O is limited to 0xffff, and 64-bit is not
         * supported by this simple implementation
         */
        if (curr & PCI_BASE_ADDRESS_SPACE_IO) {
            curr &= PCI_BASE_ADDRESS_IO_MASK;
        } else {
            curr &= PCI_BASE_ADDRESS_MEM_MASK;
        }
    }
}
```

```

    }

    len += PRINTF("\tregion %i: mask 0x%08lx, now at 0x%08lx\n",
        i, (unsigned long)mask,
        (unsigned long)curr);
    /* extract the type, and the programmable bits */
    if (mask & PCI_BASE_ADDRESS_SPACE_IO) {
        type = "I/O"; mask &= PCI_BASE_ADDRESS_IO_MASK;
    } else {
        type = "mem"; mask &= PCI_BASE_ADDRESS_MEM_MASK;
    }
    size = (~mask + 1) & 0xffff; /* Bleah */
    len += PRINTF("\tregion %i: type %s, size %i (%i%s)\n", i,
        type, size,
        (size & 0xffff) == 0 ? size >> 20 :
        (size & 0x3ff) == 0 ? size >> 10 : size,
        (size & 0x3ff) == 0 ? "MB" :
        (size & 0x3ff) == 0 ? "KB" : "B");

    if (len > PAGE_SIZE / 2) {
        len += PRINTF("... more info skipped ...\n");
        *eof = 1; return len;
    }
}
return len;
}

```

下面是 `/proc/pciregions` 给出的帧捕获器的区域报告：

```

Bus 0, device 13, fun 0 (id 8086-1223)
  region 0: mask 0xfffff000, now at 0xf1000000
  region 0: type mem, size 4096 (4KB)

```

值得注意的是，该程序所报告的内存大小有时会有点夸大事实。例如，`/proc/pciregions` 报告某个显卡有 **16 MB** 的内存，但实际上只有 **1 MB**。这是可接受的，因为这个大小信息仅仅被固件用来分配地址范围。对了解设备细节的驱动程序编写者来说，超过实际情况的大小并不是问题，他们能够正确处理固件赋予的地址范围。在这种情况下，无需修改 **PCI** 寄存器，就可添加更多的设备 **RAM**。

如果存在这种夸大的情况，也会反映在资源接口中，这时，`pci_resource_size` 将报告夸大后的大小。

15.1.6 PCI 中断

PCI 很容易处理中断。在 **Linux** 的引导阶段，计算机固件已经为设备赋予一个唯一的 **中断号**，驱动程序只需使用该 **中断号**。**中断号**保存在配置寄存器 **60** (`PCI_INTERRUPT_LINE`) 中，该寄存器为一个字节宽。这允许多达 **256** 个 **中断线**，实际受到 **CPU** 的限制。驱动程序无需检测 **中断号**，因为从 `PCI_INTERRUPT_LINE` 中找到的值是保证正确的。

如果设备不支持中断，寄存器 **61** (`PCI_INTERRUPT_PIN`) 是 **0**，否则为非零。但是，因为驱动程序知道自己的设备是否是中断驱动的，因此，通常不需要读取 `PCI_INTERRUPT_PIN` 寄存器。

这样，处理中断的 **PCI** 相关代码仅仅需要读取配置字节，以便获得 **中断号**，如下代码所示。否则，

要利用第 9 章的内容。

```
result = pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &myirq);
if (result) { /* deal with error */ }
```

本节其余内容为好奇的读者提供了一些附加信息，但这些信息对编写驱动程序没有多少帮助。

PCI 连接头有四个中断引脚，外设板可使用其中任意一个。每个引脚被独立连接到主板的中断控制器，因此，中断可被共享，而不会出现任何电气问题。然后，中断控制器负责将中断线（引脚）映射到处理器硬件。这一依赖于平台的操作由控制器完成，这样，总线本身可以获得平台无关性。

位于 `PCI_INTERRUPT_PIN` 的只读配置寄存器用来告诉计算机，实际使用的是哪个引脚。要注意每个设备板可拥有 8 个设备，而每个设备使用单独的中断引脚，并在自己的配置寄存器报告引脚的使用情况。同一设备板上的不同设备可使用不同的中断引脚，或者共享同一个中断引脚。

另一方面，`PCI_INTERRUPT_LINE` 寄存器是可读/写的。在计算机的引导阶段，固件扫描其 PCI 设备，并根据每个 PCI 槽的中断引脚连接情况设置每个设备的寄存器。这个值由固件赋予，是因为只有固件知道主板如何将不同的中断引脚连接至处理器。但是，对设备驱动程序，`PCI_INTERRUPT_LINE` 是只读的。有意思的是，新近的 Linux 内核在某些情况下，无需借助 BIOS 就可以分配中断线。

15.1.7 处理热插拔设备

在 2.3 开发周期中，内核开发人员检查了 PCI 编程接口，以便简化接口并支持热插拔设备，也就是说，某些设备可在系统运行时添加或删除（比如 CardBus 设备）。本节介绍的内容并不适合 2.2 和更早的内核，但对新驱动程序来讲，应该是首选的处理方法。

这里的基本思想是，在系统生命周期中，无论何时出现一个新的设备，所有可用的设备驱动程序都必须检查新设备，以判断新设备是否属于自己。因此，能够处理热插拔设备的驱动程序必须在内核中注册一个对象，而不是使用经典的 `init` 和 `cleanup` 入口点。该对象的 `probe` 函数将用来检查系统中的设备，其结果是，要么接管该设备，要么不予考虑。

这种方法没有终点：通常情况下，即静态的设备清单，在系统引导期间扫描设备清单一次。如果不存在对应的设备，模块化的驱动程序将被卸载，而监视总线的外部进程将在需要时再次装载这些驱动程序。这就是 PCMCIA 子系统以前的工作方式，现在已经集成到内核当中，从而可以利用类似的方法在其它不同的硬件环境下处理类似的问题。

但是读者也许会提出反对意见，即可热插拔的 PCI 设备现在不太常见。然而，新的驱动程序对象技术也可以对那些需要处理大量不同设备的非热插拔驱动程序提供很大帮助。初始化代码可被简化并流线化，从而只需根据一个已知设备清单来检查“当前”的设备，而无需主动查询 PCI 总线，这种方法需要循环调用 `pci_find_class` 一次，或循环调用 `pci_find_device` 多次。

但让我们分析下面的代码。这段代码利用了 `<linux/pci.h>` 中定义的 `pci_driver`，该结构定义了它所实现的操作，并包含由它所支持的设备清单（为避免对其代码的多余调用）。一句话，下面的代码表明如何针对一个假想的“热插拔 PCI 模块（hot plug PCI module, HPPM）”进行初始化和清

除处理。

```
struct pci_driver hppm_driver = { /* .... */ };

int hppm_init_module(void)
{
    return pci_module_init(&hppm_driver);
}

int hppm_cleanup_module(void)
{
    pci_unregister_driver(&hppm_driver);
}

module_init(hppm);
module_exit(hppm);
```

读者已经看到，整个过程非常简单。内部细节隐藏在 `pci_module_init` 的实现、以及驱动程序内部结构之中。我们将自顶而下讲述相关函数：

```
int pci_register_driver(struct pci_driver *drv);
```

该函数将驱动程序插入到由系统维护的一个链表中。已编译的设备驱动程序利用该函数执行它们的初始化，模块化的代码不直接使用该函数。函数的返回值是由该驱动程序处理的设备个数。

```
int pci_module_init(struct pci_driver *drv);
```

该函数封装了上面那个函数，并提供给模块化的代码调用。当成功时返回 0，而在未发现设备时返回 `-ENODEV`。这样，就可以避免在无设备时仍将模块驻留在内存中（而在匹配的设备出现时，期望该模块能够被自动装载）。因为该函数定义为内嵌函数，所以它的行为在 `MODULE` 未被定义时会有所不同，因此，对非模块化的代码，该函数甚至可作为 `pci_register_driver` 的替代函数。

```
void pci_unregister_driver(struct pci_driver *drv);
```

该函数将指定的驱动程序从已知驱动程序链表中删除。

```
void pci_insert_device(struct pci_dev *dev, struct pci_bus *bus);
void pci_remove_device(struct pci_dev *dev);
```

这两个函数实现了热插拔系统的另一面。它们由事件处理器调用，而事件处理器和总线报告的插入/拔出事件相关联。

```
struct pci_driver *pci_dev_driver(const struct pci_dev *dev);
```

该工具函数查找与某个设备相关联的驱动程序（如果存在）。`/proc/bus` 的支持函数使用这个函数，但对设备驱动程序来讲，没有多少意义。

pci_driver 结构

`pci_driver` 数据结构是热插拔支持的核心结构，我们将详细描述这个数据结构。该结构其实很小，只有几个方法以及一个设备 ID 清单。

```
struct list_head node;
```

用来管理驱动程序链表。这个链表是第 10 章“链表”一节中介绍过的通用链表，对设备驱动程序来讲意义不大。

```
char *name;
```

驱动程序名称，日志消息中使用该名称。

```
const struct pci_device_id *id_table;
```

是个数组，其中列出了该驱动程序所支持的设备。当有设备和这个数组中列出的项匹配时，才会调用 `probe` 方法。如果该成员被指定为 `NULL`，将对系统中的所有设备调用 `probe` 函数。如果该成员不为 `NULL`，则数组中的最后一项必须设置为 `0`。

```
int (*probe)(struct pci_dev *dev, const struct pci_device_id *id);
```

该函数必须初始化传递进入的设备，并且在成功时返回 `0`，而在失败时返回负的错误码（实际上，错误码当前还没有被用到，但最好还是返回一个错误值，而不是 `-1`）。

```
void (*remove)(struct pci_dev *dev);
```

`remove` 方法用来告诉设备驱动程序，它应该关闭设备，并且停止对其的处理，释放任何关联的内存。该函数在两种情况下被调用：当设备从系统中移走时，或者当驱动程序调用 `pci_unregister_driver` 从系统中卸载时。和 `probe` 不同，这个方法是针对某个 `PCI` 设备的，而不是针对该驱动程序所处理的所有设备集合，该特定设备通过参数传递进入。

```
int (*suspend)(struct pci_dev *dev, u32 state);
```

```
int (*resume)(struct pci_dev *dev);
```

上述函数是 `PCI` 设备的电源管理函数。如果设备驱动程序支持电源管理功能，则应该实现这两个方法，以便关闭并激活设备。这些函数由高层代码在适当的时间调用。

`PCI` 驱动程序对象相当直接，而且好用。笔者认为无需对这些成员做进一步说明，因为通常的硬件处理代码能够很好地适合这些抽象函数。

现在唯一未作解释的是 `struct pci_device_id` 对象。该结果包含了若干 `ID` 成员，要驱动的实际设备必须匹配所有的成员。设置成 `PCI_ANY_ID` 的成员，告诉系统忽略对应的 `ID`。

```
unsigned int vendor, device;
```

该驱动程序感兴趣的设备的 `vendor` 和 `device` `ID`。这两个值分别和 `PCI` 配置空间中的 `0x00` 和 `0x02` 寄存器相匹配。

```
unsigned int subvendor, subdevice;
```

即子 `ID`，和 `PCI` 配置空间中的 `0x2C` 和 `0x2E` 寄存器匹配。因为有时一对 `vendor/device` `ID` 可能会标识一组设备，而驱动程序只能支持其中的一部分，所以要使用这两个子 `ID` 进行进一步的匹配。

```
unsigned int class, class_mask;
```

如果设备驱动程序要处理整个一个类，或者某个子集，就可以将前面的成员设置为 `PCI_ANY_ID`，同时使用 `class` 标识符。`class_mask` 的存在，可让驱动程序处理某个基类，或者只是其中的子类。如果使用 `vendor/device` 标识符选择设备，则这两个成员都必须设置为 `0`（而不是 `PCI_ANY_ID`，因为相关的检查通过和掩码成员的逻辑与操作完成）。

