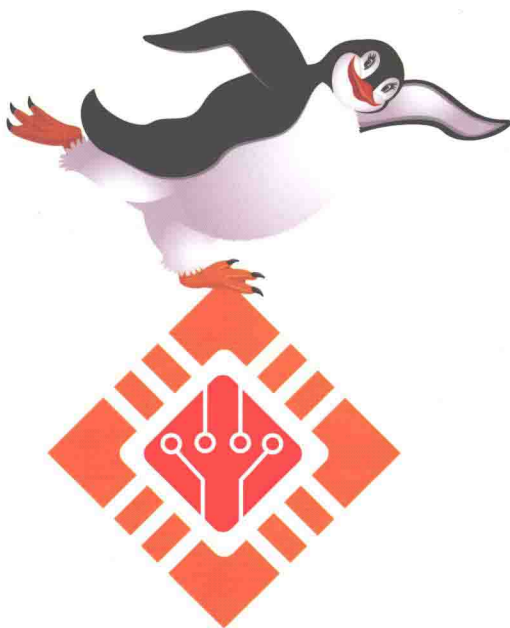


清华

开发者书库



Understanding
Linux Device Driver Programming

深入理解 Linux驱动程序设计

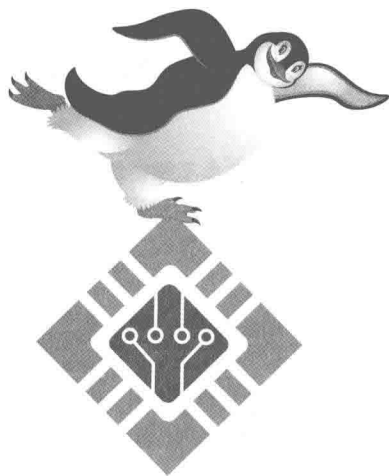
吴国伟 姚琳 毕成龙◎编著

Wu Guowei Yao lin Bi Chenglong

清华大学出版社

清华

开发者书库



Understanding
Linux Device Driver Programming

深入理解 Linux驱动程序设计

吴国伟 姚琳 毕成龙◎编著

Wu Guowei Yao lin Bi Chenglong

清华大学出版社
北京

内 容 简 介

本书基于 Linux 内核 3.8.13 源代码及相关实例向读者系统而详尽地介绍和分析了 Linux 设备驱动程序开发框架、原理和方法。全书共分 13 章,内容包括字符设备、块设备、网络设备、MMC/SD 驱动、USB 驱动、总线驱动及 Flash 驱动的开发机制和实例。

本书各章均首先概要介绍各模块的实现原理,随后列举各模块中的关键数据结构,再结合源代码及实例分析介绍,让读者更全面地了解 Linux 驱动开发。

本书内容丰富,概念和原理讲解细致、深入浅出。其中,有关代码的部分都标有注释以详细介绍功能,书中的设计和分析也配以编程实例帮助理解。

本书适合作为高年级本科生、研究生和从事嵌入式系统开发设计的工程技术人员。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

深入理解 Linux 驱动程序设计/吴国伟,姚琳,毕成龙编著.--北京:清华大学出版社,2015

清华开发者书库

ISBN 978-7-302-40163-6

I. ①深… II. ①吴… ②姚… ③毕… III. ①Linux 操作系统—程序设计 IV. ①TP316.89

中国版本图书馆 CIP 数据核字(2015)第 096830 号

责任编辑:盛东亮

封面设计:李召霞

责任校对:时翠兰

责任印制:沈 露

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座

邮 编:100084

社 总 机:010-62770175

邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

课件下载: <http://www.tup.com.cn>, 010-62795954

印 刷 者:北京富博印刷有限公司

装 订 者:北京市密云县京文制本装订厂

经 销:全国新华书店

开 本:186mm×240mm 印 张:12.75

字 数:285 千字

版 次:2015 年 11 月第 1 版

印 次:2015 年 11 月第 1 次印刷

印 数:1~2500

定 价:29.00 元

产品编号:053863-01

前言

PREFACE

Linux 从 1991 年发布第一个版本到现在的 3.19.3 版,经过无数开发者的共同努力,内核日趋完善。Linux 作为一种开源、跨平台的操作系统,受到了越来越多开发者的青睐。

随着物联网和人工智能的发展,Linux 将更多地应用在嵌入式设备中,这对 Linux 内核中的驱动设计和实现也提出了更高的要求。现有的介绍 Linux 设备驱动开发的图书中,有的偏重于内核各模块的结构和原理的阐述,难以理解和掌握;有的侧重 Linux 内核的部分特征及应用,缺少对 Linux 架构整体的介绍及系统原理的分析。基于这样的现状,编写此书供广大 Linux 爱好者参考。

本书结合 Linux 内核中各模块的原理及设备驱动实例,详细地介绍了 Linux 设备驱动开发的方法与实践。全书共分为 13 章,首先介绍了 Linux 操作系统的发展,然后针对 Linux 内核 3.8.13 全面介绍了 Linux 设备驱动开发,分析了各模块的 Linux 实现并给出了驱动开发实例。在介绍了 Linux 内核机制的基础上,着重论述块设备、网络设备、MMC/SD 驱动、USB 驱动、总线驱动及 Flash 驱动的开发。

全书各章均首先概要介绍各模块的实现原理,随后列举各模块中的关键数据结构,再结合源代码及实例分析介绍,让读者更全面地了解 Linux 驱动开发。

本书编写过程中参考了众多 Linux 开发者的研究成果和相关书籍,参考文献中无法一一列出,在此向他们致以谢意。书中实际案例,是诸多课程的研究生们在 Linux 3.8.13 版本下调试通过,在此一并表示感谢。本书的出版也离不开清华大学出版社的支持,对此表示衷心的感谢!

由于时间仓促和作者水平有限,书中难免出现遗漏与不当之处,敬请广大读者批评指正。如有任何问题,请发邮件至 wgwdut@dlut.edu.cn。

编者

2015 年 4 月

目 录

CONTENTS

第 1 章 Linux 内核组成和机制	1
1.1 Linux 内核版本与发展	1
1.1.1 Linux 操作系统的诞生	1
1.1.2 Linux 内核版本的变迁	2
1.2 Linux 内核编译	3
1.2.1 获取内核源码	3
1.2.2 内核源码树	3
1.2.3 编译内核	4
1.3 Linux 内核组成	6
1.4 Linux 内核机制	7
1.4.1 内核启动过程	7
1.4.2 模块机制	8
第 2 章 Linux 内核设备管理方式	10
2.1 devfs 设备文件系统	10
2.2 sysfs 文件系统	10
2.3 udev 设备文件系统	12
2.4 主要数据结构	12
2.4.1 kobject	12
2.4.2 ktype	14
2.4.3 kset	14
2.4.4 三者关系	15
2.5 热插拔设备管理机制	16
2.5.1 热插拔事件流程	16
2.5.2 涉及的模块	18
2.5.3 关键驱动函数	19

第 3 章 Linux 驱动开发基础	20
3.1 同步机制	20
3.1.1 内核同步机制分类	21
3.1.2 自旋锁与信号量的比较	33
3.2 make 及 makefile	33
3.2.1 makefile 文件	34
3.2.2 编写 makefile 文件	34
3.2.3 make 命令	38
3.3 调试方法	41
3.3.1 printk	42
3.3.2 /proc 文件系统	43
3.3.3 调试器及相关工具	44
第 4 章 Linux 字符设备驱动开发	46
4.1 关键数据结构	48
4.2 接口函数部分内核代码分析	51
4.3 字符设备驱动设计	57
4.3.1 字符设备驱动设计场景描述	57
4.3.2 字符设备驱动设计过程	57
第 5 章 Linux 内核中断机制	61
5.1 中断	61
5.2 中断处理	62
5.2.1 注册中断处理程序	62
5.2.2 编写中断处理程序	63
5.3 中断上半部与下半部的对比	63
5.4 中断下半部	64
5.5 BH 机制与任务队列机制	64
5.6 软中断	65
5.6.1 软中断的实现	65
5.6.2 软中断的使用	66
5.7 tasklet	66
5.7.1 tasklet 的实现	66
5.7.2 tasklet 的使用	68
5.8 工作队列	68

5.8.1 工作队列的实现	68
5.8.2 工作队列的使用	69
第 6 章 Linux 块设备驱动开发	71
6.1 块设备管理机制	71
6.1.1 块设备基本概念	71
6.1.2 块设备在 Linux 中的结构	72
6.2 块设备关键数据结构	73
6.2.1 gendisk 数据结构	73
6.2.2 block_device_operations 数据结构	74
6.2.3 request 数据结构	75
6.2.4 request_queue 数据结构	77
6.2.5 bio 数据结构	78
6.3 块设备驱动设计函数	81
6.3.1 块设备驱动注册与注销函数	81
6.3.2 块设备驱动打开与关闭函数	81
6.3.3 块设备驱动 ioctl、read 和 write 函数	81
6.3.4 块设备驱动的请求函数	82
6.4 Ramdisk 块设备驱动实例	83
6.4.1 Ramdisk 块设备驱动实例分析	83
6.4.2 Ramdisk 块设备驱动实例测试	84
第 7 章 Linux 网络设备驱动开发	86
7.1 网络设备	86
7.1.1 网络系统分层结构	86
7.1.2 网络设备管理	87
7.2 NAPI 机制	89
7.3 关键数据结构	90
7.4 内核提供的网络设备驱动设计函数	92
7.4.1 alloc_netdev	92
7.4.2 register_netdev	92
7.4.3 ether_setup	92
7.4.4 unregister_netdev	93
7.5 网络设备驱动开发实例	93
7.5.1 snull_init_module 函数	94
7.5.2 snull_init 函数	94

7.5.3 相关操作函数	95
第8章 Linux MMC/SD 驱动开发	100
8.1 MMC 子系统基本架构	100
8.2 关键数据结构	101
8.2.1 基本数据结构	101
8.2.2 基本数据结构主要成员及关系	101
8.3 MMC/CD 卡驱动实例	105
8.3.1 MMC/SD 卡设备驱动设计场景	105
8.3.2 MMC/SD 卡设备驱动实例实现	107
第9章 Linux USB 驱动开发	115
9.1 USB 设备管理机制	115
9.1.1 USB 与串口	115
9.1.2 USB 设备属性拓扑结构管理机制	115
9.1.3 USB 设备逻辑组织管理机制	116
9.2 USB 驱动关键数据结构分析	117
9.3 USB 设备驱动函数及其使用说明	119
9.3.1 客户端驱动管理	119
9.3.2 USB 设备配置和管理	120
9.3.3 主机控制器的管理	121
9.3.4 协议控制命令集和数据传输管理	122
9.4 USB 设备驱动开发实例	123
9.4.1 实例开发场景设计	123
9.4.2 USB 设备驱动开发实例的实现	123
9.4.3 驱动测试分析	128
第10章 Linux I2C 总线设备驱动	130
10.1 Linux 总线驱动及 I2C 总线	130
10.1.1 Linux 总线驱动设计过程	130
10.1.2 I2C 总线的工作原理与应用	131
10.1.3 总线基本操作	132
10.2 Linux I2C 体系结构	133
10.2.1 Linux 的 I2C 体系结构组成	133
10.2.2 Linux I2C 关键数据结构	134
10.3 Linux I2C 核心	139

10.4	Linux I2C 总线驱动	141
10.4.1	I2C 适配器驱动加载与卸载	141
10.4.2	I2C 总线通信方法	141
10.5	Linux I2C 设备驱动	143
10.5.1	Linux I2C 设备驱动模块加载与卸载	144
10.5.2	Linux I2C 设备驱动的数据传输	144
10.5.3	Linux i2c-dev.c 文件分析	145
10.6	Linux I2C 驱动实例——EEPROM	146
10.6.1	初始化	146
10.6.2	探测设备	147
10.6.3	检查适配器的功能	148
10.6.4	访问设备	148
10.6.5	其他函数	149
第 11 章	Linux PCI 总线设备驱动	151
11.1	PCI 总线设备	151
11.1.1	PCI 总线	151
11.1.2	PCI 设备	152
11.2	PCI 设备驱动结构	155
11.3	PCI 设备驱动实例	156
11.3.1	PCI 设备驱动程序基本框架	156
11.3.2	初始化设备模块	158
11.3.3	打开设备模块	159
11.3.4	数据读写和控制信息模块	160
11.3.5	中断处理模块	160
11.3.6	释放设备模块	161
11.3.7	卸载设备模块	162
第 12 章	Linux 输入设备驱动	163
12.1	Linux 输入子系统结构	163
12.2	输入设备驱动核心数据结构分析	164
12.3	Linux 输入设备驱动实例	168
12.3.1	输入设备驱动流程	168
12.3.2	USB 鼠标驱动编写实例	169

第 13 章 Linux Flash 驱动开发	174
13.1 Flash 存储器	174
13.2 Linux MTD 系统层次结构	174
13.3 关键数据结构	175
13.3.1 mtd_info 结构体	176
13.3.2 mtd_table 结构体	178
13.3.3 mtd_part 结构体	179
13.3.4 mtd_partition 结构体	179
13.3.5 map_info 结构体	179
13.4 驱动相关函数	180
13.4.1 add_mtd_device 函数	180
13.4.2 del_mtd_device 函数	180
13.4.3 add_mtd_partitions 函数	180
13.4.4 del_mtd_partitions 函数	181
13.4.5 do_map_probe 函数	182
13.5 Nor 型 Flash 驱动实例	183
13.5.1 Nor 型 Flash 驱动设计流程	183
13.5.2 Nor 型 Flash 驱动详细设计	184
13.6 Nand 型 Flash 驱动实例	188
13.6.1 Nand 型 Flash 设备驱动设计步骤	188
13.6.2 Nand 型 Flash 驱动实现	189
参考文献	192

1.1 Linux 内核版本与发展

1.1.1 Linux 操作系统的诞生

在 Linux 操作系统诞生之前,微软公司的 MS-DOS 主宰着操作系统的市场,其价格十分昂贵,而 UNIX 操作系统的经销商为了获取高额利润,也将价格抬高,在一段时间内市场上没有廉价的操作系统,二者都严格保护着自己的源代码,使得计算机爱好者无法对操作系统进行探究。此时,MINIX 操作系统出现,并且有一本详细介绍其实现原理的书,Linux 的创造者 Linus Torvalds 正是通过 MINIX 系统了解到操作系统的原理,但是 Linus 不满足 MINIX 系统的性能,并酝酿开发一款新的免费操作系统。自 1991 年 4 月至同年的 9 月, Linus 开发并发布了 Linux 0.01。随后世界各地的爱好者纷纷下载测试,并将反馈和改进的代码发回,Linus 则根据反馈进一步修改系统。很快,10 月 5 日 0.02 版就出现了,0.03 版也在几周内出现,12 月发布了 0.10 版。这时的 Linux 仅仅支持 AT 硬盘,无法登录(直接启动到 bash)。Linux 0.11 带来多语言键盘、软驱、VGA 等一系列更新,接下来版本号从 0.12 直接跳到了 0.95、0.96。代码通过芬兰的 FTP 站点传播到世界各地,世界各地的开发者下载使用并建立 FTP 镜像,至此,Linux 操作系统已经具备了雏形。

Linux 操作系统的开发还在继续,阵营很快从百人扩大到千人,继而是几十万人,无数开发者通过调制解调器联系在一起,在世界各地贡献代码和补丁,就这样看似一盘散沙的分布式开发模式写出了稳定的内核,Linux 操作系统颠覆了微软推崇的代码专有时代。在此之后,Linus 使用 GNU 通用公共许可证(GPL)将 Linux 重新授权以保证爱好者可以完全自由地复制、学习和修改源代码。由于 Linux 内核刚好填补了 GNU 缺少开源内核的空缺,二者共同构成了一个完整的 GNU 系统。经过 1992 年和 1993 年的改进,Linux 系统先后支持包括 TCP/IP 网络及图形窗口系统在内的许多重要功能。1994 年 3 月 14 日,Linux 1.0 版本正式发布,标志着 Linux 操作系统正式诞生。

1.1.2 Linux 内核版本的变迁

Linux 内核版本一般划分为 3 个阶段,第 1 个阶段是 0.01~1.0,第 2 个阶段是 1.0.x~1.2.x,第 3 个阶段是 1.2.x 之后的版本。

1991 年 2 月,Linus 初步编写的 Linux 功能就是两个进程分别显示 AAA 和 BBB,同年 9 月,Linus 发布的 Linux 0.01 版本的代码量约 10 000 行;到 1993 年参与编写修改 Linux 内核代码的程序员扩大到 100 人左右,核心组由 5 人组成;在 1994 年 3 月发布的 Linux 1.0,代码量达到了 17 万行,并且是按照完全免费的协议发布,随后正式使用 GPL 协议。此后 Linux 的开发进入了一个良性循环,核心小组收集程序员提供的改进的源代码,逐步完善 Linux 的功能。到 1995 年 Linux 操作系统已经可以运行在 Intel、Sun SPARC 等处理器上,用户量也超过了 50 万;1996 年 6 月,Linux 2.0 内核发布,其代码量达到 40 万行,并可以支持多个处理器,Linux 正式进入实用阶段,全球用户约 350 万。

2011 年 7 月 21 日,Linus 宣布 Linux 内核 3.0 正式发布,虽然在版本号数字上有了划时代的改变,但实际上 Linux 3.0 更像是 Linux 2.6.40 版本的新名称,对于内核开发并没有表现出里程碑似的特征。在 Linux 2.6.x 的基础上,Linux 3.0 正式版仍然做出了一些重要的特性改变,在此列出部分特性:

- (1) btrfs 文件系统自动碎片整理、性能改进和检查;
- (2) 支持 sendmmsg()函数调用,UDP 发送性能提升 20%,接口发送性能提升 30%;
- (3) 支持应用缓存清理(CleanCache);
- (4) 支持柏克莱封包过滤器(Berkeley Packet Filter)实时过滤,配合 libpcap/tcpdump 提升包过滤规则的运行效率;
- (5) 支持无线广域网(WLAN)唤醒;
- (6) 支持非特殊授权的 ICMP_ECHO 函数;
- (7) 支持高精度计时器 Alarm-timers;
- (8) 支持 setns() syscall,更好地命名空间管理;
- (9) 支持微软 Kinect 体感设备;
- (10) 支持 AMD Llano APU 处理器;
- (11) 支持 Intel iwlwifi 105/135 无线网卡;
- (12) 支持 Intel C600 SAS 控制器;
- (13) 支持雷凌 Ralink RT5370 无线网卡;
- (14) 支持多种 Realtek RTL81xx 系列网卡;
- (15) 大量新驱动;
- (16) 大量 bug 修正和改进。

1.2 Linux 内核编译

1.2.1 获取内核源码

获取内核源码的方式有许多,可以直接从 Linux 内核官方网站(www.kernel.org)下载完全源码的压缩包,或者获取增量补丁形式的压缩包;利用 Git 获取源码,这也是当前比较流行的方式,Git 是 Linus 及他领导的内核开发者共同开发的一个控制和管理 Linux 内核源码的系统。当前有许多在线资源提供了可靠有效的 Git 介绍,读者可根据自己的兴趣自行学习。在这里我们介绍通过登录 Linux 内核官方网站直接获取全部 Linux 内核源码的方式。

在官方网站中,可以看到两种类型的压缩形式 gzip 和 bzip2。gzip 形式压缩的文件后缀为 tar.gz,以 Linux 内核 3.8.13 为例,如果要解压这种类型的文件需要执行:

```
$tar -xvzf Linux-3.8.13.tar.gz
```

bzip2 形式压缩文件后缀为 tar.bz2,这种类型的文件解压执行命令的参数与前者存在不同,具体操作为:

```
$tar -xvjf Linux-3.8.13.tar.bz2
```

补丁作为一种在 Linux 内核开发和管理中常用的形式,为 Linux 内核开发者提供了便利,可以通过补丁的形式发布对代码的修改,也可以下载补丁接收其他人的修改,补丁相当于代码版本之间的纽带。如果想更新内核代码,不再需要下载全部压缩代码,只需要在旧版本上打上一个增量补丁即可。

增量补丁的获取方式与源码的获取方式相同,在获得补丁之后,需要从内核源码树开始应用增量,具体使用方式如下:

```
$patch -p1 < ../patch-3.8.13
```

1.2.2 内核源码树

Linux 作为一种单内核模式的系统,运行其中的所有程序都有着紧密的联系,同时它们之间的调用关系十分密切。因此在完成 Linux 内核分析前,需要熟悉源代码的目录结构。Linux 内核源代码由许多目录组成,而且大多数目录包含许多子目录,其具体的结构如表 1.1 所示。

表 1.1 Linux 内核源码目录结构

目 录	描 述
arch	存放特定体系结构的源代码,内核中与具体 CPU 和体系结构相关的代码都放在此目录的子目录下,有关的.h 文件在 include/asm 下
crypto	Crypto API,存放比较流行的分组加密和哈希函数的源代码
Documentation	有关 Linux 内核的文档
drivers	驱动程序,包括各种块设备和字符设备的驱动
fs	VFS 和各种文件系统,每个子目录分别对应一个特定的文件系统,还有一些用于虚拟文件系统(VFS)的公用源程序
include	内核头文件,包含所有的.h 文件,其每个子目录都对应着一种体系结构,还有一些公用的子目录如 sound、net、Linux 等
init	Linux 内核的引导和初始化代码,如 do_mounts.c、initramfs.c、main.c、version.c 等文件
ipc	Linux 进程间通信代码,如 mqueue.c、sem.c、shm.c 等文件
kernel	Linux 中进程管理和调度等核心子系统,如 fork.c、sched.c、time.c、wait.c 等文件
lib	通用的 Linux 内核函数库,如对错误信息的处理
mm	内存管理及虚拟内存管理的代码,如 memory.c、slab.c、shmem.c 等文件
net	网络相关代码,如网络协议、网卡驱动程序,其子目录对应的网络协议如 IEEE802、IPv4 等
scripts	内核编译时需要的脚本,系统配置的一些命令文件
security	Linux 安全模块
sound	语音子系统

Linux 内核源代码虽然很庞大,但是对于每一个具体的内核而言并不是所有的.c 和.h 文件都会被使用,在编译时会根据系统的配置有选择地使用。

1.2.3 编译内核

内核编译并没有想象中的那么难,每个版本的内核都会为用户提供编译工具,使编译内核变的容易。要完成内核的编译,首先在编译前对内核进行相关的配置。

由于 Linux 内核包含的功能非常丰富,例如对不同平台的支持等,每个用户可以根据自己的需要,完成内核编译的个性化配置。这些可以配置的选项以 CONFIG_FEATURE 形式表示,CONFIG 是所有选项的前缀,例如对称多处理器(SMP)的配置选项为 CONFIG_SMP,如果设置了该选项,SMP 启用,否则,SMP 不被启动。CONFIG_SMP 配置项只有两个选项,要么 yes(启用)要么 no(不启用),而有一些配置项存在 3 个选项,除 yes 和 no 外,还有 module 选项。module 意味着该配置项被选定,但编译时此功能的实现代码以模块的形式生成。在 3 选 1 的情况下,yes 选项把代码编译进主内核镜像中,而不是作为一个模块。通常,驱动程序都是用 3 选 1(Y/M/N)的配置项。

Linux 提供了不同的工具来简化内核的配置,下面介绍三种配置命令,注意在执行命令前务必切换到 root 用户,并在内核源码目录中执行。

(1) 基于文本命令行方式:

```
$make config
```

(2) 基于 ncurses 库编制的图形界面方式:

```
$make menuconfig
```

(3) 基于 gtk+ 的图形界面方式:

```
$make gconfig
```

以上三种方式都可以完成内核的配置,并且生成用于存储配置信息的 .config 文件,该文件存放在内核源码的根目录下,可以直接进行修改,并且可以很容易查找和修改选项。

还可以使用 make defconfig 命令对内核进行配置,该命令会根据体系结构创建一个配置,这个命令非常适合初次接触内核编译的人使用,因为它可以保证硬件所配置的选项是启用的。

在完成修改内核配置,或者用已有的文件配置新的内核源码时,还需要验证和更新配置,具体命令如下:

```
$make oldconfig
```

实际上,配置选项 CONFIG_IKCONFIG_PROC 把完整的压缩过的内核配置文件存放在 /proc/config.gz 下,也就是说,如果想编译一个新的内核源码,而且不修改当前的内核配置,可以直接将该压缩文件的内容复制到新的内核源码文件下,操作命令为:

```
$zcat /proc/config.gz > .config
$make oldconfig
```

至此,内核编译前的配置基本完成。此外,编译前还要准备一些编译所需的工具,包括 gcc、make 等以及相关的库。

```
# apt-get install build-essential libncurses-dev kernel-package fakeroot initramfs-tools
module-init-tools
```

下载了 Linux-3.8.13 后,将其解压并存放在 /usr/src 目录下,开始进行编译。

```
#make mrproper(保证旧文件不再使用)
#make menuconfig
#make(编译)
#make install(安装内核)
#make modules_install(安装模块)
```

由于 make 的时间较长,内核提供了一个并行的编译方式: make -jn,这里 n 是并行的作业数量。在实际操作中,通常一个处理器衍生出一个或者两个作业,并行的方式大大提升了编译的效率。

Linux 内核编译部分基本完成,可以尝试完成内核的修改和编译了。

1.3 Linux 内核组成

Linux 内核主要有 5 部分组成：进程调度 (SCHED)、进程间通信 (IPC)、内存管理 (MM)、虚拟文件系统 (VFS) 及网络接口 (NET)。本节简要介绍各部分的主要功能及联系。从图 1.1 中可以发现，这 5 部分彼此之间是相互依赖缺一不可的。

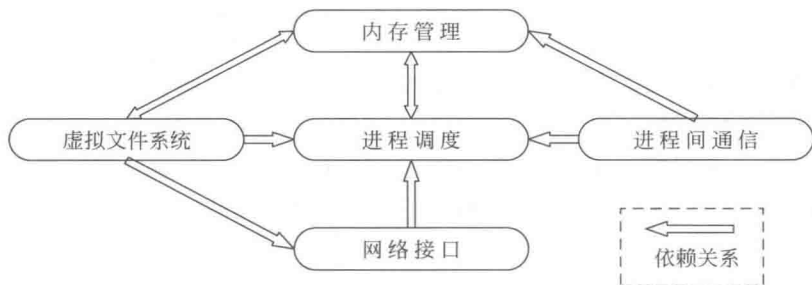


图 1.1 内核组成结构依赖关系

1. 进程调度

进程调度控制系统中的众多进程对 CPU 的访问，并且处于系统的中心位置。可以清晰地从图 1.1 中发现，因为系统内的每个子系统都需要挂起或恢复进程，进程调度为其他子系统提供了相关的服务。Linux 内核使用的是较为简单的基于优先级的进程调度算法来完成进程的选择。

2. 内存管理

内存管理实现的功能是控制多个进程安全地共享主内存区域。Linux 内存管理支持虚拟内存，即在计算机中运行的程序，其代码、数据、堆栈的总量可能超过实际内存大小，操作系统只把当前使用的程序块保留在内存中，其余程序块保留在磁盘中。当 CPU 提供内存管理单元时，Linux 内存管理完成为每个进程进行虚拟内存到物理内存的转换。内存管理从逻辑上分为硬件无关部分和硬件相关部分，硬件无关部分提供了进程的映射和逻辑内存的兑换；硬件相关部分为内存管理硬件提供虚拟接口。

3. 虚拟文件系统

虚拟文件系统 (VFS) 隐藏了各种硬件的具体细节，为所有的设备提供了统一的接口，VFS 提供了数十种不同的文件系统。虚拟文件系统独立于各个具体的文件系统，是对各个文件系统的一个抽象，它使用超级块 super block 存放文件系统的相关信息，使用索引节点 inode 存放文件的物理信息，使用目录项 dentry 存放文件的逻辑信息。图 1.2 形象地描述了 Linux 的文件系统。

4. 网络接口

网络接口提供了对各种网络标准的存取和网络硬件的支持。网络接口可以分为两类，

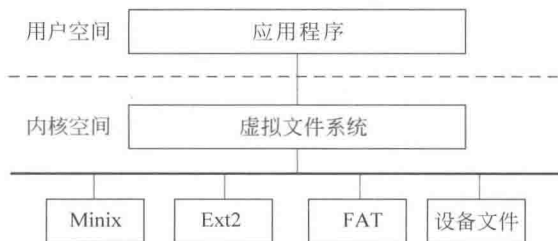


图 1.2 Linux 文件系统

网络协议和网络设备驱动程序。网络协议负责实现每一种可能的网络传输协议；网络设备驱动程序负责与硬件设备通信，每一种可能的硬件设备都有相应的设备驱动程序。

5. 进程间通信

进程间通信提供完成进程之间通信的功能，而 Linux 内核支持多种通信机制，包括信号量、共享内存及管道等，这种机制可协助多进程多资源的互斥访问、进程间同步和消息传递。

1.4 Linux 内核机制

1.4.1 内核启动过程

内核启动是非常复杂的过程，如图 1.3 所示是内核启动过程的流程图。

总体而言，内核的启动可以概括为 6 个步骤。

(1) 实模式的入口函数 `_start()`：在 `header.S` 中，这里会进入众所周知的 `main` 函数，它复制 `bootloader` 的各个参数，执行基本硬件设置，解析命令行参数。

(2) 保护模式的入口函数 `startup_32()`：在 `compressed/header_32.S` 中，这里会解压 `bzImage` 内核映像，加载 `vmlinux` 内核文件。

(3) 内核入口函数 `startup_32()`：在 `kernel/header_32.S` 中，这就是所谓的进程 0，它会进入体系结构无关的 `start_kernel()` 函数，即 Linux 内核启动函数。`start_kernel()` 会做大量的内核初始化操作，解析内核启动的命令行参数，并启动一个内核线程来完成内核模块初始化的过程，然后进入空闲循环。

(4) 内核模块初始化的入口函数 `kernel_init()`：在 `init/main.c` 中，这里会启动内核模块、创建基于内存的 `rootfs`、加载 `initramfs` 文件或 `cpio-initrd`，并启动一个内核线程来运行其中的 `/init` 脚本，完成真正的根文件系统的挂载。

(5) 根文件系统挂载脚本 `/init`：这里会挂载根文件系统、运行 `/sbin/init`，从而启动进程 1。

(6) `init` 进程的系统初始化过程：执行相关脚本，以完成系统初始化，例如设置键盘、字体、装载模块、设置网络等，最后运行登录程序，出现登录界面。

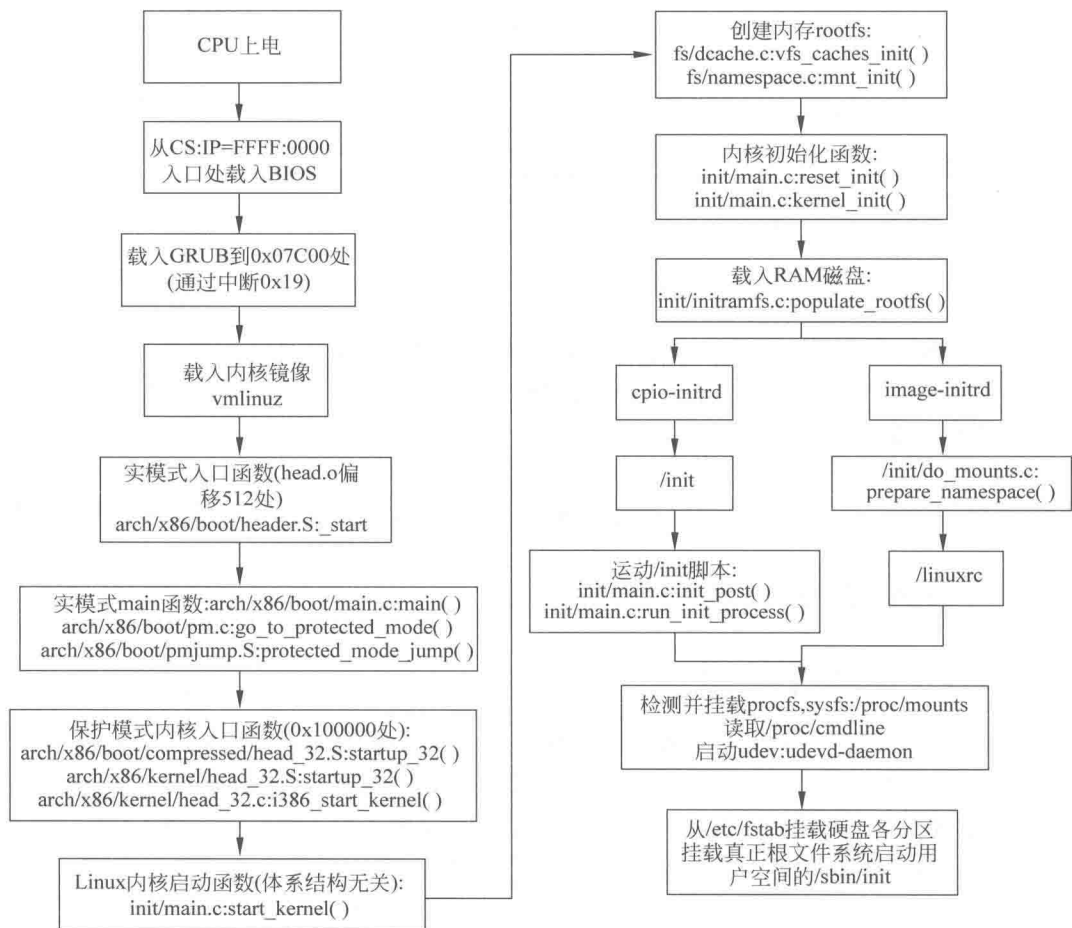


图 1.3 Linux 内核启动流程图

1.4.2 模块机制

模块是 Linux 内核中非常重要的机制，它为 Linux 内核向外提供了一个插口，其全称为动态可加载内核模块（Loadable Kernel Module）。Linux 内核之所以提供模块机制，是因为它本身是一个单内核（monolithic kernel）。单内核的最大优点是效率高，因为所有的内容都集成在一起，其缺点是可扩展性和可维护性相对较差，模块机制就是为了弥补这一缺陷。

模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。它在运行时被链接到内核作为内核的一部分在内核空间运行，这与运行在用户空间的进程是不同的。模块通常由一组函数和数据结构组成，用来实现一种文件系统、一个驱动程序或其他内核上层的功能。

模块从创建到使用需要经过源代码、Makefile 文件、编译模块、加载模块和最后的卸载

模块等步骤。下面介绍模块的加载与卸载。

在 Linux 中,对模块进行加载有两种方法,一种是手动加载,另一种是自动加载。前者使用 `insmod` 或 `modprobe` 命令实现,后者通过内核线程 `kmod` 来实现,而 `kmod` 通过调用 `modprobe` 来实现模块的加载。在这里以 `insmod` 为例分析模块的加载过程。

(1) `insmod` 首先通过系统调用 `query_module()` 遍历模块链表来获得系统中的所有符号及其在内存中的物理地址,然后利用得到的符号表修正模块中引用到的外部符号,在此过程中记录下该模块所要用的模块。由于是直接引用这些符号在内存中的物理地址进行更正,所以模块内核空间的地址引用是正确的,如果该模块还有一些符号的地址未知,则该模块不能被加载。

(2) `insmod` 填写模块 `module_ref` 表。其中 `dep` 指向本模块所使用的模块,即在修正本模块使用到的外部符号过程中标记过的模块,这些模块在内存中的物理地址也是通过系统调用 `query_module` 得到的。由于别的模块可能要使用本模块提供的服务,所以还需要提供本模块的输出符号表,此部分工作也由 `insmod` 完成。

(3) `insmod` 发出系统调用 `create_module()`。由该系统调用为模块分析足够的内存空间,并初始化位于该空间起始处的 `struct module` 结构体,然后 `insmod` 通过系统调用 `init_module()` 让系统完成余下工作。

相比于模块的加载,模块的卸载比较简单。同样模块的卸载也有两种方式,一种是用户使用 `rmmod` 命令完成卸载,另一种通过 `kerneld` 或 `kmod` 自动卸载。以 `rmmod` 为例,在系统找到欲卸载的模块后,根据其引用链表检查是否有别的模块要使用本模块,若有则打印出错信息并停止卸载;否则调用 `delete_module()` 函数完成卸载。该函数将调用 `free_module` 函数完成 4 项工作:释放系统分配给模块的资源;修改模块所依赖的所有模块的引用链表,将该模块从链表中删除;将该模块从系统的模块链表中删除;释放分配给该模块的核心内存。

2.1 devfs 设备文件系统

devfs 设备文件系统于 Linux 2.4 内核引入,在使用之初许多工程师给予了高度的评价,它的出现使得设备驱动程序能自主地管理设备文件。它挂载于 Linux 文件系统系统中的 /dev 目录下。

devfs 具有如下优点:

- (1) 可以通过程序在设备初始化时,在 /dev 目录下创建设备文件,卸载设备时将它删除。
- (2) 设备驱动程序可以指定设备名、所有者和权限位,用户空间程序可以修改所有者和权限位。
- (3) 不再需要为设备驱动程序分配主设备号及处理次设备号,在程序中可以直接给注册函数传递 0 主设备号以动态获得可用的主设备号,并在 devfs_register() 中指定次设备号。

当然,devfs 也具有较大的缺陷,例如:

- (1) 不确定的设备映射,有时同一个设备映射的设备文件可能不同,例如 U 盘可能对应 sda 可能对应 sdb。
- (2) 没有足够的主/辅设备号,当设备过多的时候,显然这是一个问题。
- (3) /dev 目录下文件太多而且不能表示当前系统上的实际设备。
- (4) 命名不够灵活,内核完成命名,用户无法指定。
- (5) 不存在的节点被打开时自动加载驱动程序。

通过以上分析可知,devfs 的出现帮助用户较为方便地进行设备操作,但是由于设计上的问题,它也存在很多缺陷,随着 Linux 的不断发展,已经不能适应新的需求。

2.2 sysfs 文件系统

在 Linux 2.6 内核以后,引入了一个新的文件系统 sysfs 以替代 devfs,完成它不支持的需求,一直延续使用至今。sysfs 挂载于 /sys 目录下,跟 devfs 一样,它也是一个虚拟文件系

统,用来对系统的设备进行管理,它把实际连接到系统上的设备和总线组织成一个分级的文件,用户空间的程序同样可以利用这些信息以实现和内核的交互,该文件系统是当前系统上实际设备树的一个直观反映。

它将系统中的设备组织成层次结构,可解决智能电源管理和设备分类的需求。它向用户模式程序提供详细的内核数据结构信息,提供接口给用户空间对驱动和设备进行配置,实现与用户空间的通信,用户甚至可以通过 echo、cat 等命令来直接操作设备的属性文件,完成设备配置。例如通过在终端输入命令 `echo 1 > /sys/devices/platform/leds-qpio/leds\LED1/brightness`,就可以实现点亮设备的 LED1 的功能。

sysfs 挂载于 /sys 目录下,目录下各子目录的名称及功能如下:

- (1) block: 系统中发现的每个块设备在该目录下对应一个目录。
- (2) bus: 内核中注册的每条总线在该目录下对应一个目录,例如 ide、pci、scsi 和 usb,等等。
- (3) bus/某一总线/devices: 该总线下所有设备在该目录下对应一个目录。
- (4) bus/某一总线/drivers: 该总线下所有驱动在该目录下对应一个目录。
- (5) classes: 将设备按照功能进行分类,例如 net 目录下包含所有的网络设备。
- (6) dev: 包含 block 和 char 两个文件,内含部分块设备和字符设备符号链接。
- (7) devices: 包含系统所有设备,并根据设备挂接总线类型组织成层次结构。
- (8) firmwares: 系统中的固件信息。
- (9) fs: 描述系统中存在的文件系统。
- (10) hypervisor: 新版本内核中的空目录。
- (11) kernel: 系统内核的配置参数。
- (12) module: 系统中所有模块的信息。
- (13) power: 系统中电源选项。

sys 下面的目录和文件反映了整台机器的系统状况。其中 10 个目录代表了 10 种完全不同的设备类型,这些目录只是给我们提供了如何去看整个设备模型的不同视角。真正的设备信息放在 devices 子目录下,Linux 系统中的所有设备都可以在这个目录里找到。如图 2.1 所示,bus 下对应驱动和设备,classes 下有设备的不同分类,分类下也对应各种设备,实际上它们都是 devices 目录下设备文件的符号链接。

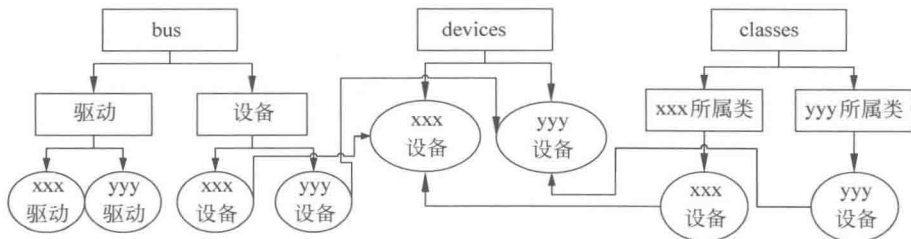


图 2.1 设备文件对应关系

2.3 udev 设备文件系统

udev 是另一种设备管理工具,最初在 Linux 2.6 中使用。它能够根据系统中设备的状况动态更新设备文件,包括设备文件的创建、删除,等等,是实现系统热插拔功能的主要工具。用户空间的工具 udev 利用了 sysfs 提供的信息来实现所有 devfs 的功能,但不同的是 udev 运行在用户空间,devfs 运行在内核空间,而且 udev 不存在 devfs 那些先天的缺陷。

udev 分为 3 个子计划,如图 2.2 所示,namedev 为设备命名子系统,为发现的设备提供默认的命名规则,libsysfs 是访问 sysfs 文件系统并从中获取设备信息的标准接口,udev 是 namedev 和 libsysfs 连接的桥梁,利用获取的信息完成设备节点文件的动态创建和删除策略。



图 2.2 udev 具体工作流程

udev 结合 sysfs 才能完整地实现设备的热插拔功能,同时解决 devfs 中存在的部分缺陷,例如无法自主命名,/dev 下不对应实际设备等。

udev 完全工作在用户态(userspace)下,利用设备加入或移除时内核所发送的 hotplug 事件(event)来工作。关于设备的详细信息由内核输出(export)到位于/sys 的 sysfs 文件系统中。所有设备的命名策略、权限控制和事件处理都是在用户态下完成的。与此相反,devfs 是作为内核的一部分工作的。

一般 Linux 系统使用创建静态设备的方法,因此在/dev 目录下创建了大量的设备节点(有时会有数千个节点),而不管对应的硬件设备实际上是否存在。这通常是由 makedev 脚本完成,这个脚本包含许多调用 mknod 程序的命令,为这个世界上可能存在的每个设备创建相应的主设备号和次设备号。而使用 udev 方式,只有被内核检测到的设备才为其创建设备节点。

2.4 主要数据结构

2.4.1 kobject

kobject 是采用面向对象的思想而设计的特殊数据结构,既然是面向对象,它就有父类节点,可以呈层次结构排列的对象。

kobject 主要包括对象引用计数、维护对象链表、对象上锁和在用户空间的表示 4 种功能。在用户空间的表示,主要是通过 sysfs 实现,每个在内核注册的 kobject 对象对应于 sysfs 文件系统中的一个目录,这些功能主要通过它对应结构体所包含的对象来完成。

内核中由 struct kobject 表示,使所有设备在底层都具有统一的接口,提供基本的对象管理。它是组成设备的基本结构,类似于 C++ 中的基类,kobject 把一些基本的对象和操作封装起来,避免了重复实现,C 中不支持继承,故需要在子类中定义 kobject 对象,从而实现基类的作用。

kobject 定义于文件 linux/kobject.h 中,可创建呈层次结构排列的对象。在内核中注册的每个 kobject 对象,对应于 sysfs 文件系统中的一个目录。以下为 kobject 结构体的源代码:

```
struct kobject {
    const char          * name;
    struct list_head    entry;
    struct kobject      * parent;
    struct kset         * kset;
    struct kobj_type     * ktype;
    struct sysfs_dirent * sd;
    struct kref          kref;
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

其中

- (1) name 表示真正存放名字的数组。
- (2) kref 表示其他结构体,entry 与 kset 一起联合使用。
- (3) parent 指向 kobject 的父对象。
- (4) kset 和 ktype 指针分别指向 kobject 所在的集合和符合的类型。
- (5) sd 指针指向 sysfs_dirent 结构体,代表 sysfs 中的 kobject。

而 kobject 的主要函数包括:

kobject_init();	//kobject 初始化函数
kobject_set_name();	//设置指定 kobject 的名称
kobject_get();	//将 kobject 对象的引用计数加 1
kobject_put();	//将 kobject 对象的引用计数减 1,如果引用计数降为 0, //则调用 kobject release()释放该 kobject 对象
kobject_register();	//kobject 注册函数
kobject_unregister();	//kobject 注销函数,调用 kobject put()减少该对象的引用计数, //如果引用计数降为 0,则释放 kobject 对象

2.4.2 ktype

kobject 结构体中定义了一个很关键的对象 `struct kobj_type * ktype`。`kobj_type` 用来表示该对象的类型,是具有相同操作的 kobject 的集合。它负责管理这一类 kobject 在 sysfs 下的操作,主要是 show 和 store 函数的实现。它的结构体定义在 `include/linux/kobject.h` 文件中,主要对象如下:

```
struct kobj_type {
    void (* release)(struct kobject *);    //释放 kobject 占用的资源
    struct sysfs_ops * sysfs_ops;        //指针指向 sysfs 操作表
    struct attribute * * default_attrs;    //一个 sysfs 文件系统默认属性列表
};
```

其中,release 函数用于释放 kobject 占用的资源,当 kref 代表的对象引用计数为 0 时,release 函数被调用来释放资源,对象引用计数的增加和减少通过 `kobject_get()` 和 `kobject_put()` 函数来完成。sysfs 操作表包含两个函数 `show()` 和 `store()`,对应文件的读和写,当用户态读取属性时,show() 被调用,而 store() 函数用于存储用户空间传入的属性值。struct attribute 为默认的属性列表,对应 sysfs 文件系统中的属性文件。

2.4.3 kset

简单地说,kset 就是 kobject 的集合,kset 就像一个容器来包容 kobject 这些子集。kset 中的所有 kobject 可以拥有相同的 ktype,也可以各自属于自己的 ktype。kset 中也有属于自己的 kobject,不过该 kobject 属于自动处理的,可以忽略它核心代码处理的过程。

kset 最重要的是建立上层和下层的关联性,kobject 也会利用它分辨自己属于哪一个类型,然后在 /sys 下建立自己所属的 kset,并把 kobj_type 指定成该 kset 下的 kobj_type。

kset 是 kobject 对象的集合体,可看做一个容器,将相关 kobject 对象置于同一位置,其结构体定义于 `<linux/kobject.h>` 中。需要注意的是,具有相同 ktype 的 kobject 可以分组到不同的 kset 中。kset 数据结构如下所示:

```
struct kset {
    struct list_head list;
    spinlock_t list_lock;
    struct kobject kobj;
    const struct kset_uevent_ops * uevent_ops;
};
```

其中

- (1) list 变量连接该集合 kset 中所有的 kobject 对象。
- (2) kobj 代表了该集合的基类。
- (3) uevent_ops 指向一个用于处理集合中 kobject 对象的操作的结构体。

kset 的主要函数包括:

```

kset_init()           //完成指定 kset 的初始化
kset_add()           //把一个 kset 加到层次结构中
kset_get()           //和 kset_put()分别增加和减少 kset 对象的引用计数(其实就是内嵌 kobject 的引用计数)
kset_register()       //完成 kset 的注册
kset_unregister()     //完成 kset 的注销

```

2.4.4 三者关系

kobject 结构为如图 2.3 的 kobject 类型部分, kset 结构为如图 2.3 的 kset 类型部分, 一个 kobject 加入一个 kset, 主要是 kobject 结构体中的相关字段记录了对应的 kset 信息, 流程为: ①记录了 kobject 所对应 kset, 指向的是 kset 所包含的 kobject 的地址; ②记录 kobject 所对应的 kset 的 kset 指针; ③记录 kobject 的类型; ④记录了 kset 所有的 kobject 的链子, 这个链子是一个双向链表, 每当有一个 kobject 加入到当前的 kset, 就会调用 list_add_tail()函数, 把要加入 kset 的 kobject 连入链表的结尾, 最终形成一个链表。整个过程如图 2.3 所示。

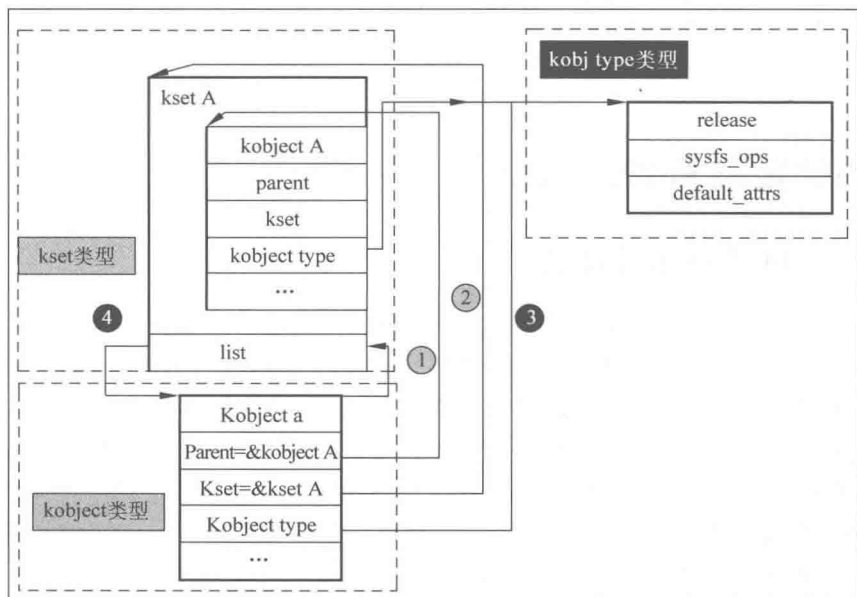


图 2.3 kobject、ktype、kset 之间的关系

当有另外一个 kobject 要加入当前的 kset, 其中的①②③步跟第一个加入当前 kset 的 kobject 是一样的, 即设置要加入的 kobject 的成员, 使之指向当前的 kset 对应数据, 第④步需要把 kobject 添加到 kset 的 list 的尾部。图 2.4 表示了新的 kobject b 加入到 kset A。

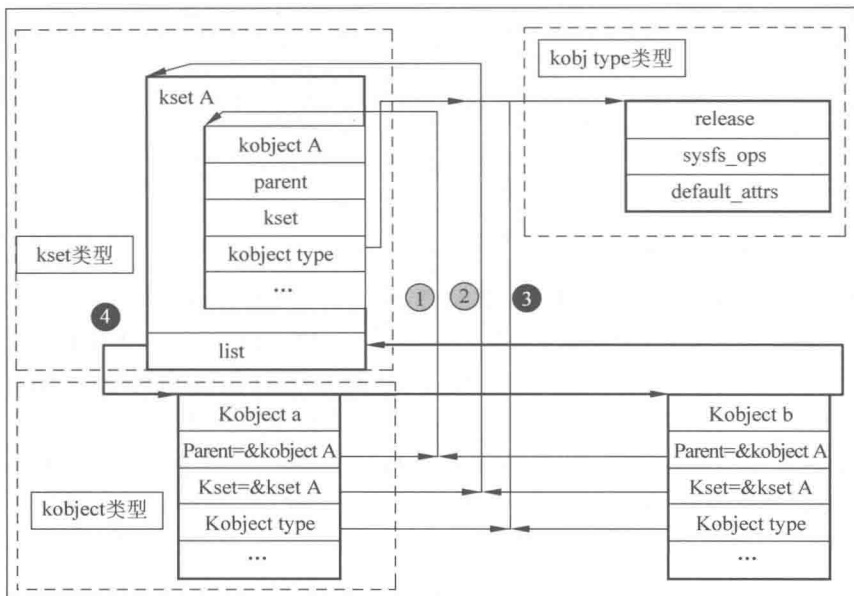


图 2.4 将新 kobject 加入到已有的 kset

2.5 热插拔设备管理机制

2.5.1 热插拔事件流程

在 Linux 环境中,热插拔设备管理机制并非是设备驱动单独进行的一项工作。可以将其概括为所处系统、设备驱动、设备模型、通信机制等几方面工作相结合处理的工作机制。下面从设备插入系统产生热插拔事件来介绍相应设备工作流程:

(1) 内核检测到新硬件插入,然后分别通知 hotplug 和 udev。前者用来装入相应的内核模块(例如 usb-storage),后者用来在 /dev 中创建相应的设备节点(例如 /dev/sda1)。

(2) udev 创建了相应的设备节点之后,会将这一消息通知 hal 的守护程序 hald。udev 还必须保证新创建的设备节点可以被普通用户访问。

(3) hotplug 装入了相应的内核模块并把这一消息通知给 hald。

(4) hald 在收到 hotplug 和 udev 发出的消息之后,认为新硬件已经正式被系统认可。此时它会通过一系列精心编写的规则文件(xxx-policy, fdi),把发现新硬件的消息通过 netlink 发送出去,同时还会调用 update-fstab 或 fstab-sync 来更新/etc/fstab,为相应的设备节点创建适合的挂载点。

(5) 卷管理器会监听 netlink 中发现新硬件的消息。根据所插入硬件(区分 U 盘和数码相机等)的不同,卷管理器会先将相应的设备节点挂载到 hald 创建的挂载点上,然后再打开

不同的应用程序。如果是在光驱中插入光盘,过程比较简单。因为光驱本身就是一个固定的硬件,无须 hotplug 和 udev 的协助。

(6) hald 会自己监视光驱,并且将光盘托架开合的消息通过 netlink 发出去。

(7) 卷管理器负责检查光驱中的盘片内容,进行挂载,并调用合适的应用程序。要注意,hald 的工作是从上游得到硬件就绪的消息,然后将这个消息转发到 netlink 中。尽管它会调用程序来更新 fstab,实际上自己并不执行挂载的工作。

按照以上流程进行操作后,系统将完成从设备热插拔响应到添加等操作。设备插入情况如图 2.5 所示。插入情况因为涉及首次添加设备等情况,所以操作流程较复杂。

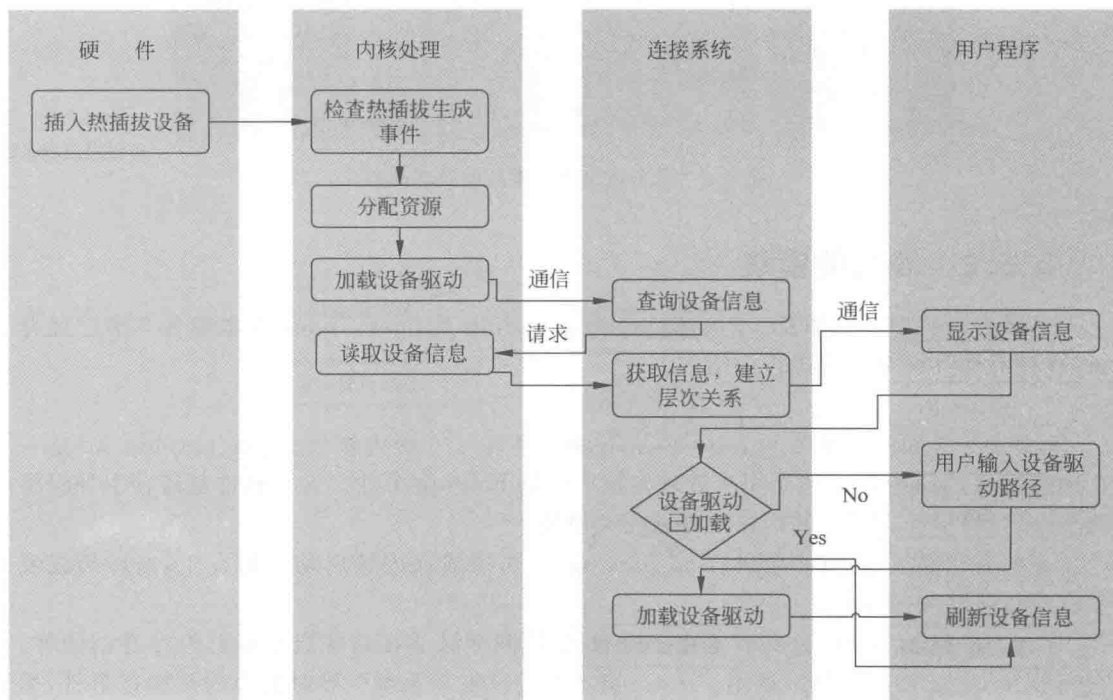


图 2.5 设备插入在各模块间处理流程

设备拔出情况如图 2.6 所示。因为在插入过程中已经将部分工作完成,为了使拔出后的下一次插入时设备正常运行,不需要对驱动等信息进行处理,所以操作流程相比于设备插入简单。

在进行这一系列相应操作之后,将会关联设备驱动。具体分为以下两种情况:

- 若驱动直接编译进内核或在启动时加载,则无需在 udev 中加载驱动模块,在 `bus_probe_device()` 中会为其找到相应的驱动。
- 若驱动需要动态加载,则先有 device 或先有 driver 均有可能。内核层面,在手动加入的驱动 `register` 函数中,找到相应 device 进行关联。用户层面,udev(目前的情况是这样,以前也有其他方式,例如 `/sbin/hotplug` 等)中,动态加载相应脚本程序。

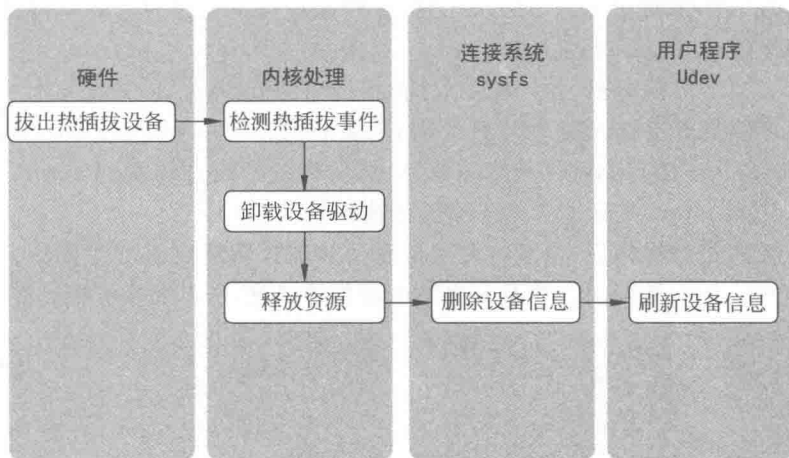


图 2.6 设备拔出在各模块间处理流程

2.5.2 涉及的模块

热插拔事件中主要涉及 3 个模块 hotplug、netlink 及 udev。udev 在本章 2.3 节已经介绍,在此对另外两个模块进行介绍。

1. hotplug

hotplug 包和内核里的 hotplug 模块不是一回事,2.6 版内核里的 pci_hotplug.ko 是一个内核模块,而 hotplug 包是用来处理内核产生的 hotplug 事件。这个软件包还在引导时检测现存的硬件,并在运行的内核中加载相关模块。

不但有热插拔,还有冷插拔(cold plugging)。热插拔在内核启动之后发生,而冷插拔发生在内核启动的过程中。

/etc/hotplug/*.rc 这些脚本用于冷插拔,检测和激活在系统启动时已经存在的硬件。它们被 hotplug 初始化脚本调用。*.rc 脚本会尝试恢复系统引导时丢失的热插拔事件,举例来说,内核没有挂载根文件系统。

/etc/hotplug/*.agent 这些脚本将被 hotplug 调用,以响应内核产生的各种不同的热插拔事件,导致插入相应的内核模块和调用用户预定义的脚本。

/sbin/hotplug 内核默认情况下,将在内核态的某些事情发生变化时(例如硬件的插入和拔出)调用此脚本。

发送热插拔事件的子系统(subsystem)包括总线驱动(USB、PCI 等)和一些设备的抽象层(网络接口、磁盘分区等)。它们通过/sbin/hotplug 的第一个参数来识别。

对于设备驱动来说,需要在代码里设置 MODULE_DEVICE_TABLE,指向驱动程序感兴趣的设备 ID 列表。

2. netlink

netlink 是一种特殊的 socket,它是 Linux 特有的,目前多用于在最新的 Linux 内核中

使用 netlink 进行内核与用户之间的通信,例如路由 (NETLINK_ROUTE)、防火墙 (NETLINK_FIREWALL)等。

netlink 是一种在内核与用户应用进行双向数据传输的有效方式。用户态应用使用标准的 socket API 即可。内核态需使用专门的内核 API 调用。

netlink 是一种异步通信机制。在内核与用户态应用之间传递的消息保存在 socket 缓存队列中,发送消息只是把消息保存在接受者的 socket 的接受队列,而不需要等待接受者收到消息。

netlink 的内核部分可以采用模块的方式实现,使用 netlink 的应用部分和内核部分没有编译时依赖,但系统调用时有依赖,新的系统调用必须静态链接到内核中,使用新系统调用的应用在编译时需要依赖内核。

2.5.3 关键驱动函数

当系统配置发生变化时,例如添加 kset 到系统,或移动 kobject,一个通知会从内核空间发送到用户空间,这就是热插拔事件。Linux 中采用 kset_uevent_ops 函数来对热插拔事件进行相关响应。

对热插拔事件的实际控制,是由保存在 kset_uevent_ops 结构中的函数完成的,它们分别是 filter、name、uevent 函数。

```
struct kset_uevent_ops {
    int (* const filter)(struct kset * kset, struct kobject * kobj);
    const char * (* const name)(struct kset * kset, struct kobject * kobj);
    int (* const uevent)(struct kset * kset, struct kobject * kobj,
        struct kobj_uevent_env * env);
};
```

当 kobject 和 kset 状态变化时 3 个函数被调用。

- (1) filter: 决定是否将事件传递到用户空间。如果 filter 返回 0,不传递事件。
- (2) name: 负责将相应的字符串传递用户空间的热插拔处理程序。
- (3) uevent: 将用户空间需要的参数添加到环境变量中。返回值正常是 0,若返回非 0,则终止热插拔事件的产生。

3.1 同步机制

共享内存的应用程序必须特别留意保护共享资源,防止共享资源被并发访问。内核也不例外。共享资源之所以要防止并发访问,是因为多个执行程序同时访问和操作数据,就有可能发生各线程之间相互覆盖共享数据的情况,造成被访问数据处于不一致的状态。并发访问共享数据是造成系统不稳定的隐患,而且这种错误难以跟踪和调试。

要做到对共享资源的恰当保护很困难。在 Linux 还未支持对称多处理器的时候,避免并发访问数据的方法相对比较简单。在单一处理器的时候,只要在中断发生,或在内核代码显式地请求重新调度执行另一个任务的时候,数据才可能被并发访问。

从 Linux 2.0 开始,内核开始支持对称多处理器,并且对它的支持不断地加强和完善。支持多处理器意味着内核代码可以同时运行在两个或更多的处理器上。因此,如果不加以保护,运行在两个不同处理器上的内核代码完全可能在同一时刻并发访问共享数据。随着 2.6 版内核的出现,Linux 内核已发展成抢占式内核,这意味着调度程序可以在任何时刻抢占正在运行的内核代码,重新调度执行其他的进程。

并发指的是多个执行单元同时被并行执行,而并发的执行单元对共享资源(硬件资源和软件上的全局变量、静态变量等)的访问很容易导致竞态。多个程序对共享资源的并发访问是 Linux 设备驱动中必须解决的问题之一。

在 Linux 中,主要的竞态发生在如下几种情况:

(1) 对称多处理器(SMP)的多个 CPU 使用共同的系统总线,因此可访问共同的外设和存储器。

(2) 单 CPU 内进程与抢占它的进程。

(3) 中断(硬中断、软中断、Tasklet、底半部)与进程之间。

总之,只要并发的多个执行单元存在对共享资源的访问,竞态就有可能发生。如果中断处理程序访问进程正在访问的资源,则竞态也会发生。多个中断之间本身也可能引起并发而导致竞态(中断被更高优先级的中断打断)。

解决竞态问题的途径是保证对共享资源的互斥访问,所谓互斥访问就是指一个执行单元在访问共享资源的时候,其他的执行单元都被禁止访问。访问共享资源的代码区域称为临界区,临界区需要以某种互斥机制加以保护。

所谓临界区就是访问和操作共享数据的代码段,为了避免在临界区中发生并发访问,编程者必须保证这些代码原子地执行。也就是说,代码在执行结束前不可被打断,就如同整个临界区是一个不可分割的指令,如果两个执行线程有可能处于同一个临界区中,那么程序就包含一个 bug。如果这种情况发生,就称之为竞争条件,避免并发和防止竞争条件称为同步。

3.1.1 内核同步机制分类

Linux 提供了多种解决竞态问题的同步机制,例如中断屏蔽、原子操作、自旋锁、信号量和大内核锁等。具体分类如图 3.1 所示,这些同步机制根据自身的特点适用于不同的应用场景。下面将详细阐述各同步机制。

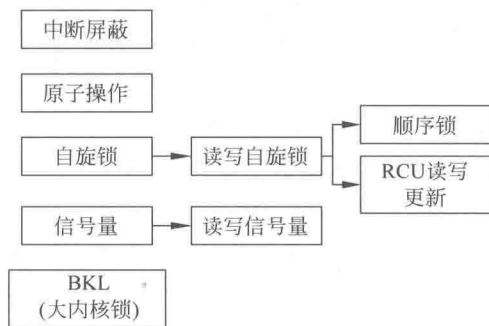


图 3.1 内核同步机制分类

1. 中断屏蔽

1) 定义

中断屏蔽是在单 CPU 范围内避免竞态的一种简单方法,在进入临界区之前屏蔽系统的中断。CPU 一般都具备屏蔽中断和打开中断的功能,这项功能可以保证正在执行的内核执行路径不被中断处理程序抢占,防止某些竞态条件的发生。总之,中断屏蔽将使得中断与进程间的并发不再发生。由于 Linux 内核的进程调度等操作都依赖中断来实现,内核抢占进程之间的并发也得以避免。

2) 中断屏蔽的使用方法

```

local_irq_disable()           //屏蔽中断
...
critical section              //临界区
...
local_irq_enable()            //打开中断
  
```

3) 中断屏蔽的特点

由于 Linux 系统的异步 IO, 进程调度等很多重要操作都依赖于中断, 在屏蔽中断期间所有的中断程序都无法得到处理, 因此长时间的屏蔽是很危险的, 有可能造成数据丢失甚至系统崩溃, 这就要求在屏蔽中断之后, 当前的内核执行路径应当尽快地执行完临界区的代码。

中断屏蔽只能禁止本 CPU 内的中断, 因此并不能解决多 CPU 引发的竞态, 所以单独使用中断屏蔽并不是一个值得推荐的方法, 它一般和自旋锁配合使用。

2. 原子操作

1) 定义

所谓原子操作, 就是该操作绝不会在执行完毕前被任何其他任务或事件打断, 也就是说, 它是最小的执行单位, 不可能有比它更小的执行单位。

Linux 内核提供了一系列函数来实现内核中的原子操作, 这些函数又分成两类, 分别是整型原子操作和位原子操作。它们的共同点是在任何情况下操作都是原子的, 内核代码可以安全地调用它们而不被打断。原子操作都需要硬件的支持, 因此很多函数都与 CPU 架构密切相关。它们都使用汇编语言实现, 因为 C 语言不能实现这样的操作。

针对整数的原子操作只能对 `atomic_t` 类型的数据进行处理。在这里之所以引入了一个特殊数据类型, 而没有直接使用 C 语言的 `int` 类型, 主要有以下几点原因: 首先, 让原子函数只接受 `atomic_t` 类型的操作数, 以确保原子操作只与这种特殊类型数据一起使用, 同时, 这也保证了该类型的数据不会传递给其他任何非原子函数; 其次, 使用 `atomic_t` 类型确保编译器不对相应的值进行访问优化, 使得原子操作最终接收到正确的内存地址; 最后, 在不同体系结构上实现原子操作时, 使用 `atomic_t` 可以屏蔽其间的差异。

位原子操作是对普通指针进行的操作, 所以不像整型原子操作对应 `atomic_t` 类型, 这里没有特殊的数据类型。它是与体系结构相关的, 对普通的内存地址进行操作。

2) 原子操作使用情况

<pre>线程 1 atomic_increment i(7->8) ...</pre>	<pre>线程 2 ... atomic_increment i(8->9)</pre>
---	---

两个原子操作绝对不可能并发地访问同一个变量, 这样也就绝对不可能引发竞争。

3) 原子操作 API 函数

(1) 整型原子操作

<pre>atomic_read(atomic_t * v); atomic_set(atomic_t * v, int i); void atomic_add(int i, atomic_t * v); atomic_sub(int i, atomic_t * v); int atomic_sub_and_test(int i, atomic_t * v);</pre>	<pre>//该函数对原子类型的变量进行原子读操作 //它返回原子类型的变量 v 的值 //该函数设置原子类型的变量 v 的值为 i //该函数给原子类型的变量 v 增加值 i //该函数从原子类型的变量 v 中减去 i //该函数从原子类型的变量 v 中减去 i, 并判 //断结果是否为 0, 如果为 0, 返回真; 否则返回假</pre>
---	--

```

void atomic_inc(atomic_t * v);           //该函数对原子类型变量 v 原子地增加 1
void atomic_dec(atomic_t * v);           //该函数对原子类型的变量 v 原子地减 1
int atomic_dec_and_test(atomic_t * v);    //该函数对原子类型的变量 v 原子地减 1,并判
                                           //断结果是否为 0,如果为 0,返回真;否则返回假
int atomic_inc_and_test(atomic_t * v);    //该函数对原子类型的变量 v 原子地增加 1,并判
                                           //断结果是否为 0,如果为 0,返回真;否则返回假
int atomic_add_negative(int i, atomic_t * v); //该函数对原子类型的变量 v 原子地增加 1
                                           //并判断结果是否为负数,如果是,返回真;否则返回假

```

(2) 位原子操作

```

void set_bit(int nr, void * addr);        //原子地设置 addr 所指的第 nr 位
void clear_bit(int nr, void * addr);      //原子地清空所指对象的第 nr 位
void change_bit(nr, void * addr);         //原子地翻转 addr 所指的第 nr 位
int test_bit(nr, void * addr);            //原子地返回 addr 位所指的第 nr 位
int test_and_set_bit(nr, void * addr);    //原子地设置 addr 所指对象的第 nr 位,并返回原先的值
int test_and_clear_bit(nr, void * addr);  //原子地清空 addr 所指对象的第 nr 位,
                                           //并返回原先的值
int test_and_change_bit(nr, void * addr); //原子地翻转 addr 所指对象的第 nr 位,
                                           //并返回原先的值

```

4) 原子操作的特点

原子操作的优点是系统的开销小并且对高速缓存的影响小,缺点是不适用于有高性能要求的代码,原子操作通常用于实现资源的引用计数。

3. 自旋锁(spin lock)

1) 定义

自旋锁是 Linux 内核中最常见的锁,自旋锁最多只能被一个可执行线程持有。如果一个执行线程试图获得一个被争用(已经被持有)的自旋锁,那么该线程就会一直进行忙循环——旋转——等待锁重新可用。如果锁未被争用,请求锁的执行线程便立刻得到它,继续执行。在任意时间,自旋锁都可以防止多于一个的执行线程同时进入临界区。同一个锁可以用于多个位置。所以,对于给定数据的所有访问都可以得到保护和同步。

2) 自旋锁 API 函数

(1) 定义和初始化自旋锁

```

spinlock_t lock;
spin_lock_init(x);           //该宏用于初始化自旋锁 x. 自旋锁在真正使用前必须先初始化
                               //该宏用于动态初始化
#define SPINLOCK(x);         //该宏声明一个自旋锁 x 并初始化
SPIN_LOCK_UNLOCKED;          //该宏用于静态初始化一个自旋锁

```

(2) 获得自旋锁

```

spin_lock(lock);              //该宏用于获得自旋锁 lock,如果能够即时获得锁,立即返回;否
                               //则,将自旋在那里,直到该自旋锁的保持者释放,这时获得锁
                               //并返回. 总之,只有获得锁才返回

```

```

spin_trylock(lock);           //该宏尽力获得自旋锁 lock,如果能即时获得锁,则获得锁并返回真;
                               //否则不能即时获得锁,返回假.它不会自旋等待 lock 被释放
spin_is_locked(x);           //该宏用于判断自旋锁 x 是否已被某执行单元保持(即被锁),如果
                               //是,返回真; 否则返回假
spin_lock_irqsave(lock, flags); //该宏获得自旋锁的同时把标志寄存器的值保存到变量
                               //flags 中并使本地中断失效
spin_lock_irq(lock);          //该宏类似于 spin_lock_irqsave,但该宏不保存标志寄存器的值
spin_lock_bh(lock);           //该宏在得到自旋锁的同时使本地软中断失效
spin_trylock_irqsave(lock, flags); //该宏如果获得自旋锁 lock,将保存标志寄存器的值到
                               //变量 flags 中,并且使本地中断失效,如果没有获得锁,
                               //则什么也不做. 如果能够即时获得锁,等同于
                               //spin_lock_irqsave; 如果不能获得锁,等同于
                               //spin_trylock. 如果该宏获得自旋锁 lock,那需要使用
                               //spin_unlock_irqrestore 来释放
spin_trylock_irq(lock);       //该宏类似于 spin_trylock_irqsave,但该宏不保存标志寄存器
                               //如果该宏获得自旋锁 lock,需要使用 spin_unlock_irq 来释放
spin_trylock_bh(lock);        //该宏如果获得了自旋锁,将使本地软中断失效; 如果得不到锁,
                               //则什么也不做. 因此,如果得到了锁,等同于 spin_lock_bh;
                               //如果得不到锁,等同于 spin_trylock. 如果该宏得到了自旋锁,需
                               //要使用 spin_unlock_bh 来释放
spin_can_lock(lock);          //该宏用于判断自旋锁 lock 是否能够被锁,实际是 spin_is_locked
                               //取反. 如果 lock 没有被锁,返回真; 否则返回假

```

(3) 释放自旋锁

```

spin_unlock(lock);            //该宏释放自旋锁 lock,它和 spin_trylock 或 spin_lock 配对使用. 如
                               //果 spin_trylock 返回假,表明没有获得自旋锁,因此不必使用
                               //spin_unlock 释放
spin_unlock_irqrestore(lock, flags); //该宏释放自旋锁 lock 的同时,也恢复标志寄存器的
                               //值为变量 flags 保存的值. 和 spin_lock_irqsave 配对使用
spin_unlock_irq(lock);        //该宏释放自旋锁 lock 的同时,也使本地中断. 和 spin_lock_irq
                               //配对应用.
spin_unlock_bh(lock);         //该宏释放自旋锁 lock 的同时,也打开本地软中断. 和 spin_lock_bh
                               //配对使用
spin_unlock_wait(x);           //该宏用于等待自旋锁 x 没有被所有执行单元保持,如果没有执行
                               //单元保持该自旋锁,该宏即时返回; 否则将循环在那里,直到
                               //该自旋锁被保持者释放

```

3) 自旋锁使用的一般形式

```

spinlock_t lock;              //定义一个自旋锁
spin_lock_init(x);            //获取自旋锁保护临界区
spin_lock(lock);              //临界区
...
spin_unlock(lock);            //解锁

```

4) 自旋锁的特点

一个被争用的自旋锁使得请求它的线程在等待锁重新可用时自旋,特别浪费处理器的时间,这种行为是自旋锁的特点。所以,自旋锁不应该被长时间持有。这点正是使用自旋锁的初衷:在短时间内进行轻量级加锁。还可以采用另外的方式处理对锁的争用:让请求线程睡眠,直到锁重新可用时再唤醒它。这样处理器就不必循环等待,可以去执行其他代码。这也会带来一定的开销——两次明显的上下文切换会额外使用较多的代码。因此,持有自旋锁的时间最好小于完成两次上下文切换的耗时。

Linux 内核实现的自旋锁是不可递归的,这点不同于自旋锁在其他操作系统中的实现。所以试图得到一个正持有的锁,必须自旋,等待自己释放这个锁。但同时又处于自旋忙等待中,所以永远没有机会释放锁,于是被自己锁死了。千万要小心自加锁。

在中断处理程序中使用自旋锁时,一定要在获取锁之前,首先禁止本地中断。否则中断处理程序会打断正持有锁的内核代码,有可能会去争用这个已经被持有的自旋锁。

4. 读写自旋锁

1) 定义

读写自旋锁是一种比自旋锁粒度更小的锁机制,它保留了“自旋”的概念,为读和写分别提供了不同的锁。一个或多个读任务可以并发地持有读锁,相反,用于写的锁最多只能被一个写任务持有,而且不能有并发的读操作。但申请线程在等待期内依然使用忙等待方式。

2) 读写自旋锁 API 函数

(1) 定义和初始化读写自旋锁

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED;    //静态初始化
rwlock_t my_rwlock;                        //定义读写锁
rwlock_init(&my_rwlock);                  //动态初始化
```

(2) 读锁定

```
void read_lock(rwlock_t * lock);           //获得指定读锁
void read_lock_irq(rwlock_t * lock);       //禁止本地中断并获得指定读锁
void read_lock_irqsave(rwlock_t *, unsigned long flags); //存储本地中断的当前状态,禁止本地中断并获得指定读锁
```

(3) 读解锁

```
void read_unlock(rwlock_t * lock);         //释放指定的读锁
void read_unlock_irq(rwlock_t * lock);     //释放指定的读锁并激活本地中断
void read_unlock_irqrestore(rwlock_t *, unsigned long flags); //释放指定的读锁并将本地中断恢复到指定的前状态
```

(4) 写锁定

```
void write_lock(rwlock_t * lock);          //获得指定的写锁
void write_lock_irq(rwlock_t * lock);      //禁止本地中断并获得写锁
void write_lock_irqsave(rwlock_t *, unsigned long flags); //存储本地中断的当前状态,禁止本地中断并获得指定写锁
```

(5) 写解锁

```
void write_unlock(rwlock_t * lock);           //释放指定的写锁
void write_lock_irq (rwlock_t * )            //释放指定的写锁并激活本地中断
void write_unlock_irqrestore(rwlock_t * , unsigned long flags); //释放指定的写锁并将本地
//中断恢复到指定的前状态
```

3) 读写自旋锁使用的一般形式

```

rwlock_t lock;                                // 定义 rwlock
rwlock_init(&lock);                            // 初始化 rwlock
//读时获取锁
read_lock(&rwlock);
...                                              //临界资源
read_unlock(&rwlock);
//写时获取锁
write_lock_irqsave(&rwlock, flags);
...                                              //临界资源
write_unlock_irqrestore(&rwlock, flags);

```

4) 读写自旋锁的特点

在使用 Linux 读写自旋锁时,要考虑的一点是这种锁机制照顾读比照顾写要多一些。当读锁被持有时,写操作为了互斥访问只能等待,但是,读执行单元却可以继续成功地占用锁。而自旋等待的写执行单元在所有读执行单元释放锁之前无法获得锁。所以,大量读执行单元必定会使挂起的写执行单元处于饥饿状态。

自旋锁提供了一种快速简单的锁实现方法。如果加锁时间不长并且代码不会睡眠(例如中断处理程序),利用自旋锁是最佳选择。如果加锁时间可能很长或者代码在持有锁时有可能睡眠,那么最好使用信号量来完成加锁功能。

5. 顺序锁 (seqlock)

1) 定义

顺序锁是对读写锁的一种优化,读执行单元不会被写执行单元阻塞。读执行单元可以在写执行单元对被顺序锁保护的共享资源进行写操作时继续读,不必等待写执行单元完成写操作,写执行单元也不需要等待所有读执行单元完成读操作才进行写操作。

2) 顺序锁 API 函数

(1) 定义和初始化顺序锁

```

DEFINE_SEQLOCK(seqlock);           // 静态初始化
void seqlock_init(seqlock_t* seqlock); // 动态初始化

```

(2) 读执行单元相关

```
unsigned read_seqbegin(const seqlock_t * sl); //读执行单元访问共享资源前调用该函数,
//返回顺序锁 sl 当前顺序号,读执行单元
//实际没有任何得到锁和释放锁的开销
```



```
int read_seqretry(const seqlock_t * sl, unsigned start);    //读执行单元访问完共享资源后检
//查在读访问期间是否有写执行单元访问了该共享
//资源,如果是,读执行单元就需要重新进行读操作;
//否则,读执行单元成功完成了读操作
```

(3) 写执行单元相关

```
void write_seqlock(seqlock_t * sl);    //写执行单元在访问被顺序锁 sl 保护的共享资源前
//需要调用该函数来获取顺序锁 sl,获取后对顺序
//号加 1,以便读执行单元能够检查出在读期间是否
//有写执行单元访问过

void write_sequnlock(seqlock_t * sl);    //写执行单元再次对顺序号加 1,以便读执行单元
//能够检查出在读期间是否有写执行单元访问过,
//并释放顺序锁 sl. 保证写执行单元在整个写的
//过程中,计数器 sequence 的值是奇数;当没有
//写操作时,计数器的值是偶数

int write_tryseqlock(seqlock_t * sl);    //写执行单元尝试获取顺序锁 sl,若失败立即返回
//0,并不再自旋;否则返回值非 0,并对顺序号加
//1,以便读执行单元能够检查出在读期间是否有
//写执行单元访问过
```

3) 顺序锁使用的一般形式

```
DEFINE_SEQLOCK(seqlock);    //定义并初始化一个顺序锁
write_seqlock(&seqlock);    //写执行单元操作
...    //临界资源,写操作代码块
write_sequnlock(&seqlock);    //写执行单元释放锁
/* 读执行单元操作 read_seqbegin()返回顺序锁的当前顺序号;只要局部变量 seqnum 的值与顺序
号的当前值不匹配,说明读操作期间有写执行单元访问过共享资源,read_seqretry()就返回 1,读执
行单元重新读数据 */
do
{
    seqnum = read_seqbegin(&seqlock);
    ...    //临界资源,读操作代码块
} while(read_seqretry(&seqlock, seqnum));
```

4) 顺序锁的特点

如果读执行单元在读操作期间,写执行单元已经发生了写操作,那么读执行单元必须重新读取数据,确保得到的数据是完整的。

被保护的共享资源不能含有指针,因为写执行单元可能使指针失效,但读执行单元如果正要访问该指针,将导致错误。

6. RCU(Read-Copy Update)

1) 定义

RCU 的原理就是读取-复制-更新,对于被 RCU 保护的共享数据结构,读执行单元不需要获得任何锁就可以访问,但写执行单元在访问时首先复制一个副本,然后对副本进行修

改,最后使用一个回调(callback)机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作。

等待适当时机的这一时期称为 grace period(宽限期),就是所有引用该数据的 CPU 都退出对共享数据的操作。图 3.2 为删除操作等待 grace period 的示意图。

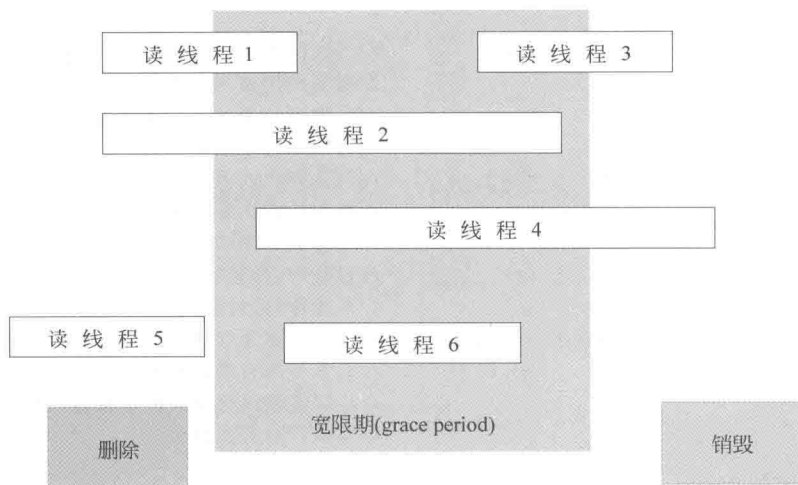


图 3.2 等待 grace period 的示意图

2) RCU 的 API 函数

(1) 读锁定

```
rcu_read_lock( );           //读执行单元在读取由 RCU 保护的共享数据时,使用该函数标记进入
                             //读端临界区
```

(2) 读解锁

```
rcu_read_unlock();          //该函数与 rcu_read_lock 配对使用,用以标记读执行单元退出读端
                             //临界区。读端临界区可以嵌套
```

(3) 同步 RCU

```
synchronize_rcu();           //该函数由 RCU 写端调用,它将阻塞写执行单元,直到经过 grace
                             //period 后,写执行单元才可以继续下一步操作。如果有多个 RCU
                             //写端调用该函数,它们将在一个 grace period 之后全部被唤醒。保
                             //证所有 CPU 都处理完正在运行的读端临界区
synchronize_sched();          //该函数用于等待所有 CPU 都处在可抢占状态,它能保证正在运
                             //行的中断处理函数处理完毕,但不能保证正在运行的 softirq(软
                             //中断)处理完毕
```

(4) 挂接回调

```
void fastcall call_rcu(struct rcu_head * head, void ( * func)(struct rcu_head * rcu));
```

```

//该函数由 RCU 写端调用,它不会使写执行单元阻塞,因而可以在中断
//上下文或软中断使用,该函数将把函数 func 挂接到 RCU 回调
//函数链上,然后立即返回
static inline void list_add_rcu(struct list_head *new, struct list_head *head);
//该函数把链表项 new 插入到 RCU 保护的链表 head 的开头.内存栅保证了
//在引用这个新插入的链表项之前,新链表项的链接指针的修改
//对所有读执行单元是可见的
static inline void list_add_tail_rcu(struct list_head *new, struct list_head *head);
//该函数类似于 list_add_rcu,它把新的链表项 new 添加到 RCU 保护的链
//表的末尾
static inline void list_del_rcu(struct list_head *entry); //该函数从 RCU 保护的链表中移走
//指定的链表项 entry
static inline void list_replace_rcu(struct list_head *old, struct list_head *new);
//用新的链表项 new 取代旧的链表项 old,内存栅保证在引用新的链表项
//之前,它的链接指针的修正对所有读执行单元可见

```

3) RCU 的特点

RCU 是一种改进的读写自旋锁,读执行单元没有任何同步开销,而写执行单元的同步开销取决于写执行单元间的同步机制。读执行单元不需要锁,不使用原子指令,而且在除 alpha 以外的所有架构上也不需要内存栅,因此不会导致锁竞争、内存延迟及流水线停滞等。因为不需要考虑死锁问题使得使用更便捷。

RCU 的优点是既允许多个读执行单元同时访问被保护的数据,也允许多个读执行单元和多个写执行单元同时访问被保护的数据。但 RCU 不能替代读写自旋锁,因为如果写比较多时,对读执行单元的性能提高不能弥补写执行单元导致的损失。

使用 RCU 时,写执行单元需要延迟数据结构的释放,复制被修改的数据结构,它必须使用某种锁机制同步并行其他写执行单元的修改操作。

4) RCU 在内核中的使用

RCU 已经在网络驱动层、网络核心层、IPC、dcache、内存设备层、软 RAID 层、系统调用审计和 SELinux 中使用。表 3.1 是 RCU 的 API 函数在 Linux 内核中使用情况汇总。

表 3.1 RCU 的 API 函数在 Linux 内核中使用情况

函 数 名	使用次数	函 数 名	使用次数
rcu_read_lock	96	call_rcu_bh	4
rcu_read_unlock	126	list API	69
rcu_read_lock_bh	8	synchronize_rcu	8
rcu_read_unlock_bh	15	rcu_dereference	39
call_rcu	18	合计	383

7. 信号量

1) 定义

Linux 中的信号量是一种睡眠锁,如果有一个任务试图获得已经被占用的信号量时,信号量会将其推进一个等待队列,让其睡眠,这时处理器能重获自由,从而去执行其他代码。当持有信号量的进程将信号量释放后,处于等待队列中的那个任务被唤醒,并获得该信号量。

2) 信号量 API 函数

(1) 定义和初始化信号量

```
DECLARE_MUTEX(sem);           //定义并初始化互斥信号量
struct semaphore * sem;       //定义信号量
sema_init(sem, val);          //初始化为计数信号量,初始值为 val
init_MUTEX(sem);              //初始化为互斥信号量
init_MUTEX_LOCKED(sem);       //初始化为锁状态的互斥信号量
```

(2) 获得信号量

```
void down(struct semaphore * sem); //尝试获取信号量 sem,如果 sem 值大于 0,则 sem
//值减 1,进程继续执行;否则进程进入不可中断的
//睡眠状态,等待信号量大于 0

int down_interruptible(struct semaphore * sem); //获取信号量 sem,如果 sem 值大于 0,
//则将 sem 的值减 1,进程继续执行;否则进程进入中
//断的睡眠状态,等待信号量大于 0;函数返回值用来
//区分是正常返回还是被信号中断,如果返回 0,表示
//获得信号量正常返回,如果被信号打断,返回 -EINTR

int down_killable(struct semaphore * sem); //获取信号量 sem,如果 sem 值大于 0,则
//将 sem 的值减 1,进程继续执行;否则进程置为
//TASK_KILLABLE 睡眠状态,等待信号量大于 0

int down_trylock(struct semaphore * sem); //尝试获取信号量 sem,如果成功则返回 0,
//否则返回非 0 值,不会导致调用者睡眠,可
//以在中断上下文使用
```

(3) 释放信号量

```
void up(struct semaphore * sem); //退出临界区时调用该函数释放信号量 sem,即把 sem
//值加 1,如果信号量上睡眠队列不为空,则唤醒其中
//一个等待进程
```

3) 信号量使用的一般形式

```
DECLARE_MUTEX(sem);           //定义并初始化信号量
down(&mount_sem);             //获取信号量
...
critical section              //临界区
```

```
up(&mount sem); //释放信号量
```

4) 信号量的特点

由于信号量会导致睡眠,所以不能用于中断上下文。并且使用 `down()` 进入睡眠的进程不能被信号打断。信号量允许任意多个执行单元持有该信号量(设置 `count` 大于 1),而自旋锁只允许最多一个任务持有。

8. 完成量

1) 定义

完成量是一种比信号量更好的同步机制,它用简单的方法使两个任务得以同步:一个执行单元等待另一个执行单元执行完某事。

2) 完成量 API 函数

(1) 定义和初始化完成量

```
DECLARE COMPLETION(my completion); //定义并初始化完成量
```

(2) 等待完成量

```
void wait_for_completion(struct completion *c); //等待一个 completion 被唤醒
```

(3) 唤醒完成量

```
void complete(struct completion *c);           //唤醒一个等待的执行单元
void complete_all(struct completion *c);      //释放所有等待同一完成量的执行单元
```

9. 读写信号量

1) 定义

读写信号量是一种相对放宽条件的粒度稍大的互斥机制。它允许多个读进程同时持有信号量,但最多只能有一个写进程。所有的读写信号量都是互斥信号量,只要没有写进程持有信号量,读进程就始终会持有该信号量,因此会造成写进程的饥饿状态。

2) 读写信号量 API 函数

(1) 定义和初始化读写信号量

```
DECLARE_RWSEM(rwsem);           //定义并初始化读写信号量
struct rw_semaphore rwsem;       //定义读写信号量
init_rwsem(&rwsem);              //初始化读写信号量
```

(2) 读信号量获取和释放

[illegible]

```
void up_read(struct rw_semaphore * sem); //读执行单元释放读写信号量 sem; 与 down_read 或
//down_read_trylock 配对使用, 如果 down_read_trylock
//返回 0, 则不需要调用 up_read 来释放读写信号量,
//因为根本就没有获得信号量
```

(3) 写信号量获取和释放

```
void down_write(struct rw_semaphore * sem); //写执行单元获取信号量 sem; 若未能获
//得, 进程进入不可中断的睡眠状态, 等待信号量大于
//0. 只能在进程上下文使用
int down_write_trylock(struct rw_semaphore * sem); //写执行单元尝试获取信号量 sem, 如果
//能够立即得到, 就获取该读写信号量, 并返回 1; 否则
//返回 0, 不会导致调用者睡眠. 可以在中断上下文使用
void up_write(struct rw_semaphore * sem); //写执行单元释放读写信号量 sem; 与
//down_write 或 down_write_trylock 配对使用, 如果
//down_write_trylock 返回 0, 则不需要调用 up_write
//来释放读写信号量, 因为根本就没有获得信号量。
```

3) 读写信号量使用的一般形式

```
static DECLARE_RWSEM(mr_rwsem);
...
down_read(&mr_rwsem); //获取读信号量
...
critical section //临界区(只读)
...
up_read(&mr_rwsem); //释放读信号量
...
down_write(&mr_rwsem); //获取写信号量
...
critical section //临界区(读和写)
...
up_write(&mr_rwsem); //获取写信号量
```

10. BKL(大内核锁)

1) 定义

BKL 是一个全局自旋锁, 用来方便实现从 Linux 最初的 SMP 过渡到细粒度加锁机制。

2) BKL 的 API 函数

```
lock_kernel(); //获得 BKL
unlock_kernel(); //释放 BKL
kernel_locked(); //如果锁被持有, 则返回非 0; 否则返回 0
```

3) BKL 的特点

持有 BKL 的任务仍然可以睡眠。BKL 是一种递归锁, 可以用在进程上下文中, BKL 已成为内核可扩展性的障碍, 在内核中不鼓励使用 BKL。事实上, 新代码中不再使用 BKL, 但是这种锁仍然在部分内核代码中沿用。

3.1.2 自旋锁与信号量的比较

Linux 内核中解决并发控制的最常用同步机制是自旋锁与信号量。自旋锁与信号量“类似而不类”，类似说的是它们功能上的相似性，不类指它们在本质和实现机理上完全不一样，不属于一类。

自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环查看该自旋锁的保持者是否已经释放了锁，“自旋”就是“在原地打转”。而信号量则引起调用者睡眠，它把进程从运行队列上拖出去，除非获得锁，这就是它们的“不类”。但是，无论是信号量，还是自旋锁，在任何时刻，最多只能有一个保持者，这就是它们的“类似”。

一般而言，自旋锁适合于保持时间非常短的情况，可以在任何上下文使用；信号量适合于保持时间较长的情况，只能在进程上下文使用。如果被保护的共享资源只在进程上下文访问，则可以用信号量；如果对共享资源的访问时间非常短，自旋锁也是好的选择。如果被保护的共享资源需要在中断上下文访问，就必须使用自旋锁。自旋锁与信号量区别总结如下：

(1) 由于争用信号量的进程在等待锁重新变为可用时会睡眠，所以信号量适用于锁会被长时间持有的情况。

(2) 相反，锁被短时间持有时，使用信号量就不太适宜，因为睡眠引起的耗时可能比锁被占用的全部时间还要长。

(3) 由于执行线程在锁被争用时睡眠，所以只能在进程上下文中才能获取信号量锁，因为在中断上下文中使用自旋锁是不能进行调度的。

(4) 可以在持有信号量时去睡眠，因为当其他进程试图获得同一信号量时不会因此而死锁，该进程也只是去睡眠而已，最终会继续执行的。

(5) 在占用信号量的同时不能占用自旋锁，因为在等待信号量时可能会睡眠，而在持有自旋锁时是不允许睡眠的。

(6) 信号量锁保护的临界区可包含可能引起阻塞的代码，而自旋锁要避免用来保护包含这样代码的临界区，因为阻塞意味着要进行进程的切换，如果进程被切换出去，另一进程企图获取本自旋锁，死锁就会发生。

(7) 信号量不同于自旋锁，它不会禁止内核抢占，所以持有信号量的代码可以被抢占，这意味着信号量不会对调度的等待时间带来负面影响。

3.2 make 及 makefile

为解决数目庞大的源文件编译问题，Linux 提供了“自动化”的编译工具 make，通过 make 命令可以便捷地解决编译问题，极大地提高软件开发效率。要执行 make 命令，还需要 makefile 文件的“指导”，makefile 文件中描述了如何去编译和链接文件。能否让 make 命令发挥最大的作用，就要看 makefile 文件是否交代清楚 make 需要完成的工作，首先需要

介绍的就是 makefile 文件。

3.2.1 makefile 文件

makefile 文件中主要包含显式规则、隐晦规则、变量定义、文件指示及注释 5 部分。显式规则解决了如何生成一个或多个目标文件的问题,开发者可以写明要生成的文件名称,所依赖的源文件及生成的命令。相比于显式规则,隐晦规则利用了 make 的自动推导功能,支持开发者简略书写 makefile 文件。makefile 文件需要定义一系列的变量,通常这些变量都为字符串,当 makefile 被执行时,文件内的变量都会被扩展到相应的引用位置上。文件指示包括 3 部分,第 1 是引用的其他 makefile 文件,可以类比于 C 语言中的 include; 第 2 是指根据某些指定 makefile 文件中的有效部分,类比于 C 语言中的预编译 #if; 最后 1 个是定义一个多行命令; makefile 文件中只有行注释,使用“#”字符,当需要使用“#”时需要在前面加上转义字符“\”,即“\#”。

通常情况下,makefile 文件的命名都是 Makefile,这样的命名并不是随意的。默认情况下,make 命令会在当前目录下按顺序找寻文件名为 makefile、Makefile 或 GNUmakefile 的文件,找到后解释文件执行命令。在这 3 个文件名中,最好使用 Makefile 文件名,因为这个文件名第 1 个字符为大写,有一种醒目的感觉。最好不要用 GNUmakefile,这个文件是 GNU 的 make 识别的。有一些 make 只对全小写的 makefile 文件名敏感,但是基本上,大多数的 make 都支持 makefile 和 Makefile 这两种默认文件名。

也可以使用别的文件名来书写 Makefile,例如 Make.Linux、Make.Solaris、Make.AIX 等,如果要指定特定的 Makefile,可以使用 make 的“-f”和“-file”参数,例如 make -f Make.Linux 或 make -file Make.AIX。

3.2.2 编写 makefile 文件

在 make 工具当中,需要做的大量工作就是书写一个完整健壮的 makefile 文件。而 makefile 是一个描述性文件,因此它要符合特定的书写规则。其一般形式如下:

```
targets :prerequisites
command
...
```

或是这样

```
targets : prerequisites ; command
command
...
```

targets 是文件名,以空格分开,可以使用通配符。一般来说,目标基本上是一个文件,但也有可能是多个文件。

command 是命令行,可以为任意的 Shell 命令。如果不与 target:prerequisites 在一行,

必须以[Tab]键开头,如果和 prerequisites 在一行,可以用分号作为分隔。

prerequisites 是目标所依赖的文件(或依赖目标)。如果其中的某个文件比目标文件新,那么,目标就被认为是“过时的”,需要重新生成。

如果命令太长,可以使用反斜杠“\”作为换行符。make 对每行有多少个字符没有限制。规则告诉 make 两件事,文件的依赖关系和如何生成目标文件。

一般,make 会以 UNIX 的标准 Shell,也就是/bin/sh 来执行命令。

1. 通配符的使用

如果想定义一系列比较类似的文件,很自然地就想起使用通配符。make 支持 3 种通配符:“*”,“?”和“[...]”。这和 UNIX 的 B-Shell 是相同的。

波浪号“~”字符在文件名中有比较特殊的用途。“~/test”,表示当前用户的 \$HOME 目录下的 test 目录,“~hchen/test”表示用户 hchen 的宿主目录下的 test 目录。

通配符代替了一系列的文件,例如“*.c”表示所有后缀为 c 的文件。需要注意的是,如果文件名中有通配符,例如“*”可以用转义字符“\”,“*”表示真实的“*”字符,而不是任意长度的字符串。

2. 文件搜寻

在一些大的工程中,有大量的源文件,通常的做法是把源文件分类存放在不同的目录中。所以,当 make 需要去找寻文件的依赖关系时,可以在文件前加上路径,但最好的方法是把一个路径告诉 make,让 make 自动去找。

Makefile 文件中的特殊变量 VPATH 就是完成这个功能的,如果没有指明这个变量,make 只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量,make 就会在当前目录找不到的情况下,到指定的目录中去找寻文件。

```
VPATH = src:../headers
```

上面的定义指定两个目录,“src”和“../headers”,make 会按照这个顺序进行搜索。目录由“冒号”分隔。当然,当前目录永远是最优先搜索的地方。

另一个设置文件搜索路径的方法是使用 make 的 vpath 关键字(注意,它是全小写的),它不是变量,是 make 的关键字,这和上面提到的那个 VPATH 变量很类似,但它更为灵活。它可以指定不同的文件在不同的搜索目录中。它的使用方法有 3 种:

(1) vpath <pattern><directories>: 为符合模式<pattern>的文件指定搜索目录<directories>。

(2) vpath <pattern>: 清除符合模式<pattern>的文件的搜索目录。

(3) vpath: 清除所有已被设置好了的文件搜索目录。

vpath 使用方法中的<pattern>需要包含“%”字符。“%”的意思是匹配零或若干字符,例如“%.h”表示所有以“.h”结尾的文件。<pattern>指定了要搜索的文件集,<directories>指定了<pattern>文件集的搜索目录。例如:

```
vpath %.h ../headers
```

该语句表示,要求 make 在“../headers”目录下搜索所有以“.h”结尾的文件(如果某文件在当前目录没有找到)。

可以连续地使用 vpath 语句,以指定不同搜索策略。如果连续的 vpath 语句中出现了相同的<pattern>,或是被重复了的<pattern>,make 会按照 vpath 语句的先后顺序来执行搜索。例如:

```
vpath % .c foo
vpath % blish
vpath % .c bar
```

其表示“.c”结尾的文件,先搜索 foo 目录,然后是 blish,最后是 bar 目录。

```
vpath % .c foo:bar
vpath % blish
```

上面的语句表示“.c”结尾的文件,先搜索 foo 目录,然后是 bar 目录,最后才是 blish 目录。

3. 伪目标

一般,makefile 中会提供一个名为 clean 的标签来提供清除编译过程中产生的中间文件及最终文件。例如:

```
clean:
rm *.o
```

只要执行 make clean 便能清除所有的.o 文件,但不生成 clean 这个文件,称这是一个“伪目标”或“标签”。

“伪目标”并不是一个文件,只是一个标签,由于“伪目标”不是文件,所以 make 无法生成它的依赖关系和决定它是否要执行。只有通过显示地指明这个“目标”才能让其生效。“伪目标”的取名不能和文件名重名,不然它就失去了“伪目标”的意义。为了避免和文件重名的情况,可以使用一个特殊的标记“.PHONY”来显示地指明一个目标是“伪目标”,并向 make 说明,不管是否有这个文件,这个目标就是“伪目标”。

```
.PHONY: clean
```

只要有这个声明,不管是否有 clean 文件,都要运行 clean 这个目标,只有 make clean 这样。于是整个过程可以这样写:

```
.PHONY: clean
clean:
rm *.o temp
```

“伪目标”一般没有依赖的文件,但可以为“伪目标”指定所依赖的文件。“伪目标”同样可以作为“默认目标”,只要将其放在第一个。例如,如果 Makefile 需要一口气生成若干个可执行文件,但你只想简单地输入 make 命令,并且所有的目标文件都写在一个 Makefile 中,那么可以使用“伪目标”这个特性。

```
all : prog1 prog2 prog3
.PHONY : all
prog1 : prog1.o utils.o
cc -o prog1 prog1.o utils.o
prog2 : prog2.o
cc -o prog2 prog2.o
prog3 : prog3.o sort.o utils.o
cc -o prog3 prog3.o sort.o utils.o
```

我们知道,Makefile 中的第 1 个目标会作为其默认目标。上面声明了一个 all 的“伪目标”,依赖于其他 3 个目标。由于“伪目标”的特性是,总是被执行,所以其依赖的那 3 个目标就总是不如 all 这个目标新。所以,其他 3 个目标的规则总是会被否决,也就达到了一口气生成多个目标的目的。“`.PHONY : all`”声明了 all 这个目标为“伪目标”。

从上面的例子可以看出,目标可以成为依赖。所以,“伪目标”同样也可成为依赖。看下面的例子:

```
.PHONY: cleanall cleanobj cleandiff
cleanall : cleanobj cleandiff
rm program
cleanobj :
rm *.o
cleandiff :
rm *.diff
```

make clean 将清除所有要被清除的文件。cleanobj 和 cleandiff 这两个“伪目标”有点像“子程序”。可以输入 make cleanall、make cleanobj 和 make cleandiff 命令来达到清除不同种类文件的目的。

4. 多目标

Makefile 规则中的目标可以不止一个,其支持多目标,有可能多个目标同时依赖于一个文件,并且生成的命令大体类似。于是就能把其合并起来。当然,多个目标生成规则的执行命令是同一个,这可能会带来麻烦,不过可以使用一个自动化变量“`$@`”(关于自动化变量,将在后面变量的使用中讲述),这个变量表示目前规则中所有目标的集合,如下所示:

```
bigoutput littleoutput : text.g
generate text.g - $(subst output, $@) > $@
```

上述规则等价于:

```
bigoutput : text.g
generate text.g - big > bigoutput
littleoutput : text.g
generate text.g - little > littleoutput
```

其中, `$(subst output, $@)` 中的“\$”表示执行一个 Makefile 的函数, 函数名为 `subst`, 后面的为参数。关于函数, 将在后面讲述。这里的这个函数是截取字符串的意思, “\$@”表示目标的集合, 像数组, “\$@”依次取出目标并执行命令。

3.2.3 make 命令

1. 显示命令

通常, `make` 会把要执行的命令行在执行前输出到屏幕上。当用“@”字符显示在命令行前, 这个命令将不被 `make` 显示出来。最具代表性的例子是, 用这个功能来向屏幕显示一些信息。例如:

```
@echo 正在编译 XXX 模块.....
```

当 `make` 执行时, 会输出“正在编译 XXX 模块.....”字符串, 但不会输出命令, 如果没有“@”, `make` 将输出“echo 正在编译 XXX 模块.....”。

如果 `make` 执行时, 带 `make` 参数“-n”或“-just-print”, 只是显示命令, 不会执行命令, 这个功能有利于调试 Makefile, 看看书写的命令执行起来是什么样子的或是什么顺序的。

`make` 参数“-s”或“-slient”则是全面禁止命令的显示。

2. 命令执行

当规则的目标需要更新时, `make` 会一条一条地执行其后的命令。需要注意的是, 如果要让上一条命令的结果应用于下一条命令, 应该使用分号分隔这两条命令。例如第 1 条命令是 `cd` 命令, 希望第 2 条命令在 `cd` 之后的基础上运行, 那么就不能把这两条命令写在两行上, 而应该把这两条命令写在一行上, 用分号分隔。例如:

示例一

```
exec:
cd /home/
cd /user
```

示例二

```
exec:
cd /home; cd /use
```

当执行“`make exec`”时, 示例一中的 `cd` 没有作用, `pwd` 会打印出当前的 Makefile 目录, 而示例二中, `cd` 就起作用了, `pwd` 会打印出“`/home/ user`”。

`make` 一般使用环境变量 `SHELL` 中所定义的系统 Shell 来执行命令, 默认情况下使用 UNIX 的标准 Shell——`/bin/sh` 来执行命令。`make` 会在 `SHELL` 所指定的路径中寻找命令解释器, 如果找不到, 其会在当前盘符中的当前目录中寻找, 如果再找不到, 其会在 `PATH` 环境变量中所定义的所有路径中寻找。

3. 命令出错

每当命令运行完, `make` 会检测每个命令的返回码, 如果命令返回成功, `make` 会执行下

一条命令,当规则中所有的命令成功返回,这个规则就成功完成了。如果一个规则中的某个命令出错(命令退出码非零),make 就会终止执行当前规则,这有可能终止所有规则的执行。

有些时候,命令的出错并不表示就是错误。例如 mkdir 命令,需要建立一个目录,如果目录不存在,mkdir 就成功执行,如果目录存在,就出错了。之所以使用 mkdir 就是确保要有这样的一个目录,那么就不希望 mkdir 出错而终止规则的运行。

为了做到这一点,忽略命令的出错,可以在 Makefile 的命令行前加一个减号“-”(在 Tab 键之后),标记为不管命令出不出错都认为是成功的。例如:

```
clean:
    - rm -f *.o
```

还有一个全局的办法,给 make 加上“-i”或“-ignore-errors”参数,Makefile 中的所有命令都会忽略错误。如果一个规则是以“. IGNORE”作为目标的,那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法,可以根据不同情况设置。

还有一个 make 的参数是“-k”或“-keep-going”,这个参数的意思是,如果某规则中的命令出错了,就终止该规则的执行,但继续执行其他规则。

4. 嵌套执行 make

在一些大的工程中,不同模块或是不同功能的源文件放在不同的目录中,可以在每个目录中都书写一个该目录的 Makefile,这有利于让 Makefile 变得更加简洁,而不至于把所有的东西全部写在一个 Makefile 中,这样会很难维护。这个技术对于模块编译和分段编译有着非常大的好处。

例如有一个子目录叫 subdir,这个目录下有个 Makefile 文件,来指明这个目录下文件的编译规则。那么总控的 Makefile 可以这样书写:

```
subsystem:
    cd subdir && $(MAKE)
```

其等价于

```
subsystem:
    $(MAKE) -C subdir
```

定义 \$(MAKE) 宏变量的意思是,也许 make 需要一些参数,所以定义成一个变量比较利于维护。这两个例子都是先进入“subdir”目录,然后执行 make 命令。

把这个 Makefile 叫做总控 Makefile,总控 Makefile 的变量可以传递到下级的 Makefile 中(如果显式地声明),但是不会覆盖下层 Makefile 中所定义的变量,除非指定了“-e”参数。

如果传递变量到下级 Makefile 中,可以使用这样的声明

```
export <variable ...>
```

如果不想让某些变量传递到下级 Makefile 中,可以这样声明

```
unexport <variable ...>
```

示例三

```
export variable = value
```

其等价于

```
variable = value
export variable
```

其等价于

```
export variable := value
```

其等价于

```
variable := value
export variable
```

示例四

```
export variable += value
```

其等价于

```
variable += value
export variable
```

如果要传递所有的变量,只要一个 `export` 就行了。后面什么也不跟,表示传递所有的变量。

需要注意的是,有两个变量,一个是 `SHELL`,另一个是 `MAKEFLAGS`,这两个变量不管是否 `export`,其总是要传递到下层 Makefile 中,特别是 `MAKEFILES` 变量,其中包含了 `make` 的参数信息,如果执行总控 Makefile 时有 `make` 参数或是在上层 Makefile 中定义了这个变量,那么 `MAKEFILES` 变量将会是这些参数,并会传递到下层 Makefile 中,这是一个系统级的环境变量。

但是 `make` 命令中有几个参数并不往下传递,它们是“-C”,“-f”,“-h”“-o”和“-W”(有关 Makefile 参数的细节将在后面说明),如果不想往下层传递参数,可以这样

```
subsystem:
    cd subdir && $(MAKE) MAKEFLAGS =
```

如果定义了环境变量 `MAKEFLAGS`,那么得确信其中的选项是大家都会用到的,如果有“-t”,“-n”,和“-q”参数,将会有让人意想不到的结果,或许会引起异常的恐慌。

还有一个在嵌套执行中比较有用的参数,“-w”或“--print-directory”会在 `make` 的过程中输出一些信息,可以看到目前的工作目录。例如下级 `make` 目录是“/home/hchen/gnu/make”,如果使用“`make-w`”来执行,那么当进入该目录时,会看到

```
make: Entering directory '/home/hchen/gnu/make'.
```

而在完成下层 make 后离开目录时,会看到

```
make: Leaving directory '/home/hchen/gnu/make'
```

当使用“-C”参数来指定 make 下层 Makefile 时,“-w”会被自动打开。如果参数中有“-s”(“-silent”)或“--no-print-directory”,“-w”总是失效的。

5. 定义命令包

如果 Makefile 中出现一些相同命令序列,可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以 define 开始,以 endef 结束。例如

```
define run-yacc
yacc $(firstword $^ )
mv y.tab.c $@
endef
```

这里,run-yacc 是这个命令包的名字,其不要和 Makefile 中的变量重名。在 define 和 endef 中的两行就是命令序列。这个命令包中的第 1 个命令是运行 yacc 程序,因为 yacc 程序总是生成“y.tab.c”文件,所以第 2 行的命令就是把文件改名字。还是把这个命令包放到一个示例中看看吧。

```
foo.c : foo.y
      $(run-yacc)
```

可以看见,要使用这个命令包,就好像使用变量一样。在这个命令包的使用中,命令包“run-yacc”中的“\$^”就是“foo.y”,“\$@”就是“foo.c”(有关这种以“\$”开头的特殊变量,会在后面介绍),make 在执行命令包时,命令包中的每个命令会被依次独立执行。

编写 Makefile 文件后,用户只需要在命令行下执行 make 命令,就可以找到当前目录的 makefile 文件执行,其过程由系统本身来完成,非常便捷,极大地提高了软件开发的效率。

3.3 调试方法

由于设备驱动程序运行于内核空间,因此有着与用户空间程序不同的调试方法。设备驱动程序的调试需要内核的支持,因此应该根据需要对内核进行重编译。本节将介绍调试方法中的 printk 及 /proc 系统,同时还将简要介绍调试器和相关的工具。

通过输入 make menuconfig 命令可以进入编译选项配置环境,选择 kernel hacking 选项进行环境配置。以下将介绍几种重要的配置选项说明。

(1) CONFIG_DEBUG_KERNEL: 这个选项使其他调试选项可用,但是它不激活任何的特性,相当于一个“开关”。

(2) CONFIG_DEBUG_SLAB: 这个选项很重要,打开了内核内存分配函数的几类检查。激活这些检查,就可能探测到一些内存覆盖和遗漏初始化的错误。当激活调试时,内核

还会在每个分配的内存对象的前后放置特别的守护值。如果这些值被改动,内核就知道已覆盖了一个内存分配区。

(3) CONFIG_DEBUG_PAGEALLOC: 满的页在释放时会从内核地址空间去除。这个选项明显拖慢系统,但是它能快速指出某些类型的内存损坏错误。

(4) CONFIG_DEBUG_SPINLOCK: 激活这个选项,内核捕捉对未初始化的自旋锁的操作,及各种其他的错误(例如 2 次解锁同 1 个锁)。

(5) CONFIG_DEBUG_SPINLOCK_SLEEP: 通常可不选。选中该选项会使系统检查到占有 spinlock 的进程进入 sleep 状态的错误,但是系统在进程只是有可能 sleep 的情况下也会发出警告,从而可能在正常情况下产生大量无用信息。

(6) CONFIG_INIT_DEBUG: 用 __init 或者 __initdata 标志的项在系统初始化或者模块加载后都被丢弃。这个选项激活了对代码的检查,这些代码试图在初始化完成后,存取初始化时内存。

(7) CONFIG_DEBUG_INFO: 这个选项使得内核在建立时包含完整的调试信息。使用 gdb 调试内核时,需要这些信息。

(8) CONFIG_DEBUG_DRIVER: 打开了驱动核心的调试信息,可以用来追踪底层支持代码的问题。

3.3.1 printk

printk 的功能与应用程序中的 printf 一样,不同之处在于 printk 可以在打印字符串前面加上内核定义的宏。例如: printk(KERN_ALERT "xx_drv initalized ok\n"); 语句的功能是: 当注册设备驱动时检查设备是否注册成功。KERN_ALERT 定义信息级别,通过使用不同的宏,可以定义需要打印的字符串的级别。有关的宏定义如下:

```
#define KERN_EMERG<0>"
/* 紧急事件消息,系统崩溃之前提示,表示系统不可用 */
#define KERN_ALERT<1>"
/* 报告消息,表示必须立即采取措施 */
#define KERN_CRIT<2>"
/* 临界条件,通常涉及严重的硬件或软件操作失败 */
#define KERN_ERR<3>"
/* 错误条件,驱动程序常用 KERN_ERR 来报告硬件的错误 */
#define KERN_WARNING<4>"
/* 警告条件,对可能出现问题的情况进行警告 */
#define KERN_NOTICE<5>"
/* 正常但又重要的条件,用于提醒. 常用于与安全相关的消息 */
#define KERN_INFO<6>"
/* 提示信息,如驱动程序启动时,打印硬件信息 */
#define KERN_DEBUG<7>"
/* 调试级别的消息值越小,级别越高 */
```

printk()能够通过为消息指定不同的级别(loglevel),对消息的重要性进行分类。没有

指定级别的 `printk` 语句的默认级别是 `DEFAULT_MESSAGE_LOGLEVEL`, 系统将为消息使用默认的 `DEFAULT_MESSAGE_LOGLEVEL` 值, 通常该值为 `KERN_WARNING` (即 4 级)。

在内核代码 `include/linux/kernel.h` 中, 定义了控制台的级别

```
extern int console_printk[];
#define console_loglevel (console_printk[0])
#define default_message_loglevel (console_printk[1])
#define minimum_console_loglevel (console_printk[2])
#define default_console_loglevel (console_printk[3])
```

当信息级别的数值小于控制台的级别时, `printk` 要打印的信息才会在终端打印出来, 否则不会显示在终端。

3.3.2 /proc 文件系统

`/proc` 是一种伪文件系统 (即虚拟文件系统), 存储的是当前内核运行状态的一系列特殊文件。用户可以通过这些文件查看有关系统硬件及当前正在运行进程的信息, 甚至可以通过更改其中某些文件来改变内核的运行状态。`/proc` 文件系统是动态的, 因此模块可以在任何时候添加或去除条目。模块可以通过此接口向用户空间传递一些调试信息。出于调试目的, 可以在驱动代码中增加向 `/proc` 文件系统导出有助于监视驱动的信息的代码。这样就可以通过查看 `/proc` 中的相关信息来监视和调试驱动。

所有使用 `/proc` 的模块应当包含 `<linux/proc_fs.h>` 头文件, 并定义正确的函数。`/proc` 下的每个文件都绑定到一个内核函数上, 当文件被读的时候就会即时产生文件内容。例如 `cat /proc/modules`, 将返回当前已加载的模块列表信息。具体如下:

(1) `read_proc` 方法创建一个只读 `/proc` 文件, 驱动程序必须实现一个当文件被读取时产生数据的函数, 当某个进程读取这个文件时 (使用 `read` 系统调用), 请求通过这个函数到达模块。

```
int (* read_proc)(char * page, char ** start, off_t offset, int count, int * eof, void * data);
```

当一个进程读 `/proc` 文件时, 内核分配了一页内存 (`PAGE_SIZE` 字节), 驱动可以写入数据返回给用户空间。只要实现上面那个函数就可以了。

(2) 使用 `seq_file` 接口创建 `/proc` 文件, 适用于 `/proc` 方法输出数据数量变大时的场合。`seq_file` 接口通过提供简单的一套函数来实现大内核虚拟文件。为使用 `seq_file`, 必须创建一个简单的 `iterator` 对象, 它能在序列里建立一个位置, 向前进, 并且输出序列里的一个项。`iterator` 对象指的是一种迭代器 (Iterator) 模式, 提供一种方法访问容器 (container) 对象中各个元素, 而又不需暴露该对象的内部细节。具体函数为

```
void * start(struct seq_file * sfile, loff_t * pos); //start 方法首先调用
void * next(struct seq_file * sfile, void * v, loff_t * pos);
//next 函数应当移动 iterator 到下一
```

```

//个位置,如果序列里为空就返回 NULL
void stop(struct seq_file * sfile, void * v); //内核处理完 iterator,调用 stop 来清理
int show(struct seq_file * sfile, void * v);
//内核调用 show 方法来输出有用的东西给用户空间.

```

用作 seq_file 输出的函数还有:

```

int seq_printf(struct seq_file * sfile, const char * fmt, ...);
int seq_putc(struct seq_file * sfile, char c);

```

其实 seq_file 接口和 file_operations 结构类似,在使用 seq_file 接口创建文件时,意味着类似创建一个 file_operations 结构来实现所有内核需要的操作,处理文件上的读和移动。实现这些函数,就可以进行驱动调试了。

3.3.3 调试器及相关工具

调试模块的最后手段是使用调试器来单步调试代码,查看变量值和机器寄存器。这个方法费时,应当避免使用。但是,通过调试器获得的代码的细粒度视角,有时是很有价值的。

在内核上使用一个交互式调试器是一个挑战。内核代表系统中的所有进程运行在自己的地址空间。结果,用户空间调试器所提供的一些普通功能,例如断点和单步,在内核中更难得到。

1. gdb

gdb 对于看系统内部是非常有用的。在这个级别需要精通调试器的使用,要求对 gdb 命令有深刻学习,需要理解目标平台的汇编代码,以及对应源码和优化汇编码的能力。

(1) 方法 1: 调试器把内核作为一个应用程序来调用,一个典型的 gdb 调用如下:

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

注意 需要指定内核映象的文件名;在命令行提供一个核心文件的名字。

第 1 个参数是非压缩的 ELF 内核可执行文件,不是 zImage 或者 bzImage。第 2 个参数是核心文件的名字。对于一个运行中的内核,核心文件是内核核心/proc/kcore。由于模块并没有作为 vmlinux 的一部分传给 gdb,因此必须通过 add-symbol-file 命令把模块信息传递给 gdb,最后通过 p 命令查看模块的变量和结构。

(2) 方法 2: gdb print *(address)填充。

address 指向一个 16 进制地址,输出是对应那个地址的代码的文件和行号。例如,找出一个函数指针到底指向什么函数。

2. kdb 内核调试器

kdb 是一款功能强大的内核调试软件,使用 kdb 可以像调试用户空间进程那样设置断点、查看变量值等,其具体使用方法如下:

(1) 首先安装 kdb 补丁才能使用 kdb。

(2) 在内核配置 kernel hacking 选中支持内核调试器选项。

built-in Kernel Debugger support

(3) 在控制台上单击 Pause(或者 Break)键,进入 kdb 调试器。

(4) 调试命令如下:

bp: 设置断点;

bt: 查找函数调用堆栈;

cp: 跳到函数入口;

mm: 修改数据。

3. Linux 追踪工具

Linux Trace Toolkit (LTT, Linux 追踪工具)是一个内核补丁及一套相关工具,允许追踪内核中的事件。这个追踪包括时间信息,可以创建一个给定时间段内发生事情的合理的完整图像。因此,它不仅用来调试也可以追踪性能问题。

4. 动态探针

动态探针(Dynamic Probes, DProbes)是由 IBM 发行、IA-32 体系的 Linux 调试工具,它是一个开源项目,其原理如下:

允许在系统中几乎任何地方安放一个“探针”,用户空间和内核空间都可以。当执行到指定位置时,这个代码可以报告信息给用户空间,改变寄存器,或者做其他很多事情。DProbes 可以联合 LTT,在任意位置插入一个新的跟踪事件。

在 Linux 内核驱动中,所有的设备都是以文件的概念出现的。这样所有的读写操作等就会变得十分方便,操作系统内核对外提供了相应的数据结构,可以方便地进行填充调用。

Linux 下的设备驱动程序被组织为一组完成不同任务的函数的集合,通过这些函数使得 Windows 的设备操作犹如文件一般。在应用程序看来,硬件设备只是一个设备文件,应用程序可以像操作普通文件一样对硬件设备进行操作,如 `open()`、`close()`、`read()`、`write()` 等。

Linux 主要将设备分为两类:字符设备和块设备。字符设备是指那些只能一个字节一个字节读写数据的设备,不能随机读取设备内存中的某一数据。其读取数据需要按照先后顺序,从这点来看,字符设备是面向数据流的设备。常见的字符设备有鼠标、键盘、串口、控制台和 LED 等。块设备是指那些可以从设备的任意位置读取一定长度数据的设备。其读取数据不必按照先后顺序,可以定位到设备的某一具体位置读取数据。常见的块设备有硬盘、磁盘、U 盘、SD 卡等。

任何设备都有一个主设备号(主码)和一个次设备号(从码),主设备号和次设备号统称为设备号。主设备号用来表示一个特定的驱动程序。次设备号用来表示使用该驱动程序的设备。例如,一个嵌入式系统有两个 LED 指示灯,LED 灯需要独立的打开或者关闭。那么,可以写一个 LED 灯的字符设备驱动程序,将其主设备号注册成 5 号设备,次设备号分别为 1 和 2。这里,次设备号就分别表示两个 LED 灯。

内核模块同进程对话有两种主要途径。一种是通过设备文件(例如 `/dev` 目录中的文件),另一种是使用 `/proc` 文件系统。把一些东西写入内核的一个主要原因就是支持一些硬件设备,所以这里从设备文件开始。

设备文件的最初目的是允许进程同内核中的设备驱动通信,并且通过它们和物理设备(modem,终端等)通信。这种方法的实现如下:每个设备驱动都对应着一定类型的硬件设备,并且被赋予一个主码。设备驱动的列表和它们的主码可以在 `/proc/devices` 中找到。每个设备驱动管理下的物理设备也被赋予一个从码。无论这些设备是否真的安装,在 `/dev` 目录中都有一个文件,称作设备文件,对应着每一个设备。例如,如果进行 `ls-l /dev/hd[ab]*` 操作,将看见可能连接到某台机器上的所有 IDE 硬盘分区。注意它们都使用了同一个主

码,但是从码却互不相同。(声明:这是在 PC 结构上的情况,不清楚在其他结构上运行的 Linux 是否如此。)

在系统安装时,所有设备文件在 `mknod` 命令下创建。它们必须创建在 `/dev` 目录下,没有技术上的原因,只是一种使用上的便利。如果是为测试而创建的设备文件,比如这里的练习,可能放在编译内核模块的目录下更加合适。

字符设备和块设备的区别是块设备有一个用于请求的缓冲区,所以它们可以选择用什么样的顺序来响应它们,这对于存储设备是非常重要的,读取相邻的扇区比互相远离的分区速度会快得多;另一个区别是块设备只能按块(块大小对应不同设备而变化)接受输入和返回输出,而字符设备却按照它们能接受的最少字节块来接受输入。大部分设备是字符设备,因为它们不需要这种类型的缓冲。可以通过 `ls -l` 命令查看输出中的第一个字符而知道一个设备文件是块设备还是字符设备,如果是 `b` 就是块设备,如果是 `c` 就是字符设备。

模块可以分成两部分:模块部分和设备及设备驱动部分。`init_module` 函数调用 `module_register_chrdev` 在内核的块设备表里注册设备驱动,同时返回该驱动所使用的主码。`cleanup_module` 函数撤销设备的注册。

注册和注销是这两个函数的主要功能。内核中的函数不像进程一样自发运行,而是通过系统调用、硬件中断或内核中的其他部分(只要是调用具体的函数)被进程调用。所以当向内核中增加代码时,应该把它注册为具体某种事件的句柄,而当把它删除的时候,需要注销这个句柄。

另一点需要记住的是,不允许管理员随心所欲地删除内核模块。这是因为如果设备文件是被进程打开的,那么删除内核模块的时候,要使用这些文件就会导致访问正常的函数(读/写)所在的内存位置。如果幸运,那里不会有其他代码被装载,将得到一个恶性的错误信息。如果不幸,另一个内核模块会被装载到同一个位置,这意味着会跳入内核中另一个程序中,结果将是不可预料的错误。通常不希望一个函数做什么事情的时候,会从函数返回一个错误码(一个负数)。但这在 `cleanup_module` 中是不可能的,因为它是一个 `void` 型的函数。一旦 `cleanup_module` 被调用,这个模块就死掉了。然而有一个计数器记录着有多少个内核模块在使用这个模块,这个计数器称为索引计数器(`/proc/modules` 中每行的最后一个数字)。如果这个数字不是 0,删除就会失败。模块的索引计数器包含在变量 `mod_use_count` 中。有定义好的处理这个变量的宏(`MOD_INC_USE_COUNT` 和 `MOD_DEC_USE_COUNT`),所以一般使用宏而不是直接使用变量 `mod_use_count`,在以后实现变化的时候会带来安全性。

Linux 内核字符设备驱动需要利用的函数接口主要利用了 4.1 节提到的数据结构,内核向外暴露了读写文件的接口,需要做的就是实现这些接口定义的读写函数。除此之外内核还提供了很多字符设备需要用到的 API,我们也会对其源码进行剖析。

4.1 关键数据结构

开发 Linux 字符驱动设备利用的关键数据结构如下：

```
struct file_operations {
    struct module * owner;
    loff_t (* llseek) (struct file *, loff_t, int);
    ssize_t (* read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (* aio_read) (struct kiocb *, char __user *, size_t, loff_t);
    ssize_t (* write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (* aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
    int (* readdir) (struct file *, void *, filldir_t);
    unsigned int (* poll) (struct file *, struct poll_table_struct *);
    int (* ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (* mmap) (struct file *, struct vm_area_struct *);
    int (* open) (struct inode *, struct file *);
    int (* flush) (struct file *);
    int (* release) (struct inode *, struct file *);
    int (* fsync) (struct file *, struct dentry *, int datasync);
    int (* aio_fsync) (struct kiocb *, int datasync);
    int (* fasync) (int, struct file *, int);
    int (* lock) (struct file *, int, struct file_lock *);
    ssize_t (* readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (* writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (* sendfile) (struct file *, loff_t *, size_t, read_actor_t, void __user *);
    ssize_t (* sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (* get_unmapped_area) (struct file *, unsigned long,
        unsigned long, unsigned long,
        unsigned long);
};
```

本结构体的具体说明如下：

```
struct module * owner
```

第一个 file_operations 成员根本不是一个操作，而是一个指向拥有这个结构的模块的指针。这个成员用来在它的操作还在被使用时，阻止模块被卸载。几乎所有时间中，它简单初始化为 THIS_MODULE(在<linux/module.h>中定义的宏)。

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

llseek 方法用作改变文件中的当前读/写位置，并且新位置作为返回值(且为正)。loff_t 参数是一个 long offset，并且即使在 32 位平台上也至少有 64 位宽。错误由一个负返回值指示。如果这个函数指针是 NULL，seek 调用会以潜在的无法预知的方式修改 file 结构中的位置计数器。

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

用来从设备中获取数据。在这个位置的一个空指针导致 read 系统调用以-EINVAL ("Invalid argument")失败。一个非负返回值代表了成功读取的字节数(返回值是一个 signed size 类型,常常是目标平台本地的整数类型)。

```
ssize_t (* aio_read)(struct kiocb *, char __user *, size_t, loff_t);
```

初始化一个异步读(可能在函数返回前不结束的读操作)。如果这个方法是 NULL,所有的操作会由 read 同步地代替进行。

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

发送数据给设备。如果为 NULL,-EINVAL 会返回给调用 write 系统调用的程序。如果非负,返回值代表成功写入的字节数。

```
ssize_t (* aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);
```

初始化设备上的一个异步写。

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

对于设备文件这个成员应当为 NULL。它用来读取目录,并且仅对文件系统有用。

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

poll 方法是 3 个系统调用的后端: poll、epoll 和 select,都用做查询对一个或多个文件描述符的读或写是否会阻塞。poll 方法应当返回一个位掩码指示是否非阻塞的读或写是可能的,并且给内核提供信息用来使调用进程睡眠直到 I/O 变为可能。如果一个驱动的 poll 方法为 NULL,设备假定为不阻塞地可读可写。

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

ioctl 系统调用提供了发出设备特定命令的方法(例如格式化软盘的一个磁道,这不是读也不是写)。另外,几个 ioctl 命令被内核识别而不必引用 fops 表。如果设备不提供 ioctl 方法,对于任何未事先定义的请求(-ENOTTY, "设备无这样的 ioctl"),系统调用返回一个错误。

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

mmap 用来请求将设备内存映射到进程的地址空间。如果这个方法是 NULL,mmap 系统调用返回-ENODEV。

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

这常常是对设备文件进行的第一个操作,不要求驱动声明一个对应的方法。如果这个项是 NULL,设备打开一直成功,但是驱动不会得到通知。

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

flush 操作在进程关闭它的设备文件描述符的复制时调用。它应当执行(并且等待)设备的任何未完成的的操作。这个务必不要和用户查询请求的 fsync 操作混淆。当前,flush 只在很少驱动中使用,SCSI 磁带驱动使用了该函数。为确保所有写的数据在设备关闭前写到磁带上。如果 flush 为 NULL,内核简单地忽略用户应用程序的请求。

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

在文件结构释放时引用这个操作。和 open 操作一样,release 可以为 NULL。

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

这个方法是 fsync 系统调用的后端,用户调用来刷新任何挂着的数据。如果这个指针是 NULL,系统调用返回 -EINVAL。

```
int (*aio_fsync) (struct kiocb *, int);
```

这是 fsync 方法的异步版本。

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

这个操作用来通知设备它的 FASYNC 标志的改变。异步通知是一个高级的主题,如果驱动不支持异步通知,这个成员可以是 NULL。

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

lock 方法用来实现文件加锁。加锁对常规文件是必不可少的特性,但是设备驱动几乎从不实现它。

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

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

这些方法实现发散/汇聚读和写操作。应用程序偶尔需要做一个包含多个内存区的单个读或写操作,这些系统调用允许它们这样做而不必对数据进行额外复制。如果这些函数指针为 NULL,read 和 write 方法被调用(可能多于一次)。

```
ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
```

这个方法实现 sendfile 系统调用的读,它从一个文件描述符代表的文件中的数据转移到另一个文件时复制次数最少。例如,当被一个需要发送文件内容到一个网络连接的 Web 服务器使用。设备驱动常常使 sendfile 为 NULL。

```
ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
```

sendpage 是 sendfile 的另一半,它由内核调用来发送数据。

```
unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long, unsigned
```



```
long, unsigned long);
```

这个方法的目的是在进程的地址空间找一个合适的位置来映射在底层设备上的内存段。这个任务通常由内存管理代码进行。这个方法存在为了使驱动能强制特殊设备可能有的任何的对齐请求。大部分驱动可以置这个方法为 NULL。

```
int (*check_flags)(int)
```

这个方法允许模块检查传递给 fcntl(F_SETFL...)调用的标志。

```
int (*dir_notify)(struct file *, unsigned long);
```

这个方法在应用程序使用 fcntl 来请求目录改变通知时调用。只对文件系统有用,驱动不需要实现 dir_notify。

scull 设备驱动只实现最重要的设备方法。它的 file_operations 结构的初始化如下:

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

4.2 接口函数部分内核代码分析

在内核中, <linux/types.h> 中定义 dev_t 的类型用来持有设备编号(包括主次部分)。其中 dev_t 是 32 位的量,12 位用作主编号,20 位用作次编号。

```
#ifndef _LINUX_TYPES_H
#define _LINUX_TYPES_H
#define __EXPORTED_HEADERS__
#include <uapi/linux/types.h>
#ifndef __ASSEMBLY__
#define DECLARE_BITMAP(name, bits) \
    unsigned long name[BITS_TO_LONGS(bits)]
typedef __u32 __kernel_dev_t;
typedef __kernel_fd_set      fd_set;
typedef __kernel_dev_t      dev_t;
typedef __kernel_ino_t      ino_t;
typedef __kernel_mode_t      mode_t;
typedef unsigned short      umode_t;
typedef __u32                nlink_t;
```

```

typedef __kernel_off_t      off_t;
typedef __kernel_pid_t      pid_t;
typedef __kernel_daddr_t    daddr_t;
typedef __kernel_key_t      key_t;
typedef __kernel_suseconds_t suseconds_t;
typedef __kernel_timer_t     timer_t;
typedef __kernel_clockid_t   clockid_t;
typedef __kernel_mqd_t       mqd_t;

```

在<linux/kdev_t.h>中的一套宏定义,为获得一个 dev_t 的主或者次编号,使用设备编号的内部表示:

```

MAJOR(dev_t dev);
MINOR(dev_t dev);

```

在有主次编号时,需要将其转换为一个 dev_t,可使用以下函数:

```

MKDEV(int major, int minor);

```

在 linux/kdev_t.h 中有以下内容:

```

#define MINORBITS 20
#define MINORMASK ((1U << MINORBITS) - 1)
#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
//高 12 位表示主设备号,低 20 位表示次设备号

```

分配和释放设备编号的函数为 register_chrdev_region(),该函数位于 fs/char_dev.c 文件中,以下是其源代码及注释。

```

184/ * *
185 * register_chrdev_region() - register a range of device numbers
186 * @from: the first in the desired range of device numbers; must include
187 *       the major number.
188 * @count: the number of consecutive device numbers required
189 * @name: the name of the device or driver.
190 *
191 * Return value is zero on success, a negative error code on failure.
192 */
193 int register_chrdev_region(dev_t from, unsigned count, const char * name)
194 {
195     struct char_device_struct * cd;
196     dev_t to = from + count;
197     dev_t n, next;
198
199     for (n = from; n < to; n = next) { /* 每次申请 256 个设备号 */

```

```

200     next = MKDEV(MAJOR(n) + 1, 0);    /* 主设备号加 1 得到的设备号, 次设备号为 0 */
201     if (next > to)
202         next = to;
203     cd = __register_chrdev_region(MAJOR(n), MINOR(n),
204         next - n, name);
205     if (IS_ERR(cd))
206         goto fail;
207 }
208 return 0;
209 fail: /* 当一次分配失败的时候, 释放所有已经分配到的设备号 */
210 to = n;
211 for (n = from; n < to; n = next) {
212     next = MKDEV(MAJOR(n) + 1, 0);
213     kfree(__unregister_chrdev_region(MAJOR(n), MINOR(n), next - n));
214 }
215 return PTR_ERR(cd);
216}

```

from 是要分配的起始设备编号, from 的次编号部分一般是 0。count 是请求的连续设备编号的总数。注意, 如果 count 太大, 要求的范围可能溢出到下一个次编号; 但是只要要求的编号范围可用, 一切都会正确工作。最后, name 是应当连接到这个编号范围的设备名字; 它会出现于 /proc/devices 和 sysfs 中。如同大部分内核函数, 如果分配成功, register_chrdev_region 的返回值是 0。出错的情况下, 返回一个负的错误码, 不能存取请求的区域。

下面是 char_device_struct 结构体的信息, 该结构体位于 fs/char_dev.c 文件中。

```

static struct char_device_struct {
    struct char_device_struct * next;    //指向散列冲突链表中的下一个元素的指针
    unsigned int major;                 //主设备号
    unsigned int baseminor;             //起始次设备号
    int minorct;                        //设备编号的范围大小
    const char * name;                  //处理该设备编号范围内的设备驱动的名称
    struct file_operations * fops;      //没有使用
    struct cdev * cdev;                 /* will die 指向字符设备驱动程序描述符的指针 */
} * chrdevs[MAX_PROBE_HASH];

```

__register_chrdev_region 函数主要是注册主设备号和次设备号, major == 0 此函数动态分配主设备号, major > 0 申请分配指定的主设备, 返回 0 表示申请成功, 返回负数表示申请失败。

```

91 static struct char_device_struct *
92 __register_chrdev_region(unsigned int major, unsigned int baseminor,
93     int minorct, const char * name)
94 {
95     /* 以下处理 char_device_struct 变量的初始化和注册 */
96     struct char_device_struct * cd, ** cp;
97     int ret = 0;

```

```

97     int i;
98     //kzalloc()分配内存并且全部初始化为 0
99     cd = kzalloc(sizeof(struct char_device_struct), GFP_KERNEL);
100    if (cd == NULL)
101        //ENOMEM 定义在 include/asm-generic/error-base.h 中
102        //15 #define ENOMEM      12      /* Out of memory */

101    return ERR_PTR(-ENOMEM);
102
103    mutex_lock(&chrdevs_lock);
104
105    /* temporary */
106    if (major == 0) { //下面动态申请主设备号
107        for (i = ARRAY_SIZE(chrdevs) - 1; i > 0; i--) {
108            //ARRAY_SIZE 定义为 ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))
109            // #define ARRAY_SIZE(a) (sizeof(a) / sizeof((a)[0]))
110            if (chrdevs[i] == NULL)
111                //chrdevs 是内核中已经注册设备号的一个数组
112                break;
113        }
114
115        if (i == 0) {
116            ret = -EBUSY;
117            goto out;
118        }
119        major = i;
120        ret = major; //这里得到一个未使用的设备号
121    }
122    //下面 4 句是对已经申请到的设备数据结构进行填充
123    cd->major = major;
124    cd->baseminor = baseminor;
125    cd->minorct = minorct; //申请设备号的个数 */
126    strcpy(cd->name, name, sizeof(cd->name));
127    /* 以下部分将 char_device_struct 变量注册到内核 */
128    i = major_to_index(major);
129
130    for(cp = &chrdevs[i]; *cp; cp = &(*cp)->next)
131        if(( *cp)->major > major || //chardevs[i]设备号大于主设备号
132            (( *cp)->major == major &&
133            ((( *cp)->baseminor >= baseminor) || //chardevs[i]设备号等于主设备号,
134                //并且此设备号大于 baseminor
135            (( *cp)->baseminor + ( *cp)->minorct > baseminor))))
136            break;
137    //在字符设备数组中找到现在注册的设备
138    /* Check for overlapping minor ranges. */
139    if( *cp && ( *cp)->major == major) {
140        int old_min = ( *cp)->baseminor;
141        int old_max = ( *cp)->baseminor + ( *cp)->minorct - 1;
142        int new_min = baseminor;
143        int new_max = baseminor + minorct - 1;

```

```

140
141     /* New driver overlaps from the left. */
142     if(new_max >= old_min && new_max <= old_max) {
143         ret = -EBUSY;
144         goto out;
145     }
146
147     /* New driver overlaps from the right. */
148     if(new_min <= old_max && new_min >= old_min) {
149         ret = -EBUSY;
150         goto out;
151     }
152 }
153 /* 所申请的设备号能够满足 */
154 cd->next = *cp; /* 按照主设备号从小到大顺序排列 */
155 *cp = cd;
156 mutex_unlock(&chrdevs_lock);
157 return cd;
158 out:
159 mutex_unlock(&chrdevs_lock);
160 kfree(cd);
161 return ERR_PTR(ret);
162 }

```

以上程序大体上分为两个步骤：

(1) char_device_struct 类型变量的分配及初始化,第 94~123 行。

(2) 将 char_device_struct 变量注册到内核,第 124~162 行。

1. char_device_struct 类型变量的分配及初始化

(1) 首先,调用 kmalloc 分配一个 char_device_struct 变量 cd。检查返回值,进行错误处理。

(2) 将分配的 char_device_struct 变量的内存区清零 memset。

(3) 获取 chrdevs_lock 读写锁,并且关闭中断,禁止内核抢占,write_lock_irq。

(4) 如果传入的主设备号 major 不为 0,跳转到第(7)步。

(5) 主设备号 major 为 0,首先需要分配一个合适的主设备号。将 i 赋值成 ARRAY_SIZE(chrdevs)-1,其中的 chrdevs 是包含有 256 个 char_device_struct * 类型的数组,然后递减 i 的值,直到在 chrdevs 数组中出现 NULL。当 chrdevs 数组中不存在空值的时候,ret=-EBUSY;goto out;。

(6) 到达这里,表明主设备号 major 已经有合法的值了,接着进行 char_device_struct 变量的初始化。设置 major、baseminor、minorct 及 name。

2. 将 char_device_struct 变量注册到内核

(1) 将 i 赋值成 major_to_index(major),将 major 对 256 取余数,得到可以存放 char_device_struct 在 chrdevs 中的索引。

(2) 进入循环,在 chrdevs[i] 的链表中找到一个合适位置。退出循环的条件为

① chrdevs[i] 为空。

② chrdevs[i]的主设备号大于 major。

③ chrdevs[i]的主设备号等于 major,但是次设备号大于等于 baseminor。

注意 `cp = &(*cp)->next`, `cp` 是 `char_device_struct **` 类型, `(*cp)->next` 是一个 `char_device_struct *` 类型, 所以 `&(*cp)->next`, 就得到一个 `char_device_struct **`, 并且这时候由于是指针, 所以对 `cp` 赋值, 就相当于对链表中元素的 `next` 字段进行操作。

(3) 进行冲突检查, 因为退出循环的情况可能造成设备号冲突(产生交集)。如果 `*cp` 不空, 并且 `*cp` 的 `major` 与要申请的 `major` 相同, 此时, 如果 `(*cp)->baseminor < baseminor + minorct`, 就会发生冲突, 因为和已经分配的设备号冲突了。出错就跳转到 `ret=-EBUSY; goto out;`。

(4) 到这里, 内核可以满足设备号的申请, 将 `cd` 链接到链表中。

(5) 释放 `chrdevs_lock` 读写锁, 开中断, 开内核抢占。

(6) 返回加入链表的 `char_device_struct` 变量 `cd`。

(7) `out` 出错退出

① 释放 `chrdevs_lock` 读写锁, 开中断, 开内核抢占。

② 释放 `char_device_struct` 变量 `cd`, `kfree`。

③ 返回错误信息。

动态申请设备号的函数实现在 `fs/char_dev.c` 文件中, 以下是源代码及注释。

```
218/ * *
219 * alloc_chrdev_region() - register a range of char device numbers
220 * @dev: output parameter for first assigned number
221 * @baseminor: first of the requested range of minor numbers
222 * @count: the number of minor numbers required
223 * @name: the name of the associated device or driver
224 *
225 * Allocates a range of char device numbers. The major number will be
226 * chosen dynamically, and returned (along with the first minor number)
227 * in @dev. Returns zero or a negative error code.
228 * /
229int alloc_chrdev_region(dev_t * dev, unsigned baseminor, unsigned count,
230                        const char * name)
231 {
232     /* dev:
233        仅作为输出参数, 成功分配后将保存已分配的第一个设备编号
234        baseminor:
235        被请求的第一个次设备号, 通常是 0
236        count:
237        所要分配的设备号的个数
238        name:
239        和所分配的设备号范围相对应的设备名称
```

返回值:

成功返回 0, 失败返回负的错误编码

```

*/
232 struct char_device_struct * cd;
233 cd = __register_chrdev_region(0, baseminor, count, name);
234 if (IS_ERR(cd))
235     return PTR_ERR(cd);
236 * dev = MKDEV(cd->major, cd->baseminor);
237 return 0;
238 }

```

4.3 字符设备驱动设计

4.3.1 字符设备驱动设计场景描述

本驱动程序目标是在内存中模拟一个设备, 利用填充 file_operation 结构体, 实现对于本设备的字符读写。首先进行程序模块的初始化, 之后编译加入到内核空间。在加入之后, 对加入到 dev 中的设备进行读写操作, 从而测试其是否可行。测试方法, 用 fopen 函数打开 dev 下的设备文件, 并且对其中的内容进行读写和位移之后读写, 将读写的内容进行打印对比。之后将内核空间的输出内容进行打印来验证是否顺利实现。

4.3.2 字符设备驱动设计过程

(1) 宏与变量声明, 如图 4.1 所示。

```

static int __init driver_init_module(void);
static void __exit driver_exit_module(void);
module_init(driver_init_module);
module_exit(driver_exit_module);

```

图 4.1 宏与变量声明

(2) 声明文件结构体中所用的函数, 定义设备驱动文件结构体, 如图 4.2 所示。

```

static int mem_open(struct inode *ind, struct file *filp);
static int mem_release(struct inode *ind, struct file *filp);
static ssize_t mem_read(struct file *filp, char __user *buf, size_t size, loff_t *fpos);
static ssize_t mem_write(struct file *filp, const char __user *buf, size_t size, loff_t *fpos);
static loff_t mem_llseek(struct file *filp, loff_t offset, int whence);

struct file_operations mem_fops =
{
    .open = mem_open,
    .release = mem_release,
    .read = mem_read,
    .write = mem_write,
    .llseek = mem_llseek,
};

```

图 4.2 定义设备驱动文件结构体

(3) 模块初始化函数,如图 4.3 所示。

```

72 //建立一个系统设备类
73 mem_class = class_create(THIS_MODULE, "myalloc");
74 if(IS_ERR(mem_class))
75 {
76     printk(KERN_INFO "Error: failed in creating class!\n");
77     return -1;
78 }
79
80 //注册设备文件系统,并建立设备节点
81 device_create(mem_class, NULL, MKDEV(MEM_MAJOR, 0), NULL, "myalloc");
82 return 0;
83

```

(a)

```

84
85 void _exit_driver_exit_module(void)
86 {
87     if(mem_cdev != NULL)
88         cdev_del(mem_cdev);
89     printk(KERN_INFO "cdev del ok!\n");
90     device_destroy(mem_class, MKDEV(MEM_MAJOR, 0));
91     class_destroy(mem_class);
92     if(mem_spvm != NULL)
93         vfree(mem_spvm);
94     printk(KERN_INFO "vfree ok!\n");
95 }
96

```

(b)

```

36 int _init_driver_init_module(void)
37 {
38     int res;
39     int devno = MKDEV(MEM_MAJOR, 0);
40     mem_spvm = (char*)vmalloc(MEM_MALLOCC_SIZE);
41     if(mem_spvm == NULL)
42         printk(KERN_INFO "vmalloc failed!\n");
43     else
44         printk(KERN_INFO "Success: mem-spvm ok!\n", (unsigned int)mem_spvm);
45
46     //动态分配一个新的字符设备对象
47     mem_cdev = cdev_alloc();
48     if(mem_cdev == NULL)
49     {
50         printk(KERN_INFO "cdev alloc failed!\n");
51         return 0;
52     }
53
54     //初始化字符设备对象
55     cdev_init(mem_cdev, &mem_fops);
56     mem_cdev->owner = THIS_MODULE;
57     mem_cdev->ops = &mem_fops;
58
59     //向内核系统中添加一个新的字符设备
60     res = cdev_add(mem_cdev, devno, 1);
61     if(res)
62     {
63         cdev_del(mem_cdev);
64         mem_cdev = NULL;
65         printk(KERN_INFO "cdev add error!\n");
66     }
67     else
68     {
69         printk(KERN_INFO "cdev add ok!\n");
70     }
71
72     //建立一个系统设备类
73     mem_class = class_create(THIS_MODULE, "myalloc");
74     if(IS_ERR(mem_class))
75     {
76         printk(KERN_INFO "Error: failed in creating class!\n");
77         return -1;
78     }
79

```

(c)

图 4.3 模块初始化函数

(4) 文件操作函数,如图 4.4 所示。

```

97 int mem_open(struct inode *ind, struct file *filp)
98 {
99     printk(KERN_INFO "open vmalloc space\n");
100     try_module_get(THIS_MODULE);
101     return 0;
102 }

```

(a)

```

104 ssize_t mem_read(struct file *filp, char *buf, size_t size, loff_t *loffp)
105 {
106     int res = -1;
107     char *tmp;
108     tmp = mem_spvm;
109     if(size > MEM_MALLOD_SIZE)
110         size = MEM_MALLOD_SIZE;
111     if(tmp != NULL)
112         res = copy_to_user(buf, tmp, size);
113     if(res == 0){
114         printk(KERN_INFO "read completed\n");
115         return size;
116     }else{
117         printk(KERN_INFO "read not completed\n");
118         return 0;
119 }

```

(b)

```

121 ssize_t mem_write(struct file *filp, const char *buf, size_t size, loff_t *loffp)
122 {
123     int res = -1;
124     char *tmp;
125     tmp = mem_spvm;
126     if(size > MEM_MALLOD_SIZE)
127         size = MEM_MALLOD_SIZE;
128     if(tmp != NULL)
129         res = copy_from_user(tmp, buf, size);
130     if(res == 0){
131         printk(KERN_INFO "write completed\n");
132         return size;
133     }else{
134         printk(KERN_INFO "write not completed\n");
135         return 0;
136 }
137 }
138

```

(c)

```

139 static loff_t mem_llseek(struct file *filp, loff_t offset, int whence)
140 {
141     loff_t newpos;
142     switch(whence) {
143     case 0: /* SEEK SET */
144         newpos = offset;
145         break;
146     case 1: /* SEEK CUR */
147         newpos = filp->f_pos + offset;
148         break;
149     case 2: /* SEEK END */
150         newpos = MEM_MALLOD_SIZE - 1 + offset;
151         break;
152     default:
153         return -EINVAL;
154     }
155     if ((newpos < 0) || (newpos > MEM_MALLOD_SIZE))
156         return -EINVAL;
157     filp->f_pos = newpos;
158     printk(KERN_INFO "llseek complete\n");
159     return newpos;
160 }
161

```

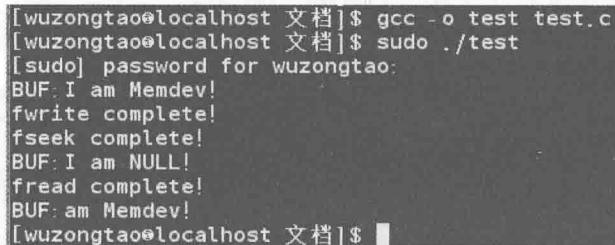
(d)

图 4.4 文件操作函数

测试代码如下：

```
#include<stdio.h>
#include<string.h>
int main(){
    FILE *fp=NULL;
    char buf[1024];
    strcpy(buf, "I'm Memdev");
    printf("BUF: %s\n", buf);
    fp = fopen("/dev/myalloc", "r+");
    if (fp == NULL)
    {
        printf("Open myalloc error!\n");
    }
    return -1;
}
fwrite(buf, sizeof(buf),1,fp);
printf(fwrite complete!\n);
fseek(fp,2,SEEK_SET);
printf(fseek complete!\n);
strcpy(buf, "I'm NULL");
fread(buf, sizeof(buf),1mfp);
printf(fread complete!\n);
printf("BUF: %s\n", buf);
close(fp);
return 0;
}
```

代码运行结果如图 4.5 所示。



```
[wuzongtao@localhost 文档]$ gcc -o test test.c
[wuzongtao@localhost 文档]$ sudo ./test
[sudo] password for wuzongtao:
BUF: I am Memdev!
fwrite complete!
fseek complete!
BUF: I am NULL!
fread complete!
BUF: am Memdev!
[wuzongtao@localhost 文档]$
```

图 4.5 运行结果

Linux 下的设备驱动程序被组织为一组完成不同任务的函数的集合,通过这些函数使得 Linux 的设备操作犹如文件一般。在应用程序看来,硬件设备只是一个设备文件,应用程序可以像操作普通文件一样对硬件设备进行操作,例如 `open()`、`close()`、`read()`、`write()`等。