```
unsigned long driver_data;
```

该成员留给设备驱动程序自己使用。举个例子，可利用该成员在编译阶段区别各个不同的设备，从而避免运行时冗长的条件判断。

值得注意的是，`pci_device_id` 数据结构只是提供给系统的一个暗示；实际的设备驱动程序仍然可以自由地从 `probe` 方法中返回非零，这样，即使设备和设备标识符数组中的某项匹配，也会拒绝该设备。举个例子，如果存在若干具有相同签名的设备，驱动程序可以在确定是否能够驱动该外设之前，进一步查询其它信息。

15.1.8 硬件抽象

到此为止，我们知道了系统是如何处理市场上各种各样的 PCI 控制器的，也已经完整讨论了 PCI 总线。本节将提供其它一些资料，以帮助感兴趣的读者了解一些内核是如何将面向对象的软件层扩展到最低层的硬件的。

用来抽象硬件的机制，就是包含方法的普通结构。这是一种强有力的技术，它在普通的函数调用开支之上，仅仅多了反引用指针这样一点最小的开支。在 PCI 管理中，唯一依赖于硬件的操作是完成配置寄存器读取和写入的操作，而 PCI 领域中的其它任何工作，都是通过直接读取和写入 I/O 及内存地址空间完成的，这些工作，都可以在 CPU 的直接控制下完成。

为此，实现硬件抽象的相关结构，仅仅包含 6 个成员：

```
struct pci_ops {
    int (*read_byte)(struct pci_dev *, int where, u8 *val);
    int (*read_word)(struct pci_dev *, int where, u16 *val);
    int (*read_dword)(struct pci_dev *, int where, u32 *val);
    int (*write_byte)(struct pci_dev *, int where, u8 val);
    int (*write_word)(struct pci_dev *, int where, u16 val);
    int (*write_dword)(struct pci_dev *, int where, u32 val);
};
```

该结构在 `<linux/pci.h>` 中定义，并由 `drivers/pci/pci.c` 使用，后者定义了实际的公用函数。

处理 PCI 配置空间的 6 个函数，比起反引用单个指针来，要花费更多的开支，这是因为代码是高度面向对象的，所以使用了级联指针。但这个开支并不是一个问题，因为这些操作的执行次数非常少，而且也从来不会在速度关键的地方调用。例如，`pci_read_config_byte` 函数的实际实现将扩展为：

```
dev->bus->ops->read_byte();
```

系统中的各个 PCI 总线在引导阶段检测，这时，`struct pci_bus` 结构以及相关功能被建立，其中包括 `ops` 成员。

通过“硬件操作”数据结构实现硬件抽象，在 Linux 中很典型。一个重要的例子是 `struct alpha_machine_vector` 数据结构。该结构在 `<asm-alpha/machvec.h>` 中定义，并用来处理各种 Alpha 计算机之间的不同。

15.2 回顾 ISA

ISA 总线在设计上相当老旧，并且其性能也很差，但是，它仍然占有很大一部分的扩展设备市场。如果要支持老主板，而速度不是非常重要时，ISA 比起 PCI 要占些优势。ISA 这个老标准的另外

一个优点是，如果你是一位电子爱好者，则可以非常容易地设计开发自己的 ISA 设备，而如果要独自开发 PCI 设备，有时简直是不可能的。

另一方面，ISA 的最大不足在于它紧紧绑定在 PC 架构上，其接口总线拥有 80286 处理器的所有限制，从而经常让系统程序员头疼。ISA 的另外一个大问题（来自最初的 IBM PC），是缺少位置的寻址，从而导致许多问题，而且在添加新设备时，要不断修改跳线并测试。值得注意的是，最老的 Apple II 计算机都采用了位置寻址方法，从而可以装备无跳线的扩展板卡。

尽管 ISA 总线有如此大的缺点，但仍然应用于若干意想不到的领域。例如，MIPS 处理器的 VR41xx 系列，在几种掌上型电脑中装备有 ISA 兼容的扩展总线，但看起来完全不同。这种意想不到的应用，其背后的原因是某些基于 ISA 的传统硬件的成本非常低廉，比如基于 8390 的以太网卡，这样，利用 ISA 电气信号的 CPU 就能够非常容易地利用这种便宜的 PC 设备，尽管从设计上讲，这种接口非常糟糕。

15.2.1 硬件资源

一个 ISA 设备可配备有 I/O 端口、内存区域以及中断线。

尽管 x86 处理器支持 64 KB 的 I/O 端口地址（也就是说，处理器有 16 条地址线），但某些老式的 PC 硬件只能处理最低的 10 条地址线。这就将可用地址空间限制在 1024 个端口，因为任何只能处理低 10 位地址线的设备，都会错误地将 1KB 到 64KB 范围内的地址看成是低地址。某些外设巧妙地利用这一限制，将端口映射到了低 1KB 字节，而使用高地址线来选择不同的设备寄存器。例如，映射到 0x340 端口的设备，也可以安全使用 0x740、0xB40 等端口。

如果说可用的 I/O 端口受到限制，内存访问情况更加糟糕。ISA 设备只能使用 640 KB 和 1 MB 之间，以及 15 MB 和 16 MB 之间的内存。640 KB 到 1 MB 的范围由 PC BIOS、VGA 兼容适配器，以及其它各种设备使用，新设备能用的空间就非常有限了。另一方面，Linux 不直接支持 15 MB 处的内存访问，如果想通过修改内核来支持对该范围的内存访问，现在已经得不偿失了。

ISA 设备板可利用的第三个资源是中断线。连接到 ISA 总线的中断线非常有限，而且由所有的接口板卡共享。这样，如果设备未经正常配置，将出现多个不同设备使用同一中断线的结果。

尽管最初的 ISA 规范不允许在设备间共享中断，但大多数设备板都允许中断的共享。^{*}软件方面的中断共享在第 9 章“中断共享”中讲述。

15.2.2 ISA 编程

对编程而言，内核和 BIOS 都可以无需帮助而访问 ISA 设备（这点上和 PCI 不同）。我们能利用的设施，只有 I/O 端口寄存器以及 IRQ 线，相关论述，可参阅“使用资源”（第 2 章）和“安装中断处理程序”（第 9 章）。

^{*} 中断共享的问题涉及到电气特性：如果某个设备通过低阻抗电平驱动信号线成为无效状态，则无法实现中断共享。另一方面，如果设备使用拉升电阻导致无效状态，则共享就是可能的。这是现在使用的标准。但是，因为 ISA 中断是边缘触发的，而不是电平触发的，因此，就有可能丢失中断信号。边缘触发的中断在硬件级别很容易实现，但却无法安全共享中断。

本书第一部分中讲述的所有编程技术都可以应用于 ISA 设备。驱动程序可以探测 I/O 端口，而中断线可利用第 9 章中“自动检测 IRQ 号”中描述的技术进行自动检测。

第 8 章“使用 I/O 存储器”中简要介绍过辅助函数 `isa_readb` 以及其它相关函数，这里不再赘述。

15.2.3 即插即用规范

某些新的 ISA 设备板遵循特殊的设计规则，并且需要一个特殊的初始化序列，以便简化附加接口板的安装和配置。这一规范称为“即插即用 (PnP)”，其中包括一堆建立和配置无跳线 ISA 设别的笨重规则。PnP 设备实现了 I/O 区域的重新分配，而 PC BIOS 负责重新分配(有点类似 PCI)。

简而言之，PnP 的目标就是为了获得类似 PCI 设备那样的灵活性，而无需修改低层的电气接口(即 ISA 总线)。为此，该规范定义了一组设备无关的配置寄存器，以及通过位置寻址接口板的方法——但物理总线并不支持各板独立的(位置相关的)连线，因为每个 ISA 信号线都会连接到每个插槽。

位置寻址通过赋予计算机中的每个 PnP 外设一个小整数，即“Card Select Number (CSN)”来工作。每个 PnP 设备配备有一个唯一的顺序标识号，有 64 位宽，并且硬编码到外设板中。CSN 赋值过程利用该唯一顺序标识号来标识 PnP 设备。但是，只能在引导阶段对 CSN 进行安全赋值，因此，需要 BIOS 能够处理 PnP 设备。出于这个原因，老的计算机需要用户获得并插入一张特殊的配置磁盘，这样才能识别 PnP 设备。

遵循 PnP 规范的接口板在硬件上比较复杂。比起 PCI 板来，它们更为精细，而且要求更为复杂的软件。安装这些设备时，一样会遇到麻烦，即使安装很顺利，也仍然要面对性能限制，以及有限的 IAS 总线 I/O 空间等问题。因此，只要可能，应该尽量安装 PCI 设备。

如果读者对 PnP 配置软件感兴趣，可浏览 `drivers/net/3c509.c`，这个驱动程序的探测函数处理 PnP 设备。Linux 2.1.33 在 `drivers/pnp` 中添加了对 PnP 的初始支持，也可一看。

15.3 PC/104 和 PC/104+

在工业界，当前有两种非常流行的总线结构：PC/104 和 PC/104+，它们都是 PC 类单板计算机的标准。

这两个总线规定了印刷电路板的外形因素，以及板间互连的电气/机械规范。这种总线的真正好处在于，可以利用设备一面的插座连接器将多个电路板垂直堆集起来。

这两个总线的电子和逻辑布局分别和 ISA (PC/104) 及 PCI (PC/104+) 一样，因此，软件不会注意到它们和通常桌面总线之间的不同。

15.4 其它 PC 总线

PCI 和 ISA 是 PC 领域两种最常用的外设接口，但它们并不是唯一的 PC 总线。这里给出其它一些能在 PC 市场上找到的总线特点。

15.4.1 MCA

微通道结构（Micro Channel Architecture, MCA）是在 PS/2 计算机和某些笔记本电脑中使用的 IBM 标准。微通道的最主要问题是其文档很少见，从而导致到现在为止，Linux 中也没有对 MCA 的良好支持。

在硬件级别，微通道比起 ISA 来有许多特点。它支持多主体（multimaster）DMA、32 位地址和数据线、共享中断线，以及用来访问各板卡上配置寄存器的位置寻址等等。这些寄存器称为“Programmable Option Select (POS)”，但却没有 PCI 寄存器的所有特征。Linux 对微通道的支持包括一些模块可用的函数。

设备驱动程序可以读取整数值 `MCA_bus`，以便判断是否运行在微通道计算机上，这点和使用 `pci_present` 判断是否存在 PCI 设备非常类似。如果该符号是一个预处理宏，则会定义 `MCA_bus__is_a_macro` 宏。如果 `MCA_bus__is_a_macro` 未被定义，则 `MCA_bus` 是一个整型变量，可由模块化代码访问。`MCA_bus` 和 `MCA_bus__is_a_macro` 在 `<asm/processor.h>` 中定义。

15.4.2 EISA

扩展 ISA (EISA) 总线是对 ISA 总线的 32 扩展，同时具有兼容的接口连接器，也就是说，ISA 设备可以插入 ISA 连接器。附加的线路在 ISA 连接器之下走线。

类似 PCI 和 MCA，EISA 总线也为无跳线设备设计，并具有和 MCA 一样的特点：32 位地址和数据线、多主体 DMA，以及共享中断线。EISA 设备由软件配置，而不需要操作系统的任何特殊支持。Linux 内核中已经有一些 EISA 驱动程序，主要是以太网设备和 SCSI 控制器。

EISA 驱动程序检查 `EISA_bus` 的值判断是否存在 EISA 总线。和 `MCA_bus` 类似，`EISA_bus` 可以是宏，也可以是变量，取决于是否定义有 `EISA_bus__is_a_macro` 宏。这两个符号均定义在 `<asm/processor.h>` 中。

对驱动程序而言，内核中无需对 EISA 的特殊支持，而程序员必须自己处理 ISA 扩展。驱动程序使用标准的 EISA I/O 操作来访问 EISA 寄存器。内核中已有的驱动程序可作为参考样例。

15.4.3 VLB

另外一个对 ISA 的扩展是 VESA 局部总线 (VESA Local Bus, VLB) 接口总线，这个总线将 ISA 连接器进行了扩展，添加了第三个纵向插座。设备可以插入这个额外的连接器（而不需要插入另外两个 ISA 连接器插槽），因为 VLB 插槽重复了 ISA 连接器中的所有重要信号。这种不使用 ISA 槽的“独立”的 VLB 外设很少见，因为大多数设备需要到达后面板，这样才能连接到外部连接器上。

比起 EISA、MCA 和 PCI 总线，VESA 总线在功能上有更多的限制，因此正在从市场上消失。对 VLB，内核中也不存在任何特殊支持。但是，Linux 2.0 中的 Lance Ethernet 驱动程序和 IDE 磁盘驱动程序可处理这些设备的 VLB 版本。

15.5 SBus

现今大部分计算机装备 PCI 或 ISA 接口总线的时候，大部分不太新的 SPARC 工作站使用 SBus 连接它们的外设。

尽管 SBus 存在很长一段时间了，但它具有相当高级的设计。尽管只有 SPARC 计算机使用该总线，但它的初衷却是处理器无关，并针对 I/O 外设板进行了优化。换句话说，我们可以将额外的 RAM 插入 SBus 插槽（RAM 扩展板已经从 ISA 领域消失，而 PCI 根本不支持 RAM 扩展板）。这种优化可简化硬件设备和系统软件的设计，其代价是主板更加复杂一些。

SBus 总线的这种 I/O 处理方法导致外设使用“虚拟”地址来传输数据，以跳过分配连续 DMA 缓冲区的需求。主板负责将虚拟地址解码并映射到物理地址。这要求在总线上附加 MMU（内存管理单元），负责该任务的芯片称为“IOMMU”。与使用物理地址的接口总线相比，这种设计似乎有些复杂，但因为 SPARC 处理器始终将 MMU 核心从 CPU 核心中分离（要么是物理上，要么至少是概念上），而使之大大简化。实际上，这种设计选择也被其它巧妙的处理器设计共享，从而获得整体上的好处。这种总线的另外一个好处是，不需要在所有外设中实现地址解码器并处理地址冲突。

SBus 外设在其 PROM 中使用 Forth 语言来初始化它们自身。选择 Forth 的原因是，其解释器是轻量级的，因此可在所有计算系统中得以实现。另外，SBus 规范描述了引导过程，因此，遵循该规范的 I/O 设备能很容易地适合系统，并且在系统引导时得以识别。对支持多平台的设备来讲，这一步意义非凡，这完全不同于 PC 为中心的 ISA 领域。但是，因为许多商业原因，这个总线并未取得成功。

尽管当前内核版本对 SBus 设备提供相当完善的支持，但该总线已经很少用到，因此不值得在这里详述。感兴趣的读者可以查看 `arch/space/kernel` 和 `arch/sparc/mm` 中的源文件。

15.6 NuBus

另外一个已经被遗忘的接口总线是 NuBus，可在老式 Mac 计算机（使用 M68k 家族 CPU）中找到。

所有的总线都是内存映射的（类似 M68k 中的所有东西），而且设备只能通过位置寻址。这是 Apple 的象征，因为更老式的 Apple II 都已经具备类似的总线设计，也是它的好处。不好的一面在于，几乎不太可能找到任何有关 NuBus 的文档，这归咎于 Apple 在 Mac 计算机上一贯遵循的封闭所有东西的策略（不同于先前的 Apple II 系统，其源代码和图表可以非常低的代价获得）。

`drivers/nubus/nubus.c` 包含了我们就该总线所知道的一切，读起来也相当有趣。读者可以从中看出，开发人员利用了多少艰难的反向工程方法。

15.7 外部总线

接口总线领域，最近出现了一个新的家族：外部总线。这包括 USB、FireWire、和 IEEE1284（基于并口的外部总线）。这些接口某种程度上和老式的、非外部的技术，比如 PCMCIA/CardBUS，甚至 SCSI 类似。

从概念上讲，这些总线既不是功能完整的接口总线（比如 **PCI**），也不是哑的通讯通道（比如串口）。很难对利用其功能的软件进行分类，通常可划分为两各级别：硬件控制器的驱动程序（比如针对 **PCI SCSI** 适配器的驱动程序，或者早先在“**PCI 接口**”中描述过的 **PCI** 控制器），以及针对特定“客户”设备的驱动程序（比如处理一般 **SCSI** 磁盘的 **sd.c**，以及处理插入总线的板卡的 **PCI** 驱动程序）。

但是还有其它一些问题。排除 **USE**，对它们的支持要么尚不成熟，要么需要修正（后面这种情况尤其符合 **SCSI** 内核子系统，许多最好的内核黑客都报告说它不太优化）。

15.7.1 USB

USB，即 **Universal Serial Bus**（一致串行总线），是唯一一种足够成熟的外部总线，因此值得作些讨论。从拓扑上讲，**USB** 子系统不能称为总线，它更象一棵由若干点对点线路组成的树。线路中包括四根线（地、电源，以及两个信号线），它将设备和集线器（和双绞线以太网类似）连接起来。通常，**PC** 类计算机提供一个“根集线器”，并提供两个连接插座。我们可以在插座上接入设备或者其它集线器。

在技术层面，**USB** 总线没有什么令人激动的地方，因为它其实是一种单主体实现，主机不停地轮询各种设备。尽管该总线有此固有限制，但它仍有一些有意思的功能特征，比如，设备能够为其数据传输请求一个固定带宽，以便可靠支持视频和音频 **I/O**。**USB** 的另外一个重要特征是，它仅仅作为设备和主机的通讯通道，而不需要它所传递的数据存在特定的含义或结构。^{*}

在这点上，**USB** 不同于 **SCSI** 通讯方式，而类似标准的串行介质。

这些特征，以及 **USB** 的热插拔能力，使得 **USB** 成为一种便利的低成本机制，我们无需关闭计算机，打开盖子，甚至旋开螺丝钉，就可以将设备连接到计算机。**USB** 正在成为 **PC** 市场上的流行接口，但是尚不适合于高速设备，因为它的最大传输率是 **12 Mb** 每秒。

版本 **2.2.18**（及以后）、**2.4.x** 的内核支持 **USB**，所有计算机中的 **USB** 控制器属于两类之一，而这两类均包含在标准内核中。

15.7.2 编写 USB 驱动程序

对“客户”设备驱动程序而言，其驱动程序处理热插拔的方法和 **pci_driver** 方法类似：设备驱动程序在 **USB** 子系统中注册自己的驱动程序对象，其后使用 **vendor** 和 **device** 标识符来标识硬件的插入。

相关的数据结构是 **struct usb_driver**，典型用法如下：

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

static struct usb_driver sample_usb_driver = {
    name:        "sample",
```