Linux 主要将设备分为两类:字符设备和块设备。字符设备是指设备发送和接收数据以字符的形式进行;块设备以整个数据缓冲区的形式进行。字符设备的驱动相对简单,利用的是内核提供的结构体及相应的 API,通过 4.3 节的字符设备驱动设计可以清晰地了解如何完成一个字符设备驱动的开发。

5.1 中断

Linux 内核管理计算机上的硬件设备,首先要和它们通信。而处理器的速度跟外围硬件设备的速度往往不在一个数量级上,因此,如果内核采取让处理器向硬件发出一个请求,然后专门等待回应的办法,显然差强人意。既然硬件的响应这么慢,内核就应该在此期间处理其他事务,等到硬件真正完成请求的操作之后,再对它进行处理。要实现这种功能,轮询(polling)可能是一种解决办法。让内核定期对设备的状态进行查询,然后做出相应的处理。不过这种方法很可能会让内核做不少无用功,因为无论硬件设备是正在忙碌着完成任务还是已经大功告成,轮询总会周期性地重复执行。更好的办法是提供一种机制,让硬件在需要的时候向内核发出信号变内核主动为硬件主动。

中断使得硬件得以与处理器进行通信,中断是一种电信号,由硬件设备生成,并直接送入中断控制器的输入引脚上,然后再由中断控制器向处理器发送相应的信号。处理器检测到此信号,便中断自己的当前工作转去处理中断。处理器便会通知操作系统已经产生中断,这样,操作系统就可以对这个中断进行适当的处理了。不同的设备对应的中断不同,而每个中断都通过一个唯一的数字标识。因此,来自键盘的中断就有别于来自硬盘的中断,从而使得操作系统能够对中断进行区分,并知道哪个硬件设备产生了哪个中断。这样操作系统才能给不同的中断提供不同的处理程序。图 5.1 为中断的总体处理流程。

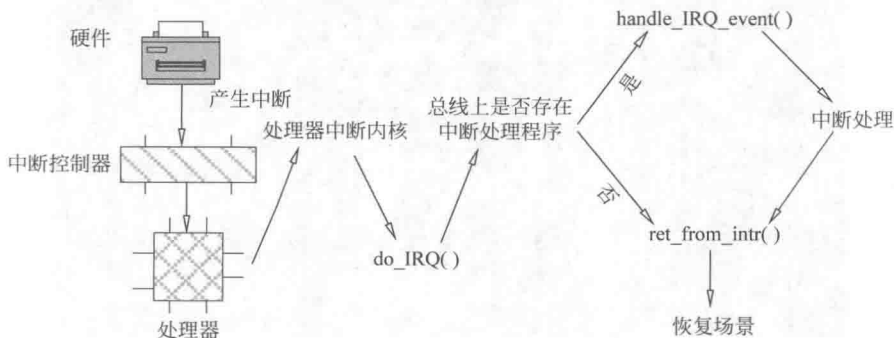


图 5.1 中断总体处理流程

5.2 中断处理

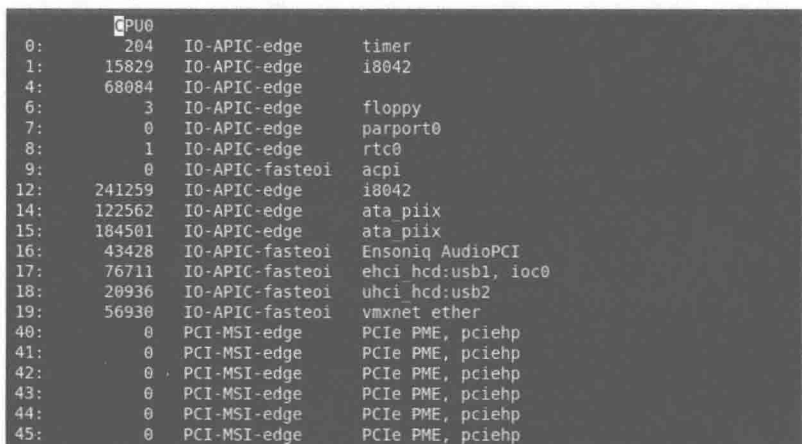
5.2.1 注册中断处理程序

中断处理程序是管理硬件驱动程序的组成部分,每一部分都有相关的驱动程序,如果设备使用,那么相应的驱动程序就注册一个中断处理程序。驱动程序可以通过下面的函数注册并激活一个中断处理程序,以便处理中断。

```
int request_irq (unsigned int irq ,
irqreturn_t ( * handler ) ( int, void *, struct pt_regs * ),
unsigned long irqflags ,
const char * devname ,
void * dev_id )
```

第 1 个参数 `irq` 表示要分配的中断号,这个值通常是预先设定的。对于大多数其他设备来说,这个值要么可以通过探测获取,要么可以通过编程动态设定。第 2 个参数 `handler` 是一个指针,指向处理这个中断的时间中断处理程序。第 3 个参数 `irqflags` 可以为 0,也可能是下列一个或多个标志的位掩码。`SA_INTERRUPT`,此标志表明给定的中断处理程序是一个快速中断处理程序,在本地处理器上,快速中断处理程序能够不受其他中断干扰,得以迅速执行;`SA_SHIRQ`,此标志表明可以在多个中断处理程序之间共享中断线。第 4 个参数 `devname` 是与中断相关的设备名字的 ASCII 文本表示法。第 5 个参数 `dev_id` 主要用于共享中断线,当一个中断处理程序需要释放时,`dev_id` 将提供唯一的标识信息,以便从共享中断线的诸多中断处理程序中删除指定的那一个。如果没有这个参数,内核不可能知道在给定的中断线上到底要删除哪一个处理程序。如果无需共享中断线,将该参数赋为空值(NULL)。

对于注册的中断处理程序,内核将注册信息保存到 `/proc/interrupts` 文件中,如图 5.2 所示。



CPU0			
0:	204	IO-APIC-edge	timer
1:	15829	IO-APIC-edge	i8042
4:	68084	IO-APIC-edge	
6:	3	IO-APIC-edge	floppy
7:	0	IO-APIC-edge	parport0
8:	1	IO-APIC-edge	rtc0
9:	0	IO-APIC-fasteoi	acpi
12:	241259	IO-APIC-edge	i8042
14:	122562	IO-APIC-edge	ata_piix
15:	184501	IO-APIC-edge	ata_piix
16:	43428	IO-APIC-fasteoi	Ensoniq AudioPCI
17:	76711	IO-APIC-fasteoi	ehci hcd:usb1, ioc0
18:	20936	IO-APIC-fasteoi	uhci hcd:usb2
19:	56930	IO-APIC-fasteoi	vmxnet ether
40:	0	PCI-MSI-edge	PCie PME, pciehp
41:	0	PCI-MSI-edge	PCie PME, pciehp
42:	0	PCI-MSI-edge	PCie PME, pciehp
43:	0	PCI-MSI-edge	PCie PME, pciehp
44:	0	PCI-MSI-edge	PCie PME, pciehp
45:	0	PCI-MSI-edge	PCie PME, pciehp

图 5.2 本地中断注册的记录文件

其中,第1列表示中断号,第2列为该中断触发的累积次数,第3列为中断控制器,第4列为设备名。

5.2.2 编写中断处理程序

编写中断处理程序如下:

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

其中,第1个参数 `irq` 就是这个处理程序要响应的中断信号。第2个参数 `dve_id` 是一个通用指针,它在与中断处理程序注册时传递给 `request_irq()` 的参数 `dev_id` 必须一致。如果该值有唯一确定性,它就相当于一个 cookie,可以用来区分共享同一中断处理程序的多个设备。最后1个参数 `regs` 是一个指向结构的指针,该结构包含处理中断之前处理器的寄存器和状态,除了调试的时候,它们很少使用到。中断处理程序可能返回两个特殊的值: `IRQ_NONE` 和 `IRQ_HANDLED`。当中断处理程序检测到一个中断,但该中断对应的设备并不是在注册处理函数期间指定的产生源时,返回 `IRQ_NONO`;当中断处理程序被正确调用,且确实是它所对应的设备产生了中断时,返回 `IRQ_HANDLED`。

中断处理程序扮演什么样的角色取决于产生中断的设备和该设备为什么要发送中断。即使其他什么工作也不做,绝大部分的中断处理程序至少要知道产生中断的设备,告诉它已经收到中断了。对于复杂一些的设备,可能还需要在中断处理程序中发送和接收数据,及执行一些扩充的工作。

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

5.3 中断上半部与下半部的对比

在中断处理程序中,既想让程序运行得快,又想让程序完成的工作量多,这两个目的显然有所抵触。鉴于两个目的之间存在不可调和的矛盾,中断处理可切分为两部分。中断处理程序是上半部(top half)——接收到一个中断,它就立即开始执行,但只做有严格时限的工作,例如对接收的中断进行应答或复位硬件,这些工作都是在所有中断被禁止的情况下完成的。可以稍后完成的工作会推迟到下半部(bottom half)去,在合适时机执行下半部分中断。

举一个例子来阐述上下两部分中断,想象一下网络设备的中断处理程序面临的挑战。该处理程序除了要对硬件应答,还要把来自硬件的网络数据包复制到内存,对其进行处理后再交给合适的协议栈或应用程序,显而易见,这种工作量不会太小,尤其对于如今的千兆比

特和万兆比特以太网卡。当网卡接收流入网络的数据包时,需要通知内核数据包到了。网卡需要立即完成这件事,从而优化网络的吞吐量和传输周期,以避免超时。因此网卡立即发出中断,通知内核这里有最新的数据包。内核通过执行网卡已注册的中断处理程序做出应答。中断开始执行,应答硬件,复制最新的网络数据包到内存,然后读取网卡更多的数据包。这些都是重要、紧迫而又与硬件相关的工作。处理和操作数据包的其他工作在随后的下半部中进行。

5.4 中断下半部

下半部的任务就是执行与中断处理密切相关,但中断处理程序本身不执行的工作。和上半部分只能通过中断处理程序实现不同,下半部可以通过多种机制实现。表 5.1 列举了中断机制演化情况。

表 5.1 中断机制演化

下半部机制	状 态
BH(bottom half)	从内核 2.5 版中去除
任务队列(task queue)	从内核 2.5 版中去除
软中断(softirq)	从内核 2.3 版中开始引入
tasklet	从内核 2.3 版中开始引入
工作队列(work queue)	从内核 2.5 版中开始引入

5.5 BH 机制与任务队列机制

以前内核中的 BH 机制设置了一个函数指针数组 bh_base[],它把所有的后半部分都组织起来,大小为 32 位,数组中的每一项就是一个后半部分,即一个 bh 函数。同时,又设置了两个 32 位无符号整数 bh_active 和 bh_mask,每个无符号整数中的一位对应着 bh_base[]中的一个元素。

BH 机制存在一些缺点。在 2.4 版以前的内核中,每次执行完 do_IRQ()中的中断服务例程以后,以及每次系统调用结束之前,就在一个叫 do_bottom_half()的函数中执行相应的 BH 函数。在 do_bottom_half()中对 BH 函数的执行是在关中断的情况下进行的,也就是对 BH 的执行进行了严格的串行化,这种方式简化了 bh 的设计。因为对单 CPU 来说,BH 函数的执行可以不嵌套;而对多 CPU 来说,在同一时间内最多只允许一个 CPU 执行 BH 函数。这种简化了的设计在一定程度上保证了从单 CPU 到多 CPU SMP 结构的平稳过渡,但随着时间的推移,就会发现这样的处理对 SMP 的性能有不利的影 响。因为,当系统中有很多个 BH 函数需要执行时,BH 函数的串行化却只能使一个 CPU 执行一个 BH 函数,其他 CPU 即使空闲,也不能执行 BH 函数。由此看出,bh 函数的串行化是针对所有 CPU 的,

根本发挥不出多 CPU 的优势。

针对 BH 机制无法并行的缺陷,内核维护人员保留了 BH 机制,并将任务队列机制引入到中断系统中来。任务队列机制通过定义一组队列实现其功能,每个队列都有自己的名字,例如调度队列、定时器队列等。不同的队列在不同的场合应用。由于用户可以随意创建新的队列,任务队列机制散落在内核各处,缺少一个类似于“总线”的机制进行总体管理。

5.6 软中断

5.6.1 软中断的实现

1. 软中断数据结构

软中断是在编译期间静态分配的,软中断由 `softirq_action` 结构表示,它定义在 `<linux/interrupt.h>` 中。

```
struct softirq_action {
    void (* action) (struct softirq_action *); /* 待执行的函数 */
    void * data; /* 传给函数的参数 */
};
```

`kernel/softirq.c` 中定义了一个包含有 32 个该结构体的数组。

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

由数据结构可见,软中断沿用了 BH 的数据结构,在调度策略方面做了相应地修改。

2. 软中断调度

软中断的工作过程模拟了实际的中断处理过程,当某一软中断事件发生后,首先需要设置对应的中断标记位,触发中断事务,然后唤醒守护线程去检测中断状态寄存器。如果通过查询发现某一软中断事务发生,那么通过软中断向量表调用软中断服务程序 `action()`,这就是软中断的过程。与硬件中断唯一不同的地方是从中断标记到中断服务程序的映射过程。在 CPU 的硬件中断发生之后,CPU 需要将硬件中断请求通过向量表映射成具体的服务程序,这个过程是硬件自动完成的。但软中断不是,其需要守护线程去实现这一过程,这就是软件模拟的中断,故称之为软中断。软中断机制的实现原理如图 5.3 所示。

软中断守护 daemon 是软中断机制的实现核心,其实现过程比较简单,通过查询软中断状态 `irq_stat` 来判断事件是否发生,如果发生,映射到软中断向量表,调用执行注册的 `action` 函数。从这点分析可以看出,软中断服务程序的执行上下文为软中断 daemon。在 Linux 中软中断 daemon 线程函数为 `do_softirq()`。触发软中断事务通过 `raise_softirq()` 来实现,该函数就是在中断关闭的情况下设置软中断状态位,然后判断如果不在中断上下文,直接唤醒守护 daemon。

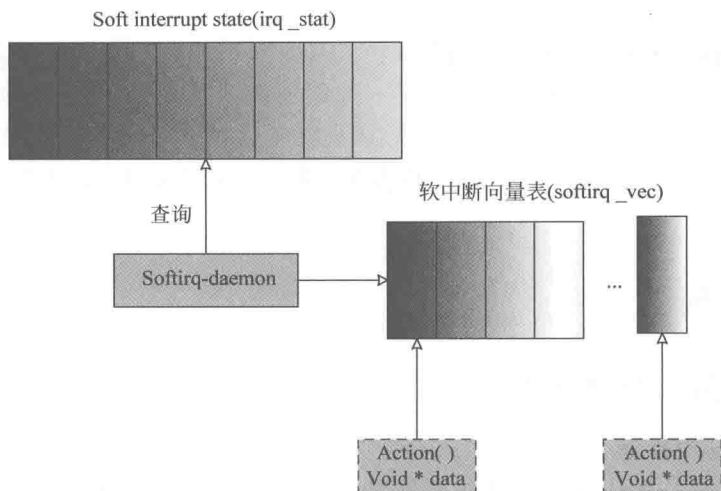


图 5.3 软中断机制的实现原理

5.6.2 软中断的使用

1. 注册处理程序

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

软中断处理程序执行的时候,允许响应中断,但它自己不能休眠。在处理程序运行的时候,当前处理器上的软中断被禁止,但其他的处理器仍可以执行别的软中断。

2. 触发软中断

```
raise_softirq(NET_TX_SOFTIRQ);
```

这会触发 NET_TX_SOFTIRQ 软中断,它的处理程序 net_tx_action() 会在内核下一次执行软中断时投入运行。

5.7 tasklet

tasklet 是通过软中断实现的,它们本身也是软中断。tasklet 由两类软中断代表: HI_SOFTIRQ 和 TASKLET_SOFTIRQ。这两者唯一的区别在于 HI_SOFTIRQ 类型的软中断优先于 TASKLET_SOFTIRQ 执行。

5.7.1 tasklet 的实现

1. tasklet 数据结构

tasklet 由 tasklet_struct 结构表示,每个结构体单独代表一个 tasklet,在 <linux/

interrupt.h>中定义。

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

state 成员只能在 0、TASKLET_STATE_SCHED 和 TASKLET_STATE_RUN 之间取值。TASKLET_STATE_SCHED 表明 tasklet 已被调度,正准备投入运行;TASKLET_STATE_RUN 表明该 tasklet 正在运行。

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

2. tasklet 调度

tasklet 机制的实现原理如图 5.4 所示,当 tasklet 的软中断事件发生之后,执行 tasklet-action 的软中断服务程序,该服务程序会扫描一个 tasklet 的任务列表,执行该任务中的具体服务程序。

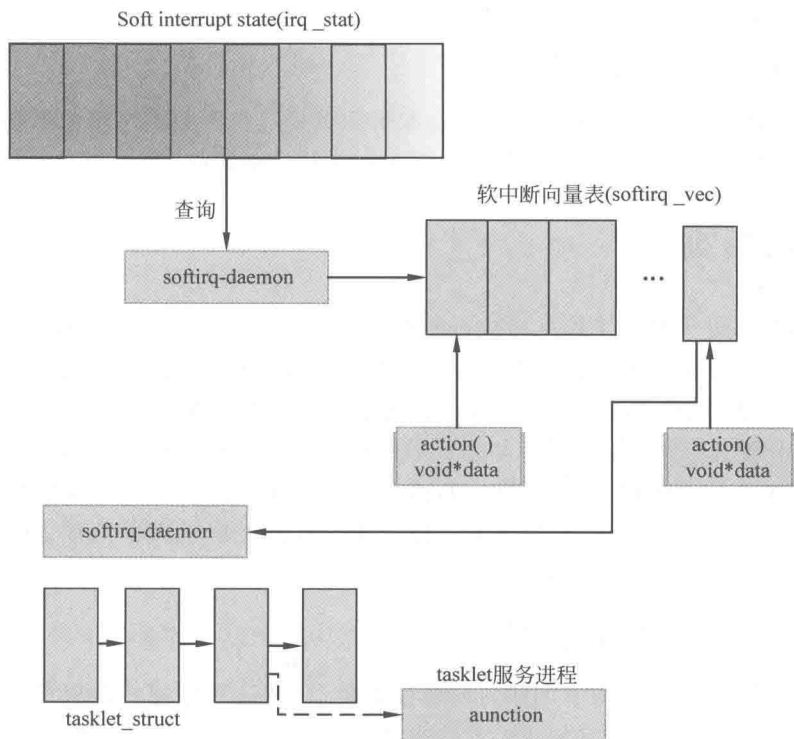


图 5.4 tasklet 机制的实现原理

举例说明：当用户读写 USB 设备之后，发生了硬件中断，硬件中断服务程序会构建一个 `tasklet_struct`，在该结构中指明完成该中断任务的具体方法函数（下半部执行函数），然后将 `tasklet_struct` 挂入 `tasklet` 的 `tasklet_struct` 链表中，这一步可以通过 `tasklet_schedule` 函数完成。最后，硬件中断服务程序退出，并且 CPU 开始调度软中断 daemon，软中断 daemon 会发现 `tasklet` 发生了事件，其会执行 `tasklet-action`，然后 `tasklet-action` 扫描 `tasklet_struct` 链表，执行具体的 USB 中断服务程序下半部。这就是应用 `tasklet` 完成中断下半部实现的整个过程。

5.7.2 tasklet 的使用

1. 声明 tasklet

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

该宏能根据给定的名称静态地创建一个 `tasklet_struct` 结构，当该 `tasklet` 被调度以后，给定的函数 `func` 被执行，它的参数由 `data` 给出。

2. tasklet 处理程序

```
void tasklet_handler(unsigned long data);           /* 注意不能睡眠 */
```

3. 调度 tasklet

```
tasklet_schedule (&my_tasklet);
```

在 `tasklet` 调度以后，只要有机会它就会尽可能早地运行。在它还没有得到运行机会之前，如果有一个相同的 `tasklet` 又被调度的了，那么它仍然只会运行一次。

5.8 工作队列

工作队列是另外一种将工作推后执行的形式，它可以把工作推后，交由一个内核线程去执行——这个下半部分总是会在进程上下文执行。这样，通过工作队列执行的代码能占尽进程上下文的优势，最重要的是工作队列允许重新调度甚至是睡眠。通常，在工作队列和软中断/`tasklet` 中做出选择非常容易。如果推后执行的任务需要睡眠，就选择工作队列；如果推后执行的任务不需要睡眠，就选择软中断或 `tasklet`。工作队列是唯一能在进程上下文运行的下半部实现机制，只有它才可以睡眠。这意味着在需要获得大量内存、获取信号量、执行阻塞式 I/O 操作时，工作队列将非常有用。

5.8.1 工作队列的实现

工作队列子系统是一个用于创建内核线程的接口，通过它创建的进程负责执行由内核其他部分排到队列里的任务，它创建的这些内核线程被习惯称为工作者线程。默认的工作者线程叫作 `events/n`，这里 `n` 是处理器的编号，每个处理器对应一个线程。

工作者线程用 `workqueue_struct` 结构表示：

```
struct workqueue_struct {
    struct cpu_workqueue_struct cpu_wq[NR_CPUS];
    const char * name;
    struct list_head list;
};
```

该结构内是一个由 `cpu_workqueue_struct` 结构组成的数组，它定义在 `kernel/workqueue.c` 中，数组中每一项对应系统中的一个处理器。每个工作者线程类型关联一个自己的 `workqueue_struct`。在该结构体里面，给每个线程分配一个 `cpu_workqueue_struct`，因而也就是给每个处理器分配一个，因为每个处理器都有一个该类型的工作者线程。

工作常常用 `work_struct` 结构表示：

```
struct work_struct {
    unsigned long pending;           /* 工作是否被挂起 */
    struct list_head entry;         /* 连接所有工作的链表 */
    void (* func) (void * );       /* 处理函数 */
    void * data;                   /* 传递给处理函数的参数 */
    void * wq_data;                /* 内部使用 */
    struct timer_list timer;        /* 延迟的工作队列所用到的定时器 */
};
```

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

以上定义了诸多结构体，使用图的方式清晰地捋顺一下各结构体间的关系，如图 5.5 所示。工作由 `work_struct` 结构体表示，`work_struct` 是 `cpu_workqueue_struct` 的一个成员，同时 `cpu_workqueue_struct` 又将工作分成不同的类型，例如处理阻塞时 I/O 的工作、处理硬盘数据的工作，等等，这些不同类型的工作集合统一使用 `workqueue_struct` 结构体描述。将分好类的工作链表交给相应的线程完成工作。

5.8.2 工作队列的使用

1. 创建推后的工作

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

静态地创建一个名为 `name`，处理函数为 `func`，参数为 `data` 的 `work_struct` 结构体。

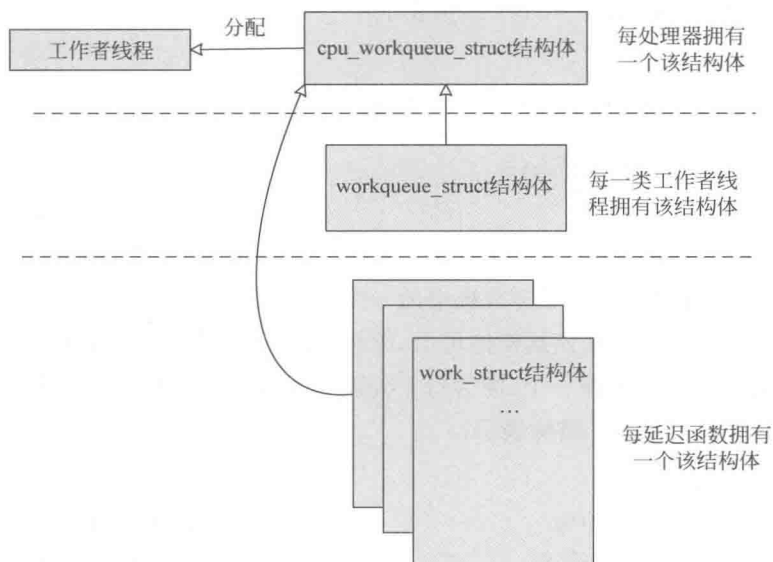


图 5.5 工作、工作队列和工作线程间的关系

2. 工作队列处理函数

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

这个函数会由一个工作者线程执行,因此,函数会运行在进程上下文中,默认情况下允许响应中断。

3. 对工作进行调度

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

work 马上就会被调度,一旦其所在的处理上的工作者线程被唤醒,它就会被执行。

4. 刷新操作

```
void flush_scheduled_work (void);
```

函数会一直等待,直到队列中所有对象都被执行以后才返回。在等待所有待处理的工作执行的时候,该函数会进入休眠状态,所以只能在进程上下文中使用它。

5. 创建新的工作队列

```
struct workqueue_struct * create_workqueue(const char * name);
```

这个函数会创建所有的工作者线程,系统中的每个处理器都有一个,并且做好所有开始处理工作之前的准备工作。

6.1 块设备管理机制

6.1.1 块设备基本概念

扇区 (Sectors)：任何块设备硬件对数据处理的基本单位。通常，1 个扇区的大小为 512B。

块 (Blocks)：由 Linux 制定对内核或文件系统等数据处理的基本单位。通常，1 个块由 1 个或多个扇区组成。

段 (Segments)：由若干个相邻的块组成，是 Linux 内存管理机制中一个内存页或者内存页的一部分。

页、段、块、扇区之间的关系图如图 6.1 所示。

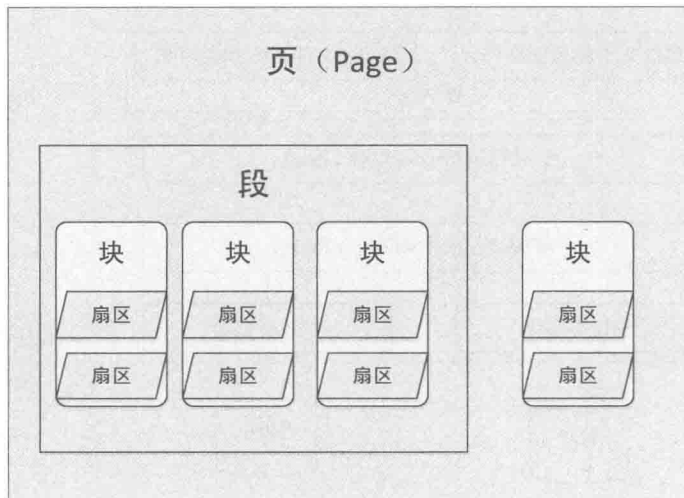


图 6.1 页、段、块、扇区关系图

块设备只能以块为单位接受输入和返回输出,而字符设备以字节为单位。大多数设备是字符设备,因为它们不需要缓冲而且不以固定块大小进行操作。

块设备对于 I/O 请求有对应的缓冲区,因此它们可以选择以什么顺序进行响应,字符设备无需缓冲且被直接读写。对于存储设备,调整读写的顺序作用巨大,因为在读写连续的扇区比分离的扇区更快。

字符设备只能被顺序读写,而块设备可以随机访问。虽然块设备可随机访问,但是对于磁盘这类机械设备,顺序地组织块设备的访问可以提高性能。注意,对 SD 卡、RAMDISK 等块设备,不存在机械上的原因,进行这样的调整没有必要。

6.1.2 块设备在 Linux 中的结构

在 Linux 中,驱动对块设备的输入或输出(I/O)操作,都会向块设备发出一个请求,在驱动中用 request 结构体描述。但对于一些磁盘设备,请求的速度很慢,这时候内核就提供一种队列的机制把这些 I/O 请求添加到队列中(即请求队列),在驱动中用 request_queue 结构体描述。在向块设备提交这些请求前,内核会先执行请求的合并和排序预操作,以提高访问的效率,然后再由内核中的 I/O 调度程序子系统负责提交 I/O 请求,I/O 调度程序将磁盘资源分配给系统中所有挂起的块 I/O 请求,其工作是管理块设备的请求队列,决定队列中的请求排列顺序及什么时候派发请求到设备,块设备在整个 Linux 中应用的总体结构如图 6.2 所示。

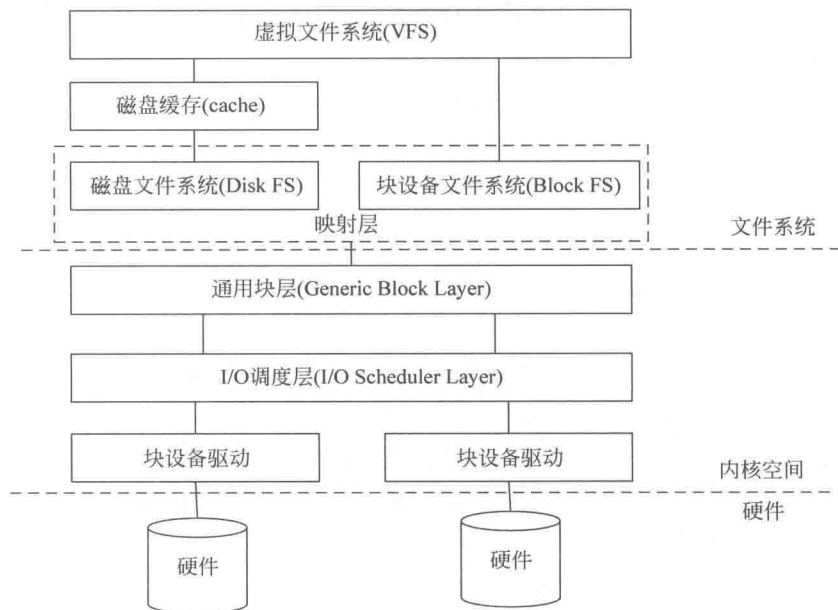


图 6.2 块设备总体结构

块设备驱动怎样维持一个 I/O 请求在上层文件系统与底层物理磁盘之间的关系呢？这就是图 6.2 中通用块层(Generic Block Layer)要做的事情。在通用块层中,通常用一个 bio 结构体对应一个 I/O 请求,它代表了正在活动的以段(Segment)链表形式组织的块 IO 操作,对于它所需要的所有段用 bio_vec 结构体表示。

块设备驱动又是怎样对底层物理磁盘进行访问的呢？上面讲的都是对上层的访问和对上层的联系。Linux 提供了一个 gendisk 数据结构体,用它来表示一个独立的磁盘设备或分区。在 gendisk 中有一个类似字符设备中 file_operations 的硬件操作结构指针,它就是 block_device_operations 结构体。它主要定义块设备所支持的操作。

6.2 块设备关键数据结构

6.2.1 gendisk 数据结构

在 Linux 内核中使用 gendisk 结构体来表示一个独立的磁盘设备或分区,它存储了一个硬盘的信息,包括请求队列、分区链表和块设备操作函数集等。这个结构体的定义在 <linux/genhd.h>中,具体结构如下:

```
struct gendisk
{
    int major;                /* 主设备号 */
    int first_minor;          /* 第一个次设备号 */
    int minors;               /* 最大的次设备号,如果不能分区则为 1 */
    char disk_name[DISK_NAME_LEN]; /* 设备名称 */
    struct disk_part_tbl * part_tbl; /* 磁盘上的分区表信息 */
    struct hd_struct part0;
    struct block_device_operations * fops; /* 块设备对底层硬件的操作结构体指针 */
    struct request_queue * queue; /* 请求队列 */
    void * private_data; /* 私有数据 */
    ...
};
```

Linux 内核提供了一组函数来操作 gendisk,主要包括以下几类。

1. 分配 gendisk

gendisk 结构体是一个动态分配的结构体,它需要特别的内核操作来初始化,驱动不能自己分配这个结构体,使用下列函数来分配 gendisk:

```
struct gendisk * alloc_disk(int minors);
```

minors 参数是这个磁盘使用的次设备号的数量,一般就是磁盘分区数量,此后 minors 不能被修改。

2. 增加 gendisk

gendisk 结构体被分配之后,系统还不能使用这个磁盘,需要调用如下函数来注册这个

磁盘设备：

```
void add_disk(struct gendisk * gd);
```

特别要注意的是对 `add_disk()` 的调用,必须发生在驱动程序的初始化工作完成并能响应磁盘的请求之后。

3. 释放 gendisk

当不再需要一个磁盘时,应当使用如下函数释放 `gendisk`：

```
void del_gendisk(struct gendisk * gd);
```

4. gendisk 引用计数

`gendisk` 中包含 1 个 `kobject` 成员,因此,它是一个可被引用计数的结构体。通过 `get_disk()` 和 `put_disk()` 函数可用来操作引用计数,这个工作一般不需要驱动做。通常对 `del_gendisk()` 的调用会去掉 `gendisk` 的最终引用计数,但并不是一定的。因此,在 `del_gendisk()` 被调用后,这个结构体可能继续存在。

5. 设置 gendisk 容量

```
void set_capacity(struct gendisk * disk, sector_t size);
```

块设备中最小的可寻址单元是扇区,扇区大小一般是 2 的整数倍,最常见的是 512 字节。扇区的大小是设备的物理属性,扇区是所有块设备的基本单元,块设备无法对比它还小的单元进行寻址和操作,不过许多块设备能够一次传输多个扇区。

6.2.2 block_device_operations 数据结构

在块设备驱动中,有 1 个类似于字符设备驱动中 `file_operations` 结构体的 `block_device_operations` 结构体,它是对块设备操作的集合,结构如下：

```
struct block_device_operations
{
    int (* open)(struct inode *, struct file *);           //打开
    int (* release)(struct inode *, struct file *);       //释放
    int (* ioctl)(struct inode *, struct file *, unsigned, unsigned long);           //控制
    long (* unlocked_ioctl)(struct file *, unsigned, unsigned long);
    long (* compat_ioctl)(struct file *, unsigned, unsigned long);
    int (* direct_access)(struct block_device *, sector_t, unsigned long *);
    int (* media_changed)(struct gendisk *);               //介质被改变
    int (* revalidate_disk)(struct gendisk *);             //使介质有效
    int (* getgeo)(struct block_device *, struct hd_geometry *);           //填充驱动器信息
    struct module * owner;                                 //模块拥有者
};
```

1. 打开和释放

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


与字符设备驱动类似,当设备被打开和关闭时将调用它们。

2. IO 控制

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

上述函数是 `ioctl()` 系统调用的实现,块设备包含大量的标准请求,这些标准请求由 Linux 块设备层处理,因此大部分块设备驱动的 `ioctl()` 函数相当短。

3. 介质改变

```
int (* media_changed) (struct gendisk * gd);
```

被内核调用来检查驱动器中的介质是否已经改变,如果是,返回一个非零值,否则返回 0。这个函数仅适用于支持可移动介质的驱动器(非可移动设备的驱动不需要实现这个方法),通常需要在驱动中增加 1 个表示介质状态是否改变的标志变量。

4. 使介质有效

```
int (* revalidate_disk) (struct gendisk * gd);
```

`revalidate_disk()` 函数被调用来响应一个介质改变,它给驱动一个机会来进行必要的工作以使新介质准备好。

5. 获得驱动器信息

```
int (* getgeo)(struct block_device *, struct hd_geometry *);
```

该函数根据驱动器的几何信息填充一个 `hd_geometry` 结构体,`hd_geometry` 结构体包含磁头、扇区、柱面等信息。

6. 模块指针

```
struct module * owner;
```

指向拥有这个结构体的模块的指针,通常被初始化为 `THIS_MODULE`。

6.2.3 request 数据结构

在 Linux 块设备驱动中,使用 `request` 结构体来表征等待进行的 I/O 请求。这个结构体的定义如下:

```
struct request
{
    struct list_head queuelist;           /* 链表结构 */
    unsigned long flags;                  /* REQ_ */
    sector_t sector;                      /* 要传送的下 1 个扇区 */
    unsigned long nr_sectors;             /* 要传送的扇区数目 */
    unsigned int current_nr_sectors;      /* 当前要传送的扇区数目 */
    sector_t hard_sector;                 /* 要完成的下 1 个扇区 */
    unsigned long hard_nr_sectors;        /* 要被完成的扇区数目 */
}
```

```

    unsigned int hard_cur_sectors;           /* 当前要被完成的扇区数目 */
    struct bio * bio;                       /* 请求的 bio 结构体的链表 */
    struct bio * biotail;                   /* 请求的 bio 结构体的链表尾 */
    ...
}

```

request 结构体的主要成员包括

```

sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;

```

上述 3 个成员标识还未完成的扇区,hard_sector 是第 1 个尚未传输的扇区,hard_nr_sectors 是尚待完成的扇区数,hard_cur_sectors 是当前 I/O 操作中待完成的扇区数。这些成员只用于内核块设备层,驱动不应当使用它们。

```

sector_t sector;
unsigned long nr_sectors;
unsigned int current_nr_sectors;

```

驱动中会经常与这 3 个成员打交道,这 3 个成员在内核和驱动交互中发挥着重大作用。它们以 512 字节为 1 个扇区,如果硬件的扇区不是 512 字节,需要进行相应的调整。

```

struct bio * bio;

```

bio 是这个请求中包含的 bio 结构体的链表,驱动中不宜直接存取这个成员,应该使用后文将介绍的 rq_for_each_bio()。

```

char * buffer;

```

指向缓冲区的指针,数据应当被传送到或者来自这个缓冲区,这个指针是一个内核虚拟地址,可被驱动直接引用。

```

unsigned short nr_phys_segments;

```

该值表示相邻的页被合并后,这个请求在物理内存中占据的段的数目。如果设备支持分散/聚集(SG, scatter/gather)操作,可依据此字段申请 sizeof(scatterlist) * nr_phys_segments 的内存,并使用下列函数进行 DMA 映射:

```

int blk_rq_map_sg(request_queue_t * q, struct request * req, struct scatterlist * sglist);

```

该函数与 dma_map_sg()类似,它返回 scatterlist 列表入口的数量。

```

struct list_head queuelist;

```

用于链接这个请求到请求队列的链表结构,调用 blkdev_dequeue_request()从队列中移除请求。

```
rq_data_dir(struct request * req);
```

使用这个宏可以从 request 获得数据传送的方向。0 返回值表示从设备中读,非 0 返回值表示向设备写。

6.2.4 request_queue 数据结构

求队列跟踪等候的块 I/O 请求,存储用于描述这个设备能够支持的请求的类型信息。例如,它们的最大大小、多少不同的段可进入一个请求、硬件扇区大小、对齐要求等参数。其结果是:如果请求队列被正确配置,它不会交给该设备一个不能处理的请求。数据结构如下:

```
struct request_queue
{ ...
    spinlock_t __queue_lock;           /* 保护队列结构体的自旋锁 */
    spinlock_t * queue_lock;
    struct kobject kobj;               /* 队列 kobject */
    unsigned long nr_requests;         /* 最大请求数量 */
    unsigned int nr_congestion_on;
    unsigned int nr_congestion_off;
    unsigned int nr_batching;
    unsigned short max_sectors;        /* 最大的扇区数 */
    unsigned short max_hw_sectors;
    unsigned short max_phys_segments; /* 最大的段数 */
    unsigned short max_hw_segments;
    unsigned short hardsect_size;      /* 硬件扇区尺寸 */
    unsigned int max_segment_size;     /* 最大的段尺寸 */
    unsigned long seg_boundary_mask;   /* 段边界掩码 */
    unsigned int dma_alignment;        /* DMA 传送的内存对齐限制 */
    ...
}
```

1. 初始化请求队列

```
request_queue_t * blk_init_queue(request_fn_proc * rfn, spinlock_t * lock);
```

该函数的第 1 个参数是请求处理函数的指针,第 2 个参数是控制访问队列权限的自旋锁,这个函数会发生内存分配的行为,故可能会失败,一定要检查它的返回值。这个函数一般在块设备驱动模块加载函数中调用。

2. 清除请求队列

```
void blk_cleanup_queue(request_queue_t * q);
```

这个函数完成将请求队列返回给系统的任务,一般在块设备驱动模块卸载函数中调用。blk_put_queue()宏定义为

```
#define blk_put_queue(q) blk_cleanup_queue((q))
```

3. 分配请求队列

```
request_queue_t * blk_alloc_queue(int gfp_mask);
```

对于 FLASH、RAM 盘等完全随机访问的非机械设备,不需要进行复杂的 I/O 调度,应该使用上述函数分配 1 个请求队列。

4. 绑定请求队列和制造请求函数

```
void blk_queue_make_request(request_queue_t * q, make_request_fn * mfn);
```

6.2.5 bio 数据结构

通常 1 个 bio 对应 1 个 I/O 请求,IO 调度算法可将连续的 bio 合并成 1 个请求。所以,1 个请求可以包含多个 bio。

bio 为通用层的主要数据结构,既描述了磁盘的位置,又描述了内存的位置,是上层 vfs 与下层驱动的连接纽带。它的数据结构如下:

```
struct bio
{
    sector_t bi_sector;           /* 要传输的第 1 个扇区 */
    struct bio * bi_next;         /* 下一个 bio */
    struct block_device * bi_bdev; /* 描述一个逻辑上的块设备 */
    unsigned long bi_flags;       /* 状态,命令等 */
    unsigned long bi_rw;          /* 低位表示 READ/WRITE,高位表示优先级 */
    unsigned short bi_vcnt;       /* bio_vec 数量 */
    unsigned short bi_idx;        /* 当前 bvl_vec 索引 */
    unsigned short bi_phys_segments; /* 不相邻的物理段的数目 */
    /* 物理合并和 DMA remap 合并后不相邻的物理段的数目 */
    unsigned short bi_hw_segments;
    unsigned int bi_size;         /* 以字节为单位所需传输的数据大小 */
    /* 为了明确最大的 hw 尺寸,考虑这个 bio 中第 1 个和最后 1 个虚拟的可合并的段的尺寸 */
    unsigned int bi_hw_front_size;
    unsigned int bi_hw_back_size;
    unsigned int bi_max_vecs;     /* 能持有的最大 bvl_vecs 数 */
    struct bio_vec * bi_io_vec;   /* 实际的 vec 列表 */
    ...
}
```

内核还提供了一组函数(宏)用于操作 bio

```
int bio_data_dir(struct bio * bio);
```

这个函数可用于获得数据传输的方向是 READ 还是 WRITE。

```
struct page * bio_page(struct bio * bio);
```

这个函数可用于获得目前的页指针。

```
int bio_offset(struct bio * bio) ;
```

这个函数返回操作对应的当前页内的偏移,通常块 I/O 操作本身就是页对齐的。

```
int bio_cur_sectors(struct bio * bio) ;
```

这个函数返回当前 bio_vec 要传输的扇区数。

```
char * bio_data(struct bio * bio) ;
```

这个函数返回数据缓冲区的内核虚拟地址。

```
char * bvec_kmap_irq(struct bio_vec * bvec, unsigned long * flags) ;
```

这个函数返回一个内核虚拟地址,这个地址可用于存取被给定的 bio_vec 入口指向的数据缓冲区。

bio 的核心是一个称为 bi_io_vec 的数组,它由 bio_vec 结构体组成,bio_vec 结构体的定义如下:

```
struct bio_vec
{
    struct page * bv_page;           /* 页指针 */
    unsigned int bv_len;             /* 传输的字节数 */
    unsigned int bv_offset;          /* 偏移位置 */
};
```

request、bio 与 bio_vec 之间的关系如图 6.3 所示。

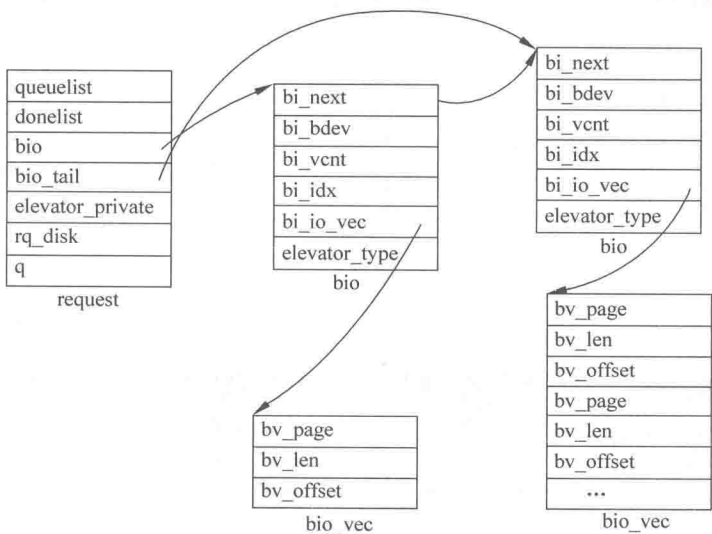


图 6.3 request、bio 与 bio_vec 之间的关系

gendisk、request、request_queue、bio 与 bio_vec 之间的关系如图 6.4 所示。

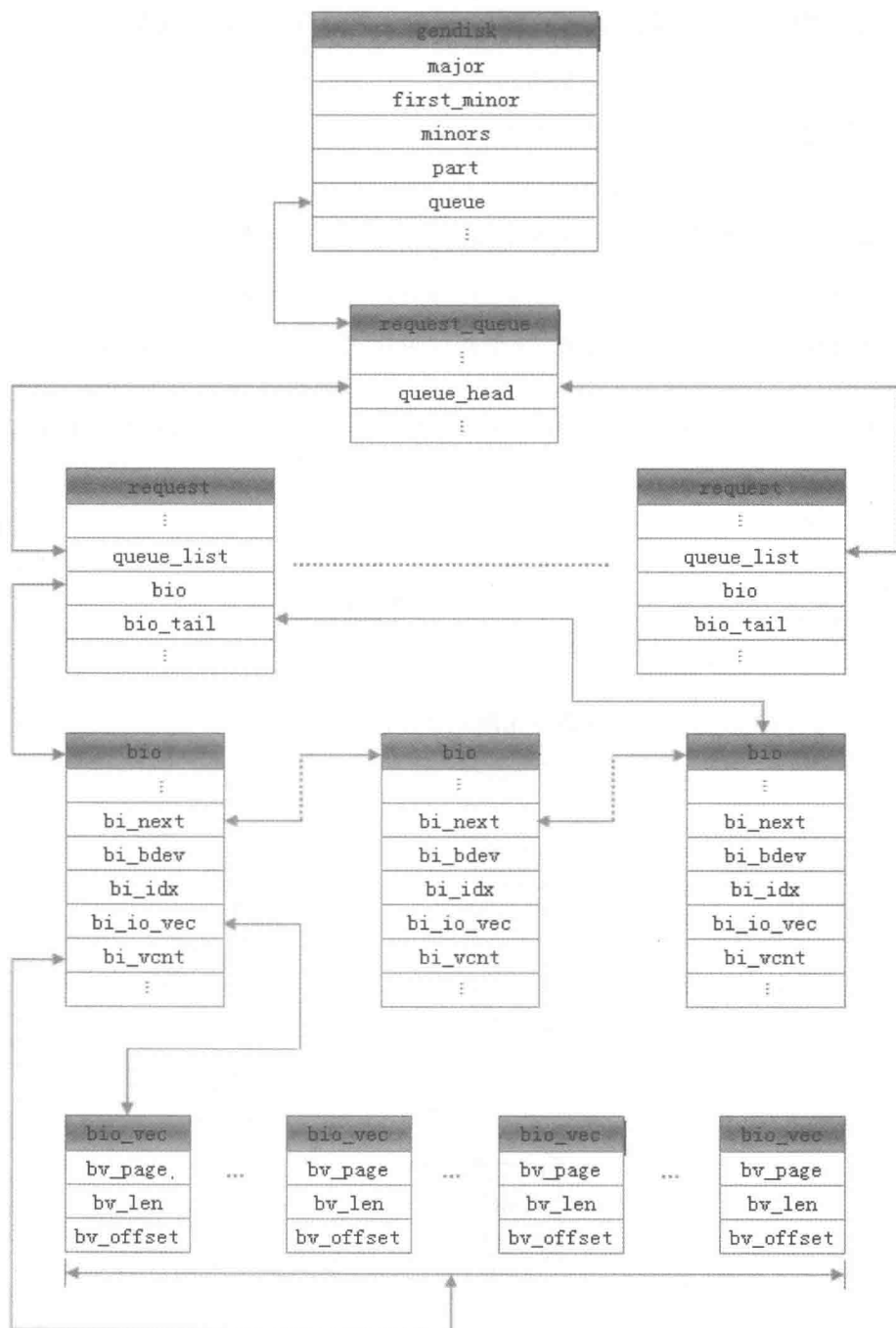


图 6.4 gendisk、request、request_queue、bio 与 bio_vec 之间的关系

6.3 块设备驱动设计函数

6.3.1 块设备驱动注册与注销函数

1. 注册函数

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

major 参数是块设备要使用的主设备号, name 为设备名, 会在 /proc/devices 中被显示。如果 major 为 0, 内核会自动分配一个新的主设备号, register_blkdev() 函数的返回值就是这个主设备号。如果 register_blkdev() 返回 1 个负值, 表明发生了一个错误。

2. 注销函数

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

传递给 register_blkdev() 的参数必须与传递给 unregister_blkdev() 的参数匹配, 否则这个函数返回 -EINVAL。

6.3.2 块设备驱动打开与关闭函数

块设备驱动的 open() 和 release() 函数是非必须的, 1 个简单的块设备驱动可以不提供 open() 和 release() 函数。块设备驱动的 open() 函数和其字符设备驱动中的对等体非常类似, 都以相关的 inode 和 file 结构体指针作为参数。当一个节点引用一个块设备时, inode->i_bdev->bd_disk 包含一个指向关联 gendisk 结构体的指针。因此, 类似于字符设备驱动, 也可以将 gendisk 的 private_data 赋给 file 的 private_data, private_data 同样最好是指向描述该设备的设备结构体 xxx_dev 的指针。

```
static int xxx_open(struct inode * inode, struct file * filp)
{
    struct xxx_dev * dev = inode->i_bdev->bd_disk->private_data;
    filp->private_data = dev; //赋值 file 的 private_data
    ...
    return 0;
}
```

在一个处理真实硬件设备的驱动中, open() 和 release() 方法还应当设置驱动和硬件的状态, 这些工作可能包括启停磁盘、加锁一个可移出设备和分配 DMA 缓冲等。

6.3.3 块设备驱动 ioctl、read 和 write 函数

与字符设备驱动一样, 块设备可以包含一个 ioctl() 函数以提供对设备的 I/O 控制能力。实际上, 高层的块设备层代码处理了绝大多数 ioctl(), 具体的块设备驱动中通常不再

需要实现很多 `ioctl` 命令。

对于块设备,不用编写 `read()` 和 `write()` 函数,而是用 VFS 提供的通用函数 `block_write()` 和 `block_read()`,即在数据结构 `file_operations` 中的 `read` 和 `write` 值是 `block_read()` 和 `block_write()`。

6.3.4 块设备驱动的请求函数

1. 使用请求队列的情况

块设备请求的原型为

```
void request (request_queue_t * queue) ;
```

这个函数不能由驱动自己调用,只有当内核认为是时候让驱动处理对设备的读写等操作时,才调用这个函数

```
static void xxx_request(request_queue_t * q)
{ ...
    /* 处理该请求 */
    xx_transfer(dev, req->sector, req->current_nr_sectors, req->buffer, rq_data_dir(req));
    end_request(req, 1);           //通知成功完成这个请求
}

static void xx_transfer(struct xx_dev * dev, unsigned long sector, unsigned long nsect, char *
buffer, int write)
{
    ...
    if (write)
    {
        write_dev(offset, buffer, nbytes);
    }
    else
    {
        read_dev(offset, buffer, nbytes);
    }
}
```

2. 不使用请求队列的情况

使用请求队列对于一个机械的磁盘设备的确有助于提高系统的性能,但是对于许多块设备,例如数码相机的存储卡、RAM 盘等完全可真正随机访问的设备,无法从高级的请求队列逻辑中获益。对于这些设备,块层支持无队列的操作模式,为使用这个模式,驱动必须提供一个制造请求函数,而不是一个请求函数。