^{*} 实际上，仍然有一些结构，但大部分只是为了满足几个预先定义的设备类型的通讯需求，比如，键盘不会分配带宽，而摄像头需要。

```

    probe:      sample_probe,
    disconnect: sample_disconnect,
};

int init_module(void)
{
    /* just register it; returns 0 or error code */
    return usb_register(&sample_usb_driver);
}

void cleanup_module(void)
{
    usb_deregister(&sample_usb_driver);
}

```

当新设备连接到系统中时（或者驱动程序被载入，而总线上已存在不明设备时），USB 内核子系统将调用该数据结构中声明的 `probe` 函数。

每个设备为系统提供 `vendor`、`device` 和 `class` 标识符来标识自己，这和 PCI 设备类似。`sample_probe` 的任务，就是查询它接收到的信息，并指出该设备是否属于自己。

如果设备属于自己，该函数返回一个非 `NULL` 指针，该指针将用来标识该设备。通常是指向设备特有数据结构的指针，该结构处于设备驱动程序的核心地位。

为了和设备交换信息，需要告诉 USB 子系统如何通讯。这个任务通过填充一个 `struct urb`（表示 USB request block）结构，并将其传递给 `usb_submit_urb` 执行。这个步骤通常由与设备特殊文件关联的 `open` 方法或者等价的函数来完成。

注意并不是所有的 USB 驱动程序需要请求自己的主设备号，并实现它自己的设备特殊文件。如果某个设备属于内核已提供一般性支持的类型，则不必建立自己的设备文件，而需要通过其它途径报告信息。

一般性管理的一个例子是输入处理。如果 USB 设备是一个输入设备（比如绘图板），就不需要为这个设备分配一个主设备，而只需调用 `input_register_device` 注册该硬件。在这种情况下，输入设备的 `open` 回调函数负责调用 `usb_submit_urb` 建立通讯。

因此，USB 输入驱动程序必须依赖于其它几个系统部分，大部分驱动程序可作为模块。USB 输入设备驱动程序的模块结构见图 15-3。

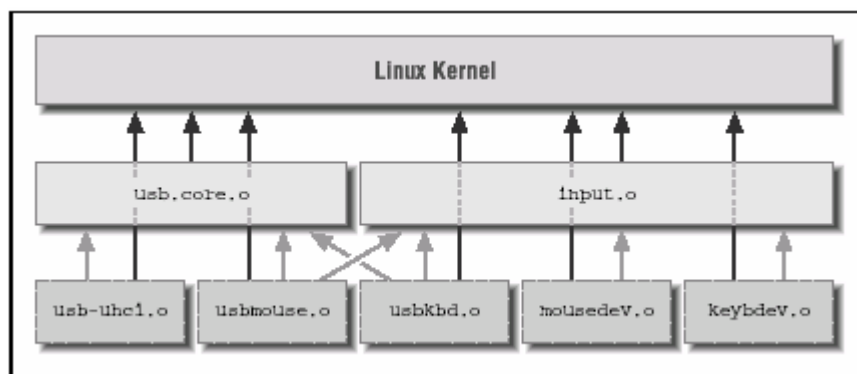


图 15-3: 与 USB 输入管理相关的模块

读者可从 O'Reilly FTP 站点上找到一个完整的 USB 设备驱动程序。它是一个很简单的键盘/鼠标驱动程序，只是为了说明如何设计一个完整的 USB 驱动程序。为了简单起见，该驱动程序并没有使用输入子系统来报告事件，而是使用 `printk` 打印消息。为测试该驱动程序，你需要准备一个 USB 键盘或鼠标。

现在有相当多的 USB 文档可获得，其中包括本书作者之一撰写的两篇论文，其风格和技术等级类似《Linux 驱动程序》。这些论文中甚至包含了更为完整的 USB 样例设备驱动程序，它使用了内核的输入子系统，如果没有 USB 设备，也可以通过其它途径得以运行。读者可在 <http://www.linux.it/kerneldocs> 上找到这两篇论文。

15.8 向后兼容性

当前内核中的 PCI 实现在版本 2.0 内核中并不存在。2.0 中的支持 API 非常原始，这是因为那时缺少本章描述过的各种对象。

早期版本中，我们可以使用访问配置空间的六个函数，它们接收 PCI 设备的 16 位低层键作为参数，而不使用指向 `struct pci_dev` 结构的指针。同时，我们必须在读写配置空间之前包含 `<asm/pci.h>` 头文件。

幸运的是，处理这些差异并不是个大问题，如果包含 `sysdep.h` 文件，则可以在 2.0 下使用与 2.4 一样的原语。版本 2.0 的 PCI 支持可从头文件 `pci-compat.h` 获得，在 2.0 下编译时，将自动被 `sysdep.h` 包含。`pci-compat.h` 实现了处理 PCI 总线的最重要函数。

如果你使用 `pci-compat.h` 开发能够在 2.0 到 2.4 任意版本上运行的驱动程序，则必须在使用完 `pci_dev` 时调用 `pci_release_device`。这是因为这个头文件为 2.0 建立的伪 `pci_dev` 结构是用 `kmalloc` 分配的，而 2.2 和 2.4 内核的真正结构是内核中的静态资源。在 2.2 和 2.4 内核中编译时，`sysdep.h` 定义其它函数不做任何事情，这样就不会有任何害处。读者可以从 `pciregions.c` 或 `pcidata.c` 中看到实际的可移植代码。

另一个 2.0 的相关差异是 PCI 的 `/proc` 支持。2.0 中没有 `/proc/bus/pci` 文件层次（实际上根本没有 `/proc/bus`），而只有一个 `/proc/pci` 文件。该文件的内容是二进制的，对人来讲，不可读。在 2.2 版本，我们可在编译阶段选择一个“向后兼容”的 `/proc/pci`，而在版本 2.4 中，该文件彻底废弃。

热插拔 PCI 驱动程序概念（以及 `struct pci_driver`）在版本 2.4 中出现。我们没有针对老版本提供向后的兼容宏。

15.9 快速参考

和往常一样，这个小节总结本章出现的符号。

```
#include <linux/config.h>
CONFIG_PCI
```

这个宏可用来对 **PCI** 代码进行条件编译。当一个 **PCI** 模块装载进入一个非 **PCI** 内核时，`insmod` 将说明有若干符号无法解析。

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

这个头文件包含 **PCI** 寄存器的符号名称，以及若干制造商和设备 ID 值。

```
int pci_present(void);
```

该函数返回一个布尔值，说明计算机是否存在 **PCI** 总线。

```
struct pci_dev;
struct pci_bus;
struct pci_driver;
struct pci_device_id;
```

这些结构代表涉及 **PCI** 管理中的对象。`pci_driver` 对 **Linux 2.4** 来说是全新的，而 `struct pci_device_id` 是围绕 `pci_driver` 工作的。

```
struct pci_dev *pci_find_device(unsigned int vendor, unsigned int device, struct pci_dev
    *from);
struct pci_dev *pci_find_class(unsigned int class, struct pci_dev *from);
```

上述函数用来查询设备链表，以找出匹配给定签名或者属于某个特定类的设备。如果没有找到，该函数返回 `NULL`。`from` 可用来继续搜索。在第一次调用这些函数时，`from` 必须为 `NULL`，如果要再次搜索，`from` 必须指向刚刚找到的设备。

```
int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
int pci_write_config_byte (struct pci_dev *dev, int where, u8 *val);
int pci_write_config_word (struct pci_dev *dev, int where, u16 *val);
int pci_write_config_dword (struct pci_dev *dev, int where, u32 *val);
```

这些函数用来读取或者写入 **PCI** 配置寄存器。尽管 **Linux** 内核处理了字节序问题，但在从单个字节装配多字节值时，程序员必须小心处理字节序问题。**PCI** 总线是 `little-endian`。

```
int pci_register_driver(struct pci_driver *drv);
int pci_module_init(struct pci_driver *drv);
void pci_unregister_driver(struct pci_driver *drv);
```

这些函数支持 **PCI** 驱动程序概念。预先编译的代码使用 `pci_register_driver` 函数（它返回该驱动程序管理的设备数目），而模块化的代码应该调用 `pci_module_init` 函数（该函数在系统存在一个和多个设备时返回 `0`，而当没有适合的设备插入系统时，返回 `-ENODEV`）。

```
#include <linux/usb.h>
#include <linux/input.h>
```

前一个头文件是 **USB** 相关信息所在的地方，因此，**USB** 设备驱动程序必须包含该文件。后一个定义了输入子系统的核心。**Linux 2.0** 中不存在这两个头文件。

```
struct usb_driver;
int usb_register(struct usb_driver *d);
void usb_deregister(struct usb_driver *d);
```

`usb_driver` 是 **USB** 设备驱动程序的主要部分。在模块装载和卸载阶段，必须进行注册和注销工作。

第 16 章 内核源代码的物理布局



到此为止，我们一直在从设备驱动程序的编写角度讨论 Linux 内核。然而，一旦我们真正开始在内核上工作，就会发现我们需要对内核在整体上有所了解。实际上，你会发现自己要不停地浏览内核源代码，并且要在源代码树上不停地进行 `grep` 操作，以便找出内核不同部分之间的关系。

本书的作者也经常要执行相当多的此类“过度 `grep`”任务，其实这是一种从源代码中检索信息的有效途径。现在，我们还可以利用 Internet 上的资源来理解内核源代码树，其中一些资源已经列在了本书前言中。但是，尽管有 Internet 资源可供利用，但使用 `grep`^{*}、`less` 或者 `ctags`、`etags` 等等工具，仍然是从内核源代码中得到有用信息的最好途径。

在笔者看来，坐在自己喜欢的 `shell` 提示符之前，获得一些必要的知识总是会有所帮助的。因此，本章在 Linux 2.4.2 版本基础上，将提供对内核源代码文件的一个概观。如果读者对其它版本感兴趣，某些描述也许不能直接照搬到除 2.4.2 之外的版本。有时可能出现整个部分都无法对应的情况，比如 2.4.0-test6 中引入的 `drivers/media` 目录，它是将先前存在的许多驱动程序挪到这个新目录而形成的。我们希望下面的内容会对你有所帮助，尽管某些内容并不适合其它内核版本。

本章中的所有路径名都是相对于源代码树的根给出的，根通常是 `/usr/src/linux`，而没有目录部分的文件名，则假定存在于“当前”目录下，也就是正在讨论的那个目录。头文件（利用 `<` 和 `>` 角括号时）是相对于源代码树的 `include` 目录给出的。我们不会仔细讲解 `Documentation` 目录，因为该目录的内容无需做出太多说明。

16.1 引导内核

看程序的通常方法是首先找到开始执行的入口，而谈及 Linux 时，却很难说出从“哪里”开始执行——这取决于我们如何定义“开始”。

和体系结构无关的起点是 `init/main.c` 中的 `start_kernel`。该函数从体系结构相关的代码中调用，而且 `start_kernel` 也从来不会返回到这些代码。它负责转起内核这个轮子，从而可以称之为“所有函数之母”，也可以把它看成是计算机生命的第一次呼吸。在 `start_kernel` 之前，整个代码处于

^{*} 通常，为了给 `grep` 建立命令行参数，还需要使用 `find` 和 `xargs` 命令。尽管 Unix 命令本身的使用很值得一讲，

“混沌”状态。

在 `start_kernel` 被调用的那个时刻，处理器已经被初始化，并进入了保护模式*处理器工作在最高的特权级（有时称为“超级模式”），并且中断被禁止。`start_kernel` 函数负责初始化所有的内核数据结构。该函数通过调用外部函数执行子任务来完成初始化，因为每个初始化函数都定义在对应的内核子系统中。

在获得内核锁，并打印 `Linux` 标语字符串之后，`start_kernel` 调用的第一个函数是 `setup_arch`。

对命令行的分析工作，通过调用与每个内核参数相关联的处理函数来完成（例如，`video=` 和 `video_setup` 相关联）。在关联设施被初始化后，每个函数通常在设置一些将要使用的变量之后返回。命令行分析的内部组织与 `initcall` 机制类似，将在后面描述。

在完成命令行分析之后，`start_kernel` 激活系统的各种基本功能设施，其中包括设置中断表，激活定时器中断，以及初始化控制台和内存管理等等。所有这些工作由平台相关代码所声明的函数完成。然后，该函数继续初始化一些基本的内核子系统，包括缓冲区管理，信号处理，以及文件和索引节点管理等。

最后，`start_kernel` 派生 `init` 内核线程（该线程的进程 ID 为 1），并执行 `idle` 函数（该函数也定义在体系结构相关的代码中）。

这样，最初的引导过程可总结如下：

- 系统固件和引导装载器准备内核并将其放在正确的内存地址上，相关代码通常在 `Linux` 源代码之外。
- 体系结构特有的汇编代码执行一些非常低层的任务，比如初始化内存，设置 CPU 寄存器以便 C 代码能够无缝执行等等，这包括选择栈区域，并适当设置栈指针。不同的平台，这类代码的总量也不尽相同，也许只有十几行，也许会有上千行。
- `start_kernel` 被调用。它获得内核锁，打印标语，并调用 `setup_arch`。
- 体系结构相关的 C 语言代码完成低层的初始化，并检索 `start_kernel` 可用的命令行。
- `start_kernel` 分析命令行，并根据能够识别的关键字调用处理程序。
- `start_kernel` 初始化基本设施并派生 `init` 线程。

其它初始化工作由 `init` 线程完成，该线程是同一文件 `init/main.c` 的一部分，而大多数初始化（`init`）调用由 `do_basic_setup` 执行。该函数初始化所有已找到的总线子系统（PCI、SBus 等等），然后调用 `do_initcalls`。设备驱动程序的初始化作为 `initcall` 过程的一部分而执行。

`initcall` 的思想在版本 2.3.13 中引入，而老的版本中并没有 `initcall`。`initcall` 主要用来避免在初始化代码中出现杂乱的 `#ifdef` 条件判断。每个可选的内核功能特色（设备驱动程序或其它）只能在已配置的情况下才能被初始化，因此，对初始化函数的调用，以往总是用类似 `#ifdef CONFIG_FEATURE` 和 `#endif` 这样的代码所包围。而利用 `initcall`，每个可选功能在它自己的初始化函数中声明，编译过程将该函数的引用放在一个特殊的 ELF 段中。在引导阶段，`do_initcalls`

但这已经超出了本书讨论的范围。

* 这个概念只存在于 x86 架构上，其它更多成熟的架构不会在上电时处于某种向后兼容的受限模式。

扫描这个 ELF 段，并调用所有相应的初始化函数。

类似思想也用于命令行参数中。每个能够在引导阶段接收命令行参数的驱动程序，定义一个数据结构，该结构将参数和一个函数关联起来。指向该数据结构的指针放在单独的 ELF 段中，因此，`parse_option` 可针对每个命令行选项扫描整个段，如果发现匹配，则调用关联的驱动程序函数。其余的参数将成为 `init` 进程的环境变量或命令行。`initcall` 和 ELF 段相关的代码包含在 `<linux/init.h>` 中。

不幸的是，`initcall` 这一思想只能用于初始化函数无顺序要求的条件下，因此，`init/main.c` 中仍然存在一些 `#ifdef` 条件。

值得看看 `initcall` 思想如何减少了代码中的条件编译代码量：

```
morgana% grep -c ifdef linux-2.[024]/init/main.c
linux-2.0/init/main.c:120
linux-2.2/init/main.c:246
linux-2.4/init/main.c:35
```

尽管新的 Linux 内核中增加了大量的新功能特色，条件编译语句的总量却戏剧性地减少，这归功于 `initcall`。这项技术的其它优点在于，设备驱动程序的维护者无需在每增加一条新的命令行参数时，对 `main.c` 文件打补丁。这项技术大大简化了内核中新功能特色的添加，而且引导代码中不再存在太多混乱的交叉引用。但是这项技术也有个副作用，2.4 不能编译成缺少 ELF 这种灵活性的老文件格式。出于这个原因，uCLinux^{*} 的开发者在从 2.0 移植到 2.4 时，只好从 COFF 转向 ELF 格式。

过分使用 ELF 段的另外一个不足是，编译内核的最后阶段并不等同于以往传统的连接阶段。每个平台现在都需要通过 `ldscript` 文件来明确定义如何连接内核映像（即 `vmlinux` 文件），该文件在每个平台的源代码树中称为 `vmlinux.lds`。ld 脚本的用法在 `binutils` 包的标准文档中有描述。