制造请求函数的原型为

```
typedef int (make_request_fn) (request_queue_t * q, struct bio * bio);
```


6.4 Ramdisk 块设备驱动实例

6.4.1 Ramdisk 块设备驱动实例分析

Ramdisk(RAM 盘)是一种模拟磁盘,其数据实际上存储在 RAM 中,它使用一部分内存空间来模拟出一个磁盘,以块设备的方式访问这片内存。以下是对驱动源码的分析。

1. 头文件定义及模块声明

```
#include<linux/module.h>
#include<linux/fs.h>
#include<linux/genhd.h>
#include<linux/blkdev.h>
#include<linux/bio.h>
#define DEVICE_MAJOR 0
#define DEVICE_NAME "ramdisk"
#define SECTOR_SIZE 512 //扇区大小
#define DISK_SIZE (3 * 1024 * 1024) //虚拟磁盘大小
#define SECTOR_ALL (DISK_SIZE/SECTOR_SIZE) //虚拟磁盘的扇区数
static struct gendisk *p_disk; //用来指向申请的 gendisk 结构体
static struct request_queue *p_queue; //用来指向请求队列
static unsigned char mem_start[DISK_SIZE]; //分配一块 3MB 的内存作为虚拟磁盘
module_init(ramdisk_init); //模块的初始化申明
module_exit(ramdisk_exit); //模块的卸载申明
MODULE_LICENSE("GPL"); //开源申明
```

2. 模块的初始化

```
static int ramdisk_init(void)
{
    p_queue = blk_alloc_queue(GFP_KERNEL); //申请请求队列
    if(!p_queue)return -1;
    blk_queue_make_request(p_queue, ramdisk_make_request); //绑定制造请求函数
    p_disk = alloc_disk(1); //申请一个分区的 gendisk 结构体
    if(!p_disk){
        blk_cleanup_queue(p_queue); //gendisk 申请失败,清除已申请的请求队列
        return -1;
    }
    strcpy(p_disk->disk_name, DEVICE_NAME); //块设备名
    p_disk->major = DEVICE_MAJOR; //主设备号
    p_disk->first_minor = 0; //次设备号
    p_disk->fops = &ramdisk_fops; //fops 地址
    p_disk->queue = p_queue; //请求队列地址
    set_capacity(p_disk, SECTOR_ALL); //设置磁盘扇区数
    add_disk(p_disk); //设置好后添加这个磁盘
    return 0;
}
```

3. 模块的请求

```
static int ramdisk_make_request(struct request_queue *q, struct bio *bio)
{
    struct bio_vec *bvec;                //bio 结构中包含多个 bio_vec 结构
    int i;                                //用于循环的变量,不需要赋值
    void *disk_mem;                       //指向虚拟磁盘正在读写的位置

    /* 检查超出容量的情况 */
    if((bio->bi_sector * SECTOR_SIZE) + bio->bi_size > DISK_SIZE){
        printk("ramdisk over flowed!\n");
        bio_endio(bio, 1);                //第 2 个参数为 1, 通知内核 bio 处理出错
        return 0;
    }

    /* mem_start 是虚拟磁盘的起始地址 */
    disk_mem = mem_start + bio->bi_sector * SECTOR_SIZE;
    bio_for_each_segment(bvec, bio, i){
        void *iovec;                      //指向内核存放数据的地址
        iovec = kmap(bvec->bv_page) + bvec->bv_offset;    //将 bv_page 映射到高端内存
        switch(bio_data_dir(bio)){        //bio_data_dir(bio) 返回要处理数据的方向
            case READA:                    //READA 是预读, RAED 是读, 采用统一处理
            case READ: memcpy(iovec, disk_mem, bvec->bv_len); break;
            case WRITE: memcpy(disk_mem, iovec, bvec->bv_len); break;
            default: bio_endio(bio, 1); kunmap(bvec->bv_page); return 0;    //处理失败的情况
        }
        kunmap(bvec->bv_page);            //释放 bv_page 的映射
        disk_mem += bvec->bv_len;
        //移动虚拟磁盘的指向位置, 准备下一个循环 bvec 的读写做准备
    }
    bio_endio(bio, 0);                    //第 2 个参数为 0, 通知内核处理成功
    return 0;
}
```

4. 模块的卸载

```
static void ramdisk_exit(void)
{
    del_gendisk(p_disk);                //删除 gendisk 注册信息
    put_disk(p_disk);                   //释放 disk 空间
    blk_cleanup_queue(p_queue);          //清除请求队列
}
```

6.4.2 Ramdisk 块设备驱动实例测试

编译驱动后, 得到的文件如图 6.5 所示。

其中, ramdisk.ko 文件就是编译好的驱动模块文件。

通过 `ls-l /dev | grep ramdisk` 查看设备文件, 如图 6.6 所示。

```

smx@ubuntu:~/ramdisk$ ls
built-in.o      Module.symvers  ramdisk.mod.c
Makefile        ramdisk.c       ramdisk.mod.o
modules.order   ramdisk.ko      ramdisk.o
smx@ubuntu:~/ramdisk$ sudo insmod ramdisk.ko

```

图 6.5 编译驱动后文件目录

```

smx@ubuntu:~/ramdisk$ ls -l /dev|grep ramdisk
brw-rw---- 1 root disk 240, 0 2013-04-13 21:18 ramdisk
smx@ubuntu:~/ramdisk$ lsmod /dev|grep ramdisk
Usage: lsmod
smx@ubuntu:~/ramdisk$ ls /dev|grep ramdisk
ramdisk
smx@ubuntu:~/ramdisk$ ls -l /dev|grep ramdisk
brw-rw---- 1 root disk 240, 0 2013-04-13 21:18 ramdisk
smx@ubuntu:~/ramdisk$ lsmod |grep ramdisk
ramdisk                3147055 0

```

图 6.6 ramdisk 详细信息

将 ramdisk 格式化成 ext3 格式,如图 6.7 所示。

```

smx@ubuntu:~/ramdisk$ sudo mkfs.ext3 /dev/ramdisk
mke2fs 1.41.12 (17-May-2010)
文件系统标签=
操作系统:Linux
块大小=1024 (log=0)
分块大小=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
768 inodes, 3072 blocks
153 blocks (4.98%) reserved for the super user
第一个数据块=1
Maximum filesystem blocks=3145728
1 block group
8192 blocks per group, 8192 fragments per group
768 inodes per group

```

图 6.7 ramdisk 转化为 ext3 格式

挂载/dev/ramdisk 到/mnt/test 下,可以对 ramdisk 进行相应的读写,结果如图 6.8 所示。

```

smx@ubuntu:~/ramdisk$ sudo mount /dev/ramdisk /mnt/test/
smx@ubuntu:~/ramdisk$ sudo cp ./ * /mnt/test/
smx@ubuntu:~/ramdisk$ ls /mnt/test/
built-in.o  modules.order  ramdisk.c  ramdisk.mod.c  ramdisk.o
Makefile    Module.symvers  ramdisk.ko  ramdisk.mod.o
smx@ubuntu:~/ramdisk$ sudo lsmod |grep ramdisk
ramdisk                3147055 1
smx@ubuntu:~/ramdisk$ sudo rm -rf /mnt/test/*
smx@ubuntu:~/ramdisk$ sudo umount /mnt/test/
smx@ubuntu:~/ramdisk$ lsmod |grep ramdisk
ramdisk                3147055 0

```

图 6.8 显示挂载后 ramdisk 信息

7.1 网络设备

7.1.1 网络系统分层结构

Linux 内核中网络系统是一个分层结构,一般把 Linux 网络系统分为物理层/数据链路层、IP 层、INET Socket 层、BSD Socket 层和应用层 5 个部分。前 4 部分包含在 Linux 内核中。INET Socket 层实现对 IP 分组排序、控制网络系统效率等功能,它比 IP 协议层次高。而 IP 层是 TCP/IP 网络协议栈中网络层的实现。因为物理层在 TCP/IP 协议栈本身就和数据链路层区分不明确,所以可以称这个包括了硬件驱动和硬件发送组织工作的层次为物理层。图 7.1 说明了 Linux 中基于 TCP/IP 协议的网络系统体系结构。

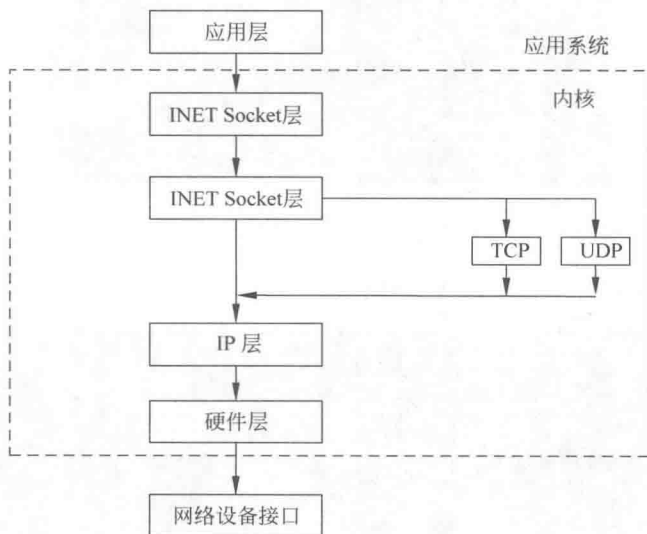


图 7.1 Linux 网络协议栈示意图

下面介绍网络数据从应用层到网络设备接口硬件之间的流程走向。应用层中的操作对象是 socket 文件描述符,通过文件系统定义的通用接口,使用系统调用从用户空间切换到内核空间,控制 socket 文件描述符对应的就是对 BSD Socket 的操作,从而进入到 BSD Socket 层的操作。在 BSD Socket 层中,操作的对象是 struct socket 结构,每一个这样的结构对应一个网络连接。通过网络地址的不同来区分不同的操作方法,判断是否应该进入到 INET Socket 层,这一层的数据存放在 struct msghdr 结构体中,内核使用该结构体来接收数据。在 INET Socket 层中,根据建立连接的类型,分成面向连接和无连接两种类型,这是区分 TCP 和 UDP 协议的主要原则。这一层中的操作对象是 socket 类型的数据,数据存放在 sk_buff 结构体中,sk_buff 结构体可能是 Linux 网络代码中最重要的数据结构,它表示接收或发送数据包的包头信息。从 INET Socket 层到 IP 层,主要是路由过程,发送时根据发送的目标地址确定需要使用的网络设备接口和下一个需要传送到机器地址;接收数据的时候需要在 IP 层判断该数据包是要发送给上一层协议还是需要做一个 IP 转发,将数据传递给下一台机器。从 IP 层到硬件层,也就是到网络接口设备驱动程序,即是有关硬件相关的控制方法。

7.1.2 网络设备管理

为了屏蔽网络环境中物理网络设备的多样性,Linux 对所有的物理设备进行抽象并定义了一个统一的概念,称为接口(Interface)。所有对网络硬件的访问都是通过接口进行的,接口提供了一个对所有类型的硬件一致化的操作集合来处理基本数据的发送和接收。网络接口看作是一个发送和接收数据包(packets)的实体。对于每个网络接口,都用一个 device 的数据结构表示。通常,网络设备是一个物理设备,例如以太网卡;但软件也可以作为网络设备,例如回送设备(loopback)。在内核启动时,通过网络设备驱动程序,登记存在的网络设备。设备用标准的支持网络的机制来传递收到的数据到相应的网络层。所有被发送和接收的包都用数据结构 sk_buf 表示。这是一个具有很好灵活性的数据结构,可以很容易地增加或删除网络协议数据包的首部。

网络设备作为其中的 3 类设备之 1,有其非常特殊的地方。它与字符设备及块设备都有很大的不同。

(1) 网络接口不存在于 Linux 的文件系统中,而是在核心中用一个 device 数据结构表示。每一个字符设备或块设备在文件系统中都存在一个相应的特殊设备文件来表示该设备,例如/dev/hda1、/dev/sda1、/dev/tty1 等。网络设备在做数据包发送和接收时,直接通过接口访问,不需要进行文件的操作;对字符设备和块设备的访问都须通过文件操作界面。

(2) 网络接口是在系统初始化时实时生成的,对于核心支持的但不存在的物理网络设备,将不可能有与之相对应的 device 结构。对于字符设备和块设备,即使该物理设备不存在,在/dev下也有相应的特殊文件。且在系统初始化时,核心将会对所有内核支持的字符设备和块设备进行登记,初始化该设备的文件操作界面(struct file_operations),而不管该设备是否在物理上存在。

以上两点是网络设备与其他设备之间存在的最主要的不同。然而,它们之间又有一些共同之处,例如在系统中一个网络设备的角色和一个安装的块设备相似。一个块设备在 `blk_dev` 数组及核心其他的数据结构中登记自己,然后根据请求,通过自己的 `request_function` 函数发送和接收数据块。相似地,为了能与外界进行数据交流,一个网络接口也必须在一个特殊的数据结构中登记自己。在系统内核中,存在字符设备管理表 `chardevs` 和块设备管理表 `blkdevs`,这两张表保存着指向 `file_operations` 结构的指针设备管理表,分别用来描述各种字符驱动程序和块设备驱动程序。类似地,在内核中也存在着一张网络接口管理表 `dev_base`,但与前两张表不同,`dev_base`是指向 `device` 结构的指针,因为网络设备是通过 `device` 数据结构表示的。`dev_base` 实际上是一条 `device` 结构链表的表头,在系统初始化完成以后,系统检测到的网络设备将自动地保存在这张链表中,其中每一个链表单元表示一个存在的物理网络设备。当要发送数据时,网络子系统将根据系统路由表选择相应的网络接口进行数据传输;当接收到数据包时,通过驱动程序登记的中断服务程序进行数据的接收处理(软件网络接口除外)。网络设备工作原理如图 7.2 所示。

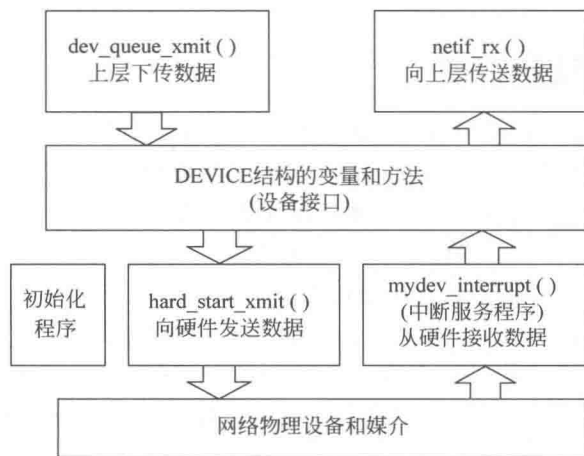


图 7.2 Linux 网络设备工作原理图

Linux 网络设备驱动程序的体系结构可以分为 4 层：网络协议接口层、网络设备接口层、设备驱动功能层和网络设备与媒介层,如图 7.2 所示。网络设备驱动程序最主要的工作是完成设备驱动功能层。在 Linux 中,所有网络设备都抽象为一个接口,这个接口提供了对所有网络设备的操作集合。由数据结构 `struct net_device` 表示网络设备在内核中的运行情况,即网络设备接口。它既包括纯软件网络设备接口,例如环路(loopback),也包括硬件网络设备接口,例如以太网卡。由以 `dev_base` 为头指针的设备链来集体管理所有网络设备,该设备链中的每个元素代表一个网络设备接口。数据结构 `net_device` 中有很多供系统访问和协议层调用的设备方法,包括初始化,打开和关闭网络设备的 `open` 和 `stop` 函数,处理数据包发送的 `hard_start_xmit` 函数,以及中断处理函数等。

当网络子系统上层有数据包要发送时,通过调用网络设备驱动中实现的 `ndo_start_xmit` 函数,将要发送的数据包封装在套接字缓冲区 `skb` 参数中。在驱动程序的发送数据包函数的具体实现中,将首先在 `skb` 数据包所在主存中的数据块和网络设备内存之间建立一个 DMA 通道,然后启动该 DMA 通道将数据包由主存传输到设备内存,之后由网络设备硬件通过网络接口或者天线将数据包发送出去。数据包发送成功后会向处理器发出一个硬件中断,在中断处理程序里做一些善后处理工作。数据包的接收是一个异步的过程,出于系统性能的考虑,绝大部分网络设备都支持数据接收中断,因此在驱动程序中是通过中断处理程序接收数据包。由于系统主存与网络设备之间已经建立好 DMA 通道,所以当有数据包到达网络设备时,数据包会被自动传输到系统主存,此时将产生一个中断信号,从而进入驱动程序的中断处理函数。在中断处理函数里驱动首先会分配一个套接字缓冲区 `skb` 来容纳收到的数据包,然后将 `skb` 传递到网络子系统的上层代码中,具体传递的过程是驱动程序通过调用 `netif_rx(skb)` 函数实现的,上层代码负责释放该 `skb` 所占用的内存。

7.2 NAPI 机制

NAPI 是 Linux 采用的一种提高网络处理效率的技术,它的核心概念就是不采用中断的方式读取数据,而代之以首先采用中断唤醒数据接收的服务程序,然后以 POLL 的方法来轮询数据。随着网络接收速度的增加,NIC 触发的中断能做到不断减少。目前 NAPI 技术已经在网卡驱动层和网络层得到广泛应用,驱动层次上已经有 E1000 系列网卡,RTL8139 系列网卡,3c50X 系列等主流的网络适配器都采用了这个技术;网络层次上,NAPI 技术已经完全应用到了著名的 `netif_rx` 函数中,并且提供了专门的 POLL 方法——`process_backlog` 来处理轮询的方法。根据实验数据表明采用 NAPI 技术可以大大提高短长度数据包接收的效率,减少中断触发的时间。

但是 NAPI 存在如下一些比较严重的缺陷:

(1) 对于上层的应用程序,系统不能在每个数据包接收到的时候都及时地去处理,而且随着传输速度增加,累计的数据包将会耗费大量的内存,经过实验表明在 Linux 平台上这个问题会比在 FreeBSD 上要严重。

(2) 另外一个问题是对于大的数据包处理比较困难,原因是大的数据包传送到网络层上耗费的时间比短数据包长很多(即使是采用 DMA 方式),所以正如前面所说,NAPI 技术适用于对高速率的短长度数据包处理。

使用 NAPI 的先决条件如下:

驱动可以继续使用老的 2.4 版内核的网络驱动程序接口,NAPI 的加入并不会导致向前兼容性的丧失,但是 NAPI 的使用至少要得到下面的保证。

(1) 要使用 DMA 的环形输入队列(也就是 `ring_dma`),或者是有足够的内存空间缓存驱动获得的包。

(2) 在发送/接收数据包产生中断的时候,有能力关断 NIC 中断的事件处理,并且在关

断 NIC 以后,并不影响数据包接收到网络设备的环形缓冲区(以下简称 rx-ring)处理队列中。

NAPI 对数据包到达事件的处理采用轮询方法,在数据包到达的时候,NAPI 就会强制执行 dev->poll 方法。而不像以前的驱动那样为了减少包到达时间的处理延迟,通常采用中断的方法来进行。

7.3 关键数据结构

在 Linux 内核中采用一个 net_struct 的实例来表示一个网络设备,这其中包括了虚拟网络设备和实际网络设备。该数据结构比较复杂,主要任务分为两部分:第 1 对上层协议屏蔽底层设备的区别,提供统一的操作接口;第 2 对下层设备,提供实际驱动方法。网络设备驱动程序只需要填充 net_device 的具体成员,并注册 net_device 即可实现硬件操作函数与内核的挂接。net_device 结构体中包括的比较重要的结构体及其关系如图 7.3 所示。

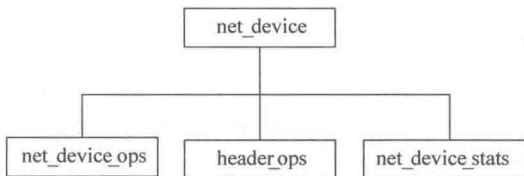


图 7.3 几个重要结构体关系图

1. net_device 结构体

由于 net_device 结构体太过庞大,这里只选取了部分成员变量加以注释。

```

struct net_device {
    char name[IFNAMSIZ];           //设备名字
    unsigned long state;           //设备状态
    struct net_device * next;      //全局列表中指向下一个设备的指针
    unsigned long mem_end;         //设备共享内存的起始和结束地址
    unsigned long mem_start;
    unsigned long base_addr;       //网络设备 I/O 基地址
    unsigned int irq;              //设备使用的中断号
    unsigned char if_port;         //指定多端口设备使用哪个端口 unsigned char dma;
                                   //分配给该设备的 DMA 通道
    unsigned short hard_header_len; //硬件头部长度
    unsigned mtu;                  //最大传输单元 (MTU)
    unsigned long tx_queue_len;    //设备发送队列中可以排队的最大帧数
    unsigned short type;           //接口的硬件类型
    unsigned short flags;          //接口标志
    int features;
    struct net_device_stats stats; //网络设备统计状态结构体
    const struct net_device_ops * netdev_ops; //操作函数结构体

```



```

const struct header_ops * header_ops;           //包头操作函数结构体
...
}

```

2. net_device_ops 结构体

```

struct net_device_ops
{
    int (*ndo_init)(struct net_device *dev); //初始化函数,在注册设备调用
    int (*ndo_open)(struct net_device *dev);
    //打开网络接口设备,获得设备需要的 I/O 地址,IRQ,DMA 通道等
    int (*ndo_stop)(struct net_device *dev); //关闭网络设备
    netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb, struct net_device *dev);
    //驱动数据包发送,参数是 sk_buff,使得驱动程序获取从上层传递下来的数据包
    void (*ndo_tx_timeout)(struct net_device *dev); //当数据包发送超时,该函数会被
                                                    //调用,该函数需重新启动数据包发送过程,
                                                    //或重新启动硬件等恢复网络到正常状态
}

```

3. header_ops 结构体

```

struct header_ops
{
    int (*create)(struct sk_buff *skb, struct net_device *dev, unsigned short type,
                  const void *daddr, const void *saddr, unsigned int len);
    int (*parse)(const struct sk_buff *skb, unsigned char *haddr);
    //从包含在 skb 中的报文中抽取源地址,复制到 haddr 的缓存区
    int (*rebuild)(struct sk_buff *skb);
    //用来在 ARP 解析完成后,但是在报文发送前重建硬件头的函数
    int (*cache)(const struct neighbour *neigh, struct hh_cache *hh, __be16 type);
    //被调用来填充 hh_cache 结构,使用一个 ARP 请求的结果
    void (*cache_update)(struct hh_cache *hh, const struct net_device *dev,
                        const unsigned char *haddr);
}

```

4. net_device_stats 结构体

```

struct net_device_stats
{
    unsigned long rx_packets; //收到的总数据包数
    unsigned long tx_packets; //发送的总数据包数
    unsigned long rx_bytes; //收到总字节数
    unsigned long tx_bytes; //发送总字节数
    unsigned long rx_errors; //收到的错误数据包数
    unsigned long tx_errors; //发送出错数
    unsigned long rx_dropped; //缓冲溢出
    unsigned long tx_dropped; //发送缓冲溢出
    unsigned long multicast; //收到多个发送包
}

```

```

    unsigned long collisions;
    /* 详细的接收出错信息: */
    unsigned long rx_length_errors;
    unsigned long rx_over_errors;           //接受缓冲溢出
    unsigned long rx_crc_errors;           // CRC 错误
    unsigned long rx_frame_errors;
    unsigned long rx_fifo_errors;          // fifo 出错
    unsigned long rx_missed_errors;        //收到丢失的数据包
    /* 详细的发送出错信息 */
    unsigned long tx_aborted_errors;
    unsigned long tx_carrier_errors;
    ...
};

```

7.4 内核提供的网络设备驱动设计函数

7.4.1 alloc_netdev

net_device 结构, 包含一个 kobject, 因此它可被引用计数并通过 sysfs 输出。它必须动态分配, 进行这种分配的内核函数是 alloc_netdev, 它有下列原型:

```

struct net_device * alloc_netdev(int sizeof_priv, const char * name, void (* setup)(struct
net_device * ));

```

sizeof_priv 是驱动的私有数据区的大小; 对于网络驱动, 这个是同 net_device 结构一起分配的。实际上, 这两个是在一个大内存块中一起分配的, 但是驱动作者应当假装不知道这一点。name 是这个接口的名字, 如同用户空间看到的一样, 这个名字可以有一个 printf 风格的 %d 在里面。内核用下一个可用的接口号来替换这个 %d。最后, setup 是一个初始化函数的指针, 被调用来设置 net_device 结构的剩余部分。

7.4.2 register_netdev

该函数实现对网络接口的登记功能。其实现步骤如下:

- (1) 检查设备名是否已确定, 若没赋值则以以太网设备待之并给它一个默认的值 ethN, N 为最小的可用以太网设备号。
- (2) 网络设备自己的 init_function, 即刚在 init_module 中赋值的 dev->init 被调用, 用来实现对网络接口的实际初始化工作。
- (3) 若初始化成功, 则将该网络接口加到网络设备管理表 dev_base 的尾部。

7.4.3 ether_setup

ether_setup 是一个通用于以太网接口的网络接口设置函数。由于以太网卡有很好的

共性,device 结构中许多有关的网络接口信息都是通过调用 `ether_setup` 函数来统一设置。若满意这些默认设置,在写驱动程序时只要调用一下这个函数就可以将这些域段的设置工作置之不理,也可在调用该函数之后再进行修改。

7.4.4 unregister_netdev

模块卸载函数很简单,从系统中去除接口,释放 `net_device` 结构回系统。

7.5 网络设备驱动开发实例

本实例设计的网络驱动程序 `snull` 创建了两个接口,这两个接口能够实现网络回环的作用,无论通过接口发送什么都会回环到另一个接口。其连接方式如图 7.4 所示。

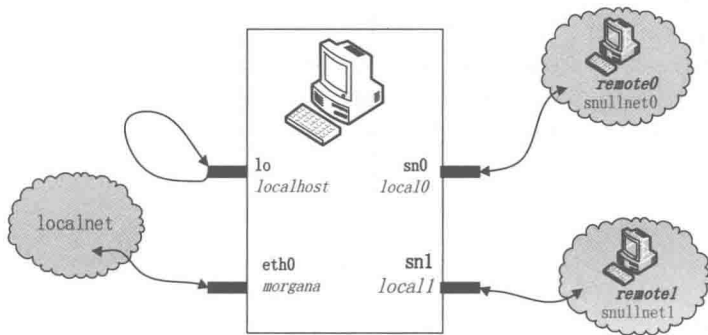


图 7.4 网络回环

网络设备驱动程序设计主要是完成对 `net_device` 结构体的分配,初始化及注册。在初始化阶段的函数调用关系如图 7.5 所示。

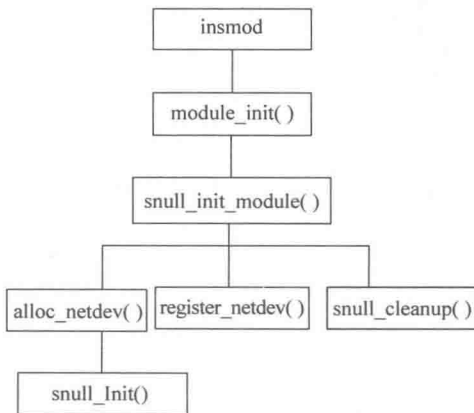


图 7.5 驱动模块初始化阶段函数调用关系

7.5.1 snull_init_module 函数

snull_init_module 函数应用举例如下：

```
int snull_init_module(void)
{
    int result, i, ret = -EONMEM;
    snull_interrupt = use_napi ? snull_napi_interrupt : snull_regular_interrupt;
    /* 分配设备 */
    snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
    snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
    if(snull_devs[0] == NULL || snull_devs[1] == NULL)
        goto out;
    ret = -ENODEV;
    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
            ret = 0;
    out:
    if(ret)
        snull_cleanup();
    return ret;
}
```

该函数申请了两个 net_device 结构体，设置初始化这个结构体的函数指针。初始化完成后将这两个结构体注册到系统内核，最后设置注销函数。

7.5.2 snull_init 函数

snull_init 函数应用举例如下：

```
void snull_init(struct net_device *dev)
{
    struct snull_priv *priv;                                //初始化默认值
    ether_setup(dev);
    dev->netdev_ops = &snull_dev_ops;
    dev->header_ops = &snull_header_ops;
    dev->watchdog_time0 = timeout;
    /* 不适用 ARP 协议 */
    dev->flags |= IFF_NOARP;
    dev->features |= NETIF_F_NO_CSUM;
    /* 初始化 priv 结构体 */
    priv = netdev_priv(dev);
```

```

memset(priv,0,sizeof(struct snull_priv));
priv->dev = dev;
spin_lock_init(&priv->lock);           //使能接收中断
snull_rx_ints(dev);
if (use_napi) {
    netif_napi_add(dev,&priv->napi,snull_poll,2);
}
}

```

该函数首先调用 `ether_setup(dev)` 函数,初始化 `net_device` 结构体的一些默认值,然后通过初始化相关的操作函数,并使能接受中断。

7.5.3 相关操作函数

操作函数的初始化主要通过两个结构体来完成,其代码如下。其完成的工作主要是将相关函数的指针指向用户编写的驱动函数。

```

static const struct net_device_ops snull_dev_ops = {
    .ndo_open = snull_open,
    .ndo_stop = snull_release,
    .ndo_set_config = snull_config,
    .ndo_start_xmit = snull_tx,
    .ndo_do_ioctl = snull_ioctl,
    .ndo_get_stats = snull_stats,
    .ndo_change_mtu = snull_change_mtu,
    .ndo_tx_timeout = snull_tx_out,
};

```

1. 打开/关闭操作

```

int snull_open(struct net_device *dev)
{
    memcpy(dev->dev_addr,"\0SNUL0",ETH_ALEN);
    if(dev == snull_devs[1])
        dev->dev_addr[ETH_ALEN-1]++;
    netif_start_queue(dev);
    return 0;
}

int snull_release(struct net_device *dev)
{
    netif_stop_queue(dev);
    return 0;
}

```

打开接口,在 `ifconfig` 激活接口时,接口被打开。`open` 方法应当注册它需要的任何系统资源(I/O 接口,IRQ,DMA,等等),打开硬件,并对设备进行其他所需的设置。在缺乏真实硬件的情况下,`open` 函数要做的事情很少。`stop` 方法同样,它只是 `open` 的反操作。因此,

实现 stop 的函数常常称为 close 或者 release。

2. 发送数据包操作

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
    int len;
    char *data;
    struct snull_priv *priv = netdev_priv(dev);
    len = skb->len < ETH_ZLEN ? ETH_ZLEN : skb->len;
    data = skb->data;
    dev->trans_start = jiffies;
    priv->skb = skb;
    snull_hw_tx(data, len, dev);
    return 0;
}
```

传送指的是通过网络连接发送一个报文的行为。无论何时内核需要传送一个数据报文,它调用驱动的 hard_start_transmit 方法将数据放在外出队列上。每个内核处理的报文都包含在一个 socket 缓存结构(结构 sk_buff)中。socket 缓存是一个复杂的结构,内核提供了一些函数来操作它,一个基本的 sk_buff 就足够编写一个能工作的驱动。

3. 接受数据包操作

从网络上接收报文比发送报文要难一些,因为必须分配一个 sk_buff 并从一个原子性上下文中递交给上层。网络驱动可以实现两种报文接收的模式:中断驱动和查询,大部分采用中断驱动技术。

snull 的实现将硬件细节从设备独立的常规事务中分离。因此,函数 snull_rx 在硬件收到报文后,从 snull 的中断处理中调用,并且报文现在已经在计算机的内存中。snull_rx 收到一个数据指针和报文长度,它的责任是发走这个报文和运行附加信息给上层的网络代码。这个代码独立于获得数据指针和长度的方式。

```
void snull_rx(struct net_device *dev, int len, unsigned char *buf)
{
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);
    skb = dev_alloc_skb(len+2); //分配一个保存数据包的缓冲区
    if (!skb)
    {
        printk("snull rx: low on mem - packet dropped\n");
        priv->stats.rx_dropped++;
        return;
    }
    skb_reserve(skb, 2);
    memcpy(skb_put(skb, len), buf, len);
    //skb_put 函数刷新缓冲区内的数据末尾指针,并且返回新建数据区的指针
    skb->dev = dev;
```

```

skb->protocol = eth_type_trans(skb, dev);
skb->ip_summed = CHECKSUM_UNNECESSARY;
priv->stats.rx_packets++;
priv->stats.rx_bytes += len;
netif_rx(skb);           //将套接字缓冲区传递给上层软件处理
return;
}

```

4. 中断处理流程

大多数硬件接口通过中断处理流来控制。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。

中断处理的第一个任务是取一个指向正确 net_device 结构的指针。这个指针通常来自作为参数收到的 dev_id 指针。中断处理发送结束的情况下,统计量被更新,调用 dev_kfree_skb 来返回 socket 缓存给系统。

```

static void snull_regular_interrupt(int irq, void * dev_id, struct pt_regs * regs) {
    int statusword;
    struct snull_priv * priv;
    struct snull_packet * pkt = NULL;
    /* 根据设备号获取设备指针 */
    struct net_device * dev = (struct net_device *) dev_id;
    if (!dev) return;
    /* 锁住该设备 */
    priv = netdev_priv(dev);
    spin_lock(&priv->lock);
    /* 接收状态信息 */
    statusword = priv->status;
    priv->status = 0;
    if (statusword & SNULL_RX_INTR) {
        /* 发送给 snull_rx 处理 */
        pkt = priv->rx_queue;
        if (pkt) {
            priv->rx_queue = pkt->next;
            snull_rx(dev, pkt);
        }
    }
    if (statusword & SNULL_TX_INTR) {
        /* 发送完毕,释放 skb */
        priv->stats.tx_packets++;
        priv->stats.tx_bytes += priv->tx_packetlen;
        dev_kfree_skb(priv->skb);
    }
    /* 接收完成释放该设备 */
    spin_unlock(&priv->lock);
}

```

```

        if (pkt) snull_release_buffer(pkt);
    return;
}

```

5. 轮询处理流程

当一个网络驱动如上面所述编写出来,接口收到每个报文都中断处理器。在大多情况下,这是希望的操作模式。然而,高带宽接口能够在每秒内收到几千个报文。这个中断负载下,系统的整体性能会受影响。

作为一个高端 Linux 系统性能的方法,网络子系统开发者已创建了一种可选的基于轮询的接口(称为 NAPI)。在驱动开发者看来,轮询的名声并不好,轮询常常视为一种低效和无能的技术。但是,轮询的低效率仅表现在接口没有事情可做时。当系统有一个处理大流量的高速接口时,每个时刻都有大量的数据包需要处理。在这种情况下,没有必要中断处理器,使用轮询技术处理到达接口的每一个数据包就足够了。

```

static int snull_poll(struct napi_struct *napi, int work_limit)
{
    int nworked = 0;
    struct snull_priv *priv = netdev_priv(napi->dev);
    struct sk_buff *skb;
    struct snull_packet *pkt;
    unsigned long flags;
    printk("snullnet:work_limit = %d\n", work_limit);
    while(nworked < work_limit && priv->rx_queue) {
        spin_lock_irqsave(&priv->lock, flags);
        pkt = priv->rx_queue;
        priv->rx_queue = priv->rx_queue->next;
        spin_unlock_irqrestore(&priv->lock, flags);
        skb = dev_alloc_skb(pkt->datalen + 2);
        if (!skb) {
            /* 没有分配到内存 */
            priv->stats.rx_dropped++;
            snull_release_buffer(pkt);
            continue;
        }
        skb_reserve(skb, 2);
        memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
        skb->dev = napi->dev;
        skb->protocol = eth_type_trans(skb, napi->dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY;
        netif_receive_skb(skb);
        nworked++;
        priv->stats.rx_packets++;
        priv->stats.rx_bytes += pkt->datalen;
        snull_release_buffer(pkt);
    }
}

```


函数的核心部分是创建包含数据包的 `skb`；这部分代码和之前的 `snuff_rx` 一样。但也存在如下一些区别：

- (1) `budget` 参数提供了一个允许传给内核的最大数据包数。
- (2) 数据包应当用 `netif_receive_skb` 递交给内核，而不是 `netif_rx`。
- (3) 如果 `poll` 方法能够在限制内处理所有的报文，它应当重新能接收中断，调用 `netif_rx_complete` 来关闭轮询函数，并且返回 0。如果返回值为 1，表示数据包仍在被处理状态。

MMC 卡即多媒体卡(MultiMedia Card),是一种非易失性存储器件,具有体积小、容量大、耗电量低、传输速度快等特点。正因这些特点,它广泛应用于消费类电子产品中。

SD 卡即安全数码卡(Secure Digital Memory Card),在 MMC 卡的基础上发展而来,但具有 MMC 不具备的安全性和更快传输速度。SD 卡可以设定存储内容的使用权限,防止数据被他人复制;SD 卡在大小尺寸上比 MMC 卡更厚。SD 卡向前兼容 MMC 卡,就是所有支持 SD 卡的设备都支持 MMC 卡。

8.1 MMC 子系统基本架构

MMC 子系统的实现在 kernel/driver/mmc 目录下,当前 MMC 子系统不仅支持 MMC/SD 卡,也支持 SDIO 等记忆卡,图 8.1 是 MMC 子系统的结构图。

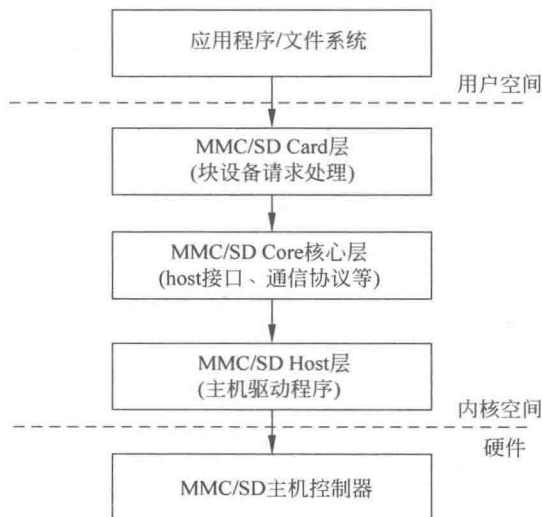


图 8.1 MMC 子系统结构图

结构图清晰显示了 MMC 子系统各层级的关系。HOST 层提供针对不同主机的驱动程序,根据特定平台完成。CORE 层是整个 MMC 的核心层,该层完成不同协议和规范的实现,并且为 HOST 层的驱动提供接口函数。CARD 层的功能是对一些请求进行处理,因为记忆卡都是块设备,系统需要提供块设备的驱动程序,该层就是将 SD 卡如何实现为块设备。

8.2 关键数据结构

8.2.1 基本数据结构

基本数据结构如下:

(1) struct mmc_host,用来描述卡控制器,该类里面的成员是所有 SD 卡控制器都需要的,放之四海而皆准的数据结构,位于 kernel/include/linux/mmc/host.h 下面。

(2) struct mmc_card,用来描述卡,位于 kernel/include/linux/mmc/card.h 下面。

(3) struct mmc_driver,用来描述 MMC 卡驱动,位于 rnel/include/linux/mmc/card.h 下面。

(4) struct mmc_host_ops,用来描述卡控制器操作集,用于从主机控制器向 CORE 层注册操作函数,从而将 CORE 层与具体的主机控制器隔离。CORE 要操作主机控制器,就是这个 ops 中给的函数指针操作,不能直接调用具体主控制器的函数。位于 kernel/include/linux/mmc/host.h 下面。

(5) struct mmc_ios,用于描述控制器对卡的 I/O 状态。位于 kernel/include/linux/mmc/host.h 下面。

(6) struct mmc_request,用于描述读写 MMC 卡的请求,包括命令,数据及请求完成后的回调函数。位于 kernel/include/linux/mmc/core.h 中。

(7) struct mmc_queue,是 MMC 卡的请求队列结构,封装了通用请求队列结构,加入了 MMC 卡相关结构。位于 kernel/drivers/mmc/card/queue.h 中。

(8) struct mmc_data,描述了 MMC 卡读写的数据相关信息,例如请求,操作命令,数据及状态等。位于 kernel/include/linux/mmc/core.h 中。

(9) struct mmc_command,描述了 MMC 卡操作相关命令及数据,状态信息等。位于 kernel/include/linux/mmc/core.h 中。

8.2.2 基本数据结构主要成员及关系

Card 控制器相关数据结构如下:

```
struct mmc_host {
```

```

struct device      * parent;
struct device      class_dev;
int                index;
const struct mmc_host_ops * ops;           //主控制器的操作函数,
                                           //即该控制器所具备的驱动能力

unsigned int       f_min;
unsigned int       f_max;
unsigned int       f_init;
...

struct mmc_ios     ios;                   //配置时钟、总线、电源、片选、时序等
struct mmc_card    * card;                //连接到此主控制器的 SD 卡设备
...
}

struct mmc_host_ops {
    int (* enable)(struct mmc_host * host);
    int (* disable)(struct mmc_host * host);
    void (* post_req)(struct mmc_host * host, struct mmc_request * req, int err);
    void (* pre_req)(struct mmc_host * host, struct mmc_request * req, bool is_first_req);
    /* 核心函数,完成主控制器与 SD 卡之间的数据通信 */
    void (* request)(struct mmc_host * host, struct mmc_request * req);
    void (* set_ios)(struct mmc_host * host, struct mmc_ios * ios);
    int (* get_ro)(struct mmc_host * host);
    int (* get_cd)(struct mmc_host * host);
    void (* enable_sdio_irq)(struct mmc_host * host, int enable);
    void (* init_card)(struct mmc_host * host, struct mmc_card * card);
    int (* start_signal_voltage_switch)(struct mmc_host * host, struct mmc_ios * ios);
    int (* execute_tuning)(struct mmc_host * host, u32 opcode);
    void (* enable_preset_value)(struct mmc_host * host, bool enable);
    int (* select_drive_strength)(unsigned int max_dtr, int host_drv, int card_drv);
    void (* hw_reset)(struct mmc_host * host);
    void (* card_event)(struct mmc_host * host);
}

struct mmc_card {
    struct mmc_host * host;                /* 描述 MM 卡所属的主控制器 */
    struct device dev;                     /* 描述该设备 */
    unsigned int rca;                      /* 卡的相对地址 */
    unsigned int type;                     /* 描述卡的类型 */
    ...

    u32 raw_cid[4];                       /* CID 寄存器 */
    u32 raw_csd[4];                       /* CSD 寄存器 */
    u32 raw_scr[2];                       /* SCR 寄存器 */
    struct mmc_cid cid;                    /* 卡 ID */
    struct mmc_csd csd;                    /* 与 raw-csd 对应,描述 E 的特征 */
}

```

```

struct mmc_ext_csd    ext_csd;           /* MMC v4 扩展卡特征 */
struct sd_scr         scr;               /* SD 卡的附加信息 */
struct sd_ssr         ssr;               /* 描述卡其他描述 */
struct sd_switch_caps sw_caps;           /* 完成 caps 的交换 */

unsigned int          sdio_funcs;        /* SDIO 功能的个数 */
struct sdio_cccr       cccr;              /* 描述卡的常见信息 */
struct sdio_cis         cis;              /* 常见元组信息 */
struct sdio_func        * sdio_func[SDIO_MAX_FUNCS]; /* SDIO 实现的驱动功能 */
struct sdio_func        * sdio_single_irq; /* 描述 SDIO 响应单中断的功能 */
...
}

```

在 Card 控制器中各成员之间的关系如图 8.2 所示。

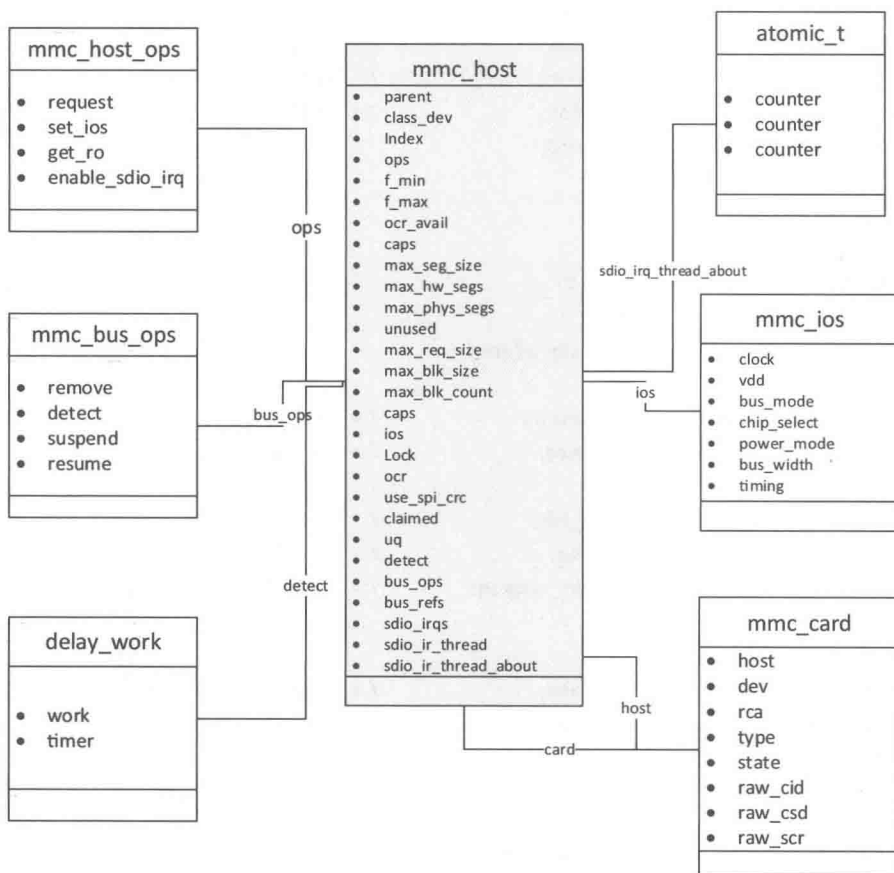


图 8.2 Card 控制器各成员之间关系

MMC 卡的请求相关数据结构如下：

```

struct mmc_command {
    u32                opcode;
    u32                arg;
    u32                resp[4];
    unsigned int       flags;           /* 用于标记期望的响应类型 */
    ...
    unsigned int       cmd_timeout_ms; /* 请求超时设置(单位 ms) */
    struct mmc_data     * data;        /* 该命令所需的数据 */
    struct mmc_request  * mrq;        /* 该命令涉及的请求 */
};

struct mmc_data {
    unsigned int       timeout_ns;     /* 数据传输超时(单位 ns 且阈值为 80ms) */
    unsigned int       timeout_clks;   /* 超时设置(单位是时钟) */
    unsigned int       blksz;         /* 数据块大小 */
    unsigned int       blocks;        /* 数据所占块数 */
    unsigned int       error;         /* 数据错误标记 */
    unsigned int       flags;
};

#define MMC_DATA_WRITE (1 << 8)
#define MMC_DATA_READ  (1 << 9)
#define MMC_DATA_STREAM (1 << 10)

    unsigned int       bytes_xfered;

    struct mmc_command * stop;        /* 终止命令 */
    struct mmc_request  * mrq;        /* 相关请求信息 */

    unsigned int       sg_len;        /* 散列表长度 */
    struct scatterlist  * sg;         /* I/O 散列表 */
    s32                host_cookie;   /* 主控制器缓存,为私有信息 */
};

struct mmc_request {
    struct mmc_command  * sbc;        /* 修改 SET_BLOCK_COUNT 以用于多块 */
    struct mmc_command  * cmd;
    struct mmc_data     * data;
    struct mmc_command  * stop;

    struct completion    completion;
    void (* done)(struct mmc_request *); /* 请求的回调函数 */
};

```

各数据结构之间的相互关系如图 8.3 所示。

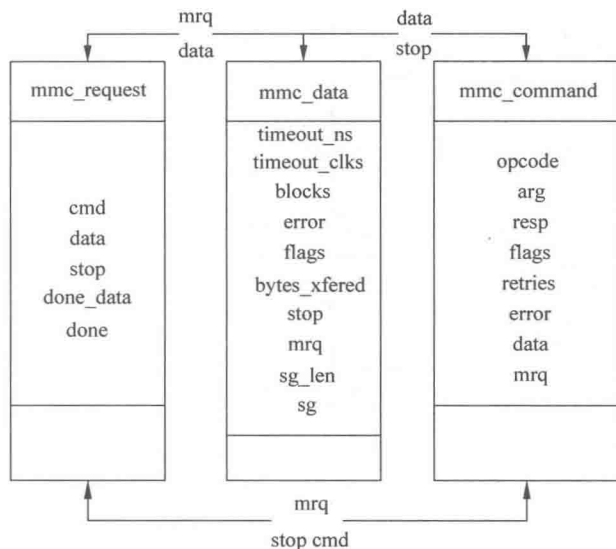


图 8.3 MMC 卡请求相关数据结构成员间关系

8.3 MMC/CD 卡驱动实例

8.3.1 MMC/SD 卡设备驱动设计场景

MMC 子系统代码主要在 `drivers/mmc` 目录下,共有 3 个目录。Card: 存放闪存卡(块设备)的相关驱动,例如 MMC/SD 卡设备驱动,SDIOUART; Host: 针对不同主机端的 SDHC、MMC 控制器的驱动; Core: 整个 MMC 的核心层,完成不同协议和规范的实现,为 host 层和设备驱动层提供接口函数。根据 MMC/SD 协议,MMC/SD 卡的驱动分为:卡识别阶段和数据传输阶段。卡识别阶段通过命令使 MMC/SD 处于:空闲(idle)、准备(ready)、识别(ident)、等待(stby)和不活动(ina)等不同的状态;数据传输阶段通过命令使 MMC/SD 处于:发送(data)、传输(tran)、接收(rcv)、程序(prg)和断开连接(dis)等不同的状态。所以总结 MMC/SD 在工作的整个过程中分为两个阶段的 10 种状态。下面如图 8.4 和 8.5 所示两个阶段 10 种状态之间的转换关系。

卡识别阶段,如图 8.4 所示。

数据传输阶段,如图 8.5 所示。

MMC/SD 卡驱动的整体架构共由 3 个文件组成,主要完成的任务是卡的检测及数据的读写。

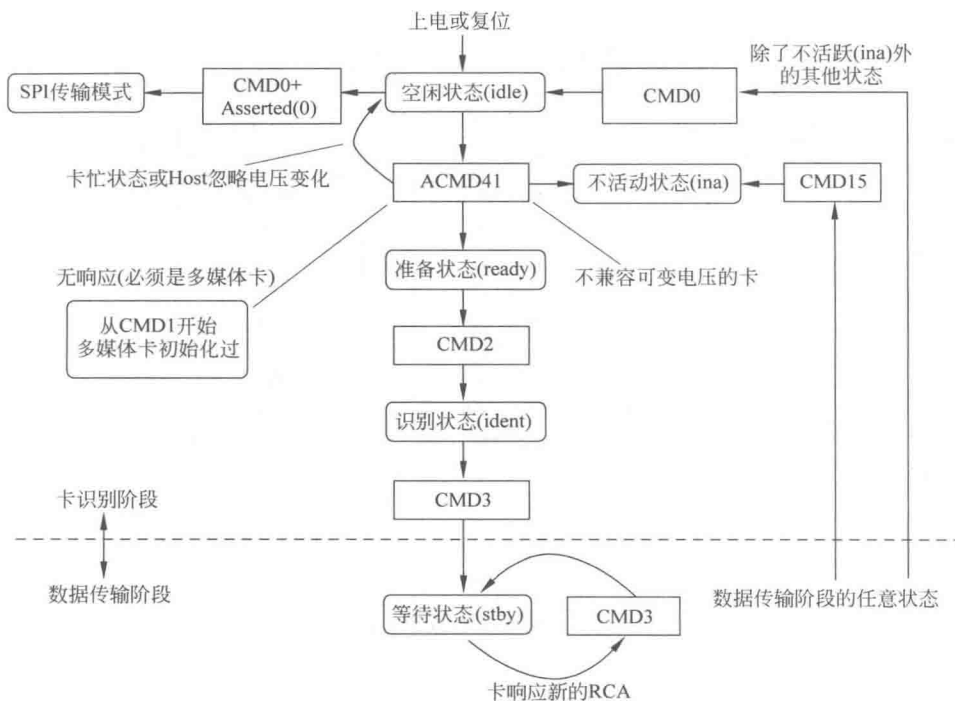


图 8.4 卡识别阶段

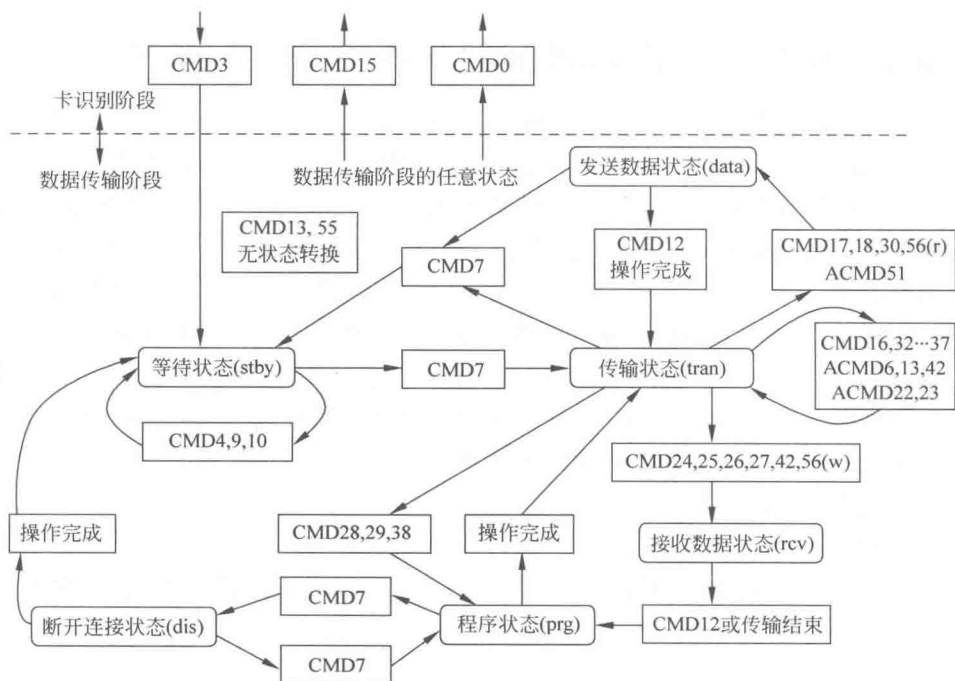


图 8.5 数据传输阶段

8.3.2 MMC/SD 卡设备驱动实例实现

系统初始化时扫描 platform 总线上是否有该 SD 主控制器名字为 pxa2xx-mci 的设备, 如果有, 驱动程序将主控制器挂载到 platform 总线上, 并注册该驱动程序。

```
static int __init pxamci_init(void)
{
    // 注册驱动
    return platform_driver_register(&pxamci_driver);
}

static struct platform_driver pxamci_driver = {
    .probe = pxamci_probe,
    .remove = pxamci_remove,
    .suspend = pxamci_suspend,
    .resume = pxamci_resume,
    .driver = {
        .name = "pxa2xx-mci",
        .owner = THIS_MODULE,
    },
};
```

其中, remove 为 probe 的反操作; suspend 和 resume 涉及电源管理的内容; probe 是最核心的函数。SD 主控制器驱动程序的初始化函数 probe(struct platform_device *pdev), 主要完成如下五大任务:

- (1) 初始化设备的数据结构, 并将数据挂载到 pdev->dev.driver_data 下;
- (2) 实现设备驱动的功能函数, 例如 mmc->ops = &pxamci_ops;
- (3) 申请中断函数 request_irq();
- (4) 注册设备, 即注册 kobject, 建立 sys 文件, 发送 uevent 等;
- (5) 其他需求, 例如在 /proc/driver 下建立用户交互文件等。

下面进行详细介绍。

1. 注册设备

对于设备的注册, 所有设备驱动的相关代码都类似。

```
static int pxamci_probe(struct platform_device *pdev)
{
    mmc = mmc_alloc_host(sizeof(struct pxamci_host), &pdev->dev);
    mmc_add_host(mmc);
    ...
}
```

这两个函数都由 /drivers/mmc/core 核心层下的 host.c 负责具体实现。

2. 设备初始化

其实, 整个设备驱动的 probe() 函数, 本质就是为设备建立起数据结构并对其赋初值。

pxamci_probe(struct platform_device * pdev)主要为SD主控制器完成时钟、存储等方面的初始化配置。

```
static int pxamci_probe(struct platform_device * pdev)
{
    struct mmc_host * mmc;
    struct pxamci_host * host = NULL;
    mmc->ops = &pxamci_ops;
    mmc->max_phys_segs = NR_SG;
    mmc->max_hw_segs = NR_SG;
    mmc->max_seg_size = PAGE_SIZE;
    host = mmc_priv(mmc);
    host->mmc = mmc;
    host->dma = -1;
    host->pdata = pdev->dev.platform_data;
    host->clkrt = CLKRT_OFF;
    host->clk = clk_get(&pdev->dev, "MMCCLK");
    host->clkrate = clk_get_rate(host->clk);
    mmc->caps |= MMC_CAP_MMC_HIGHSPEED | MMC_CAP_SD_HIGHSPEED;
    host->sg_cpu = dma_alloc_coherent(&pdev->dev, PAGE_SIZE, &host->sg_dma, GFP_KERNEL);
    host->dma = pxa_request_dma(DRIVER_NAME, DMA_PRIO_LOW, pxamci_dma_irq, host);
    r = platform_get_resource(pdev, IORESOURCE_MEM, 0); // 得到控制器芯片的起始地址
    r = request_mem_region(r->start, SZ_4K, DRIVER_NAME); // 为芯片申请 4K 的内存空间
    irq = platform_get_irq(pdev, 0); // 得到芯片的中断号
    host->res = r;
    host->irq = irq;
    host->base = ioremap(r->start, SZ_4K); // 将芯片的物理地址映射为虚拟地址
    ...
}
```

3. 设备驱动的功能函数

一般情况下,设备驱动里都有一个行为函数结构体,例如字符设备驱动里的 struct file_operations * fops; 该结构描述了设备所具备的工作能力,例如 open, read, write 等。SD主控制器驱动程序里也有一个类似的结构 struct mmc_host_ops * ops,描述了该控制器所具备驱动的能力。

```
static int pxamci_probe(struct platform_device * pdev)
{
    mmc->ops = &pxamci_ops;
    ...
}
static const struct mmc_host_ops pxamci_ops = {
    .request = pxamci_request,
    .get_ro = pxamci_get_ro,
    .set_ios = pxamci_set_ios,
    .enable_sdio_irq = pxamci_enable_sdio_irq,
};
```

其中, * set_ios 为主控制器设置总线和时钟等配置, * get_ro 得到只读属性, * enable_sdio_irq 开启 sdio 中断。* request 这个回调函数是整个 SD 主控制器驱动的核心, 实现了 SD 主控制器与 SD 卡进行通信的能力。

```
static void pxamci_request(struct mmc_host * mmc, struct mmc_request * mrq)
{
    struct pxamci_host * host = mmc_priv(mmc); unsigned int cmdat;

    set_mmc_cken(host, 1);
    host->mrq = mrq;
    cmdat = host->cmdat;
    host->cmdat &= ~CMDAT_INIT;
    if (mrq->data) {
        pxamci_setup_data(host, mrq->data);
        cmdat &= ~CMDAT_BUSY;
        cmdat |= CMDAT_DATAEN | CMDAT_DMAEN;
        if (mrq->data->flags & MMC_DATA_WRITE)
            cmdat |= CMDAT_WRITE;
        if (mrq->data->flags & MMC_DATA_STREAM)
            cmdat |= CMDAT_STREAM;
    }
    pxamci_start_cmd(host, mrq->cmd, cmdat);
}
```

其中, pxamci_setup_data() 实现数据传输, pxamci_start_cmd() 实现指令传输。

```
static void pxamci_setup_data(struct pxamci_host * host, struct mmc_data * data)
{
    host->data = data;
    writel(data->blocks, host->base + MMC_NOB); // 设置块的数量
    writel(data->blksz, host->base + MMC_BLKLEN); // 设置一个块的大小, 一般为 512B
    if (data->flags & MMC_DATA_READ) {
        host->dma_dir = DMA_FROM_DEVICE;
        dcmd = DCMD_INCTRGADDR | DCMD_FLOWTRG;
        DRCMR(host->dma_drcmrtx) = 0;
        DRCMR(host->dma_drcmrrx) = host->dma | DRCMR_MAPVLD;
    } else {
        host->dma_dir = DMA_TO_DEVICE;
        dcmd = DCMD_INCSRCADDR | DCMD_FLOWSRC;
        DRCMR(host->dma_drcmrrx) = 0;
        DRCMR(host->dma_drcmrtx) = host->dma | DRCMR_MAPVLD;
    }
    dcmd |= DCMD_BURST32 | DCMD_WIDTH1;
    host->dma_len = dma_map_sg(mmc_dev(host->mmc), data->sg, data->sg_len, host->dma_dir);
    for (i = 0; i < host->dma_len; i++) {
        unsigned int length = sg_dma_len(&data->sg[i]);
    }
}
```

```

        host->sg_cpu[i].dcmd = dcmd | length;
    if (length & 31)
        host->sg_cpu[i].dcmd |= DCMD_ENDIRQEN;
    if (data->flags & MMC_DATA_READ) {
        host->sg_cpu[i].dsadr = host->res->start + MMC_RXFIFO;
        host->sg_cpu[i].dtadr = sg_dma_address(&data->sg[i]);
    } else {
        host->sg_cpu[i].dsadr = sg_dma_address(&data->sg[i]);
        host->sg_cpu[i].dtadr = host->res->start + MMC_TXFIFO;
    }
    host->sg_cpu[i].ddadr = host->sg_dma + (i + 1) * sizeof(struct pxa_dma_desc);
}

host->sg_cpu[host->dma_len - 1].ddadr = DDADR_STOP;
wmb();
DDADR(host->dma) = host->sg_dma;
DCSR(host->dma) = DCSR_RUN;
}

```

for()循环里的内容是整个 SD 卡主控制器设备驱动的实质,通过 DMA 的方式实现主控制器与 SD 卡之间数据的读写操作。

```

static void pxamci_start_cmd(struct pxamci_host * host, struct mmc_command * cmd, unsigned
int cmdat)
{
    host->cmd = cmd;
    if (cmd->flags & MMC_RSP_BUSY)
        cmdat |= CMDAT_BUSY;
    #define RSP_TYPE(x) ((x) & ~(MMC_RSP_BUSY|MMC_RSP_OPCODE))
    switch (RSP_TYPE(mmc_resp_type(cmd))) {
        case RSP_TYPE(MMC_RSP_R1): /* r1, r1b, r6, r7 */
            cmdat |= CMDAT_RESP_SHORT;
            break;
        case RSP_TYPE(MMC_RSP_R3):
            cmdat |= CMDAT_RESP_R3;
            break;
        case RSP_TYPE(MMC_RSP_R2):
            cmdat |= CMDAT_RESP_R2;
            break;
        default: break;
    }
    writel(cmd->opcode, host->base + MMC_CMD);
    writel(cmd->arg >> 16, host->base + MMC_ARGH);
    writel(cmd->arg & 0xffff, host->base + MMC_ARGL);
    writel(cmdat, host->base + MMC_CMDAT);
    pxamci_start_clock(host);
    pxamci_enable_irq(host, END_CMD_RES);
}

```

根据 SD 卡的协议,当 SD 卡收到从控制器发来的 cmd 指令,SD 卡会发出 response 响应,response 的类型分为 R1、R1b、R2、R3、R6、R7,这些类型分别对应不同的指令,各自的数据包结构也不同(具体内容参考 SD 卡协议)。通过 RSP_TYPE 对指令 cmd 的 opcode 解析得到对应的 response 类型,再通过 switch 赋给寄存器 MMC_CMDAT 对应的[1:0]位。以上程序中那 4 行 writel()语言是整个 SD 卡主控制器设备驱动的实质,通过对主控制器芯片寄存器 MMC_CMD,MMC_ARGH,MMC_ARGL,MMC_CMDAT 的设置,实现主控制器发送指令到 SD 卡的功能。

以上通过 pxamci_request()实现了主控制器的通信功能,之后只须通过 host->ops->request(host, mrq);回调函数即可。协议层里,每条指令都会通过 mmc_wait_for_req(host, &mrq)调用到 host->ops->request(host, mrq)来发送指令和数据。

4. 中断实现

pxamci_probe(struct platform_device *pdev)中有两个中断,一个为 SD 主控制器芯片内电路固有的内部中断,另一个为探测引脚探测到外部有 SD 卡插拔引起的中断。

由主控芯片内部电路引起的中断 request_irq(host->irq, pxamci_irq, 0, "pxa2xx-mci", host);host->irq 就是从 platform_device 里得到的中断号,irq = platform_get_irq(pdev, 0);host->irq = irq;pxamci_irq 便是为该中断 host->irq 申请的中断函数。

```
static irqreturn_t pxamci_irq(int irq, void *devid)
{
    struct pxamci_host *host = devid;           // 得到主控制器的数据
    unsigned int ireg; int handled = 0;
    ireg = readl(host->base + MMC_I_REG) & ~readl(host->base + MMC_I_MASK);
                                           // 读取中断寄存器的值

    if (ireg) {
        unsigned stat = readl(host->base + MMC_STAT);           // 读取状态寄存器的值
        if (ireg & END_CMD_RES)
            handled |= pxamci_cmd_done(host, stat);
        if (ireg & DATA_TRAN_DONE)
            handled |= pxamci_data_done(host, stat);
        if (ireg & SDIO_INT) {
            mmc_signal_sdio_irq(host->mmc);
        }
    }
    return IRQ_RETVAL(handled);
}
```

当调用 *request,host->ops->request(host, mrq),即上文中的 pxamci_request()后,控制器与 SD 卡之间开始进行一次指令或数据传输。通信完毕后,主控芯片将产生一个内部中断,以告知此次指令或数据传输完毕。

这个中断的具体值保存在一个名为 MMC_I_REG 的中断寄存器中以供读取,中断寄存器 MMC_I_REG 相关描述如下:如果中断寄存器 MMC_I_REG 中的第 0 位有值,意味着数

据传输完成,执行 `pxamci_cmd_done(host, stat)`;如果中断寄存器 `MMC_I_REG` 中的第 2 位有值,意味着指令传输完成,执行 `pxamci_data_done(host, stat)`;其中 `stat` 是从状态寄存器 `MMC_STAT` 中读取的值,在代码里主要起到处理错误状态的作用。

`pxamci_cmd_done` 收到结束指令的内部中断信号,主控制器从 SD 卡那里得到 `response`,结束这次指令传输。注意,寄存器 `MMC_RES` 里已经存放了来自 SD 卡发送过来的 `response`,以供读取。

```
static int pxamci_cmd_done(struct pxamci_host * host, unsigned int stat)
{
    struct mmc_command * cmd = host->cmd;
    cmd->resp[i] = readl(host->base + MMC_RES) & 0xffff;
    if (stat & STAT_TIME_OUT_RESPONSE) {
        cmd->error = -ETIMEDOUT;
    } else if (stat & STAT_RES_CRC_ERR && cmd->flags & MMC_RSP_CRC) {
        cmd->error = -EILSEQ;
    }
    pxamci_disable_irq(host, END_CMD_RES);
    if (host->data && !cmd->error) {
        pxamci_enable_irq(host, DATA_TRAN_DONE);
    } else {
        pxamci_finish_request(host, host->mrq);
        //结束一次传输,清空主控制器中的指令和数据
    }
    return 1;
}
```

`pxamci_data_done` 收到结束数据的内部中断信号,得到传输数据的大小,结束这次数据传输。

```
static int pxamci_data_done(struct pxamci_host * host, unsigned int stat)
{
    struct mmc_data * data = host->data;
    DCSR(host->dma) = 0;
    dma_unmap_sg(mmc_dev(host->mmc), data->sg, host->dma_len, host->dma_dir);
    if (stat & STAT_READ_TIME_OUT)
        data->error = -ETIMEDOUT;
    else if (stat & (STAT_CRC_READ_ERROR|STAT_CRC_WRITE_ERROR))
        data->error = -EILSEQ;
    if (!data->error)
        /* 数据传输量 = 块的数量 × 每个块的大小(一般为 512B) */
        data->bytes_xfered = data->blocks * data->blksz;
    else
        data->bytes_xfered = 0;
    pxamci_disable_irq(host, DATA_TRAN_DONE);
    host->data = NULL;
}
```

```

    pxamci_finish_request(host, host->mrq);
    ...
}
static void pxamci_finish_request(struct pxamci_host * host, struct mmc_request * mrq)
{
    host->mrq = NULL;
    host->cmd = NULL;
    host->data = NULL;
    mmc_request_done(host->mmc, mrq);
    set_mmc_cken(host, 0);
    unset_dvfm_constraint();
}
/* drivers/mmc/core/core.c */
void mmc_request_done(struct mmc_host * host, struct mmc_request * mrq)
{
    ...
    if (mrq->done)
        mrq->done(mrq);
}

```

这里用到了完成量 completion, 完成量是一个任务的轻量级机制, 允许一个线程告知另一个线程工作已经完成。这里的一个线程就是内部中断处理函数, 是结束主控制器与 SD 卡间通信的线程, 通过 `mrq->done(mrq)`; 即 `complete(mrq->done_data)`; 告知另一个线程——回调 * request 实现主控制器与 SD 卡进行通信的线程, 通信已经完毕。

探测引脚引起的中断如下:

```

if (host->pdata && host->pdata->init)
    host->pdata->init(&pdev->dev, pxamci_detect_irq, mmc);

```

该 `init()` 回调函数是系统初始化时, 通过 `saar_mci_init()` 实现的。

```

static int saar_mci_init(struct device * dev, irq_handler_t saar_detect_int, void * data)
{
    request_irq(cd_irq, saar_detect_int, IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING, "MMC
card detect", data);
    ...
}

```

其中, `cd_irq` 是通过 GPIO 转换得到的中断号, `pxamci_detect_irq` 是该中断实现的函数, 之前已经提到过的 `mmc_detect_change` 将最终调用 `queue_delayed_work` 执行工作队列里的 `mmc_rescan` 函数。

```

static irqreturn_t pxamci_detect_irq(int irq, void * devid)
{

```

```
    struct pxamci_host * host = mmc_priv(devid);  
    mmc_detect_change(devid, host->pdata->detect_delay);  
    return IRQ_HANDLED;  
}
```

当有 SD 卡插入或拔出时,硬件主控制器芯片的探测 pin 脚产生外部中断,进入中断处理函数,执行工作队列里的 `mmc_rescan`,扫描 SD 总线,对插入或拔出 SD 卡作相应的处理。

9.1 USB 设备管理机制

9.1.1 USB 与串口

USB 是通用串行总线 Universal Serial Bus 的英文缩写,1994 年 11 月由 Compaq、DEC、IBM、Intel、NEC、Microsoft、Northern Telecom 等公司共同提出。USB 最初是为了替代许多不同的低速总线(包括并行、串行和键盘连接)而设计的,以单一类型的总线连接各种不同类型的设备。USB 的发展已经超越了这些低速的连接方式,可以支持几乎所有连接到 PC 上的设备。

USB 为所有的 USB 外设提供了单一的、易于操作的、标准的连接类型。简化了 USB 外设的设计,同时也简化了判断哪个插头对应哪个插槽时的任务,实现了单一的数据通用接口。

一个 USB 系统包含 3 类硬件设备:USB 主机(USB HOST)、USB 设备(USB DEVICE)和 USB 集线器(USB HUB)。USB 支持热插拔(hotplug)和即插即用(PNP)。USB 支持 3 种设备传输速率:1.5 Mb/s(低速设备)、12 Mb/s(中速设备)和 480 Mb/s(高速设备)。一个 USB 口理论上可以连接 127 个 USB 设备。

9.1.2 USB 设备属性拓扑结构管理机制

USB 采用属性拓扑结构,主机侧和设备侧的 USB 控制器分别称为主机控制器(Host Controller)和 USB 设备控制器(UDC),每条总线上只有一个主机控制器,负责协调主机和设备间的通信,而设备不能主动向主机发送任何消息。如图 9.1 所示,在 Linux 系统中,USB 驱动可以从两个角度去观察,一个角度是主机侧,另一个角度是设备侧。

在 Linux 驱动中,处于最底层的是 USB 主机控制器硬件,在其之上运行的是 USB 主机控制器驱动,主机控制器之上为 USB 核心层,再上层为 USB 设备驱动层,即插入主机上的 U 盘、鼠标、USB 转串口等设备驱动。因此,在主机侧的层次结构中,要实现的 USB 驱动包括两类:USB 主机控制器驱动和 USB 设备驱动,前者控制插入其中的 USB 设备,后者控制

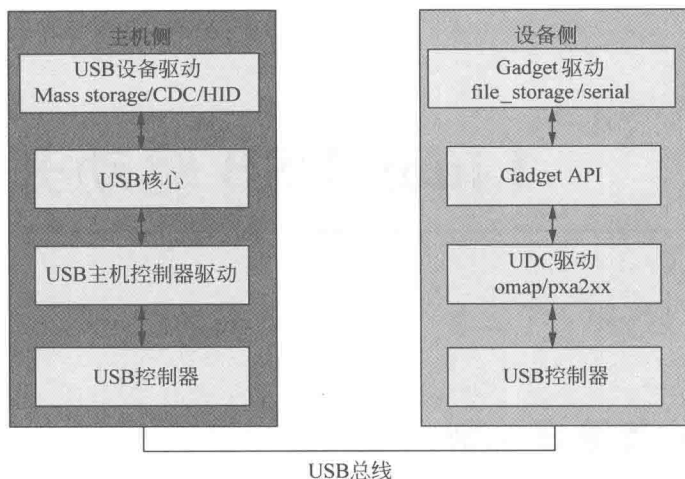


图 9.1 Linux USB 驱动总体结构

USB 设备如何与主机通信。Linux 内核 USB 核心负责 USB 驱动管理和协议处理。主机控制器驱动和设备驱动之间的 USB 核心非常重要，其功能包括：通过定义一些数据结构、宏和功能函数，向上为设备驱动提供编程接口，向下为 USB 主机控制器驱动提供编程接口；通过全局变量维护整个系统的 USB 设备信息；完成设备热插拔控制、总线数据传输控制等。

9.1.3 USB 设备逻辑组织管理机制

在 USB 设备的逻辑组织中，包含设备、配置、接口和端点 4 个层次。每个 USB 设备都提供了不同级别的配置信息，可以包含一个或者多个配置，不同的配置使设备表现出不同的功能组合（探测/连接器件需从其中选定一个），配置由多个接口组成。在 USB 协议中，接口由多个端点组成，代表一个基本的功能，是 USB 设备驱动程序控制的对象，一个功能复杂的 USB 设备可以具有多个接口。每个配置中可以有多多个接口，而设备接口都是端点的汇集。端点是 USB 通信的最基本形式，每个 USB 设备接口在主机看来就是一个端点的集合。主机只能通过端点与设备进行通信，以使用设备的功能。在 USB 系统中每一个端点都有唯一的地址，由设备地址和端点号给出。每个端点都有一定的属性，其中包括传输方式、总线访问频率、带宽、端点号和数据包的最大容量等。一个 USB 端点只能在一个方向承载数据，或者从主机到设备（称为输出端点），或者从设备到主机（称为输入端点），因此端点可以看作一个单向的管道。端点 0 通常为控制端点，用于设备初始化参数等。只要连接到 USB 上，并且上电端点 0 就可以被访问。端点 1、2 等一般用作数据端点，存放主机与设备间往来的数据。

USB 设备非常复杂，由许多的逻辑单元组成，这些单元之间的关系如图 9.2 所示。

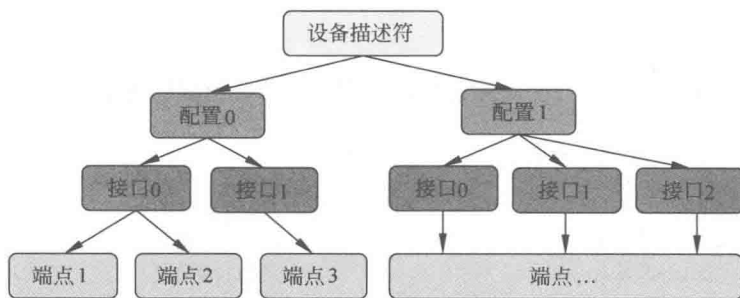


图 9.2 USB 设备、配置、接口和端点关系图

9.2 USB 驱动关键数据结构分析

(1) 设备描述符：关于设备的通用信息。

```

struct usb_device_descriptor
{
    __u8 bLength;           //描述符长度
    __u8 bDescriptorType;  //描述符类型编号
    __le16 bcdUSB;          //USB 版本
    __u8 bDeviceClass;      //USB 分配的设备类 code
    __u8 bDeviceSubClass;  // USB 分配的子类 code
    __u8 bDeviceProtocol;  // USB 分配的协议 code
    __u8 bMaxPacketSize0;  //endpoint0 最大包大小
    __le16 idVendor;        //厂商编号
    __le16 idProduct;       //产品编号
    __le16 bcdDevice;       //设备出厂编号
    __u8 iManufacturer;     //描述厂商字符串的索引
    __u8 iProduct;          //描述产品字符串的索引
    __u8 iSerialNumber;     //描述设备序列号字符串的索引
    __u8 bNumConfigurations; //可能的配置数量
} __attribute__((packed));
  
```

(2) 配置描述符：此配置中的接口数、支持的挂起和恢复能力及功率要求。

```

struct usb_config_descriptor
{
    __u8 bLength;           //描述符长度
    __u8 bDescriptorType;  //描述符类型编号
    __le16 wTotalLength;    //配置所返回的所有数据大小
    __u8 bNumInterfaces;   //配置所支持的接口数
    __u8 bConfigurationValue; //Set_Configuration 命令需要的参数
    __u8 iConfiguration;   //描述该配置的字符串的索引值
    __u8 bmAttributes;     //供电模式的选择
  
```

```

    __u8 bMaxPower;                //设备从总线提取的最大电流
} __attribute__((packed));

```

(3) 接口描述符：接口类、子类和适用的协议，接口备用配置的数量和端点数目。

```

struct usb_interface_descriptor
{
    __u8 bLength;                //描述符长度
    __u8 bDescriptorType;        //描述符类型
    __u8 bInterfaceNumber;       //接口编号
    __u8 bAlternateSetting;      //备用的接口描述符编号
    __u8 bNumEndpoints;         //该接口使用端点数, 不包括端点
    __u8 bInterfaceClass;        //接口类型
    __u8 bInterfaceSubClass;     //接口子类型
    __u8 bInterfaceProtocol;     //接口所遵循的协议
    __u8 iInterface;             //描述该接口的字符串索引值
} __attribute__((packed));

```

(4) 端点描述符：端点地址、方向和类型，支持的最大包大小，如果是终端类型的端点还包括轮询频率。

```

struct usb_endpoint_descriptor
{
    __u8 bLength;                //描述符长度
    __u8 bDescriptorType;        //描述符类型
    __u8 bEndpointAddress;       //端点地址: 0-3 是端点号, 第 7 位
    __u8 bmAttributes;           //端点属性
    __le16 wMaxPacketSize;       //本端点接收或发送的最大信息包的大小
    __u8 bInterval;             //轮询数据传送端点的时间间隔
    __u8 bRefresh;
    __u8 bSynchAddress;
} __attribute__((packed));

```

(5) usb_driver 结构体：描述一个 USB 设备驱动。

```

struct usb_driver
{
    const char * name;           //驱动名称
    int (* probe) (struct usb_interface * intf, const struct usb_device_id * id); //探测函数
    void (* disconnect) (struct usb_interface * intf); //探测断开
    int (* ioctl) (struct usb_interface * intf, unsigned int code, void * buf); //I/O 控制函数
    int (* suspend) (struct usb_interface * intf, pm_message_t message); //挂起函数
    int (* resume) (struct usb_interface * intf); //恢复函数
    void (* pre_reset) (struct usb_interface * intf);
    void (* post_reset) (struct usb_interface * intf);
    const struct usb_device_id * id_table; //usb_device_id 表指针
};

```

```

struct usb_dynids dynids;
struct usbdrv_wrap drvwrap;
unsigned int no_dynamic_id:1;
unsigned int supports_autosuspend:1;
};

```

(6) URB(USB Request Block)结构体：描述与 USB 设备通信所用的基本载体和核心数据结构。

```

struct urb{
    /* 私有的：只能由 USB 核心和主机控制器访问的字段 */
    struct kref kref;                //URB 引用计数
    spinlock_t lock;                //URB 锁
    void * hcpriv;                   //主机控制器私有数据
    int bandwidth;                   //INT/ISO 请求的带宽
    atomic_t use_count;              //并发传输计数
    u8 reject;                       //传输将失败

    /* 公共的：可以被驱动使用的字段 */
    struct list_head urb_list;       //链表头
    struct usb_device * dev;         //关联的 USB 设备
    unsigned int pipe;               //管道信息
    int status;                      //URB 的当前状态
    unsigned int transfer_flags;     //URB_SHORT_NOT_OK | ...
    void * transfer_buffer;          //发送数据到设备或者从设备接收数据的缓冲区
    dma_addr_t transfer_dma;        //用来以 DMA 方式向设备传输数据的缓冲区
    int transfer_buffer_length;      //缓冲区大小
    int actual_length;               //URB 结束后,发送或者接收数据的实际长度
    unsigned char * setup_packet;    //指向控制 URB 的数据包指针
    dma_addr_t setup_dma;            //控制 URB 的设置数据包 DMA 缓冲区
    int start_frame;                 //等时传输中用于设置或返回初始帧
    int number_of_packets;           //等时传输中等时缓冲区数据
    int interval;                   //URB 被轮询到的时间间隔
    int error_count;                 //等时传输错误数量
    void * context;                  //completion 函数上下文
    usb_complete_t complete;         //当 URB 被完全传输或者发生错误时,被调用
    struct usb_iso_packet_descriptor iso_frame_desc[0]; //单个 URB 一次可以定义多个等
                                                    //时传输时,描述各个等时传输
};

```

9.3 USB 设备驱动函数及其使用说明

9.3.1 客户端驱动管理

USB 内核通过一个双向链表 `usb_driver_list` 管理所有客户端驱动,具体管理功能为安装和卸载两部分,对应于 `usb_register` 和 `usb_deregister`,USB 内核是动态安装和卸载设备

驱动。

```
int usb_register(struct usb_driver * new_driver) ;
```

客户端驱动程序应该在初始化函数中调用 `usb_register`, 先检查驱动是否初次安装, 根据 USB 保存的次版本号数组 (目前是 16 个) 中该驱动对应项是否为空, 如果不是, 返回错误; 如果是, 将它加入到 `usb_driver_list` 中, 并进行设备接口扫描 `usb_scan_devices`, 用来探测系统中哪些设备的接口可以被此程序驱动, 将会调用驱动提供的 `probe` 函数。 `usb_register` 通过深度优先算法按系统所具有的树型结构搜索系统中所有设备未被驱动的接口, 由驱动检验能否驱动。如果能, 则分配必要的软件资源, 配置并让其工作。

```
void usb_deregister(struct usb_driver * driver);
```

当客户端驱动需要从系统中卸载时, 会调用 `usb_deregister`, 将 USB 保存的次版本号数组该驱动对应项设为 `NULL`, 然后将它从 `usb_driver_list` 卸载, 断开驱动中所有被它驱动的设备接口连接, 释放所有资源。 `usb_deregister` 还会通知系统中可用的客户驱动程序, 检验这些失去资源的设备接口能否被其他程序驱动, 如可能, 则为其分配资源, 让它们正常工作。

驱动可以调用的其他接口管理函数:

```
void usb_driver_claim_interface(struct usb_driver * driver, struct usb_interface * iface,
void* priv);
int usb_interface_claimed(struct usb_interface * iface);
void usb_driver_release_interface(struct usb_driver * driver, struct usb_interface * iface);
```

根据给定的接口或设备, 从 `usb_device_id` 数组中查找第一个相符的设备 ID。它一般在驱动绑定接口时调用。

9.3.2 USB 设备配置和管理

支持 USB 设备的热插拔, USB 提供了对设备进行配置和管理, 包括插入设备、拔下设备和设备复位。

1. 插入设备相关函数

设备插入时, 与之相连的集线器首先发现设备的插入信息, 通过中断传输将信息传送给集线器的驱动, 通过信息分析, 确认有新设备插入到总线上, 集线器驱动调用 `usb_connect` 和 `usb_new_device` 来配置设备, 并将其与对应的设备驱动建立联系。

```
void usb_connect(struct usb_device * dev);
```

设定新设备信息。查找并分配设备地址 `dev→devnum`, 真正发 USB 命令进行配置由 `usb_new_device` 完成。

```
int usb_new_device(struct usb_device * dev);
```

参照协议,完成新设备的配置工作,包括 `usb_set_address` 分配地址,`usb_get_descriptor` 获得设备描述符,`usb_get_configuration` 获得设备所有配置描述符,`usb_set_configuration` 激活默认配置,`usbdevfs_add_device` 加入一个 `/proc/bus/usb` 入口,通过 `usb_find_drivers` 为默认配置 0 的每个接口查找相应的驱动程序来进行驱动。

总线上的第一个设备根集线器和主机控制器是一体的,在启动时就认为是插上的,默认地址为 0。Linux 支持多 USB 总线,即多个主机控制器和根集线器,它们各自的设备地址是不相关的。

2. 拔下设备相关函数

设备拔下时,与之相联的集线器首先检测到设备的拔下信号,通过中断传输将信息传送给集线器的驱动,集线器的驱动先验证设备是否被拔下,如果是则调用 `usb_disconnect` 进行处理。

```
void usb_disconnect(struct usb_device **pdev);
```

断开设备后的处理。找到设备当前活动配置的每个接口的驱动程序,调用提供的 `disconnect` 接口函数,中断与各个接口的数据传输操作,释放为每个接口分配的资源。如果此设备是集线器,则递归调用 `usb_disconnect` 处理它的子设备。释放设备地址,并通过 `usbdevfs_remove_device` 释放给设备创建的 `inode(/proc/bus/usb 入口)`,`usb_free_dev` 释放 USB 给设备分配的资源。

3. 设备复位相关函数

```
int usb_reset_device(struct usb_device *dev);
```

USB 提供了 `usb_reset_device` 进行设备的复位操作。首先复位设备连接的集线器端口,然后与 `usb_new_device` 函数相似的步骤重新完成对设备的配置操作。调用此函数一定要慎重,如果处理不当将会影响设备的工作。

9.3.3 主机控制器的管理

每个主机控制器拥有一个 USB 系统,称为一个 USB 总线。USB 支持多个主机控制器,即多个 USB 总线。每增加一个主机控制器,会分配一个 `usb_bus` 结构。USB 动态安装和卸载主机驱动。

驱动安装时,初始化函数一方面完成主机控制器硬件的配置和初始化工作,另一方面调用 `usb_alloc_bus` 和 `usb_register_bus` 将自己注册到 USB 中去,供 USB 子系统访问。

```
struct usb_bus *usb_alloc_bus(struct usb_operations *op);
```

创建主机控制器对应的总线结构 `usb_bus`,保存主机控制器给 USB 提供的函数接口,并进行初始化。每个主机控制器都为 USB 提供了一套函数接口,进行 USB 通信操作。

```
struct usb_operations
{
```

```

int (* allocate)(struct usb_device *);           //为设备分配物理层的资源
int (* deallocate)(struct usb_device *);         //释放设备占有的物理层资源
int (* get_frame_number) (struct usb_device * usb_dev);      //提供当前主机控制器所使
                                                    //用的帧号,一般用于实时传输

int (* submit_urb) (struct urb * urb);           //进行实际数据传输
int (* unlink_urb) (struct urb * urb);           //结束数据传输请求
};

void usb_register_bus(struct usb_bus * bus);

```

将 USB 总线结构 `usb_bus` 注册到 USB 内核中,即将其加入到 USB 内核的总线双向链表 `usb_bus_list` 中,并创建一个 `/proc/bus/usb` 入口。

主机驱动卸载时,调用 `usb_deregister_bus` 和 `usb_free_bus` 释放资源,为主机驱动提供的接口函数有:

```

struct usb_device *usb_alloc_dev(struct usb_device *parent, struct usb_bus * bus);
void usb_free_dev(struct usb_device * dev) ;
void usb_inc_dev_use(struct usb_device * dev);

```

9.3.4 协议控制命令集和数据传输管理

1. 协议控制命令集

USB 内核为设备的客户端驱动提供了一套控制命令的接口函数,实现对设备的配置、控制和通信。这些接口函数通过 `usb_control_msg` 进行发送,客户端也可以通过调用 `usb_control_msg` 完成自己的设备命令。`usb_control_msg` 属于同步通信函数,不采用异步回调方式。

```
int usb_clear_halt(struct usb_device * dev, int pipe);
```

只提供针对停止工作的端点的清除操作,不提供清除设备的远程唤醒操作。

2. 数据传输的管理

数据传输都是使用 USB 内核提供的 URB。有些变量是针对特定传输类型的。USB 内核提供的用于处理 URB 的接口函数有:

(1) `urb_t * usb_alloc_urb(int iso_packets)` 用于给客户驱动分配 URB, `iso_packets` 为该 URB 中需要传输的实时数据包的个数,其他传输为 0。

(2) `void usb_free_urb(urb_t * urb)` 与 `usb_alloc_urb` 对应,释放分配的 URB。

(3) `int usb_submit_urb(urb_t * urb)` 将一个或多个 urb 异步发送给 USB 内核处理。其中 urb 可以是一个,也可以是多个,并且可以对应于不同的端点。

(4) `int usb_unlink_urb(urb_t * urb)` 表示在 urb 传输处理完成之前,取消数据处理。一般是设备在工作过程中被拔下,或软件主动取消对某个数据传输的处理,或 URB 传输超时调用。

当主机控制器驱动将 URB 中的数据传输处理完成后,将调用 `urb->complete` 回调函


```

    struct input_dev * dev;                //USB 鼠标同时又是一种输入设备,需要内嵌一个输入
                                           //设备结构体来描述其输入设备的属性
    struct urb * irq;                     // URB 请求包结构体,用于传送数据

    signed char * data;                   //普通传输用的地址
    dma_addr_t data_dma;                  // dma 传输用的地址
};
// 驱动程序生命周期的开始点,向 USB core 注册这个鼠标驱动程序
static int __init usb_mouse_init(void)
{
    int retval = usb_register(&usb_mouse_driver); //注册函数
    if (retval == 0)                               //注册失败
        printk(KERN_INFO KBUILD_MODNAME ": " DRIVER_VERSION ": "
                DRIVER_DESC "\n");
    printk("Mr. Ren, the mousedriver is inserted, Welcome to use it!!!\n");
    return retval;
}
//驱动程序生命周期的结束点,向 USB core 注销这个鼠标驱动程序
static void __exit usb_mouse_exit(void)
{
    printk("Mr. Ren, the mousedriver is removed, See you later!!!\n");
    usb_deregister(&usb_mouse_driver);
}

```

2. 探测函数实现

```

static int usb_mouse_probe(struct usb_interface * intf, const struct usb_device_id * id)
{
    /* 接口结构体包含于设备结构体中,usb_host_interface 用于描述接口设置的结构体,内嵌在接口
       结构体 usb_interface 中。usb_endpoint_descriptor 是端点描述符结构体,内嵌在端点结构体
       usb_host_endpoint 中,而端点结构体内嵌在接口设置结构体中。
    */
    //interface_to_usbdev 通过接口结构体获得设备结构体
    struct usb_device * dev = interface_to_usbdev(intf);
    struct usb_host_interface * interface;
    struct usb_endpoint_descriptor * endpoint;
    struct usb_mouse * mouse;
    struct input_dev * input_dev;
    int pipe, maxp;
    int error = - ENOMEM;
    //打印相关信息,用于驱动测试
    printk("Mr. Ren, the driver is in usb_mouse_probe. Welcome to come back next time!\n");
    interface = intf->cur_altsetting;
    // 鼠标仅有一个 interrupt 类型的 in 端点,不满足此要求的设备均报错
    if (interface->desc.bNumEndpoints != 1)
        return - ENODEV;
    endpoint = &interface->endpoint[0].desc;

```

```

if (!usb_endpoint_is_int_in(endpoint))
    return -ENODEV;
/* 返回对应端点能够传输的最大数据包,鼠标返回的最大数据包为 4 个字节,数据包具体内容
   在 urb 回调函数中有详细说明
*/
pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
//为 mouse 设备结构体分配内存
mouse = kzalloc(sizeof(struct usb_mouse), GFP_KERNEL);
input_dev = input_allocate_device();
if (!mouse || !input_dev)
    goto fail1;
/* 申请内存空间用于数据传输,data 为指向该空间的地址,data_dma 是这块内存空间的 dma 映射,
   即这块内存空间对应的 dma 地址.在使用 dma 传输的情况下,则使用 data_dma 指向的 dma 区域,
   否则使用 data 指向的普通内存区域进行传输.GFP_ATOMIC 表示不等待,GFP_KERNEL 是普通的优
   先级,可以睡眠等待,由于鼠标使用中断传输方式,不允许睡眠状态,data 又是周期性获取鼠标事
   件的存储区,因此使用 GFP_ATOMIC 优先级,如果不能分配到内存则立即返回 0
*/
mouse->data = usb_alloc_coherent(dev, 8, GFP_ATOMIC, &mouse->data_dma);
if (!mouse->data)
    goto fail1;
/* 为 urb 结构体申请内存空间,第一个参数表示等时传输时需要传送包的数量,其他传输方式则为
   0.申请的内存通过 usb_fill_int_urb 函数进行填充
*/
mouse->irq = usb_alloc_urb(0, GFP_KERNEL);
if (!mouse->irq)
    goto fail2;
/* 填充 usb 设备结构体和输入设备结构体 */
mouse->usbdev = dev;
mouse->dev = input_dev;
/* 获取鼠标设备的名称 */
if (dev->manufacturer)
    strcpy(mouse->name, dev->manufacturer, sizeof(mouse->name));
if (dev->product) {
    if (dev->manufacturer)
        strcat(mouse->name, " ", sizeof(mouse->name));
    strcat(mouse->name, dev->product, sizeof(mouse->name));
}

if (!strlen(mouse->name))
    snprintf(mouse->name, sizeof(mouse->name),
             "USB HIDBP Mouse %04x: %04x",
             le16_to_cpu(dev->descriptor.idVendor),
             le16_to_cpu(dev->descriptor.idProduct));
/* 填充鼠标设备结构体中的节点名.usb_make_path 用来获取 USB 设备在 Sysfs 中的路径,格式为:
   usb-usb 总线号-路径名
*/

```

```

usb_make_path(dev, mouse->phys, sizeof(mouse->phys));
strlcat(mouse->phys, "/input0", sizeof(mouse->phys));
//将鼠标设备的名称赋给鼠标设备内嵌的输入子系统结构体
input_dev->name = mouse->name;
//将鼠标设备的设备节点名赋给鼠标设备内嵌的输入子系统结构体
input_dev->phys = mouse->phys;
/* input_dev 中的 input_id 结构体,用来存储厂商、设备类型和编号,这个函数是将设备描述
   符中的编号赋给内嵌的输入子系统结构体
   */
usb_to_input_id(dev, &input_dev->id);
//cdev 是设备所属类别(class device)
input_dev->dev.parent = &intf->dev;
//鼠标事件检测
input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REL);
input_dev->keybit[BIT_WORD(BTN_MOUSE)] = BIT_MASK(BTN_LEFT) |
    BIT_MASK(BTN_RIGHT) | BIT_MASK(BTN_MIDDLE);
input_dev->relbit[0] = BIT_MASK(REL_X) | BIT_MASK(REL_Y);
input_dev->keybit[BIT_WORD(BTN_MOUSE)] |= BIT_MASK(BTN_SIDE) |
    BIT_MASK(BTN_EXTRA);
input_dev->relbit[0] |= BIT_MASK(REL_WHEEL);

input_set_drvdata(input_dev, mouse);
/* 填充输入设备打开函数指针 */
input_dev->open = usb_mouse_open;
//填充输入设备关闭函数指针
input_dev->close = usb_mouse_close;
/* 构建 urb,将填充好的 mouse 结构体的数据填充进 urb 结构体中,在 open 中递交给 urb。
   当 urb 包含一个即将传输的 DMA 缓冲区时应该设置 URB_NO_TRANSFER_DMA_MAP。USB 核心
   使用 transfer_dma 变量所指向的缓冲区,而不是 transfer_buffer 变量所指向的。URB_NO_
   SETUP_DMA_MAP 用于 Setup 包,URB_NO_TRANSFER_DMA_MAP 用于所有 Data 包
   */
usb_fill_int_urb(mouse->irq, dev, pipe, mouse->data, (maxp > 8 ? 8 : maxp),
    usb_mouse_irq, mouse, endpoint->bInterval);
mouse->irq->transfer_dma = mouse->data_dma;
mouse->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
/* 向系统注册输入设备 */
error = input_register_device(mouse->dev);
if (error)
    goto fail3;
/* 一般在 probe 函数中,需要将设备相关信息保存在一个 usb_interface 结构体中,以便以后通过
   usb_get_intfdata 获取使用。鼠标设备结构体信息将保存在 intf 接口结构体内嵌的设备结构
   体中的 driver_data 数据成员中,即 intf->dev->driver_data = mouse
   */
usb_set_intfdata(intf, mouse);
return 0;
fail3:
usb_free_urb(mouse->irq);

```

```

fail2:
    usb_free_coherent(dev, 8, mouse->data, mouse->data_dma);
fail1:
    input_free_device(input_dev);
    kfree(mouse);
    return error;
}

```

3. urb 回调函数实现

在完成提交 urb 后, urb 回调函数将被调用。此函数作为 usb_fill_int_urb 函数的形参, 为构建的 urb 制定的回调函数。

```

static void usb_mouse_irq(struct urb *urb)
{
    /* urb 中的 context 指针用于为 USB 驱动程序保存数据 */
    struct usb_mouse *mouse = urb->context;
    signed char *data = mouse->data;
    struct input_dev *dev = mouse->dev;
    int status;

    /* status 值为 0, 表示 urb 成功返回, 直接跳出循环把鼠标事件报告输入子系统. ECONNRESET
    出错信息表示 urb 被 usb_unlink_urb 函数给 unlink 了, ENOENT 出错信息表示 urb 被 usb_
    kill_urb 函数给 kill 了. usb_kill_urb 表示彻底结束 urb 的生命周期, usb_unlink_urb 是
    停止 urb, 这个函数不等 urb 完全终止就会返回给回调函数. 这在运行中断处理程序时或者
    等待某自旋锁时非常有用, 在这两种情况下是不能睡眠的, 而等待一个 urb 完全停止很可能
    会出现睡眠的情况.
    ESHUTDOWN 这种错误表示 USB 主控制器驱动程序发生了严重的错误, 或者提交完 urb 的一瞬
    间设备被拔出.
    遇见除了以上 3 种错误以外的错误, 将申请重传 urb
    */
    printk("Mr. Ren, the driver is in usb_mouse_irq. Welcome to come back next time!\n");
    switch (urb->status) {
        case 0: /* 成功返回 */
            break;
        case -ECONNRESET: /* 连接中断 */
        case -ENOENT:
        case -ESHUTDOWN:
            return;
        /* -EPIPE: 需清除所有中止项 */
        default: /* 其他错误 */
            goto resubmit;
    }

    /* 向输入子系统汇报鼠标事件情况, 以便作出反应.
    data 数组的第 0 个字节: bit 0、1、2、3、4 分别代表左、右、中、SIDE
    EXTRA 键的按下情况;
    data 数组的第 1 个字节: 表示鼠标的水平位移;
    data 数组的第 2 个字节: 表示鼠标的垂直位移;
    data 数组的第 3 个字节: REL_WHEEL 位移
    */
}

```

```

    */
    input_report_key(dev, BTN_LEFT, data[0] & 0x01);
    input_report_key(dev, BTN_RIGHT, data[0] & 0x02);
    input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
    input_report_key(dev, BTN_SIDE, data[0] & 0x08);
    input_report_key(dev, BTN_EXTRA, data[0] & 0x10);
    input_report_rel(dev, REL_X, data[1]);
    input_report_rel(dev, REL_Y, data[2]);
    input_report_rel(dev, REL_WHEEL, data[3]);

    /* 这里用于事件同步. 上面几行是一次完整的鼠标事件, 包括按键信息、绝对坐标信息和滚轮信息,
       输入子系统正是通过这个同步信号来在多个完整事件报告中区分每一次完整事件报告的。示意
       如下: 按键信息坐标位移信息滚轮信息 EV_SYN | 按键信息坐标位移信息滚轮信息 EV_SYN ...
    */
    input_sync(dev);
resubmit:
    status = usb_submit_urb(urb, GFP_ATOMIC);
    if (status)
        err("can't resubmit intr, %s - %s/input0, status %d",
            mouse->usbdev->bus->bus_name,
            mouse->usbdev->devpath, status);
}

```

4. 其他相关函数

(1) static int usb_mouse_open(struct input_dev *dev) 打开鼠标设备时, 开始提交在 probe 函数中构建的 urb, 进入 urb 周期。

(2) static void usb_mouse_close(struct input_dev *dev) 关闭鼠标设备时, 结束 urb 生命周期。

(3) static void usb_mouse_disconnect(struct usb_interface *intf) 鼠标设备拔出时的处理函数

9.4.3 驱动测试分析

(1) 驱动程序编写完成后, 再编写 makefile 文件, 然后在 Linux 的终端输入 make 命令, 编译成功后生成 usbmouseDriver.ko 模块文件, 使用命令 sudo insmod usbmouseDriver.ko 将该驱动模块加载进入系统, 如图 9.3 所示。

从图 9.3 中可以看出, USB 鼠标驱动模块已经成功加载进入系统。

(2) 将 USB 鼠标插入 PC 的 USB 接口, 然后输入 dmesg 命令, 结果如图 9.4 所示。

由于 USB 设备属于热插拔设备, 当鼠标插入主机后, 系统会自动检测到设备, 然后调用对应的 usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id) 函数处理, 使用鼠标操作(比如单击左键)会调用 usb_mouse_open(struct input_dev *dev)、usb_mouse_close(struct input_dev *dev) 及 usb_mouse_irq(struct urb *urb) 函数进行相应处理。