但将初始化代码放到特殊段中还有一个好处：一旦初始化结束，这些代码就不再需要了，因为这些代码已经被隔离，所以内核就可以将其废弃，并回收代码所占用的内存。

16.2 引导之前

在前面一节，我们将 `start_kernel` 看成是第一个内核函数。然而，读者也许会对 `start_kernel` 函数被调用之前发生的事情感兴趣，为此，我们将简单讨论“引导之前”这个主题。对这段内容不感兴趣的读者，可跳过本节而阅读下面的内容。

我们提到，`start_kernel` 之前运行的代码，其大部分是汇编代码，但是也有一些平台调用 C 库函数（最常见的是 `inflate`，即 `gunzip` 的核心）。

在大多数常见平台上，`start_kernel` 之前运行的代码主要用来展开内核——在计算机固件（有时在

^{*} uCLinux 是能够运行在无 MMU 处理器上的 Linux 版本，常用于嵌入式系统，比如没有硬件内存管理的 M68k 和 ARM 处理器上。uCLinux 表示 microcontroller Linux，其含义是，它运行在微型控制器上，而不是功能完整的计算机上。

引导装载器的帮助下) 已经将内核从其它存储介质, 比如本地磁盘或者网络上的远程工作站等, 装载到 RAM 之后。

我们也可以从一些体系结构相关的代码树中的 `boot` 目录中找到原始的引导装载器代码。例如, `arch/i386/boot` 目录中就包含有从软盘上装载内核并执行的引导代码。其中的 `bootsect.S` 文件只能从软盘上装载内核, 因此不能算作是一个完整的引导装载器 (例如, 它不能为内核传递命令行参数)。虽然如此, 将新内核复制到软盘上, 并从软盘快速启动新内核仍然是 PC 上一个非常便捷的方法。

众所周知, x86 平台有一个著名的限制, 在其启动时, CPU 只能看到 640K 的系统内存, 而无论系统中安装有多少内存。为了应付这个限制, 就必须压缩内核, 因此, `arch/i386/boot` 中包含有用来支持解压缩的代码, 还有用来设置 VGA 模式的代码。由于在 PC 上存在有这么一个限制, 所以我们不能在 `vmlinux` 内核映像上做任何事情, 而实际引导的文件称为 `zImage` 或 `bzImage`, 先前描述的引导扇区实际包含的是 `zImage` 或 `bzImage`, 而不是 `vmlinux`。我们不会继续花时间介绍 x86 平台上的引导过程, 因为可供选择的引导装载器很多, 而且相关的内容在其它地方有很好的论述。

某些平台具有大大不同于 PC 的引导代码设计。有时代码必须处理相同架构的不同变种, 例如, ARM、MIPS 和 M64k 等就是这样。这些平台涵盖了大量不同的 CPU 和系统类型, 其范围从高能服务器和工作站, 到 PDA 或嵌入式设备。不同的环境需要不同的引导代码, 有时甚至是编译内核映像的不同 `ld` 脚本。其中某些尚未包含到由 Linux 发布的正式内核源代码中, 而只能通过第三方的并行版本系统 (CVS) 获得, 这些 CVS 系统紧紧跟踪正式的源代码树, 但尚未合并到正式版本中。举例来说, 有用于 MIPS 工作站的 SGI CVS 树, 有用于基于 MIPS palm 计算机的 LinuxCE CVS 树。虽然如此, 我们还是要在该主题上多花点口舌, 因为这些不同的引导代码很有意思, 值得一读。`start_kernel` 之后的所有功能都基于这些额外的复杂性, 但却不会注意到这些代码。

尤其对嵌入式系统来说, 需要特殊的 `ld` 脚本和 `makefile` 规则, 尤其对 uCLinux 所支持的没有内存管理单元的系统来讲。如果没有硬件的 MMU 将虚拟地址映射成物理地址, 则必须将内核连接成在目标平台上, 能够从装载地点所在的物理地址执行的映像。在小型系统上, 连接内核以便能够将其装载到只读存储器 (通常是 FLASH 存储器) 的情况很常见, 这样, 内核能够在上电后无需任何引导装载器的帮助, 就可以开始运行。

如果内核要直接从 FLASH 存储器上执行, 则 `makefile`、`ld` 脚本以及引导代码之间要紧密配合。`ld` 规则将代码和只读段 (比如 `initcall` 信息) 放在 FLASH 存储器中, 而将数据段 (数据及以符号 BSS 开始的数据块) 放在系统 RAM 中, 其结果是两种段不再连续。然后, `makefile` 会提供一些特殊的规则, 以便将所有段合并到连续的地址, 并将其转换成适合于上载到目标系统的格式。这种合并操作是强制性的, 因为数据段中包含了已初始化的数据结构, 这些数据结构必须写入只读存储器, 否则就会丢失。最后, 运行在 `start_kernel` 之前的汇编代码必须将这些数据段从 FLASH 存储器复制到 RAM (即连接器指定的地址), 并将 BSS 段对应的地址范围清零。只有在这些重映射完成之后, C 语言代码才能开始运行。

在我们向目标系统上载新内核时, 系统中的固件从网络或者串口通道中检索数据文件, 并将其写入 FLASH 存储器。用来上载内核到目标计算机的中间格式, 会随着系统的变化而不同, 因为它依赖

于实际的上载如何发生。但是，无论是哪种情况，这一中间格式将是一种使用标准工具，传输已编译映像的二进制数据容器。例如，**BIN** 格式用来在网络上传输，而 **S3** 格式则是通过串口电缆，传输映像到目标系统的十六进制 **ASCII** 文件。^{*}

大多数情况下，系统上电时，用户可以选择引导 **Linux** 还是键入固件命令。

16.3 init 进程

当 **start_kernel** 派生出一个 **init** 线程（由 **init/main.c** 中的 **init** 函数实现）之后，它仍然运行在内核模式，**init** 线程也一样。当前面描述过的所有初始化均已完成之后，**init** 线程释放内核锁，并准备执行用户空间的 **init** 进程。将要执行的文件一般会保存在 **/sbin/init**、**/etc/init**，或者 **/bin/init**。如果上述文件均未找到，则会运行 **/bin/sh**，以作为实际 **init** 被丢失或破坏时的一种恢复手段。另外，用户也可以通过内核命令行指定 **init** 线程要执行的文件名称。

进入用户空间的过程很简单。代码通过 **open** 系统调用打开 **/dev/console** 作为标准输入，并调用 **dup** 将控制台复制为 **stdout** 和 **stderr**，最后，它调用 **execve** 执行用户空间程序。

该线程在内核模式下运行却能够调用系统调用的原因，是因为 **init/main.c** 在包含 **<asm/unistd.h>** 之前，已经声明了 **__KERNEL_SYSCALLS__**。这个头文件定义了一些特殊代码，以便允许内核代码调用有限的一些系统调用，就如同运行在用户空间一样。有关内核系统调用的更多信息，可参阅 <http://www.linux.it/kerneldocs/ksys>。

最后对 **execve** 的调用将完成向用户空间的转变，其中并没有涉及到什么特殊处理。和 **Unix** 中任意的 **execve** 调用一样，这个调用利用欲执行的二进制文件所定义的新内存映射覆盖当前进程的内存映射（读者应该知道执行一个文件，意味着将其映射到当前进程的虚拟地址空间）。在这种情况下，并不关心调用进程是否运行在内核空间。对 **execve** 的实现来讲，这是透明的，它只需在激活新的内存映射之前，找出需要释放的老内存映射。

无论系统设置或命令行参数如何，**init** 进程现在开始在用户空间执行，将来所有的内核操作，都是为了响应来自 **init** 本身，或者它派生出的进程的系统调用。

有关 **init** 进程如何启动整个系统的相信信息，可参阅 <http://www.linux.it/kerneldocs/init>。我们将继续讨论源代码目录中实现的系统调用，并描述设备驱动程序的布局，以及它们在源代码树中的组织方式。

16.4 kernel 目录

某些内核设施，比如和文件系统、内存管理以及网络相关的设施，保存在它们自己的源代码树中。源代码树中的 **kernel** 目录包含了其它所有的基本设施。

^{*} 我们不会详细描述这些格式或工具，因为研究嵌入式 **Linux** 的人们很容易获得这些信息。

这些设施中最为重要的是调度。这样，我们可将 `sched.c` 及 `<linux/sched.h>` 看成是 Linux 内核中最重要的源文件。除了调度器 `proper`（由 `schedule` 实现）之外，该文件定义了用来控制进程优先级以及所有用于睡眠和唤醒的机制。

`fork` 和 `exit` 系统调用分别由 `fork.c` 和 `exit.c` 源文件定义。这两个文件涉及的内容广泛，而且结构很好，涉及到进程创建和销毁的各个方面。

内核消息的传递在 `printk.c` 中实现，该文件用来进行控制台管理。控制台代码并非没有价值，因为“控制台”的概念现在已经很是抽象，包括文本屏幕（不管是原始的字符屏幕还是基于帧缓冲区的屏幕）、串行端口，甚至打印机端口。

该目录实现的其它设施有时间处理（`time.c`）、内核定时器（`timer.c`）、信号抵送和处理（`signal.c`）、模块管理和相关系统调用（`module.c`）、`kmod` 线程（`kmod.c`）、全系统的电源管理（`pm.c`）、`tasklet`（`softirq.c`），以及 `panic` 函数（`panic.c`）等。

16.5 fs 目录

文件处理处于任何 Unix 系统的核心地位，Linux 中的 `fs` 目录是所有目录当中最“胖”的一个。其中包含了当前 Linux 版本所支持的所有文件系统（每个文件系统保存在各自的子目录中），还包含除 `fork` 和 `exit` 之外的最重要的系统调用实现。

`execve` 系统调用位于 `exec.c` 文件，这个系统调用依赖于各种可用的二进制格式，以便于解释可执行文件中的二进制数据。当前最重要的二进制格式是 `ELF`，由 `binfmt_elf.c` 实现。`binfmt_script.c` 支持对解释性文件的执行。在检测到需要解释器时（通常由 `#!` 行指定，又称为“`shebang`”行），该文件依赖于其它二进制格式去装载解释器。

其它二进制格式（比如 `Java` 可执行格式），可由用户利用 `/proc` 中的一个接口定义，该接口在 `binfmt_misc.c` 中定义。`misc` 二进制格式可识别在可执行文件内容基础上经解释后的二进制格式，并传入适当的参数而调用适当的解释器。该工具可通过 `/proc/sys/fs/binfmt_misc` 配置。

用于文件访问的基本系统调用的定义在 `open.c` 和 `read_write.c` 中。前者还定义了 `close` 和其它几个文件访问系统调用（例如 `chown`）。`select.c` 实现了 `select` 和 `poll` 系统调用。`pipe.c` 和 `fifo.c` 实现了管道和命名管道。`readdir.c` 实现了 `getdents` 系统调用，它由用户空间程序用来读取目录（其名称表示“`get directory entries`，获取目录项”）。其它用来访问目录数据的编程接口（比如 `readdir` 接口），均在 `getdents` 系统调用的基础上，作为库函数在用户空间实现。

大多数用来操作文件的系统调用，比如 `mkdir`、`rmdir`、`rename`、`link`、`symlink` 和 `mknod` 等，在 `namei.c` 中实现，而 `namei.c` 利用了 `dcache.c` 中的目录项缓存。

文件系统的挂装和卸装，以及用于支持临时根文件系统的 `initrd`，在 `super.c` 中实现。

设备驱动程序编写者尤其感兴趣的是 `devices.c` 文件，它实现了字符和块设备驱动程序的注册，并作为所有设备方法的分发器。为此，它实现了一般性的 `open` 方法，该方法在找出对应的 `file_operations` 结构之前使用。块设备的 `read` 和 `write` 方法在 `block_dev.c` 中实现，而该文件

又委托 `buffer.c` 实现所有和缓冲区管理相关的工作。

该目录中还有其它一些文件，但这些文件的重要性较低。其中最为重要的是 `inod.c` 和 `file.c`，它们管理文件和索引节点数据结构的内部组织，另外还有用来实现 `ioctl` 的 `ioctl.c` 文件，以及实现配额的 `dquot.c` 文件。

我们提到，`fs` 的大部分子目录包含了各个文件系统的实现。但是 `fs/partitions` 并不是一个文件系统类型，其中却是分区管理代码。其中的某些文件会忽略内核配置而始终编译，而其它用来支持特定分区方案的一些文件，可单独使能或禁止。

16.6 mm 目录

内核源代码中最后一个主要目录是 `mm`，其中包含有内存管理代码。该目录中的文件实现了系统中所有与内存管理相关的数据结构。我们知道，内存管理和给定 CPU 的特有功能和寄存器有关，我们已经在第 13 章看到了大部分代码是如何做到与平台无关的。感兴趣的读者可从 `asm/arch-arch/mm` 中了解特定计算机平台的内存管理的最低层实现。

`kmalloc` 和 `kfree` 内存分配引擎在 `slab.c` 中定义，该文件是一个全新的实现，替换了原本的 `kmalloc.c` 文件。在版本 2.0 之后的内核中，`kmalloc.c` 已经不存在了。

大部分程序员对操作系统以块和页面管理内存的方式已经很熟悉了，但 Linux 使用了另外一个更加灵活的概念，即 `slab`（该思想来自 Sun Microsystem 的 Solaris）。每个 `slab` 是包含多个具有相同大小的内存对象的高速缓存。某些 `slab` 经过特殊处理，从而包含有内核特定部分使用的特定类型的结构，其它一些则更为通用，包含了 32 字节、64 字节等等的内存区域。使用 `slab` 的优点是结构或者其它内存区域能够以很小的开支缓存或重用，而“笨重”的分配和释放页面的技术，则很少被调用。

另外一个重要的分配工具，即 `vmalloc`，以及该函数赖以执行的函数，即 `get_free_pages`，分别定义在 `vmalloc.c` 和 `page_alloc.c` 中。这两个函数都很直接、简单，而且值得一读。

除了分配内存的服务之外，内存管理系统必须提供内存映射，这是因为 `mmap` 是许多系统活动的基础，比如文件的执行。但实际的 `sys_mmap` 函数并不在这个目录中实现，它隐藏在体系结构特有的代码中，这是因为超过 5 个参数的系统调用必须经过特殊调用，这主要涉及到 CPU 寄存器的使用。对所有平台通用的、用来实现 `mmap` 的函数是定义在 `mmap.c` 中的 `do_mmap_pgoff`。`mmap.c` 文件还实现了 `sys_sendfile` 和 `sys_brk`。后者可看成是独立的，因为 `brk` 主要用来增加进程可使用的最大虚拟地址。实际上，Linux（以及现代的大部分 Unix 系统）通过对 `/dev/zero` 进行内存页映射而为进程建立新的虚拟地址空间。

将普通文件映射为内存的机制可见 `filemap.c`，该文件操作内存管理系统中相当低层的数据结构。`mprotect` 和 `remap` 分别在 `mprotect.c` 和 `remap.c` 文件中实现，而内存锁则在 `mlock.c` 中实现。

当进程存在有多个内存映射时，需要一种在内存地址空间中寻找空闲区域的有限途径。为此，进程所有的内存映射都以 Adelson-Velski-Landis (AVL) 树的形式组织，该软件结构在 `mmap_avl.c` 中

实现。

交换文件的初始化和删除（即 `swapon` 和 `swapoff` 系统调用）实现包含在 `swapfile.c` 中。`swap_state.c` 用于实现交换缓存；页老化（`page aging`）算法在 `swap.c` 中实现。“交换”本身并没有在这里实现，相反，它作为内存页管理的一部分，由 `kswapd` 线程实现。

页表管理的最低层由 `memory.c` 文件实现，其中仍然包含有 Linus 在 1991 年十二月实现第一个真正的内存管理功能时的注释。发生在更低层的事情则是体系结构特有代码的一部分（经常以头文件中宏的形式隐藏这些架构相关的代码）。

用于高端内存管理的代码在 `highmem.c` 中实现。高端内存是指内核不能直接访问的高地址内存。尤其在 `x86` 平台上，当配备有超过 4 GB 的 RAM，但又不想放弃 32 位架构时，就会出现高端内存问题。

`vmscan.c` 实现了 `kswapd` 内核线程。它负责查找未使用以及老化的页面，以便释放这些页面，或将其放到交换空间。该文件的源代码包含有丰富注释，因为对其中算法的良好调节，是影响整体系统性能的关键因素。在这一非凡而关键部分中的每一个设计上的选择，都需要有一个充分的理由，因此，就要有大量的注释来说明。

`mm` 目录中的其它源代码处理一些次要的，但有时又很重要的细节，比如 `oom_killer`，这个函数在系统可用内存很少时，选择要杀掉的进程。

值得一提的是，Linux 内核缺少 MMU 处理器上的 `uClinux` 移植，引入了一个 `mmuommu` 目录。它其实是正式 `mm` 目录的一个严格复制，但却去掉了许多 MMU 相关的代码。开发人员选择这种方法，主要是为避免在 `mm` 源代码树中添加过多杂乱的条件编译代码。因为 `uClinux` 还没有集成到主流内核中，因此，如果想要对比这两个目录，则需要下载 `uClinux CVS` 树或者一个 `tar` 归档文件（这两个目录均包含在 `uClinux` 树中）。

16.7 net 目录

Linux 源代码树中的 `net` 目录包含有套接字抽象和网络协议的实现，这些功能特性涉及到大量的代码，这是因为 Linux 支持多种不同的网络协议。每个协议（`IP`、`IPX` 等等）的实现位于自己的子目录中。用于 `IP` 实现的目录称为 `ipv4`，它表示该协议的第 4 版本。新的标准（在本书编写时尚未广泛使用），即 `ipv6`，也在 Linux 中实现了。`Unix` 域套接字被认为是另一个网络协议，它的实现可在 `unix` 子目录中找到。

Linux 中的网络实现基于用于设备文件的相同文件操作。这是很自然的，因为网络连接（即套接字）由普通的文件描述符描述。`socket.c` 文件包含了套接字文件操作。它将系统调用通过 `struct proto_ops` 结构分发到某个网络协议。每个网络协议定义了如何通过这个结构将系统调用映射到其特有的低层数据处理操作。

并不是所有的 `net` 子目录都定义一个协议族，其中有一些值得一提的例外：`core`、`bridge`、`ethernet`、`sunrpc` 以及 `khttpd`。

`core` 中的文件实现了通用的网络功能，比如设备处理、防火墙、多播和别名等，其中包含有套接字缓冲区处理（`core/skbuff.c`），以及独立于低层协议的套接字操作（`core/sock.c`）。与设备特有代码相邻的设备无关数据管理在 `core/dev.c` 中定义。

`ethernet` 和 `bridge` 目录用于实现特定的低层功能，即以太网相关的辅助函数（已在第 14 章中描述），以及网桥功能。

`sunrpc` 和 `khttpd` 比较特殊，因为它们所实现的功能通常是位于用户空间的。

在 `sunrpc` 目录中，我们可以找到用来支持内核级 NFS 服务器（一种基于 RPC 的服务）的函数，而 `khttpd` 实现的则是内核空间的 web 服务器。将这些服务引入内核空间，主要是为避免在时间关键的任务中由系统调用和上下文切换导致的开支。但是，`khttpd` 子系统，已经被 TUX 描述为过时的，而后者在本书编写时，仍然保持着世界最快 web 服务器的记录。TUX 可能会集成到 2.5 内核系列中。

`net` 目录中还有两个源文件，即 `sysctl_net.c` 和 `netsyms.c`。前者是 `sysctl` 机制*的后端，

而后者只是 `EXPORT_SYMBOL` 声明的清单。在整个内核中，有许多类似的文件，通常在每个主要目录中都有一个。

16.8 ipc 和 lib

Linux 源代码树中最小的目录（从尺寸上讲）是 `ipc` 和 `lib`。前一个目录是 System V 进程间通讯原语的实现，分别是信号量、消息队列、共享内存——人们经常遗忘这些机制，但许多应用程序都在使用它们（尤其是共享内存）。后面那个目录包含了一些通用支持函数，和标准 C 函数库中的函数相同。

通用库函数是用户空间库函数的一个很小的子集，但是却包含了编写代码时通常不可缺少的那些函数：字符串函数（包括简单的 `atoi`，以及具有错误检查功能的字符串到长整数转换函数等）以及 `<ctype.h>` 函数。该目录中最重要的文件是 `vsprintf.c`，它实现了 `vsprintf` 函数，而这个函数是 `sprintf` 和 `printf` 的核心。另外一个重要文件是 `inflate.c`，其中包含了 `gzip` 的解压缩代码。

16.9 include 和 arch 目录

我们在快速浏览内核源代码时，较少谈到头文件和体系结构相关代码。本书从头到尾都在介绍头文件，因此，读者应该已经清楚它们的角色（以及 `include/linux` 和 `include/asm` 之间的区分）。

另一方面，体系结构特有的代码，却从来没有详细介绍过，但它们却不太容易讨论。在每个体系结构的目录中，我们通常会发现一个类似顶层的文件树结构（也就是说，其中包含有 `mm` 和 `kernel` 子目录），当然也有引导相关的代码，以及汇编源文件。每个已被支持的体系结构中最重要汇编文件是 `kernel/entry.S`，它是系统调用机制的后端（也就是说，用户进程进入内核模式的地方）。除

* 本书尚未讨论过 `sysctl`，感兴趣的读者可以阅读 Alessandro 有关该机制的论述：
<http://www.linux.it/kerneldocs/sysctl>。

此之外，各个不同体系结构间很少有其它相同点，对它们的描述将很乏味。

16.10 drivers 目录

当前的 Linux 内核支持大量的设备。设备驱动程序占据了源代码树的一半大小（如果不算无用的体系结构相关代码，则实际占三分之二），大概有 1500 个 C 文件和超过 800 个头文件。

`drivers` 目录本身不包含任何源文件，而只包含子目录（当然还有一个 `makefile`）。

将如此大量的源文件结构化地组织起来并非易事，而且开发人员尚未遵循任何严格的规则。最初的划分（即 `drivers/char` 和 `drivers/block`）已经远远不能满足现今的需要，因此，已根据若干不同的需求建立了更多的目录。但是，最一般的字符和块驱动程序仍然存于 `drivers/char` 和 `drivers/block` 目录中，因此，我们首先看这两个目录。

16.10.1 drivers/char

`drivers/char` 目录也许是 `drivers` 树中最为重要的目录，因为其中包含了大量驱动程序无关的代码。

一般性的 `tty` 层（以及线路规则、`tty` 软件驱动程序和类似特性）在这个目录中实现。`console.c` 定义了 linux 终端类型（通过实现其特定的 `escape` 序列和键盘编码）。`vt.c` 定义了虚拟控制台，包含了在虚拟控制台之间切换的代码。选择支持（Linux 文本控制台的剪切/粘贴能力）由 `selection.c` 实现。默认的线路规则则由 `n_tty.c` 实现。

还有其它一些设备无关的文件（读者也许想象不到）。`lp.c` 实现了一个通用并口打印机驱动程序，它包含有“行式打印机上的控制台”功能。通过使用 `parport` 设备驱动程序将操作映射到实际的硬件（见图 2-2），该驱动程序保持了设备无关性。类似地，`keyboard.c` 实现了高级的键盘处理，它导出了 `handle_scancode` 函数，以便平台相关的键盘驱动程序（比如同一目录中的 `pc_keyb.c`）能够从一般化的管理中获得好处。`mem.c` 实现了 `/dev/mem`、`/dev/null` 和 `/dev/zero`，它们是多功能赖以实现的基本资源。

实际上，因为 `mem.c` 从不考虑编译过程（指条件编译），因此被选择作为 `chr_dev_init` 所在的位置，这个函数初始化其它一些设备驱动程序——如果它们被选择编译的话。

`drivers/char` 中还有其它一些设备无关和平台无关的源文件。如果读者对这些源文件所扮演的角色感兴趣，一个最好的起点是该目录中的 `makefile` 文件，该文件中包含了许多有意思的解释信息。

16.10.2 drivers/block

和前述的 `drivers/char` 目录类似，`drivers/block` 在 Linux 开发中已经存在有很长时间了。它用来保存所有的块设备驱动程序，出于这个原因，其中包含有一些设备无关代码。

最重要的文件是 `ll_rw_blk.c`（low-level-read-write block）。该文件实现了第 12 章所描述的所有请求管理函数。

该目录中相对较新的一项是 `blkpg.c`（在 2.3.3 中增加）。该文件实现了块设备分区和几何参数处理的一般性代码。其中的代码，连同先前描述过的 `fs/partitions` 目录一起，替换了“一般性硬盘”支持的早期部分。`genhd.c` 文件仍然存在，但现在只包含块驱动程序的一般初始化函数（类似字符驱动程序中的 `mem.c`）。由 `blkpg.c` 导出的公共函数之一是 `blk_ioctl`，该函数在第 12 章“`ioctl` 方法”中有过描述。

`drivers/block` 中的最后一个设备无关文件是 `elevator.c`。该文件实现了修改某个块驱动程序电梯算法的机制。通过“`ioctl` 方法”中简单介绍过的 `ioctl` 命令可利用这个功能。

`drivers/block` 中除了包含硬件相关的设备驱动程序以外，还包含有一些软件的设备驱动程序，它们从根本上就是跨平台的，就象本书介绍过的 `sball` 和 `spull` 驱动程序一样。它们是 RAM 磁盘 `rd.c`、“网络块设备”`nbd.c`，以及回环块设备 `loop.c`。利用回环设备，可将文件当作块设备挂装。（见 `mount` 命令的手册页，其中描述了 `-o loop` 选项。）利用网络块设备，可将远程资源当作块设备访问（例如，可利用网络块设备实现远程交换设备）。

该目录中的其它文件实现了特定硬件的驱动程序，比如各种不同的软盘驱动器、老式 x86 XT 磁盘控制器等等。大部分重要的块驱动程序已经被移到单独的目录。

16.10.3 drivers/ide

先前保存在 `drivers/block` 目录中的 IDE 设备驱动程序现在已经移到单独的目录。值得一提的是，IDE 已经被增强并扩展，以便能够支持除传统硬盘之外的设备，比如 IDE 磁带。

`drivers/ide` 目录已经形成自己的独立王国，其中包含了一些一般性代码及其自己的编程接口。读者也许会注意到该目录中的某些文件居然有好几千字节长，但却只包含 IDE 控制器的检测代码，并依赖于一般性的 IDE 驱动程序。如果读者对 IDE 驱动程序感兴趣，这些文件值得一读。

16.10.4 drivers/md

该目录用来实现 RAID 功能和逻辑卷管理器（Logical Volume Manager）抽象。该代码注册自己的字符和块设备编号，因此，我们可以将它看成是类似其它传统驱动程序的驱动程序。虽然如此，因为这些代码并不进行任何直接的硬件管理，所以被独立出来。

16.10.5 drivers/cdrom

这个目录包含了一般的 CD-ROM 接口。IDE 和 SCSI cdrom 驱动程序均依赖于 `drivers/cdrom/cdrom.c` 文件中的某些功能。该文件的主要入口是 `register_cdrom` 和 `unregister_cdrom`，调用者为这两个函数传递一个指向 `struct cdrom_device_info` 结构的指针，它将作为 CD-ROM 管理的主要对象。

该目录中的其它文件实现了对特殊硬件驱动器的支持，这些驱动器既不是 IDE 接口，也不是 SCSI 接口。现在这些设备已经很少用到，现代的 IDE 控制器已经让这些设备过时了。

16.10.6 drivers/scsi

与 SCSI 总线相关的所有内容保存在这个目录中。其中包含了对特定设备（比如硬盘和磁带）的控制器无关支持，也包含了对特定 SCSI 控制器板的支持。

对 SCSI 总线接口的管理分散在若干文件中：`scsi.c`、`hosts.c`、`scsi_ioctl.c` 和其它十几个文件。如果读者对整个清单感兴趣，可浏览 `makefile` 文件，其中定义有 `scsi_mod-objs`。这组文件中所有的公共入口点收集在 `scsi_syms.c` 文件中。

支持特定类型硬件驱动器的代码通过调用 `scsi_register_module` 函数插入 SCSI 核心系统，并传递一个 `MODULE_SCSI_DEV` 参数。`sd.c` 用这个方法将磁盘支持增加到核心系统，`sr.c`（该文件内部涉及到 `cdrom_` 函数族）用同样的方法将 CD-ROM 支持增加到核心系统，`st.c` 加入磁带支持，而 `sg.c` 加入一般设备支持。

“一般”设备提供给用户空间程序以直接访问 SCSI 设备的能力。低层的设备可以是任何东西，当前的 CD 刻录程序以及扫描仪程序依赖于 SCSI 的一般设备，以便直接访问它们所驱动的硬件。通过打开 `/dev/sg` 设备，用户空间的驱动程序可在缺少内核特定支持的情况下做任何事情。

主（host）适配器（也就是 SCSI 控制器硬件）可通过调用 `scsi_register_module` 而插入核心系统，并传递一个 `MODULE_SCSI_HA` 参数。大部分驱动程序当前通过使用 `scsi_module.c` 中的设施注册自己：驱动程序的源代码定义它们自己的（静态）数据结构，然后包含 `scsi_module.c` 文件。该文件基于 `<linux/init.h>` 和 `initcall` 定义了标准的初始化和清除函数。这一技术允许驱动程序不需要任何 `#ifdef` 行的帮助，就可以以模块或者已编译函数的形式存在。

有意思的是，`drivers/scsi` 所支持的一种主适配器是 IDE SCSI 仿真代码，即用来映射 IDE 设备的软件主适配器。举例来说，它可用于 CD 控制：系统将所有驱动器看成是 SCSI 设备，这样，用户空间的程序只需考虑 SCSI 设备。

需要注意有些 SCSI 驱动程序是由生产厂商贡献给 Linux 的，而不是出自我们喜欢的黑客社区，因此，并不是所有的驱动程序都读之如饴。

16.10.7 drivers/net

读者也许能想象到，这个目录包含有大多数网络接口适配器的驱动程序。不象 `drivers/scsi`，这个目录不包括实际的通讯协议，它们包含在顶层的 `net` 目录中。尽管如此，`drivers/net` 中仍然有一些软件的抽象实现，就是由串行网络通讯使用的各种线路规则的实现。

线路规则是负责数据在通讯线路上传输的软件层。每个 `tty` 设备都有一个附加的线路规则。每个线路规则由一个编号标识，通常，使用符号名称来指定。默认的 Linux 线路规则是 `N_TTY`，也就是定义在 `drivers/char/n_tty.c` 中的常规 `tty` 管理例程。

但是，在考虑 PPP、SLIP 或其它通讯协议时，默认的线路规则必须被替换。用户空间的程序要将规则替换成 `N_PPP` 或 `N_SLIP`，而在设备最后被关闭时，则会恢复默认规则。在设置通讯链路之后，`pppd` 和 `slattach` 并不退出的原因就在于，当它们退出时，设备将被关闭，而这时就会恢复默认的线路规则。

初始化网络驱动程序的工作尚未转换成 `initcall` 机制，因为一些微妙的技术细节阻止这个转换。因此，初始化仍然以老的方式进行：`space.c` 文件通过扫描并探测一张已知硬件的清单来执行初始化。该清单由 `#ifdef` 预编译指令控制，将选择那些在编译期间实际包含的设备。

16.10.8 drivers/sound

和 `drivers/scsi` 和 `drivers/net` 类似，该目录包含所有的声卡驱动程序。该目录中的内容在某种程度上和 `SCSI` 目录类似：一个文件建立核心声音系统，而独立的设备驱动程序堆在这个核心之上。核心声音系统负责对主设备号 `SOUND_MAJOR` 的请求，并将其分发到低层的设备驱动程序上。硬件驱动程序通过调用 `sound_install_audiodrv` 函数将自己插入核心，该函数在 `dev_table.c` 中声明。

该目录中设备无关的文件清单相当长，因为它们包含有对混音器、音序器等的通用支持，如果有人想进一步研究该目录，建议使用 `makefile` 作为参考。

16.10.9 drivers/video

这里保存有所有的帧缓冲区视频设备驱动程序。该目录关注的是视频输出，而不是视频输入。和 `drivers/sound` 类似，整个目录实现了一个单独的字符设备驱动程序，而核心帧缓冲区系统将实际的访问分发到计算机中可用的各个帧缓冲区上。

`/dev/fb` 设备的入口点在 `fbmem.c` 中。该文件注册主设备号，并维护一个内部清单，其中记录了哪个帧缓冲区设备负责哪个次设备号。硬件驱动程序通过调用 `register_framebuffer` 注册自己，并传递一个 `fb_info` 结构，它是包含所有特定设备管理信息的一个数据结构。包括 `open`、和 `release` 方法，但没有 `read`、`write` 或 `mmap` 方法，这些方法在 `fbmem.c` 中以一般性的途径实现。

除了帧缓冲区内存之外，该目录还负责帧缓冲区控制台。因为帧缓冲区内存中的像素布局在某种程度上是标准化的，内核开发者就可以为不同的显示内存布局实现一般性的控制台支持。一旦硬件驱动程序注册了它自己的 `struct fb_info`，它会自动获得附加其上的一个文本控制台，并且这个控制台是根据显示内存布局而确定的。

不幸的是，在这个领域并没有真正的标准，因此，内核当前支持 17 种不同的屏幕布局，其范围从相当标准的 16 位和 32 位彩色显示模式，到简陋的 VGA 和 Mac 像素显示模式。负责将文本显示在帧缓冲区的文件称为 `fbcon-name.c`。

在第一个帧缓冲区设备注册之后，`register_framebuffer` 调用 `take_over_console`（由 `drivers/char/console.c` 导出），以便真正将当前帧缓冲区设置为系统控制台。在引导阶段，帧缓冲区初始化之前，控制台是原始的文本屏幕，或者如果没有屏幕存在，则是第一个串口。通过启动内核时的命令行，可选择特定的控制台设备而覆盖默认行为。内核开发者建立 `take_over_console` 函数，以便能够为帧缓冲区提供支持，而无需增加引导代码的复杂性。（通常帧缓冲区驱动程序依赖于 PCI 或等价支持，因此，它们不能在引导过程的早期被激活。）但是，`take_over_console` 功能并不限于帧缓冲区，它可由涉及任意硬件的任何代码使用。如果读者想利用摩尔斯式发报机或 UDP 网络数据包传输内核信息，则可以从你的内核模块调用 `take_over_console` 而实现。

16.10.10 drivers/input

输入管理是为了简化和标准化若干驱动程序的共有行为而设计的另外一个设施，它为用户空间提供一个一致的接口。这里的核心文件是 `input.c`，它将自己注册为主设备号为 `INPUT_MAJOR` 的字符驱动程序。它负责从低层设备驱动程序收集事件，并将这些事件发送到高层。

输入接口在 `<linux/input.h>` 中定义。每个低层驱动程序调用 `input_register_device` 注册自己，在注册之后，驱动程序就可以调用 `input_event` 为系统产生事件。

高层模块可调用 `input_register_handler` 注册自己到 `input.c`，但需要指定它们感兴趣的事件类型。这就是 `keybdev.c` 表示它对键盘事件感兴趣的方式（键盘事件最终会传递到 `drivers/char/keyboard.c`）。

高层模块也可以注册自己的次设备号，因此可以使用自己的文件操作并成为 `/dev` 中与输入相关的设备文件的所有者。但是，目前第三方的模块不能方便地注册次设备号，而这一功能也只能由 `drivers/input` 中的文件才能可靠使用。次设备号当前用于支持鼠标、游戏杆，甚至用户空间的一般通道。

16.10.11 drivers/media

该目录在版本 `2.4.0-test7` 中引入，收集了对其它通讯介质的支持，目前有无线和视频输入设备。`media/radio` 和 `media/video` 驱动程序当前均位于 `video/videodev.c` 之上，后者实现了“Video For Linux”的 API。

`video/videodev.c` 是一个一般性的容器。它请求了一个主设备号，并使之对硬件驱动程序可用。各个低层的驱动程序通过调用 `video_register_device` 注册自己，并传递指向自身 `struct video_device` 的指针，以及用来指定设备类型的整数。已支持的设备有帧捕获器（`VFL_TYPE_GRABBER`）、无线电（`VFL_TYPE_RADIO`）、文字电视广播（`VFL_TYPE_VTX`），以及 undecoded vertical-blank information（`VFL_TYPE_VBI`）。

16.10.12 总线相关目录

某些 `drivers` 的子目录专用于插入某个特定总线结构中的设备。将它们从一般的 `char` 和 `block` 目录中分离，是因为相当数量的代码对某个总线结构来讲是通用的（相对硬件设备来讲）。

包含内容最少的目录是 `drivers/pci`。它仅仅包含与 `PCI` 控制器（或系统 `BIOS`）通讯的代码，而 `PCI` 硬件驱动程序则分散在各个地方。`PCI` 接口分布广泛，所以没有必要将这些 `PCI` 卡归入某个特定的地方。

如果读者想知道 `ISA` 是否有特定的目录的话，其答案是“否”。没有任何针对 `ISA` 的支持文件，因为这个总线没有提供任何的资源管理或者标准，以便在其上建立一个软件层。`ISA` 硬件驱动程序保存在 `drivers/char`、`drivers/sound` 或其它地方。

其它总线相关目录包括从很少听说的计算机内部总线，到大量使用的外部接口标准。

前一种类型包括 `drivers/sbus`、`drivers/nubus`、`drivers/zorro`（Amiga 计算机使用的总线）、`drivers/dio`（HP300 类计算机使用的总线），以及 `drivers/tc`（在 MIPS DECstation 上使用的 Turbo Channel 总线）。`sbus` 中包含了 SBus 支持函数和某些 SBus 设备的驱动程序，而其它目录中只包含支持函数。基于这些总线的所有硬件驱动程序分布在 `drivers/net`、`drivers/scsi` 等适合实际硬件的地方（刚刚说过，某些 SBus 驱动程序是个例外）。其中几个总线目前只有一个驱动程序使用。

用于外部总线的目录有 `drivers/usb`、`drivers/pcmcia`、`drivers/parport`（通用跨平台并口支持，它定义了一组全新的驱动程序）、`drivers/isdn`（所有 Linux 支持的 ISDN 控制器以及它们的公共支持函数）、`drivers/atm`（用于 ATM 网络连接），以及 `drivers/ieee1394`（防火墙）。

16.10.13 平台相关目录

有时，某个计算机平台在 `drivers` 树中有其自己的目录。这通常发生在针对该平台的内核开发正在进展，而尚未合并到主源代码树的情况下。这时，单独列出这些目录有助于维护代码。举例来说，这种目录有 `drivers/acorn`（老的 ARM 计算机）、`drivers/macintosh`、`drivers/sgi`（Silicon Graphics 工作站），以及 `drivers/s390`（IBM 大型机）。阅读这些代码通常没有什么价值，除非你对某种特定的平台感兴趣。

16.10.14 其它子目录

`drivers` 中还有其它一些子目录，但从笔者的角度看，它们非常次要，也仅仅用于一些特殊情况。`drivers/mtd` 实现了 Memory Technology Device 层，它用来管理固态的磁盘（FLASH 存储器或者类似的 EEPROM）。`drivers/i2c` 提供了 i2c 协议的实现，i2c 是一些高级外设内部使用的“Inter Integrated Circuit” two-wire 总线，其中主要是帧捕获器。类似地，`drivers/i20` 处理 I20 设备（一种用于特定 PCI 设备的专有高速通讯标准，目前已在自由软件社区的压力下公开其技术）。`drivers/pnp` 则是 ISA 即插即用设备驱动程序的公共代码，但不幸的是，制造厂商现在并没有真正使用 PnP 技术。

在 `drivers/` 目录之下，我们还可以找到对新设备类型的初始支持，这些设备类型目前只由少量的设备实现。

其中包括光纤通道支持（`drivers/fc4`）和 `drivers/telephony`。还有一个几乎为空的目录 `drivers/misc`，它用于“for misc devices that really don't fit anywhere else，不适合放在其它地方的杂类设备”。该目录中没有代码，但包含有一个（空的）`makefile`，其中仅仅含有上述注释。

Linux 内核太大了，我们无法在短短几页中涵盖所有内容。此外，这还是一个移动目标，当发现自己已经掌握了某个版本的内核代码结构时，接踵而来的却是黑客们没完没了的新补丁，以及大量的资料。也许在读者阅读本书时，2.4 内核中的 `misc` 目录就不再是空的。

尽管我们认为这不太可能，但 2.6 或 3.0 将可能与 2.4 非常不同。不幸的是，本书的这个版本不能自动更新自己以涵盖新的版本，因此，随时间的迁移，这个版本将最终过时。尽管我们已经尽最大努力覆盖内核的当前版本——不管是本章还是整书内容，但仍然没有比直接参考源代码更好的办法。

附录 A 参考书目

本书大部分内容取材于内核源代码，而内核源代码则是 Linux 内核的最佳文档。

内核源代码可从遍布全球的几百个 FTP 站点下载，因此我们不打算在这里列出这些站点。

阅读补丁文件是找出版本依赖关系的最好方法。这些补丁可从获得 Linux 源代码的同一地点下载。repatch 程序可帮助读者了解单个文件在不同内核补丁之间的修改情况，该程序的源代码也可从 O'Reilly FTP 站点上下载。

在 sunsite.unc.edu 及其所有的镜像站点上，读者会发现许多设备驱动程序，这些驱动程序可以帮助我们编写自己的驱动程序。

A.1 Linux 内核书籍

Bar, Moshe. *Linux Internals*. McGraw-Hill. 2000.

本书由《Byte》杂志专栏作家 Moshe Bar 编写，讲述了 Linux 内核的工作原理，并包含大量的 2.4 功能特性。

Bovet, Daniel P., and Cesati, Marco. *Understanding the Linux Kernel*. O'Reilly & Associates. 2000.

本书详细讲述了 Linux 内核的设计和实现。本书更专注于讲解内核算法，而不是内核 API 的介绍。

Maxwell, Scott. *Linux Core Kernel Commentary*. Coriolis. 1999.

《Linux Core Kernel Commentary》的大部分是内核核心代码清单，最后是 150 页的注解。读者可以从本书了解特定内核部分正在发生的事件。

Nutt, Gary J. *Kernel Projects for Linux*. Addison-Wesley. 2000.

《Kernel Projects for Linux》是用于大学教学的书籍，因此，并没有完整介绍 Linux 内核。但本书对那些欲从事内核 hack 工作的人来讲，是一本不错的辅助读物。

A.2 Unix 设计和内幕

Bach, Maurice. *The Design of the Unix Operating System*. Prentice Hall. 1987.

本书有点老，但却涵盖了 Unix 实现中的所有问题。本书是 Linus 在 Linux 第一个版本中的许多思路和灵感的来源。

Stevens, Richard. *Advanced Programming in the UNIX Environment*. Addison-Wesley. 1992.

本书详细讲述了 Unix 系统调用。当我们需要在设备方法中实现一些高级特性时，本书可作为重要的参考手册。

Stevens, Richard. *Unix Network Programming*. P T R Prentice-Hall. 1990.

本书也许是 Unix 网络编程 API 方面的最权威著作。

附录 B 封面故事

我们的外表源于读者的评价、我们自己的实验以及来自发行渠道的反馈。与众不同的封面是对我们独特的技术论点的有力补充，并且将个性和生活带入了看起来有点干涩的技术论述。

《Linux 设备驱动程序》的封面图片是一匹脱缰的烈马。这副图的彩色描写可见 William Thayer 所著的《Marvels of the New West: A Vivid Portrayal of the Stupendous Marvels in the Vast Wonderland West of the Missouri River》(The Henry Bill Publishing Co. Norwich, CT, 1888)。Thayer 援引一名仓库管理员的话，详细描述了一匹脱缰的烈马：“一匹马受惊时，他会将自己的头放在前腿中间，弓起他的后背，就象一只发疯的猫，然后，他跳向空中，并发出一声可怕的嘶声。有时，他会不停地跳跃、不停地重复，直到完全疲惫为止。如果骑手还在马鞍上，他也会一样疲惫。当然，骑在受惊的马身上而不被甩下来是很困难的，尤其那马是一匹真正有技术的烈马时，他每跳一下，都会往旁边倾斜一次。这种墨西哥式的马鞍上有两条肚带，这样就能牢牢拴住马鞍，但是两条肚带经常会惊吓着马，而只有一条肚带却不会。有些时候，不把马惹急了，就没办法系好侧肚带。”

Darren Kelly 是本书的产品编辑，Cynthia Kogut 是技术编辑，Susan Carlson Greene 是本书第二版的校对者。Catherine Morris 和 Claire Cloutier 负责质量控制。Judy Hoer 编写了索引。Matt Hutchinson、Lucy Muellner 和 Joe Wizda 负责产品支持。

Edie Freedman 设计了本书封面。封面上的图片来自 19 世纪的雕版画。Emma Colby 使用 QuarkXPress 4.1 设计了封面的版面，并使用了 Adobe 的 ITG Garamond 字体。

David Futato 根据 Nancy Priest 的系列设计完成了内部版式设计。每章打头的图片取自 Dover Pictorial Archive 收藏的《Marvels of the New West: A Vivid Portrayal of the Stupendous Marvels in the Vast Wonderland West of the Missouri River》一书 (The Henry Bill Publishing Co. Norwich, CT, 1888, William Thayer 著)，以及《The Pioneer History of America: Apopular Account of the Heroes and Adventruers》(The Jones Brothers Publishing Company, Cincinnati, OH, 1884, Augustus Lynch Mason 著)。本书的印刷版本是从 DocBook XML 标记的源文件中转换成一组 gtroff 宏而形成的，期间使用了 O'Reilly & Associates 的 Norman Walsh 开发的一个过滤器。Steve Talbott 在 GNU troff-gs 宏的基础上设计并编写了底层的宏集合；Lenny Muellner 将其改写成适合 XML 并实现了针对本书的设计。我们使用了 GNU groff text formatter 版本 1.11.1 生成 PostScript 输出。文本和标题字体是 ITC Garamond Light 和 Garamond Book 字体。书中出现的图例由 Robert Romano 和 Jessamyn Read 使用 Macromedia FreeHand 9 和 Adobe Photoshop 6 绘制。

我们尽可能使用耐用和柔韧的平折装订来装订书籍。如果超过了这种装订的限度，则使用非粘性的装订。