```

ubuntu@ubuntu: ~/Desktop/mouseDriver
File Edit View Search Terminal Help
ubuntu@ubuntu:~/Desktop/mouseDriver$ sudo rmmod usbmouseDriver
ubuntu@ubuntu:~/Desktop/mouseDriver$ sudo insmod usbmouseDriver.ko
ubuntu@ubuntu:~/Desktop/mouseDriver$ dmesg
[794297.316631] Mr.Ren, the mousedriver is removed, See you later!!!
[794297.316637] usbcore: deregistering interface driver usbmouse
[794301.730310] usbcore: registered new interface driver usbmouse
[794301.730315] usbmouseDriver: v1.0:Longtao Ren's USB HID Boot Protocol mouse driver
[794301.730318] Mr.Ren, the mousedriver is inserted, Welcome to use it!!!
ubuntu@ubuntu:~/Desktop/mouseDriver$

```

图 9.3 将驱动程序加载到系统

```

ubuntu@ubuntu: ~/Desktop/mouseDriver
File Edit View Search Terminal Help
ubuntu@ubuntu:~/Desktop/mouseDriver$ dmesg
[796742.148964] Mr.Ren, the mousedriver is removed, See you later!!!
[796742.148974] usbcore: deregistering interface driver usbmouse
[796746.186323] usbcore: registered new interface driver usbmouse
[796746.186328] usbmouseDriver: v1.0:Longtao Ren's USB HID Boot Protocol mouse driver
[796746.186331] Mr.Ren, the mousedriver is inserted, Welcome to use it!!!
[796746.186333] Mr.Ren, the driver is in usb mouse probe. Welcome to come back next time!
[796746.186335] Mr.Ren, the driver is in usb mouse open. Welcome to come back next time!
[796746.186337] Mr.Ren, the driver is in usb mouse close. Welcome to come back next time!
[796746.186340] Mr.Ren, the driver is in usb mouse irq. Welcome to come back next time!
ubuntu@ubuntu:~/Desktop/mouseDriver$

```

图 9.4 USB 鼠标插入 USB 接口

(3) 鼠标设备从 PC 上拔出后,驱动执行 `usb_mouse_disconnect(struct usb_interface * intf)` 函数,使用 `dmesg` 命令可得到图 9.5 所示结果。

从测试结果中可以看出,本次鼠标驱动开发完整实现了驱动开发设计的场景。在本次开发过程中,大家应对 Linux 内核代码及 Linux 下 USB 开发过程有了更深刻地了解。

```

ubuntu@ubuntu: ~/Desktop/mouseDriver
File Edit View Search Terminal Help
ubuntu@ubuntu:~/Desktop/mouseDriver$ dmesg
[797264.829113] Mr.Ren, the mousedriver is removed, See you later!!!
[797264.829125] usbcore: deregistering interface driver usbmouse
[797267.765916] usbcore: registered new interface driver usbmouse
[797267.765920] usbmouseDriver: v1.0:Longtao Ren's USB HID Boot Protocol mouse driver
[797267.765922] Mr.Ren, the mousedriver is inserted, Welcome to use it!!!
[797267.765923] Mr.Ren, the driver is in usb mouse probe. Welcome to come back next time!
[797267.765925] Mr.Ren, the driver is in usb mouse open. Welcome to come back next time!
[797267.765927] Mr.Ren, the driver is in usb mouse close. Welcome to come back next time!
[797267.765928] Mr.Ren, the driver is in usb mouse irq. Welcome to come back next time!
[797267.765930] Mr.Ren, the driver is in usb mouse disconnect. Welcome to come back next time!
ubuntu@ubuntu:~/Desktop/mouseDriver$

```

图 9.5 将鼠标拔出执行驱动函数

10.1 Linux 总线驱动及 I2C 总线

10.1.1 Linux 总线驱动设计过程

总线是处理器与设备之间的通道,在设备模型中,所有的设备都通过总线相连。总线驱动数据结构定义为 `struct bus_type`,主要定义的是 `name` 和 `match` 成员, `uevent` 为热插拔前对环境变量的设置。定义一个总线设备,对于 CPU 核心来说,总线也只是一个外设而已,所以需要定义总线设备 `struct device`,主要定义的成员是 `BUS_ID` 和 `release`。定义总线属性 `BUS_ATTR(version, S_IRUGO, show_bus_version, NULL)`。

总线需要的几个基本的对象定义以后,在 `module_init` 里注册总线,注册步骤:第 1 步注册总线 `bus_register(struct bus_type)`;第 2 步建立属性文件(不是必须) `bus_create_file`,可以在 `sys/bus/. /` 看到总线属性;第 3 步注册总线设备。

在总线驱动中定义设备注册和设备驱动注册的函数。定义包括 `struct device` 成员的自定义 `_device` 数据结构的设备结构, `struct device_driver` 的自定义 `_driver` 数据结构的驱动结构,在驱动结构中定义 `device_driver` 的 `probe`、`remove` 等方法,这些方法为中间的一个重定向作用,便于设备驱动程序中可以自定义这些方法。数据都定义好了,接下来就实现两个注册过程。

第 1 个过程, `int register_xx_device(struct xx_device *)`: ①设置 `device` 里的 `bus` 类型,此类型把设备注册到总线上; ②设置 `device parent`,指向总线注册时,注册的总线设备; ③定义一个 `dev.release`; ④复制一个名字到 `dev.bus_id` 中,这个名字要与自定义的 `driver` 结构的 `driver` 成员中的 `name` 成员一致,才能注册成功; ⑤注册设备自定义中的 `device` 成员, `device_register`。其卸载过程使用 `device_unregister`。

第 2 个过程,设备驱动程序的注册过程: `int register_xxx_driver(struct xxx_driver *)`: ①设置设备驱动结构中的 `device_driver` 成员的 `bus_type`; ②在设备驱动程序中定义了 `probe` 等方法,则调用总线的 `probe` 把设备驱动中的 `probe` 等方法注册到内核中; ③其他一些相关的 `device_driver` 定义; ④注册 `device_driver`; ⑤属性文件的建立(可选)。

有了以上总线驱动,可以直接调用定义的设备注册函数和设备驱动注册函数来注册设备。驱动的编写和常规的方法一样,只须在模块加载的时候增加设备注册到总线及设备驱动到总线的注册过程。要注意的是定义的设备,结构中的 `bus_id` 和设备驱动中的 `device_driver.name` 必须一致,否则设备不能被正常注册,而对应的设备驱动在其总线上会找不到和其匹配的设备,注册失败。在成功注册后,一般会自定义 `probe` 函数,其带的参数是自定义的设备的结构体,在驱动注册到内核的过程中,系统找到与之匹配的设备以后,会把这个设备的指针传给 `probe` 作为参数使设备驱动能得到这个参数。在此以后,可以把其看成一个句柄来使用,通过这个句柄可以调用总线上的方法,因为总线的方法都是和设备相关的,需要这个设备指针。

10.1.2 I2C 总线的工作原理与应用

I2C(Inter—Integrated Circuit)总线是一种由 Philips 公司开发的两线式串行总线,用于连接微控制器及其外围设备。I2C 总线产生于在 80 年代,最初为音频和视频设备开发,如今主要在服务器管理中使用,其中包括单个组件状态的通信。例如管理员可对各个组件进行查询,以管理系统的配置或掌握组件的功能状态,如电源和系统风扇。可随时监控内存、硬盘、网络及系统温度等多个参数,增加了系统的安全性,方便了管理。

I2C 总线最主要的优点是简单性和有效性。由于接口直接在组件之上,因此 I2C 总线占用的空间非常小,减少了电路板的空间和芯片引脚的数量,降低了互联成本。总线的长度可高达 25 英尺,并且能够以 10Kbps 的最大传输速率支持 40 个组件。I2C 总线的另一个优点是支持多主控(multimastering),其中任何能够进行发送和接收的设备都可以成为主总线。一个主控能够控制信号的传输和时钟频率。当然,在任何时间点上只能有一个主控。

I2C 总线是由数据线 SDA 和时钟 SCL 构成的串行总线,可发送和接收数据。在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送,最高传送速率 100Kbps。各种被控制电路均并联在这条总线上,但就像电话机一样只有拨通各自的号码才能工作,所以每个电路和模块都有唯一的地址,在信息的传输过程中,I2C 总线上并接的每一模块电路既是主控器(或被控器),又是发送器(或接收器),这取决于所要完成的功能。CPU 发出的控制信号分为地址码和控制量两部分,地址码用来选址,即接通需要控制的电路,确定控制的种类;控制量决定该调整类别(例如对比度、亮度等)及需要调整的量。这样,各控制电路虽然挂在同一条总线上,却彼此独立,互不相关。

I2C 总线在传送数据过程中共有 3 种类型信号,分别是:开始信号、结束信号和应答信号。

- (1) 开始信号: SCL 为高电平时,SDA 由高电平向低电平跳变,开始传送数据。
- (2) 结束信号: SCL 为低电平时,SDA 由低电平向高电平跳变,结束传送数据。
- (3) 应答信号: 接收数据的 IC 在接收到 8b 数据后,向发送数据的 IC 发出特定的低电

平脉冲,表示已收到数据。CPU 向受控单元发出一个信号后,等待受控单元发出一个应答信号,CPU 接收到应答信号后,根据实际情况作出是否继续传递信号的判断。若未收到应答信号,判断受控单元出现故障。

目前,有很多半导体集成电路上都集成了 I2C 接口。带有 I2C 接口的单片机有: Cygnal 的 C8051F0XX 系列、PhilipsP87LPC7XX 系列, Microchip 的 PIC16C6XX 系列等。很多外围器件,例如存储器、监控芯片等也提供 I2C 接口。

10.1.3 总线基本操作

I2C 规程运用主/从双向通信。器件发送数据到总线上,则定义为发送器;器件接收数据,则定义为接收器。主器件和从器件都可以工作于接收和发送状态。总线必须由主器件(通常为微控制器)控制,主器件产生串行时钟(SCL)控制总线的传输方向,并产生起始和停止条件。SDA 线上的数据状态仅在 SCL 为低电平的期间才能改变,SCL 为高电平的期间,SDA 状态的改变被用来表示起始和停止条件,如图 10.1 所示。

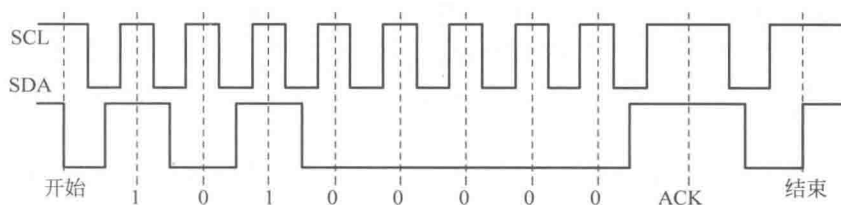


图 10.1 串行总线上的数据传送顺序

1. 控制字节

在起始条件之后,必须是器件的控制字节,其中高 4 位为器件类型识别符(不同的芯片类型有不同的定义,EEPROM 一般应为 1010)。接着 3 位为片选;最后 1 位为读写位,当为 1 时为读操作,为 0 时为写操作,如图 10.2 所示。

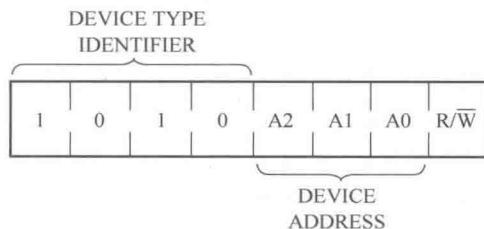


图 10.2 控制字节配置

2. 写操作

写操作分为字节写和页面写两种操作,对于页面写根据芯片的一次装载字节的不同有所不同。关于页面写的地址、应答和数据传送的时序参见图 10.3 所示。

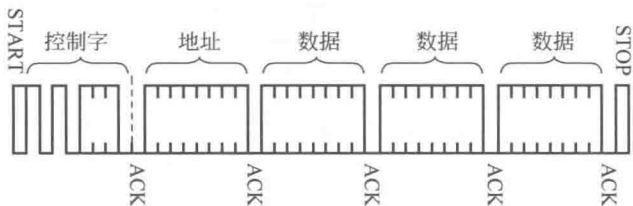


图 10.3 页面写

3. 读操作

读操作有 3 种基本操作：当前地址读、随机读和顺序读。图 10.4 给出的是顺序读的时序图。应当注意的是：最后一个读操作的第 9 个时钟周期不是“不关心”。为了结束读操作，主机必须在第 9 个周期发出停止条件或者在第 9 个时钟周期内保持 SDA 为高电平，然后发出停止条件。

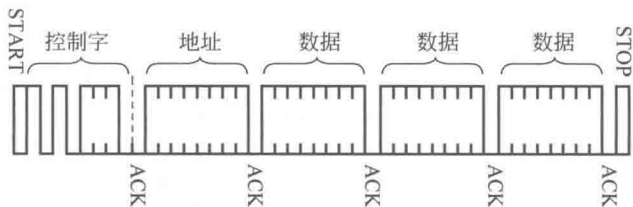


图 10.4 顺序读

10.2 Linux I2C 体系结构

10.2.1 Linux 的 I2C 体系结构组成

Linux 的 I2C 体系结构组成如图 10.5 所示。

I2C 核心提供了 I2C 总线驱动和设备驱动的注册、注销方法。I2C 通信方法（即 algorithm，笔者认为直译为“运算方法”并不合适，为免引起误解，下文将直接使用 algorithm）上层的、与具体适配器无关的代码及探测设备、检测设备地址的上层代码等。

I2C 总线驱动是对 I2C 硬件体系结构中适配器端的实现，适配器可由 CPU 控制，甚至直接集成在 CPU 内部。

I2C 总线驱动主要包含了 I2C 适配器数据结构 `i2c_adapter`、I2C 适配器的 algorithm 数据结构 `i2c_algorithm` 和控制 I2C 适配器产生通信信号的函数。

经由 I2C 总线驱动的代码，可以控制 I2C 适配器以主控方式产生开始位、停止位、读写周期，以及从设备方式被读写、产生 ACK 等。

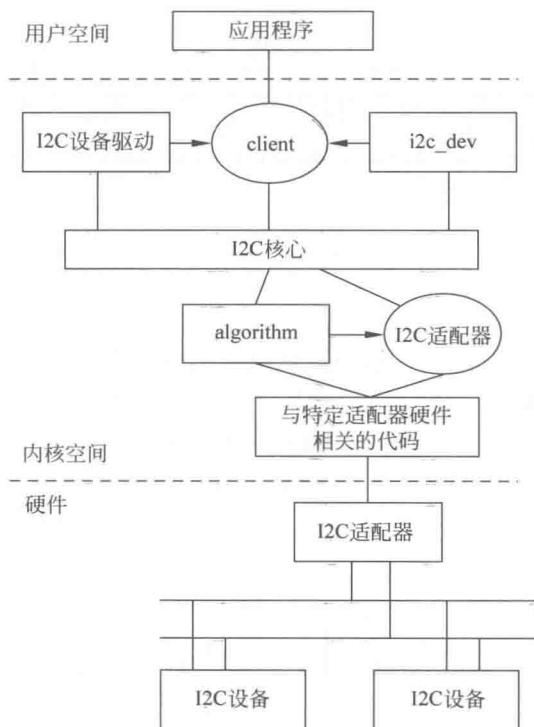


图 10.5 Linux I2C 体系结构

I2C 设备驱动是对 I2C 硬件体系结构中设备端的实现,设备一般挂载在受 CPU 控制的 I2C 适配器上,通过 I2C 适配器与 CPU 交换数据。

I2C 设备驱动主要包含了数据结构 `i2c_driver` 和 `i2c_client`,需要根据具体设备实现其中的成员函数。

10.2.2 Linux I2C 关键数据结构

在 Linux 内核中,所有的 I2C 设备都在 `sysfs` 文件系统中显示,存在于 `/sys/bus/i2c/` 目录,以适配器地址和芯片地址的形式列出。在 Linux 内核源代码中的 `drivers` 目录下包含一个 `i2c` 目录,而在 `i2c` 目录下又包含如下文件和文件夹:

1. `i2c-core.c`

实现了 I2C 核心的功能及 `/proc/bus/i2c *` 接口。

2. `i2c-dev.c`

实现了 I2C 适配器设备文件的功能,每一个 I2C 适配器都被分配一个设备。通过适配器访问设备时的主设备号都为 89,次设备号为 0~255。应用程序通过“`i2c-%d`”(`i2c-0`, `i2c-1`, ..., `i2c-10`, ...)文件名并使用文件操作接口函数 `open()`、`write()`、`read()`、`ioctl()` 和 `close()` 等来访问这个设备。

i2c-dev.c 并没有针对特定的设备而设计,只是提供了通用的 read()、write() 和 ioctl() 等接口函数,应用层可以借用这些接口函数访问挂载在适配器上的 I2C 设备的存储空间或寄存器,并控制 I2C 设备的工作方式。

3. chips 文件夹

包含了一些特定的 I2C 设备驱动,例如 Dallas 公司的 DS1337 实时钟芯片、EPSON 公司的 RTC8564 实时钟芯片和 I2C 接口的 EEPROM 驱动等。

4. busses 文件夹

包含了一些 I2C 总线的驱动,例如 S3C2410 的 I2C 控制器驱动为 i2c-s3c2410.c。

5. algos 文件夹

实现了一些 I2C 总线适配器的 algorithm。

下面介绍 I2C 驱动关键数据结构。

1. I2C 驱动关键数据结构

此外,内核中的 i2c.h 头文件对 i2c_driver、i2c_client、i2c_adapter 和 i2c_algorithm 这 4 个数据结构进行了定义。

1) i2c_adapter 结构体

```
struct i2c_adapter {
    struct module * owner;           /* 所属模块 */
    unsigned int id;
    unsigned int class;
    struct i2c_algorithm * algo;     /* 总线通信方法结构体指针 */
    void * algo_data;               /* algorithm 数据 */
    int (* client_register)(struct i2c_client *); /* client 注册时调用 */
    int (* client_unregister)(struct i2c_client *); /* client 注销时调用 */
    struct semaphore bus_lock;       /* 控制并发访问的自旋锁 */
    struct semaphore clist_lock;
    int timeout;
    int retries;                    /* 重试次数 */
    struct device dev;              /* 适配器设备 */
    struct class_device class_dev;   /* 类设备 */
    int nr;
    struct list_head clients;        /* client 链表头 */
    struct list_head list;
    char name[I2C_NAME_SIZE];       /* 适配器名称 */
    struct completion dev_released;  /* 用于同步 */
    struct completion class_dev_released;
};
```

2) i2c_algorithm 结构体

```

struct i2c_algorithm {
    int (* master_xfer)(struct i2c_adapter * adap, struct i2c_msg * msgs,
        int num);                                /* i2c 传输函数指针 */
    int (* smbus_xfer)(struct i2c_adapter * adap, u16 addr, 5 unsigned short
        flags, char read_write,
        u8 command, int size, union i2c_smbus_data * data);
    int (* slave_send)(struct i2c_adapter *, char *, int);
    int (* slave_recv)(struct i2c_adapter *, char *, int);
    int (* algo_control)(struct i2c_adapter *, unsigned int, unsigned long);
    u32 (* functionality)(struct i2c_adapter *);    /* 返回适配器支持的功能 */
};

```

上述代码第 4 行对应为 SMBus 传输函数指针, SMBus 大部分基于 I2C 总线规范, SMBus 不需增加额外引脚。与 I2C 总线相比, SMBus 增加了一些新的功能特性, 在访问时序也有一定的差异。

3) i2c_driver 结构体

```

struct i2c_driver {
    int id;
    unsigned int class;
    int (* attach_adapter)(struct i2c_adapter *);
    int (* detach_adapter)(struct i2c_adapter *);    /* 脱离 i2c_adapter 函数指针 */
    int (* detach_client)(struct i2c_client *);        /* i2c client 脱离函数指针 */
    int (* command)(struct i2c_client * client, unsigned int cmd, void * arg);
    struct device_driver driver;    /* 设备驱动结构体 */
    struct list_head list;    /* 链表头 */
};

```

4) i2c_client 结构体

```

struct i2c_client {
    unsigned int flags;    /* 标志 */
    unsigned short addr;    /* 低 7 位为芯片地址 */
    struct i2c_adapter * adapter;    /* 依附的 i2c_adapter */
    struct i2c_driver * driver;    /* 依附的 i2c_driver */
    int usage_count;    /* 访问计数 */
    struct device dev;    /* 设备结构体 */
    struct list_head list;    /* 链表头 */
    char name[I2C_NAME_SIZE];    /* 设备名称 */
    struct completion released;    /* 用于同步 */
};

```

2. I2C 驱动关键数据结构关系

下面分析 i2c_driver、i2c_client、i2c_adapter 和 i2c_algorithm 这 4 个数据结构的作用及关系。

1) i2c_adapter 与 i2c_algorithm

i2c_adapter 对应于物理上的一个适配器, i2c_algorithm 对应一套通信方法。一个 I2C 适配器需要 i2c_algorithm 提供的通信函数来控制适配器上产生特定的访问周期。缺少 i2c_algorithm 的 i2c_adapter 什么也做不了, 因此 i2c_adapter 中包含其使用的 i2c_algorithm 的指针。

i2c_algorithm 中的关键函数 master_xfer() 用于产生 I2C 访问周期需要的信号, 以 i2c_msg (即 I2C 消息) 为单位。

i2c_msg 结构体

```
struct i2c_msg {
    __u16 addr;           /* 设备地址 */
    __u16 flags;          /* 标志 */
    __u16 len;            /* 消息长度 */
    __u8 * buf;           /* 消息数据 */
};
```

2) i2c_driver 与 i2c_client

i2c_driver 对应一套驱动方法, 是用于辅助作用的数据结构, 不对应于任何的物理实体。i2c_client 对应于真实的物理设备, 每个 I2C 设备都需要一个 i2c_client 描述。i2c_client 一般包含在 i2c 字符设备的私有信息结构体中。

i2c_driver 与 i2c_client 发生关联的时刻在 i2c_driver 的 attach_adapter() 函数被运行时。attach_adapter() 会探测物理设备, 当确定一个 client 存在时, 把该 client 使用的 i2c_client 数据结构的 adapter 指针指向对应的 i2c_adapter, driver 指针指向该 i2c_driver, 并调用 i2c_adapter 的 client_register() 函数。相反的过程发生在 i2c_driver 的 detach_client() 函数被调用的时候。

假设 I2C 总线适配器 xxx 上有两个使用相同驱动程序的 yyy I2C 设备, 在打开该 I2C 总线的设备结点后相关数据结构之间的逻辑组织关系如图 10.6 所示。

3) i2c_adapter 与 i2c_client

i2c_adapter 与 i2c_client 的关系与 I2C 硬件体系中适配器和设备的关系一致, 即 i2c_client 依附于 i2c_adapter。由于一个适配器上可以连接多个 I2C 设备, 所以一个 i2c_adapter 也可以被多个 i2c_client 依附, i2c_adapter 中包括依附于它的 i2c_client 的链表。

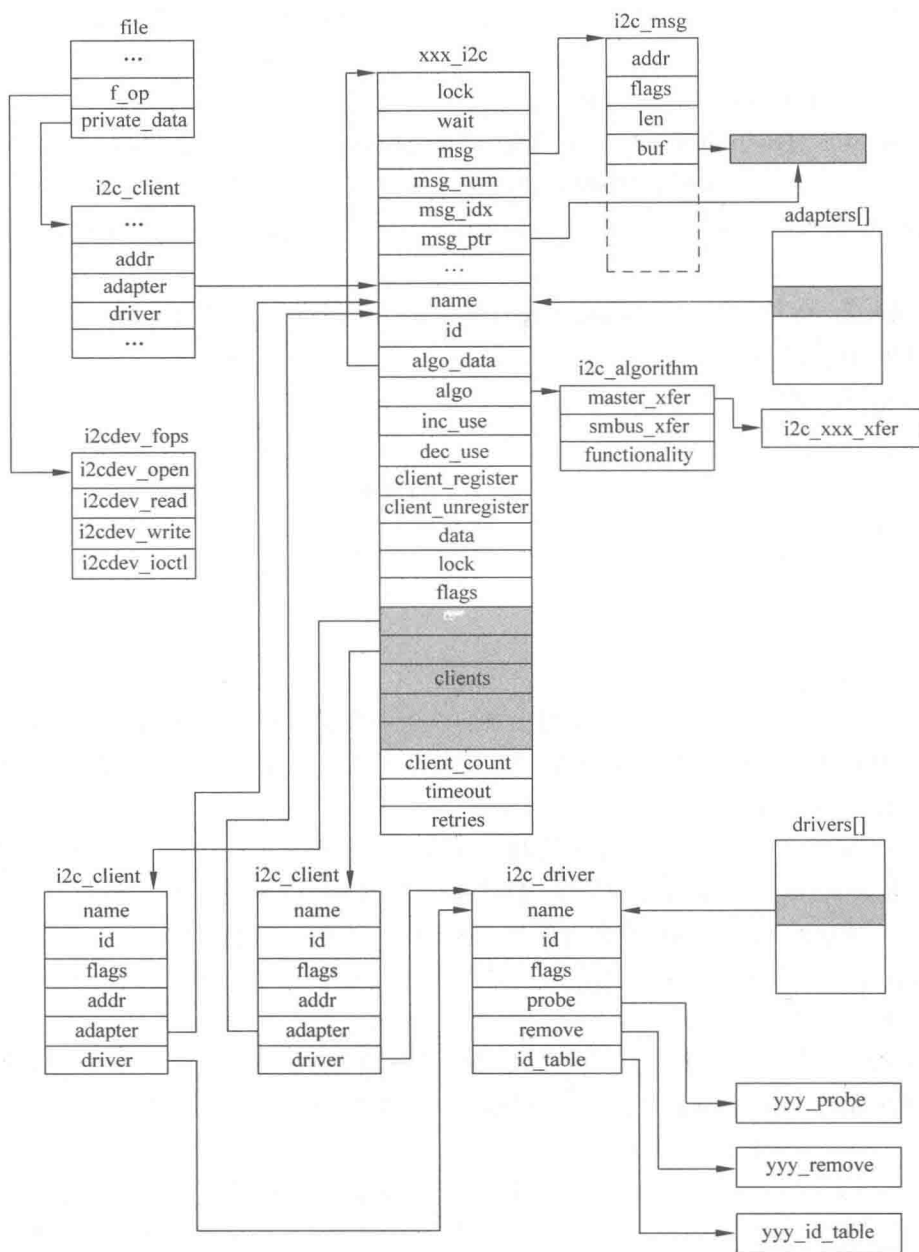


图 10.6 I2C 驱动各数据结构关系

10.3 Linux I2C 核心

I2C 核心(drivers/i2c/i2c-core.c)提供了一组不依赖于硬件平台的接口函数,这个文件一般不需要修改,但是理解其中的主要函数非常关键,因为 I2C 总线驱动和设备驱动之间依赖于 I2C 核心作为纽带。I2C 核心中的主要函数包括:

1. 增加/删除 i2c_adapter

```
int i2c_add_adapter(struct i2c_adapter * adap);
int i2c_del_adapter(struct i2c_adapter * adap);
```

2. 增加/删除 i2c_driver

```
int i2c_register_driver(struct module * owner, struct i2c_driver * driver);
int i2c_del_driver(struct i2c_driver * driver);
inline int i2c_add_driver(struct i2c_driver * driver);
```

3. i2c_client 依附/脱离

```
int i2c_attach_client(struct i2c_client * client);
int i2c_detach_client(struct i2c_client * client);
```

当一个具体的 client 被侦测到并被关联的时候,设备和 sysfs 文件将被注册。相反,在 client 被取消关联的时候,sysfs 文件和设备也被注销。I2C 核心 client attach/detach 函数如下所示:

```
int i2c_attach_client(struct i2c_client * client)
{ ...
    device_register(&client->dev);
...}
int i2c_detach_client(struct i2c_client * client)
{ ...
    device_unregister(&client->dev);
...}
```

4. i2c 传输、发送和接收

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg * msgs, int num);
int i2c_master_send(struct i2c_client * client, const char * buf, int count);
int i2c_master_recv(struct i2c_client * client, char * buf, int count);
```

i2c_transfer() 函数用于进行 I2C 适配器和 I2C 设备之间的一组消息交互,i2c_master_send() 函数和 i2c_master_recv() 函数内部会调用 i2c_transfer() 函数分别完成一条写消息和一条读消息。

(1) I2C 核心 i2c_master_send 函数

```
int i2c_master_send(struct i2c_client * client, const char * buf, int count)
{
    int ret;
    struct i2c_adapter * adap = client->adapter;
    struct i2c_msg msg;
    /* 构造一个写消息 */
    msg.addr = client->addr;
    msg.flags = client->flags & I2C_M_TEN;
    msg.len = count;
    msg.buf = (char *)buf;
    /* 传输消息 */
    ret = i2c_transfer(adap, &msg, 1);
    return (ret == 1) ? count : ret;
}
```

(2) I2C 核心 i2c_master_recv 函数

```
int i2c_master_recv(struct i2c_client * client, char * buf, int count)
{
    struct i2c_adapter * adap = client->adapter;
    struct i2c_msg msg;
    int ret;
    /* 构造一个读消息 */
    msg.addr = client->addr;
    msg.flags = client->flags & I2C_M_TEN;
    msg.flags |= I2C_M_RD;
    msg.len = count;
    msg.buf = buf;
    ret = i2c_transfer(adap, &msg, 1);
    /* 成功(1条消息被处理),返回读的字节数 */
    return (ret == 1) ? count : ret;
}
```

i2c_transfer()函数本身不具备驱动适配器物理硬件完成消息交互的能力,只是寻找到i2c_adapter对应的i2c_algorithm,并使用i2c_algorithm的master_xfer()函数真正驱动硬件流程。

(3) I2C 核心 i2c_transfer 函数

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg * msgs, int num)
{
    int ret;
    if (adap->algo->master_xfer) {
        down(&adap->bus_lock);
        ret = adap->algo->master_xfer(adap, msgs, num);
        up(&adap->bus_lock);
        return ret;
    }
    /* 消息传输 */
}
```

```

    } else {
        dev_dbg(&adap->dev, "I2C level transfers not supported\n");
        return -ENOSYS;
    }
}

```

5. I2C 控制命令分派

下面函数有助于将发给 I2C 适配器设备文件 `ioctl` 的命令,分派给对应适配器的 `algorithm` 的 `algo_control()` 函数或 `i2c_driver` 的 `command()` 函数。

```

int i2c_control(struct i2c_client *client, unsigned int cmd, unsigned long arg);
void i2c_clients_command(struct i2c_adapter *adap, unsigned int cmd, void *arg);

```

10.4 Linux I2C 总线驱动

10.4.1 I2C 适配器驱动加载与卸载

I2C 总线驱动模块的加载函数要完成两个工作:

(1) 初始化 I2C 适配器所使用的硬件资源、申请 I/O 地址、中断号等。

(2) 通过 `i2c_add_adapter()` 添加 `i2c_adapter` 的数据结构,这个 `i2c_adapter` 数据结构成员已经被 `xxx` 适配器的相应函数指针所初始化。

I2C 总线驱动模块的卸载函数要完成的工作与加载函数的相反:

(1) 释放 I2C 适配器所使用的硬件资源、释放 I/O 地址、中断号等。

(2) 通过 `i2c_del_adapter()` 删除 `i2c_adapter` 的数据结构。

I2C 总线驱动模块加载和卸载函数模板:

```

static int __init i2c_adapter_xxx_init(void)
{
    xxx_adapter_hw_init();
    i2c_add_adapter(&xxx_adapter);
}

static void __exit i2c_adapter_xxx_exit(void)
{
    xxx_adapter_hw_free();
    i2c_del_adapter(&xxx_adapter);
}

```

上述代码中 `xxx_adapter_hw_init()` 和 `xxx_adapter_hw_free()` 函数的实现都与具体的 CPU 和 I2C 设备硬件相关。

10.4.2 I2C 总线通信方法

为特定的 I2C 适配器实现其通信方法,主要实现 `i2c_algorithm` 的 `master_xfer()` 函数

和 `functionality()` 函数。

`functionality()` 函数非常简单,用于返回 `algorithm` 所支持的通信协议,例如 `I2C_FUNC_I2C`、`I2C_FUNC_10BIT_ADDR`、`I2C_FUNC_SMBUS_READ_BYTE`、`I2C_FUNC_SMBUS_WRITE_BYTE` 等。

`master_xfer()` 函数在 I2C 适配器上完成传递给它的 `i2c_msg` 数组中的每个 I2C 消息,如下给出了 `xxx` 设备的 `master_xfer()` 函数模板。

```
static int i2c_adapter_xxx_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
{
    ...
    for (i = 0; i < num; i++)
    {
        i2c_adapter_xxx_start();          /* 产生开始位 */
        /* 读消息 */
        if (msgs[i] -> flags & I2C_M_RD)
        {
            i2c_adapter_xxx_setaddr((msg -> addr << 1) | 1);      /* 发送从设备读地址 */
            i2c_adapter_xxx_wait_ack(); /* 获得从设备的 ack */
            i2c_adapter_xxx_readbytes(msgs[i] -> buf, msgs[i] -> len);    /* 读取 msgs[i]
                                                                              // -> len 长的数据到 msgs[i] -> buf */
        }
        else
        /* 写消息 */
        {
            i2c_adapter_xxx_setaddr(msg -> addr << 1);          /* 发送从设备写地址 */
            i2c_adapter_xxx_wait_ack(); /* 获得从设备的 ack */
            i2c_adapter_xxx_writebytes(msgs[i] -> buf, msgs[i] -> len);    /* 读取 msgs[i]
                                                                              // -> len 长的数据到 msgs[i] -> buf */
        }
    }
    i2c_adapter_xxx_stop();          /* 产生停止位 */
}
```

上述代码实际给出了一个 `master_xfer()` 函数处理 I2C 消息数组的流程,对于数组中的每个消息,判断消息类型,若为读消息,则赋从设备地址为 `(msg -> addr << 1) | 1`,否则赋为 `msg -> addr << 1`。对每个消息产生 1 个开始位,紧接着传送设备地址,然后开始数据的发送或接收,对最后的消息还需产生 1 个停止位。图 10.7 描述了整个 `master_xfer()` 完成的时序。

`master_xfer()` 函数模板中的 `i2c_adapter_xxx_start()`、`i2c_adapter_xxx_setaddr()`、`i2c_adapter_xxx_wait_ack()`、`i2c_adapter_xxx_readbytes()`、`i2c_adapter_xxx_writebytes()` 和 `i2c_adapter_xxx_stop()` 函数用于完成适配器的底层硬件操作,与 I2C 适配器和 CPU 的具体硬件直接相关,需要根据芯片的数据手册来实现。

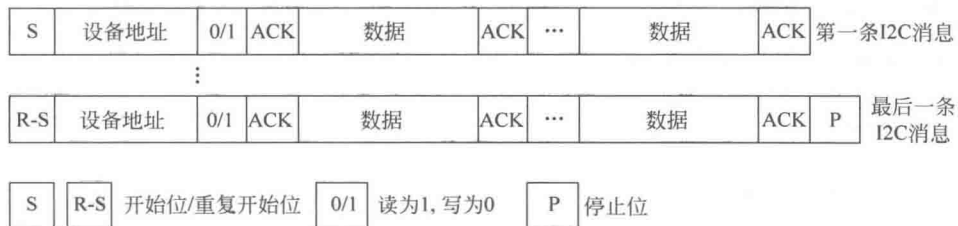


图 10.7 algorithm 中 master_xfer 的时序

i2c_adapter_xxx_readbytes() 用于从从设备上接收一串数据, i2c_adapter_xxx_writebytes() 用于向从设备写入一串数据, 这两个函数的内部会涉及 I2C 总线协议中的 ACK 应答。

master_xfer() 函数的实现在形式上会很多样, 即便是 Linux 内核源代码中已经给出的一些 I2C 总线驱动的 master_xfer() 函数, 由于由不同的组织或个人完成, 风格上的差别也非常大, 不一定能与模板完全对应, 例如 master_xfer() 函数模板给出的消息处理是顺序进行的, 而有的驱动以中断方式来完成这个流程。不管具体怎么实施, 流程的本质都是不变的。因为这个流程不以驱动工程师的意志为转移, 最终由 I2C 总线硬件上的通信协议决定。

多数 I2C 总线驱动会定义一个 xxx_i2c 结构体, 作为 i2c_adapter 的 algo_data (类似“私有数据”), 其中包含 I2C 消息数组指针、数组索引及 I2C 适配器 algorithm 访问控制用的自旋锁、等待队列等, 而 master_xfer() 函数完成消息数组中消息的处理也可通过对 xxx_i2c 结构体相关成员的访问来控制。

xxx_i2c 结构体模板:

```
struct xxx_i2c
{
    spinlock_t lock;
    wait_queue_head_t wait;
    struct i2c_msg * msg;
    unsigned int msg_num;
    unsigned int msg_idx;
    unsigned int msg_ptr;
    ...
    struct i2c_adapter adap;
};
```

10.5 Linux I2C 设备驱动

I2C 设备驱动要使用 i2c_driver 和 i2c_client 数据结构并填充其中的成员函数。i2c_client 一般包含在设备的私有信息结构体 yyy_data 中, 而 i2c_driver 适宜被定义为全局变量并初始化。

初始化的 i2c_driver:

```
static struct i2c_driver yyy_driver =
{
    .driver =
    {
        .name = "yyy",
    },
    .probe = yyy_probe,
    .remove = yyy_remove,
    .id_table = yyy_id,
};
```

10.5.1 Linux I2C 设备驱动模块加载与卸载

I2C 设备驱动模块加载函数通用的方法是在 I2C 设备驱动模块加载函数中完成两件事:

- (1) 通过 register_chrdev() 函数将 I2C 设备注册为一个字符设备。
- (2) 通过 I2C 核心的 i2c_add_driver() 函数添加 i2c_driver。

模块卸载函数中需要做相反的两件事:

- (1) 通过 I2C 核心的 i2c_del_driver() 函数删除 i2c_driver。
- (2) 通过 unregister_chrdev() 函数注销字符设备。

I2C 设备驱动加载与卸载函数的模板:

```
static int __init yyy_init(void)
{
    res = register_chrdev(YYY_MAJOR, "yyy", &yyy_fops);
    res = i2c_add_driver(&yyy_driver);
}

static void __exit yyy_exit(void)
{
    i2c_del_driver(&i2cdev_driver);
    unregister_chrdev(YYY_MAJOR, "yyy");
}
```

10.5.2 Linux I2C 设备驱动的数据传输

在 I2C 设备上读取数据的时序和数据, 通常通过 i2c_msg 数组组织, 最后通过 i2c_transfer() 函数完成。

I2C 设备驱动数据传输示例:

```
struct i2c_msg msg[2];
/* 第一条消息是写消息 */
```

```

msg[0].addr = client->addr;
msg[0].flags = 0;
msg[0].len = 1;
msg[0].buf = &offs;
/* 第二条消息是读消息 */
msg[1].addr = client->addr;
msg[1].flags = I2C_M_RD;
msg[1].len = sizeof(buf);
msg[1].buf = &buf[0];
i2c_transfer(client->adapter, msg, 2);

```

10.5.3 Linux i2c-dev.c 文件分析

i2c-dev.c 文件完全可以看作一个 I2C 设备驱动,实现的 i2c_client 是虚拟的、临时的,随着设备文件的打开而产生,并随设备文件的关闭而撤销,没有添加到 i2c_adapter 的 clients 链表中。i2c-dev.c 针对每个 I2C 适配器生成一个主设备号为 89 的设备文件,实现了 i2c_driver 的成员函数及文件操作接口,所以 i2c-dev.c 的主体是“i2c_driver 成员函数+字符设备驱动”。

i2c-dev.c 提供 i2cdev_read()、i2cdev_write() 函数来对应用户空间要使用的 read() 和 write() 文件操作接口,这两个函数分别调用 I2C 核心的 i2c_master_recv() 和 i2c_master_send() 函数来构造 1 条 I2C 消息并引发适配器 algorithm 通信函数的调用,完成消息的传输,对应于图 10.8 所示的时序。但是,大多数复杂一点 I2C 设备的读写流程并不对应于 1 条消息,往往需要 2 条甚至更多的消息来进行一次读写周期,即如图 10.9 所示的重复开始位 RepStart 模式。这种情况下,应用层仍然调用 read()、write() 文件 API 来读写 I2C 设备,将不能正确地读写。显然是对 i2cdev_read() 和 i2cdev_write() 函数的作用有所误解。



图 10.8 i2cdev_read 和 i2cdev_write 函数对应时序

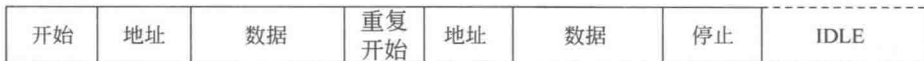


图 10.9 RepStart 模式

鉴于上述原因,i2c-dev.c 中 i2cdev_read() 和 i2cdev_write() 函数不具备太强的通用性,只能适用于非 RepStart 模式的情况。对于两条以上消息组成的读写,在用户空间需要组织 i2c_msg 消息数组,并调用 I2C_RDWR IOCTL 命令。

10.6 Linux I2C 驱动实例——EEPROM

示例中的 EEPROM 有两个从地址 SLAVE_ADDR1 和 SLAVE_ADDR2, 分别对应两个内存块。

10.6.1 初始化

与其他驱动程序一样, I2C 驱动程序也有 init() 入口点, 初始化主要用于分配数据结构, 向 I2C 核心注册驱动程序。

```
static struct file_operations eep_fops = {
    .owner = THIS_MODULE,
    .llseek = eep_llseek,
    .read = eep_read,
    .ioctl = eep_ioctl,
    .open = eep_open,
    .release = eep_release,
    .write = eep_write,
};

static dev_t dev_number;
static struct class *eep_class;

struct ee_bank {
    struct i2c_client *client;
    unsigned int addr;
    unsigned short current_pointer;
    int bank_number;
};

struct ee_bank *ee_bank_list;

int __init
eep_init(void)
{
    int err, i;
    ee_bank_list = kmalloc(sizeof(struct ee_bank) * NUM_BANKS, GFP_KERNEL);
    memset(ee_bank_list, 0, sizeof(struct ee_bank) * NUM_BANKS);
    if (alloc_chrdev_region(&dev_number, 0, NUM_BANKS, "eep") < 0) {
        printk(KERN_DEBUG "Can't register device\n");
        return -1;
    }
}
```



```

    }
    eep_class = class_create(THIS_MODULE, DEVICE_NAME);
    for (i = 0; i < NUM_BANKS; i++) {
        cdev_init(&ee_bank[i].cdev, &ee_fops);
        if (cdev_add(&ee_bank[i].cdev, (dev_number + i), 1)) {
            printk("Bad kmalloc\n");
            return 1;
        }
        device_create(eep_class, NULL, MKDEV(MAJOR(dev_number), i), "eeprom%d", i);
    }
    err = i2c_add_driver(&eep_driver);
    if (err) {
        printk("Registering I2C driver failed, errno is %d\n", err);
        return err;
    }

    printk("EEPROM Driver Initialized.\n");
    printk(KERN_INFO "eep_init\n");
    return 0;
}

```

打开 EEPROM 设备：

```

int eep_open(struct inode * inode, struct file * file)
{
    n = MINOR(file->f_dentry->d_inode->i_rdev);
    file->private_data = (struct ee_bank *) ee_bank_list[n];
}

```

10.6.2 探测设备

注册 probe() 方法：

```

static struct i2c_driver eep_driver =
{
    .driver = {
        .name      = "EEP",
    },
    .id           = I2C_DRIVERID_EEP,
    .attach_adapter = eep_probe,
    .detach_client = eep_detach,
};

```

探测 EEPROM 块的存在：

```
static unsigned short normal_i2c[] = {
    SLAVE_ADDR1, SLAVE_ADDR2, I2C_CLIENT_END
};

static struct i2c_client_address_data addr_data = {
    .normal_i2c = normal_i2c,
    .probe = ignore,
    .ignore = ignore,
    .forces = ignore,
};

static int eep_probe(struct i2c_adapter *adapter)
{
    return i2c_probe(adapter, &addr_data, eep_attach);
}
```

同客户关联：

```
int eep_attach(struct i2c_adapter *adapter, int address, int kind)
{
    static struct i2c_client *eep_client;

    eep_client = kmalloc(sizeof(*eep_client), GFP_KERNEL);
    eep_client->driver = &eep_driver;
    eep_client->addr = address;
    eep_client->adapter = adapter;
    eep_client->flags = 0;
    strncpy(eep_client->name, "eep", I2C_NAME_SIZE);
    i2c_attach_client(new_client);
}
```

10.6.3 检查适配器的功能

每个主机适配器的功能都有限。一个适配器可能不支持所有的数据访问命令，例如可能支持 read_word 命令，但不支持 read_block 命令。客户驱动程序在使用这些命令前，必须检查适配器是否对其提供支持。

I2C 核心提供两个能完成此功能的函数：

- (1) i2c_check_functionality() 检查某个特定的功能是否被支持；
- (2) i2c_get_functionality() 返回包含所有被支持功能的掩码。

在 include/linux/i2c.h 可看到所有可能支持功能的列表。

10.6.4 访问设备

为了从 EEPROM 读取数据，首先需要从此设备节点关联的私有数据域中收集调用线

程的信息。其次,使用 I2C 核心提供的数据访问例程读取数据。最后,发送至用户空间,并增加内部文件指针,以便下一次 read()/write()操作可以从上一次结束处开始。

从 EEPROM 读取数据:

```
ssize_t eep_read(struct file * file, char * buf,
                 size_t count, loff_t * ppos)
{
    int i, transferred, ret, my_buf[BANK_SIZE];
    struct ee_bank * my_bank =
        (struct ee_bank *)file->private_data;
    if (i2c_check_functionality(my_bank->client,
                                I2C_FUNC_SMBUS_READ_WORD_DATA)) {
        while (transferred < count) {
            ret = i2c_smbus_read_word_data(my_bank->client,
            my_bank->current_pointer + i);
            my_buf[i++] = (u8)(ret & 0xFF);
            my_buf[i++] = (u8)(ret >> 8);
            transferred += 2;
        }
        if (copy_to_user(buffer, (void *)my_buf, transferred))
            transferred = -EFAULT;
        else {
            my_bank->current_pointer += transferred;
            printk(KERN_INFO "eep_read\n");
        }
    }
    return transferred;
}
```

10.6.5 其他函数

为了获得功能齐全的驱动程序,需要添加剩余的入口点。这些和普通的字符驱动程序差别不大。

(1) 为了支持给内部文件指针赋新值的 lseek()系统调用,需要实现 llseek()函数。内部文件指针存储了有关 EEPROM 访问的状态信息。

(2) 为了验证数据的完整性,EEPROM 驱动程序可实现 ioctl()函数,用于校准并验证存储的数据的校验和。

(3) 如果选择将驱动程序编译为一个模块,需要提供 exit()方法,以注销设备,并清理客户设备特定的数据结构。从 I2C 核心卸载驱动只需执行如下操作:

```
i2c_del_driver(&eep_driver);
```

Linux I2C 驱动体系结构主要由 3 部分组成：I2C 核心、I2C 总线驱动和 I2C 设备驱动。I2C 核心是 I2C 总线驱动和 I2C 设备驱动的中间枢纽，以通用的、与平台无关的接口实现了 I2C 中设备与适配器的沟通。I2C 总线驱动填充 `i2c_adapter` 和 `i2c_algorithm` 结构体。I2C 设备驱动填充 `i2c_driver` 和 `i2c_client` 结构体。`i2c-dev.c` 文件定义的主设备号为 89 的设备可以给应用程序提供读写 I2C 设备寄存器的能力，大多数时候不需要为具体的 I2C 设备驱动定义文件操作接口。设计 I2C 设备驱动的程序，并不一定要遵守 Linux I2C 驱动的体系结构，可以把它当作一个普通的字符设备处理。

11.1 PCI 总线设备

11.1.1 PCI 总线

PCI 是外围设备互连 (Peripheral Component Interconnect) 的简称, 作为一种通用的总线接口标准, 在目前的计算机系统中得到了广泛的应用。PCI 提供了一组完整的总线接口规范, 目的是描述如何将计算机系统的外围设备以一种结构化和可控化的方式连接在一起, 同时还刻画了外围设备在连接时的电气特性和行为规约, 并且详细定义了计算机系统各个不同部件之间如何正确地进行交互。无论是基于 Intel 芯片的 PC, 或是基于 Alpha 芯片的工作站, PCI 毫无疑问都是目前使用最广泛的一种总线接口标准。同旧式的 ISA 总线不同, PCI 将计算机系统总线子系统与存储子系统完全地分开, CPU 通过一块称为 PCI 桥 (PCI-Bridge) 的设备来完成同总线子系统的交互, 如图 11.1 所示。

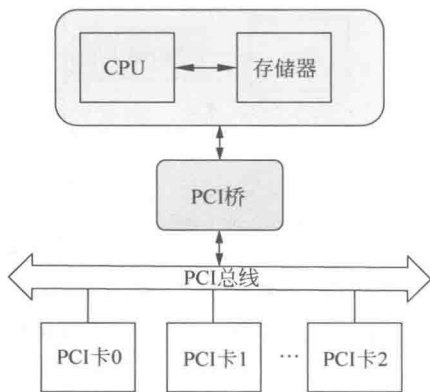


图 11.1 PCI 子系统的体系结构

由于使用了更高的时钟频率, PCI 总线能够获得比 ISA 总线更好的整体性能。PCI 总线的时钟频率一般在 25~33MHz 范围内, 有些甚至达到 66MHz 或者 133MHz, 在 64 位系

统中最高能达到 266mHz。尽管目前 PCI 设备大多采用 32 位数据总线,但 PCI 规范中已经给出了 64 位的扩展实现,从而使 PCI 总线能够更好地实现平台无关性。现在 PCI 总线已经能够用于 IA-32、Alpha、PowerPC、SPARC64 和 IA-64 等体系结构中。PCI 总线具有 3 个非常显著的优点,能够完成最终取代 ISA 总线这一历史使命。

- (1) 在计算机和外设间传输数据时具有更好的性能;
- (2) 能够尽量独立于具体的平台;
- (3) 可以很方便地实现即插即用。

图 11.2 是一个典型的基于 PCI 总线的计算机系统逻辑示意图,系统的各个部分通过 PCI 总线和 PCI-PCI 桥连接在一起。从图 11.2 中可以看出,CPU 和 RAM 需要通过 PCI 桥连接到 PCI 总线 0(即主 PCI 总线),而具有 PCI 接口的显卡可以直接连接到主 PCI 总线上。PCI-PCI 桥是一个特殊的 PCI 设备,负责将 PCI 总线 0 和 PCI 总线 1(即从 PCI 主线)连接在一起,通常 PCI 总线 1 称为 PCI-PCI 桥的下游(downstream),PCI 总线 0 称为 PCI-PCI 桥的上游(upstream)。图 11.2 中连接到从 PCI 总线上的是 SCSI 卡和以太网卡。为了兼容旧的 ISA 总线标准,PCI 总线还可以通过 PCI-ISA 桥来连接 ISA 总线,从而能够支持以前的 ISA 设备。图 11.2 中 ISA 总线上连接着一个多功能 I/O 控制器,用于控制键盘、鼠标和软驱。

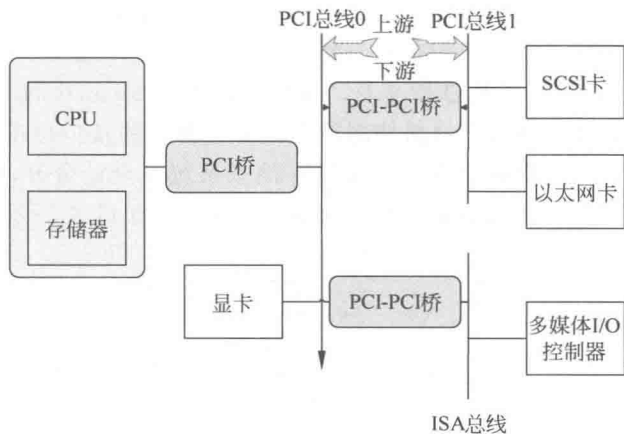


图 11.2 PCI 系统示意图

在此对 PCI 总线系统体系结构作了概括性介绍,如果想进一步了解,David A Rusling 在 The Linux Kernel(<http://tldp.org/LDP/tlk/dd/pci.html>)中对 Linux 的 PCI 子系统有比较详细的介绍。

11.1.2 PCI 设备

所有符合 PCI 总线标准的设备都被称为 PCI 设备,PCI 总线架构中可以包含多个 PCI 设备。由于一个 PCI 接口卡上可能包含多个功能模块,每个模块都被当作一个独立的逻辑

设备。因此,每个 PCI 功能即 PCI 逻辑设备都唯一地对应一个 pci_dev 设备描述符,该结构体的具体定义如下:

```

struct pci_dev {
    struct list_head global_list;
    /* 全局链表元素 global_list: 每一个 pci_dev 结构都通过该成员连接到全局 pci 设备链表 pci_
       devices 中
    */
    struct list_head bus_list;
    /* 总线设备链表元素 bus_list: 每一个 pci_dev 结构除了连接到全局设备链表中,还会通过这个成员
       连接到其所属 PCI 总线的设备链表中。每一条 PCI 总线都维护一条自己的设备链表视图,以
       便描述所有连接在该 PCI 总线上的设备,其表头由 PCI 总线的 pci_bus 结构中的 devices 成员所
       描述
    */
    struct pci_bus *bus;
    /* 总线指针 bus: 指向这个 PCI 设备所在的 PCI 总线的 pci_bus 结构。因此,对于桥设备,bus 指针
       将指向桥设备的主总线(primary bus),也即指向桥设备所在的 PCI 总线
    */
    struct pci_bus *subordinate;
    /* 指针 subordinate: 指向这个 PCI 设备所桥接的下级总线。这个指针成员仅对桥设备才有意义,
       对于一般的非桥 PCI 设备,该指针成员总是为 NULL
    */
    void *sysdata;
    /* 无类型指针 sysdata: 指向一片特定于系统的扩展数据 */
    struct proc_dir_entry *procent;
    /* 指针 procent: 指向该 PCI 设备在 /proc 文件系统中对应的目录项 */
    unsigned int devfn;
    /* devfn: PCI 设备的设备功能号,也称为 PCI 逻辑设备号(0~255)。其中 bit[7:3]是物理设备号
       (取值范围 0~31),bit[2:0]是功能号(取值范围 0~7) */
    unsigned short vendor;
    /* vendor: 16 位无符号整数,表示 PCI 设备的厂商 ID */
    unsigned short device;
    /* device: 16 位无符号整数,表示 PCI 设备的设备 ID */
    unsigned short subsystem_vendor;
    /* subsystem_vendor: 16 位无符号整数,表示 PCI 设备的子系统厂商 ID */
    unsigned short subsystem_device;
    /* subsystem_device: 16 位无符号整数,表示 PCI 设备的子系统设备 ID */
    unsigned int class;
    /* class: 32 位的无符号整数,表示该 PCI 设备的类别,其中,bit[7:0]为编程接口,bit[15:8]为子
       类别代码,bit[23:16]为基类别代码,bit[31:24]无意义。显然,class 成员的低 3 位字节刚好
       对应与 PCI 配置空间中的类代码
    */
    u8 hdr_type;
    /* hdr_type: 8 位符号整数,表示 PCI 配置空间头部的类型。其中,bit[7] = 1 表示多功能设备,
       bit[7] = 0 表示单功能设备。Bit[6:0]表示 PCI 配置空间头部的布局类型,值 00h 表示一般 PCI
       设备的配置空间头部,值 01h 表示 PCI-to-PCI 桥的配置空间头部,值 02h 表示 CardBus 桥的配
       置空间头部
    */

```

```

* /
u8 rom_base_reg;
/* rom_base_reg: 8 位无符号整数, 表示 PCI 配置空间中的 ROM 基地址寄存器在 PCI 配置空间中的
   位置。ROM 基地址寄存器在不同类型的 PCI 配置空间头部的位置是不一样的, 对于 type 0 的配置
   空间布局, ROM 基地址寄存器的起始位置是 30h; 对于 PCI-to-PCI 桥所用的 type 1 配置空间布
   局, ROM 基地址寄存器的起始位置是 38h
* /
struct pci_driver * driver;
/* 指针 driver: 指向这个 PCI 设备所对应的驱动程序定义的 pci_driver 结构。每一个 pci 设备驱
   动程序都必须对定义自己的 pci_driver 结构进行描述
* /
u64 dma_mask;
/* dma_mask: 用于 DMA 的总线地址掩码, 一般来说, 这个成员的值是 0xffffffff。数据类型 dma_addr
   _t 定义在 include/asm/types.h 中, 在 x86 平台上, dma_addr_t 类型就是 u32 类型
* /
pci_power_t current_state;
/* 当前操作状态 */
struct device dev;
/* 通用的设备接口 */
unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];
/* 定义这个 PCI 设备与哪些设备相兼容 */
unsigned int irq;
/* 无符号的整数 irq: 表示 PCI 设备通过哪根 IRQ 输入线产生中断, 一般为 0~15 之间的某个值
* /
struct resource resource[DEVICE_COUNT_RESOURCE];
/* 表示该设备可能用到的资源, 包括: I/O 端口区域、设备内存地址区域及扩展 ROM 地址区域
* /
int cfg_size;
/* 配置空间的大小 */
unsigned int transparent:1;
/* 透明 PCI 桥 */
unsigned int multifunction:1;
/* 多功能设备 */
unsigned int is_enabled:1;
/* pci_enable_device 已经被调用 */
unsigned int is_busmaster:1;
/* 设备是主设备 */
unsigned int no_msi:1;
/* 设备不使用 msi */
unsigned int block_ucfg_access:1;
/* 配置空间访问用块的形式 */
u32 saved_config_space[16];
/* 在挂起时保存配置空间 */
struct bin_attribute * rom_attr;
/* sysfs ROM 入口的属性描述 */
int rom_attr_enabled;

```



```

/* 能显示 rom 属性 */
struct bin_attribute *res_attr[DEVICE_COUNT_RESOURCE];
/* 资源的 sysfs 文件 */
};

```

11.2 PCI 设备驱动结构

PCI 设备上有 3 种地址空间：PCI 的 I/O 空间、存储空间和配置空间。CPU 可以访问 PCI 设备上的所有地址空间，其中 I/O 空间和存储空间提供给设备驱动程序使用，配置空间由 Linux 内核中的 PCI 初始化代码使用。内核在启动时负责对所有 PCI 设备进行初始化，配置好所有的 PCI 设备，包括中断号及 I/O 基址，并在文件 `/proc/pci` 中列出所有找到的 PCI 设备，以及这些设备的参数和属性。Linux 驱动程序通常使用结构(struct)表示一种设备，结构体中的变量代表某一具体设备，该变量存放了与该设备相关的所有信息。好的驱动程序应该能驱动多个同种设备，每个设备之间用次设备号进行区分，如果采用结构数据代表所有该驱动程序的设备，就可以简单地使用数组下标表示次设备号。

1. pci_driver

这个数据结构在文件 `include/linux/pci.h` 里，这是 Linux 内核 2.4 版本之后为新型的 PCI 设备驱动程序所添加的。其中，最主要的是用于识别设备的 `id_table` 结构，以及用于检测设备的函数 `probe()` 和卸载设备的函数 `remove()`。

```

struct pci_driver {
    struct list_head node;
    char * name;
    const struct pci_device_id * id_table;
    int (* probe) (struct pci_dev * dev, const struct pci_device_id * id);
    void (* remove) (struct pci_dev * dev);
    int (* save_state) (struct pci_dev * dev, u32 state);
    int (* suspend) (struct pci_dev * dev, u32 state);
    int (* resume) (struct pci_dev * dev);
    int (* enable_wake) (struct pci_dev * dev, u32 state, int enable);
};

```

2. pci_dev

这个数据结构也在文件 `include/linux/pci.h` 里，详细描述了一个 PCI 设备几乎所有的硬件信息，包括厂商 ID、设备 ID 及其他各种资源。

```

struct pci_dev {
    struct list_head global_list;
    struct list_head bus_list;
    struct pci_bus * bus;
    struct pci_bus * subordinate;
};

```

```

void      * sysdata;
struct proc_dir_entry * procent;
unsigned int  devfn;
unsigned short vendor;
unsigned short device;
unsigned short subsystem_vendor;
unsigned short subsystem_device;
unsigned int  class;
u8  hdr_type;
u8  rom_base_reg;
struct pci_driver * driver;
void      * driver_data;
u64  dma_mask;
u32  current_state;
unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];
unsigned int  irq;
struct resource resource[DEVICE_COUNT_RESOURCE];
struct resource dma_resource[DEVICE_COUNT_DMA];
struct resource irq_resource[DEVICE_COUNT_IRQ];
char  name[80];
char  slot_name[8];
int  active;
int  ro;
unsigned short regs;
int (* prepare)(struct pci_dev * dev);
int (* activate)(struct pci_dev * dev);
int (* deactivate)(struct pci_dev * dev);
};

```

11.3 PCI 设备驱动实例

11.3.1 PCI 设备驱动程序基本框架

用模块方式实现 PCI 设备驱动程序时,通常要实现以下几个部分:初始化设备模块、设备打开模块、数据读写和控制模块、中断处理模块、设备释放模块及设备卸载模块等。下面给出一个典型的 PCI 设备驱动程序的基本框架,从中体会这几个关键模块是如何组织起来的。

```

/* 指明该驱动程序适用于哪一些 PCI 设备 */
static struct pci_device_id demo_pci_tbl[] __initdata = {
    {PCI_VENDOR_ID_DEMO, PCI_DEVICE_ID_DEMO,
     PCI_ANY_ID, PCI_ANY_ID, 0, 0, DEMO},

```

```

    {0,}
};
/* 对特定 PCI 设备进行描述的数据结构 */
struct demo_card {
    unsigned int magic;
    /* 使用链表保存所有同类的 PCI 设备 */
    struct demo_card *next;
    /* ... */
}
/* 中断处理模块 */
static void demo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    /* ... */
}
/* 设备文件操作接口 */
static struct file_operations demo_fops = {
    owner: THIS_MODULE,                /* demo_fops 所属的设备模块 */
    read: demo_read,                   /* 读设备操作 */
    write: demo_write,                 /* 写设备操作 */
    ioctl: demo_ioctl,                 /* 控制设备操作 */
    mmap: demo_mmap,                  /* 内存重映射操作 */
    open: demo_open,                  /* 打开设备操作 */
    release: demo_release              /* 释放设备操作 */
    /* ... */
};
/* 设备模块信息 */
static struct pci_driver demo_pci_driver = {
    name: demo_MODULE_NAME,            /* 设备模块名称 */
    id_table: demo_pci_tbl,            /* 能够驱动的设备列表 */
    probe: demo_probe,                 /* 查找并初始化设备 */
    remove: demo_remove               /* 卸载设备模块 */
    /* ... */
};
static int __init demo_init_module(void)
{
    /* ... */
}
static void __exit demo_cleanup_module(void)
{
    pci_unregister_driver(&demo_pci_driver);
}
/* 加载驱动程序模块入口 */
module_init(demo_init_module);
/* 卸载驱动程序模块入口 */
module_exit(demo_cleanup_module);

```

上面这段代码给出了一个典型的 PCI 设备驱动程序的框架,是一种相对固定的模式。注意,同加载和卸载模块相关的函数或数据结构都要在前面加上__init、__exit 等标志符,以便同普通函数区分开来。构造框架后,接下去就是如何完成框架内的各个功能模块。

11.3.2 初始化设备模块

Linux 系统下,对一个 PCI 设备的初始化,需要完成以下工作:

- (1) 检查 PCI 总线是否被 Linux 内核支持。
- (2) 检查设备是否插在总线插槽上,如果在,保存所占用插槽的位置等信息。
- (3) 读出配置头中的信息提供给驱动程序使用。

当 Linux 内核启动并完成对所有 PCI 设备进行扫描、登录和分配资源等初始化操作时,会建立起系统中所有 PCI 设备的拓扑结构,此后当 PCI 驱动程序需要对设备进行初始化时,一般都会调用如下代码:

```
static int __init demo_init_module (void)
{
    /* 检查系统是否支持 PCI 总线 */
    if (!pci_present())
        return -ENODEV;
    /* 注册硬件驱动程序 */
    if (!pci_register_driver(&demo_pci_driver)) {
        pci_unregister_driver(&demo_pci_driver);
        return -ENODEV;
    }
    /* ... */

    return 0;
}
```

驱动程序首先调用函数 pci_present()检查 PCI 总线是否被 Linux 内核支持,如果系统支持 PCI 总线结构,函数的返回值为 0;如果驱动程序在调用函数时得到一个非 0 的返回值,那么驱动程序就必须中止自己的任务。在 2.4 版以前的内核中,需要手动调用 pci_find_device()函数查找 PCI 设备。在 2.4 版以后更好的办法是调用 pci_register_driver()函数注册 PCI 设备的驱动程序,此时需要提供一个 pci_driver 结构,该结构中的 probe 探测例程负责完成对硬件的检测工作。

```
static int __init demo_probe(struct pci_dev *pci_dev, const struct pci_device_id *pci_id)
{
    struct demo_card *card;
    /* 启动 PCI 设备 */
    if (pci_enable_device(pci_dev))
        return -EIO;
    /* 设置 DMA 标识 */
}
```

```

if (pci_set_dma_mask(pci_dev, DEMO_DMA_MASK)) {
    return -ENODEV;
}
/* 在内核空间中动态申请内存 */
if ((card = kmalloc(sizeof(struct demo_card), GFP_KERNEL)) == NULL) {
    printk(KERN_ERR "pci_demo: out of memory\n");
    return -ENOMEM;
}
memset(card, 0, sizeof(*card));
/* 读取 PCI 配置信息 */
card->iobase = pci_resource_start(pci_dev, 1);
card->pci_dev = pci_dev;
card->pci_id = pci_id->device;
card->irq = pci_dev->irq;
card->next = devs;
card->magic = DEMO_CARD_MAGIC;
/* 设置成总线主 DMA 模式 */
pci_set_master(pci_dev);
/* 申请 I/O 资源 */
request_region(card->iobase, 64, card_names[pci_id->driver_data]);
return 0;
}

```

11.3.3 打开设备模块

打开设备模块主要实现申请中断、检查读写模式及申请对设备的控制权等。在申请控制权时,非阻塞方式遇忙返回;否则进程主动接受调度,进入睡眠状态,等待其他进程释放对设备的控制权。

```

static int demo_open(struct inode * inode, struct file * file)
{
    /* 申请中断,注册中断处理程序 */
    request_irq(card->irq, &demo_interrupt, SA_SHIRQ,
        card_names[pci_id->driver_data], card)) {
    /* 检查读写模式 */
    if(file->f_mode & FMODE_READ) {
        /* ... */
    }
    if(file->f_mode & FMODE_WRITE) {
        /* ... */
    }

    /* 申请对设备的控制权 */
    down(&card->open_sem);
    while(card->open_mode & file->f_mode) {
        if (file->f_flags & O_NONBLOCK) {

```

```

        /* NONBLOCK 模式, 返回 -EBUSY */
        up(&card->open_sem);
        return -EBUSY;
    } else {
        /* 等待调度, 获得控制权 */
        card->open_mode |= f_mode & (FMODE_READ | FMODE_WRITE);
        up(&card->open_sem);
        /* 设备打开计数增 1 */
        MOD_INC_USE_COUNT;
        /* ... */
    }
}
}
}

```

11.3.4 数据读写和控制信息模块

PCI 设备驱动程序通过 `demo_fops` 结构中的函数 `demo_ioctl()`, 向应用程序提供对硬件进行控制的接口。例如通过它可以从 I/O 寄存器里读取一个数据, 并传送到用户空间里。

```

static int demo_ioctl(struct inode * inode, struct file * file, unsigned int cmd, unsigned long arg)
{
    /* ... */

    switch(cmd) {
        case DEMO_RDATA:
            /* 从 I/O 端口读取 4 字节的数据 */
            val = inl(card->iobae + 0x10);

            /* 将读取的数据传输到用户空间 */
            return 0;
    }

    /* ... */
}

```

事实上, 在 `demo_fops` 里还可以实现诸如 `demo_read()`、`demo_mmap()` 等操作, Linux 内核源码中的 `driver` 目录里提供了许多设备驱动程序的源代码, 那里可以找到类似的例子。在对资源的访问方式上, 除了 I/O 指令外, 还有对外设 I/O 内存的访问。对这些内存的操作, 一方面可以通过把 I/O 内存重新映射后, 作为普通内存进行操作; 另一方面也可以通过总线主 DMA (Bus Master DMA) 的方式, 让设备把数据通过 DMA 传送到系统内存中。

11.3.5 中断处理模块

PC 的中断资源比较有限, 只有 0~15 的中断号, 因此大部分外部设备都是以共享的形式申请中断号。当中断发生时, 中断处理程序首先负责对中断进行识别, 然后再做进一步的

处理。

```
static void demo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct demo_card *card = (struct demo_card *)dev_id;
    u32 status;
    spin_lock(&card->lock);
    /* 识别中断 */
    status = inl(card->iobase + GLOB_STA);
    if(!(status & INT_MASK))
    {
        spin_unlock(&card->lock);
        return; /* not for us */
    }
    /* 告诉设备已经收到中断 */
    outl(status & INT_MASK, card->iobase + GLOB_STA);
    spin_unlock(&card->lock);

    /* 其他进一步的处理,例如更新 DMA 缓冲区指针等 */
}
```

11.3.6 释放设备模块

释放设备模块主要负责释放对设备的控制权,释放占用的内存和中断等,所做的事情正好与打开设备模块相反。

```
static int demo_release(struct inode *inode, struct file *file)
{
    /* ... */

    /* 释放对设备的控制权 */
    card->open_mode &= (FMODE_READ | FMODE_WRITE);

    /* 唤醒其他等待获取控制权的进程 */
    wake_up(&card->open_wait);
    up(&card->open_sem);

    /* 释放中断 */
    free_irq(card->irq, card);

    /* 设备打开计数增 1 */
    MOD_DEC_USE_COUNT;

    /* ... */
}
```

11.3.7 卸载设备模块

卸载设备模块与初始化设备模块是相对应的,实现起来相对比较简单,主要是调用函数 `pci_unregister_driver()` 从 Linux 内核中注销设备驱动程序。

```
static void __exit demo_cleanup_module (void)
{
    pci_unregister_driver(&demo_pci_driver);
}
```


12.1 Linux 输入子系统结构

input 子系统即输入子系统是将 Linux 内核中的很多输入设备的软件进行分层,将属性相同且与硬件无关的部分抽象成事件核心层和事件处理层,由 Linux 系统统一实现,并以统一的接口提供给用户进程使用。输入设备驱动开发人员只需要关注和硬件相关的实现细节,最大限度地简化输入设备驱动程序的开发。

input 子系统从下到上由 3 层组成,分别为设备驱动层、核心层和事件处理层。设备驱动层主要实现对硬件设备的读写访问,中断设置,并把硬件产生的事件转换为核心层定义的规范提交给事件处理层。核心层为设备驱动层提供了规范和接口。设备驱动层关心如何驱动硬件并获得硬件数据(例如按下的按键数据),然后调用核心层提供的接口,核心层自动把数据提交给事件处理层。事件处理层是用户编程的接口(设备节点),并处理驱动层提交的数据处理。

图 12.1 显示了 input 子系统的分层结构。各层之间通信的基本单位是事件,任何一个输入设备的动作都可以抽象成一种事件,例如键盘的按下、触摸屏的按下、鼠标的移动等。事件有三种属性:类型(type),编码(code)和值(value)。input 子系统支持的所有事件都定义在 `input.h` 中,包括所有支持的类型,所属类型支持的编码等。设备驱动层把事件报告到事件核心层。核心层对事件进行分发,传到事件处理层,相应的事件处理层把事件放到 event buffer 中,等待用户进程来取。在编写 input 子系统的驱动时,要编写图 12.1 中的设备驱动层的驱动,在编写该驱动的过程中,需要调用事件核心层和事件处理层提供的接口函数来完成对事件的传输和处理。

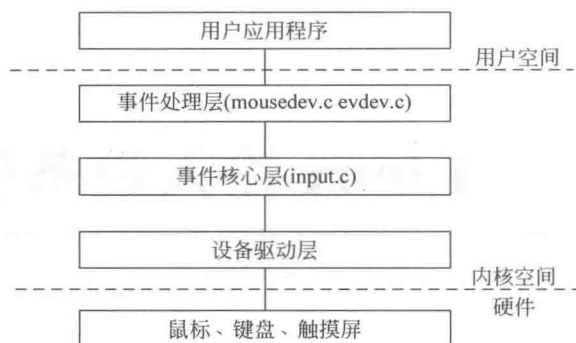


图 12.1 input 子系统的结构框架

12.2 输入设备驱动核心数据结构分析

12.1 节简要地介绍了 input 子系统的分层结构及设备事件报告处理流程,下面来了解一下在事件报告处理过程中子系统用到的 3 个基本的数据结构。

(1) input_dev 是硬件驱动层,代表一个 input 设备。

```

struct input_dev
{
    const char * name;
    const char * phys;
    const char * uniq;
    struct input_id id; //用于与 input_handler 匹配时
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; //支持的所有事件类型
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; //按键事件支持的子事件
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; //相对坐标事件支持的子事件
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; //绝对坐标事件支持的子事件
    unsigned long msckbit[BITS_TO_LONGS(MSC_CNT)]; //其他事件支持的子事件
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; //LED 灯事件支持的子事件
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)]; //声音事件支持的子事件
    unsigned long ffbbit[BITS_TO_LONGS(FF_CNT)]; //受力事件支持的子事件
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)]; //开关事件支持的子事件
    ...
    struct timer_list timer;
    int sync;
    int abs[ABS_MAX + 1]; //绝对坐标上报的当前值
    int rep[REP_MAX + 1]; //主要是处理重复按键
    unsigned long key[BITS_TO_LONGS(KEY_CNT)]; //按键有两种状态,按下和抬起,
    //这个字段就是记录这两个状态
    unsigned long led[BITS_TO_LONGS(LED_CNT)];
    unsigned long snd[BITS_TO_LONGS(SND_CNT)];
}

```

```

unsigned long sw[BITS_TO_LONGS(SW_CNT)];
int absmax[ABS_MAX + 1];           //绝对坐标的最大值
int absmin[ABS_MAX + 1];           //绝对坐标的最小值
int absfuzz[ABS_MAX + 1];
int absflat[ABS_MAX + 1];
int (*open)(struct input_dev *dev);
void (*close)(struct input_dev *dev);
int (*flush)(struct input_dev *dev, struct file *file);
int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);
struct input_handle *grab;          //当前使用的
...
struct list_headh_list;              //h_list 是一个链表头,用来把 handle 挂载上
struct list_headnode;               //用来连到 input_dev_list 上
};

```

(2) input_handler 是事件处理层,代表一个事件处理器。

```

struct input_handler
{
    void *private;
    void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int
value);
    int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct
input_device_id *id);
    void (*disconnect)(struct input_handle *handle);
    void (*start)(struct input_handle *handle);
    const struct file_operations *fops; int minor; //次设备号
    const char *name; const struct input_device_id *id_table; const struct input_device_id *
blacklist;
    struct list_headh_list;           //h_list 是一个链表头,用来把 handle 挂载上
    struct list_headnode;             //用来连到 input_handler_list 上
};

```

(3) input_handle 是核心层,实现 input_dev 与 input_handler 的配对。

```

struct input_handle
{
    void *private;
    int open;
    const char *name; struct input_dev *dev; //指向 input_dev
    struct input_handler *handler;          //指向 input_handler
    struct list_headh_node;                 //连到 input_dev 的 h_list 上
    struct list_headh_node;                 //连到 input_handler 的 h_list 上
};

```

一类 handler 可以和多个硬件设备相关联,一个硬件设备可以和多个 handler 相关联。例如一个触摸屏设备可以作为一个 event 设备,作为一个鼠标设备,也可以作为一个触摸设

备,所以一个设备需要与多个平台驱动进行连接。而一个平台驱动也不只为一个设备服务,一个触摸平台驱动可能要为 A,B,C 3 个触摸设备提供上层驱动,所以需要一对多的连接。input_dev 通过全局的 input_dev_list 链接在一起。input_handler 通过全局的 input_handler_list 链接在一起。

图 12.2 和图 12.3 表示了设备与事件处理器的连接关系。

如图 12.2 所示,一个设备对应多个驱动句柄。

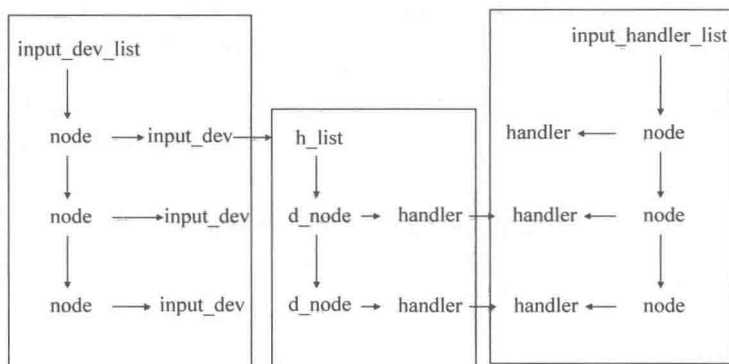


图 12.2 一个设备对应多个驱动句柄

如图 12.3 所示,一个驱动句柄对应多个设备。

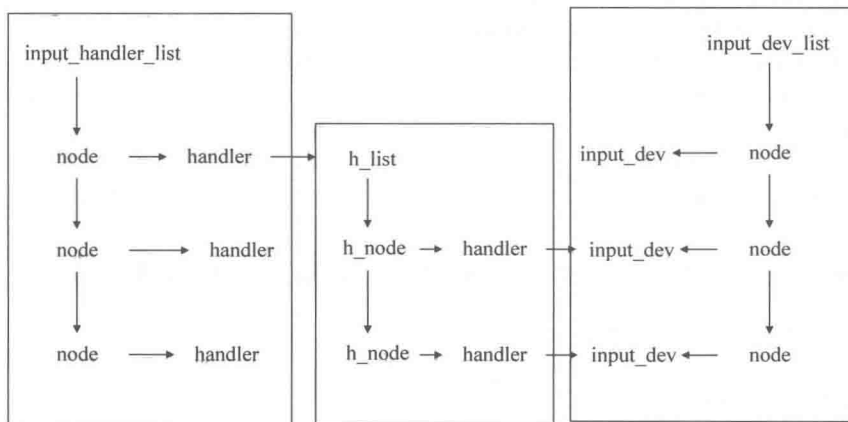


图 12.3 一个句柄对应多个设备

通过以上的介绍,可以了解到 input 子系统驱动的实现,就是对以上 3 个结构体的操作,但这 3 个结构体又是如何相互联系的呢? 下面介绍 3 个结构体建立联系的过程。

在编写设备驱动中调用 input_register_device() 注册设备。input_register_device() 完成的主要功能是初始化一些默认的值,将 device 结构添加到 Linux 设备模型当中,将 input_

dev 添加到 `input_dev_list` 链表中,然后寻找合适的 handler 与 `input_handler` 配对,配对的核心函数是 `input_attach_handler`。

`input_attach_handler` 的主要功能是调用两个函数,一个是 `input_match_device` 进行配对,另一个是 `connect` 处理配对成功后续工作。这些函数可以在 Linux 内核源码中进行查看。

对于 `connect` 函数,每种事件处理器的实现都有差异,但原理相同,以事件处理器 `evdev` 为例,其 `connect` 函数为 `evdev_connect()`。`evdev_connect` 函数做配对的善后工作,分配一个 `evdev` 结构体,并初始化相关成员, `evdev` 结构体中有 `input_handle` 结构,在函数 `evdev_connect()` 中实现该结构体的初始化并调用 `input_register_handle` 实现注册。

`input_register_handle()` 是把一个 handle 结构体通过 `d_node` 链表项,分别链接到 `input_dev` 的 `h_list`, `input_handler` 的 `h_list` 上,从而实现 3 个结构体的连接。3 个结构体连接过程如图 12.4 所示。

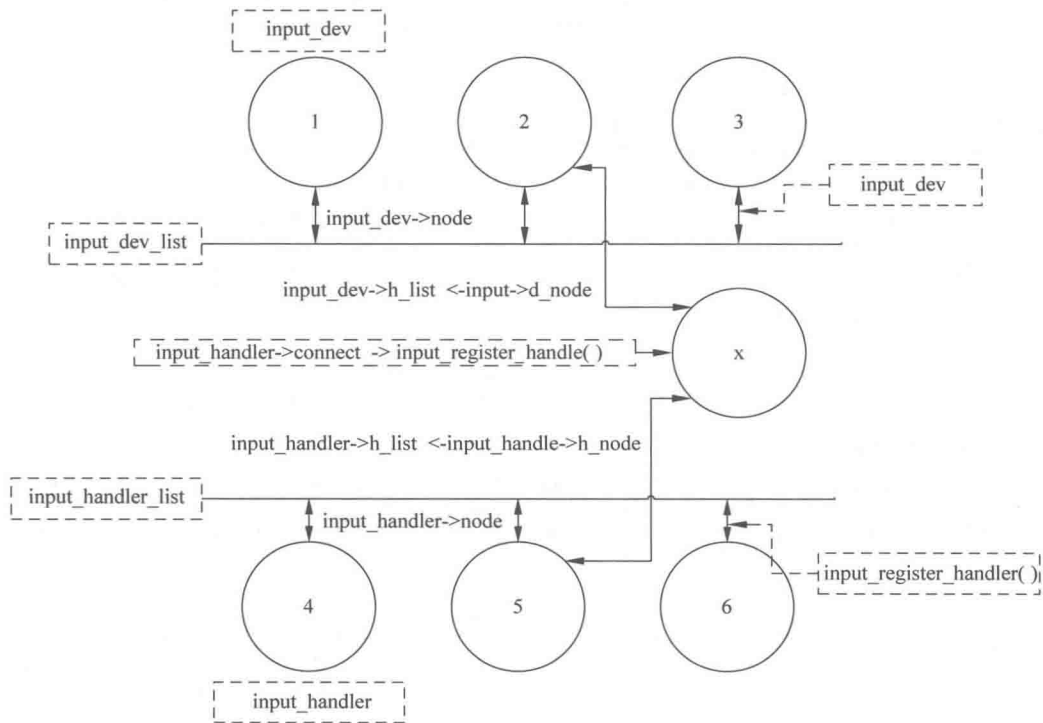


图 12.4 `input_dev` 与 `input_handler` 连接过程

以上是对整个子系统的介绍,需要理清各层的关系及在编写设备驱动时所需要操作的层次和函数。下面介绍 `input` 子系统驱动编写的一般过程。

12.3 Linux 输入设备驱动实例

12.3.1 输入设备驱动流程

1. 分配一个输入设备

```

struct input_dev * dev;                                //声明一个输入设备结构体
Struct input_dev
{
    const char * name;                                //输入设备的名称
    ...
    struct input_idid;                                //输入设备的 ID 号
    ...
    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];       //输入设备所支持的事件类型
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];     //保存设备所支持事件码
    ...
    //输入设备文件接口操作函数指针
    int (* open)(struct input_dev * dev);
    void (* close)(struct input_dev * dev);
    int (* flush)(struct input_dev * dev, struct file * file);
    int (* event)(struct input_dev * dev, unsigned int type, unsigned int code, int value);
    struct input_handle__rcu * grab;
    struct device dev;                                //表示 SYS 设备模型中的一个设备
    struct list_head h_list;
    struct list_head node;
}
dev = input_allocate_device(void);                     //为声明的结构体分配空间

```

2. 驱动支持什么事件

这一步设置输入设备支持哪些事件类型,以鼠标为例可以设置设备支持按键事件和相对坐标事件。在设置完支持的事件类型后还需要设置事件类型的码表,即该事件所包含的子事件。

```

input_dev->evbit[0] = BIT(EV_KEY);                    //赋值使输入设备支持按键类型
input_dev->keybit[BITS_TO_LONGS(KEY_CNT)] = BIT(BTN_LEFT) | BIT(BTN_RIGHT) | BIT(BTN_MIDDLE);
                                                    //按键类型的键码,左键、右键和中键

```

3. 注册一个输入设备

这一步调用的是 `input_register_device()` 函数,是输入子系统核心层提供的函数。该函数将 `input_dev` 结构体注册到输入子系统核心中。`input_register_device()` 函数如果注册失败,必须调用 `input_free_device()` 函数释放分配的 `input_dev` 的空间;如果该函数注册成功,在卸载函数中应调用 `input_unregister_device()` 函数注销输入设备结构体。注册输入设备的过程就是为输入设备设置默认值,并将其挂在 `input_dev_list` 链上,与挂载在 `input_`

handler_list 中的 handler 相匹配。如果匹配成功,就会调用 handler 的 connect 函数。

```
int input_register_device(struct input_dev *dev); //向 input 子系统注册一个新的输入设备
```

4. 驱动事件报告

这一步是设备驱动层向核心层报告发生的事件,例如按键事件,相对坐标事件等,这些事件全汇总到核心层,由核心层统一分配到事件处理层。

```
input_report_key(dev, BTN_LEFT, value);
```

实质是函数:

```
input_event(dev, type, value);
```

5. 释放和注销设备

这一步注销输入设备并清除为该设备分配的空间。

```
void input_unregister_device(struct input_dev *dev);
```

```
void input_free_device(struct input_dev *dev);
```

以上介绍了整个 input 子系统驱动程序的编写流程。下面根据 usb 鼠标驱动程序介绍 input 子系统驱动程序的实现过程。

12.3.2 USB 鼠标驱动编写实例

鼠标结构体,包含了描述鼠标相关信息的变量。

```
struct usb_mouse
{
    char name[128];           // 包括生产厂商、产品类别、产品等信息
    char phys[64];           // 设备节点名称
    struct usb_device *usbdev; // 描述 usb 鼠标的 usb 属性
    struct input_dev *dev;    // 描述 usb 鼠标的输入设备属性
    struct urb *irq;          // URB 请求包结构体,用于传送数据
    signed char *data;        // 普通传输用的地址
    dma_addr_t data_dma;      // dma 传输用的地址
};
```

urb 回调函数,在完成提交 urb 后,urb 回调函数将被调用。

此函数作为 usb_fill_int_urb 函数的形参,为构建的 urb 制定回调函数。

```
static void usb_mouse_irq(struct urb *urb)
{
    struct usb_mouse *mouse = urb->context;
    signed char *data = mouse->data;
    struct input_dev *dev = mouse->dev;
    int status;
    switch (urb->status)
```

```

    {
        case 0:
            break;
        case -ECONNRESET:
        case -ENOENT:
            case -ESHUTDOWN:
                return;
        default:
            goto resubmit;
    }
    //报告鼠标的事件
    input_report_key(dev, BTN_LEFT, data[0] & 0x01);
    input_report_key(dev, BTN_RIGHT, data[0] & 0x02);
    input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
    input_sync(dev); //告诉系统事件报告结束
    resubmit:
    ...
}

```

打开鼠标设备函数,开始提交在 probe 函数中构建的 urb,进入 urb 周期。

```

static int usb_mouse_open(struct input_dev *dev)
{
    struct usb_mouse *mouse = dev->name;
    mouse->irq->dev = mouse->usbdev;
    if (usb_submit_urb(mouse->irq, GFP_KERNEL))
        return -EIO;
    return 0;
}

```

关闭鼠标设备函数,结束 urb 生命周期。

```

static void usb_mouse_close(struct input_dev *dev)
{
    struct usb_mouse *mouse = dev->name;
    usb_kill_urb(mouse->irq);
}

```

驱动程序的探测函数。

```

static int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
    struct usb_device *dev = interface_to_usbdev(intf);
    struct usb_host_interface *interface;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_mouse *mouse;
    struct input_dev *input_dev;
    int pipe, maxp;
}

```



```

interface = intf->cur_altsetting;
if (interface->desc.bNumEndpoints != 1)
    return -ENODEV;
endpoint = &interface->endpoint[0].desc;
if (!usb_endpoint_is_int_in(endpoint))
    return -ENODEV;
//返回对应端点能够传输的最大数据包, 鼠标返回的最大数据包为 4 个字节
// 数据包具体内容在 urb 中
pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
/* 为 mouse 设备结构体分配内存 */
mouse = kzalloc(sizeof(struct usb_mouse), GFP_KERNEL);
input_dev = input_allocate_device();          //为 input_dev 申请空间
...
/* 填充 usb 设备结构体和输入设备结构体 */
mouse->usbdev = dev;
mouse->dev = input_dev;
/* 获取鼠标设备的名称 */
if (dev->manufacturer)
    strlcpy(mouse->name, dev->manufacturer, sizeof(mouse->name));
if (dev->product)
{
    if (dev->manufacturer)
        strlcat(mouse->name, " ", sizeof(mouse->name));
    strlcat(mouse->name, dev->product, sizeof(mouse->name));
}
if (!strlen(mouse->name))
    snprintf(mouse->name, sizeof(mouse->name),
        "USB HIDBP Mouse %04x: %04x",
        le16_to_cpu(dev->descriptor.idVendor),
        le16_to_cpu(dev->descriptor.idProduct));
usb_make_path(dev, mouse->phys, sizeof(mouse->phys));
strlcat(mouse->phys, "/input0", sizeof(mouse->phys));
/* 将鼠标设备的名称赋给鼠标设备内嵌的输入子系统结构体 */
input_dev->name = mouse->name;
/* 将鼠标设备的设备节点名赋给鼠标设备内嵌的输入子系统结构体 */
input_dev->phys = mouse->phys;
usb_to_input_id(dev, &input_dev->id);
input_dev->dev = &intf->dev;
//设置输入设备支持的事件类型
input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_REL);
/* keybit 表示键值, 包括左键、右键和中键 */
input_dev->keybit[BITS_TO_LONGS(KEY_CNT)] = BIT(BTN_LEFT) | BIT(BTN_RIGHT) | BIT(BTN_
MIDDLE);
input_dev->name = mouse;
input_dev->open = usb_mouse_open;
input_dev->close = usb_mouse_close;

```

```

usb_fill_int_urb(mouse->irq, dev, pipe, mouse->data,
                (maxp > 8 ? 8 : maxp),
                usb_mouse_irq, mouse, endpoint->bInterval);
mouse->irq->transfer_dma = mouse->data_dma;
mouse->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
input_register_device(mouse->dev); //向系统注册输入设备
usb_set_intfdata(intf, mouse);
return 0;
fail2: usb_free_coherent(dev, 8, mouse->data, mouse->data_dma);
fail1: input_free_device(input_dev);
kfree(mouse);
return -ENOMEM;
}

//鼠标设备拔出时的处理函数
static void usb_mouse_disconnect(struct usb_interface * intf)
{
    /* 获取鼠标设备结构体 */
    struct usb_mouse * mouse = usb_get_intfdata (intf);
    usb_set_intfdata(intf, NULL);
    if (mouse)
    {
        usb_kill_urb(mouse->irq); //结束 urb 生命周期
        input_unregister_device(mouse->dev); //将鼠标设备从输入子系统中注销
        usb_free_urb(mouse->irq); //释放 urb 存储空间
        usb_free_coherent(interface_to_usbdev(intf), 8, mouse->data, mouse->data_dma);
        //释放存放鼠标事件的 data 存储空间
        kfree(mouse); //释放存放鼠标结构体的存储空间
    }
}

//这个宏的参数意义为: 类别,子类别,协议
//USB_INTERFACE_PROTOCOL_MOUSE 表示是鼠标设备,遵循鼠标的协议
static struct usb_device_id usb_mouse_id_table [] = {
{
    USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT, USB_INTERFACE_
    PROTOCOL_MOUSE) },
{ }
};

/* 这个宏让运行在用户空间的程序知道这个驱动程序能够支持的设备,USB 驱动程序,第一个参数
必须是 usb */
MODULE_DEVICE_TABLE (usb, usb_mouse_id_table);

//鼠标驱动程序结构体
static struct usb_driver usb_mouse_driver = {
    .name          = "usbmouse",
    .probe         = usb_mouse_probe,
    .disconnect    = usb_mouse_disconnect,
    .id_table      = usb_mouse_id_table,
};

```

//驱动程序生命周期的开始点,向 USB core 注册这个鼠标驱动程序

```
static int __init usb_mouse_init(void)
{
    int retval = usb_register(&usb_mouse_driver);
    if (retval == 0)
        printk(DRIVER_VERSION ":" DRIVER_DESC);
    return retval;
}
```

//驱动程序生命周期的结束点,向 USB core 注销这个鼠标驱动程序

```
static void __exit usb_mouse_exit(void)
{
    usb_deregister(&usb_mouse_driver);
}
module_init(usb_mouse_init);
module_exit(usb_mouse_exit);
```

13.1 Flash 存储器

现在市场上有两种主要的非易失闪存技术。1988 年, Intel 公司首先开发出 NOR Flash 技术, 彻底改变了原先由 EPROM(Erasable Programmable Read-Only-Memory, 电可编程只读存储器)和 EEPROM(Electrically Erasable Programmable Read-Only Memory, 电可擦只读存储器)一统天下的局面。1989 年, 东芝公司发表了 NAND Flash 结构, 强调降低每比特的成本, 并有更好的性能, 而且像磁盘一样可以通过接口轻松升级。NOR Flash 的特点是芯片内执行(XIP, eXecute In Place), 应用程序可以直接在 Flash 闪存内运行, 不必再把代码读到系统 RAM 中。NOR 的传输效率很高, 在 1~4MB 的小容量时具有很高的成本效益, 但是很低的写入和擦除速度大大影响到性能。NAND 的结构能提供极高的单元密度, 达到高存储密度, 并且写入和擦除的速度也很快。应用 NAND 的困难是 Flash 的管理和需要特殊的系统接口。通常读取 NOR 的速度比 NAND 稍快, 而 NAND 的写入速度比 NOR 快很多, 在设计中应该考虑这些情况。若要详细地理解 Flash 存储器, 首先需要对 Linux 系统中的 Flash 驱动结构进行了解。

13.2 Linux MTD 系统层次结构

在 Linux 系统中, 提供了 MTD(Memory Technology Device, 内存技术设备)系统建立 Flash 针对 Linux 的统一、抽象的接口。MTD 将文件系统与底层的 Flash 存储器进行了隔离, 无须关心 Flash 作为字符设备和块设备与 Linux 内核的接口。

如图 13.1 所示, 在引入 MTD 后, Linux 系统中的 Flash 设备驱动及接口可分为 4 层, 从上到下依次是: 设备节点、MTD 设备层、MTD 原始设备层和硬件驱动层, 作用如下:

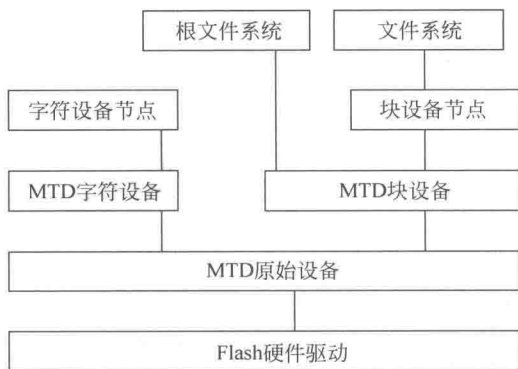


图 13.1 Linux MTD 系统层次

1. 硬件驱动层

Flash 硬件驱动层负责 Flash 硬件设备的读、写及擦除。Linux MTD 设备的 NOR Flash 芯片驱动位于 `drivers/mtd/chips` 子目录下, NAND 型 Flash 的驱动程序位于 `drivers/mtd/nand` 子目录下。

2. MTD 原始设备层

MTD 原始设备层由两部分组成,一部分是 MTD 原始设备的通用代码,另一部分是各个特定的 Flash 的数据,例如分区。

3. MTD 设备层

基于 MTD 原始设备, Linux 系统可以定义出 MTD 的块设备(主设备号 31)和字符设备(设备号 90),构成 MTD 设备层。MTD 字符设备的定义在 `mtdchar.c` 中实现,通过注册一系列 `file_operation` 函数(`lseek`、`open`、`close`、`read`、`write`、`ioctl`)可实现对 MTD 设备的读写和控制。MTD 块设备定义了一个描述 MTD 块设备的结构 `mtdblks_dev`,并声明了一个名为 `mtdblks` 的指针数组,数组中的每一个 `mtdblks_dev` 和 `mtd_table` 中的每一个 `mtd_info` 一一对应。

4. 设备节点

通过 `mknod` 在 `/dev` 子目录下建立 MTD 字符设备节点(主设备号为 90)和 MTD 块设备节点(主设备号为 31),用户访问此设备节点即可访问 MTD 字符设备和块设备。

13.3 关键数据结构

引入 MTD 后,底层 Flash 驱动直接与 MTD 原始设备层交互,利用其提供的接口注册设备和分区,如图 13.2 所示。

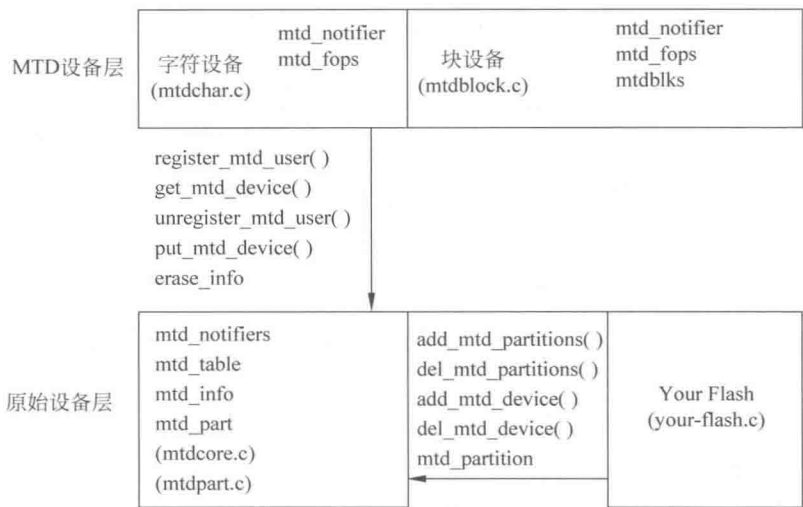


图 13.2 底层 Flash 驱动结构

13.3.1 mtd_info 结构体

用于描述 MTD 原始设备的数据结构是 `mtd_info`, 定义在 `include/linux/mtd/mtd.h` 文件中, 其中定义了大量关于 MTD 的数据和操作函数。`mtd_info` 是表示 MTD 原始设备的结构体, 每个分区也认为是一个 `mtd_info`。

例如有两个 MTD 原始设备, 每个有 3 个分区, 在系统中就共有 6 个 `mtd_info` 结构体。`mtd_info` 结构体的定义如下:

```
struct mtd_info {
    u_char type;                //内存技术类型
    uint32_t flags;             //标志位
    uint64_t size;              // mtd 设备大小
    uint32_t erasesize;         //主要查出块大小(同一个 mtd 设备可能有书中不同的 erasesize)
    uint32_t writesize;
    uint32_t writebufsize;

    uint32_t oobsize;           // oob 数据大小
    uint32_t oobavail;
    unsigned int erasesize_shift;
    unsigned int writesize_shift;
    /* 下面的掩码是基于上述两种位移实现的, 并且一一对应 */
    unsigned int erasesize_mask;
    unsigned int writesize_mask;
    ...
    //不同 erasesize 的区域
    int numeraseregions;        //不同 erasesize 区域的数目, 通常是 1
};
```

```

struct mtd_erase_region_info *eraseregions;
int (*_erase) (struct mtd_info *mtd, struct erase_info *instr); /
int (*_point) (struct mtd_info *mtd, loff_t from, size_t len,
               size_t *retlen, void **virt, resource_size_t *phys);
int (*_unpoint) (struct mtd_info *mtd, loff_t from, size_t len);
unsigned long (*_get_unmapped_area) (struct mtd_info *mtd,
                                     unsigned long len,
                                     unsigned long offset,
                                     unsigned long flags);
int (*_read) (struct mtd_info *mtd, loff_t from, size_t len,
              size_t *retlen, u_char *buf);
int (*_write) (struct mtd_info *mtd, loff_t to, size_t len,
               size_t *retlen, const u_char *buf);
int (*_panic_write) (struct mtd_info *mtd, loff_t to, size_t len,
                     size_t *retlen, const u_char *buf);
int (*_read_oob) (struct mtd_info *mtd, loff_t from,
                  struct mtd_oob_ops *ops);
int (*_write_oob) (struct mtd_info *mtd, loff_t to,
                  struct mtd_oob_ops *ops);
int (*_get_fact_prot_info) (struct mtd_info *mtd, struct otp_info *buf,
                             size_t len);
int (*_read_fact_prot_reg) (struct mtd_info *mtd, loff_t from,
                             size_t len, size_t *retlen, u_char *buf);
int (*_get_user_prot_info) (struct mtd_info *mtd, struct otp_info *buf,
                             size_t len);
int (*_read_user_prot_reg) (struct mtd_info *mtd, loff_t from,
                             size_t len, size_t *retlen, u_char *buf);
int (*_write_user_prot_reg) (struct mtd_info *mtd, loff_t to,
                              size_t len, size_t *retlen, u_char *buf);
int (*_lock_user_prot_reg) (struct mtd_info *mtd, loff_t from,
                              size_t len);
int (*_writev) (struct mtd_info *mtd, const struct kvec *vecs,
                unsigned long count, loff_t to, size_t *retlen);
void (*_sync) (struct mtd_info *mtd);
int (*_lock) (struct mtd_info *mtd, loff_t ofs, uint64_t len);
int (*_unlock) (struct mtd_info *mtd, loff_t ofs, uint64_t len);
int (*_is_locked) (struct mtd_info *mtd, loff_t ofs, uint64_t len);
int (*_block_isbad) (struct mtd_info *mtd, loff_t ofs);
int (*_block_markbad) (struct mtd_info *mtd, loff_t ofs);
int (*_suspend) (struct mtd_info *mtd);
void (*_resume) (struct mtd_info *mtd);
/*
 * 对于一些智能设备,可能拥有自己的引用计数,下述的两个方法仅适用于驱动
 */
int (*_get_device) (struct mtd_info *mtd);
void (*_put_device) (struct mtd_info *mtd);

```

```

/* 用于 writeback, 存储 backing_device 的信息
 */
struct backing_dev_info * backing_dev_info;

struct notifier_block reboot_notifier;          /* 重启前的默认模式 */

/* ECC 状态信息 */
struct mtd_ecc_stats ecc_stats;
/* NAND 中的页位移 */
int subpage_sft;

void * priv;

struct module * owner;
struct device dev;
int usecount;
};

```

mtd_info 的 type 字段给出底层物理设备的类型, 包括 MTD_RAM、MTD_ROM、MTD_NORFlash、MTD_NANDFlash 及 MTD_PEROM 等。

flags 字段包括 MTD_ERASEABLE(可擦除)、MTD_WRITEB_WRITEABLE(可编程)、MTD_XIP(可片内执行)、MTD_OOB(NAND 带外数据)及 MTD_ECC(支持自动 ECC)等。某些内存技术支持带外数据(OOB), 例如 NAND Flash 每 512 字节就有 16 个字节的“额外数据”, 用于存放纠错码或元数据。这是因为, 所有 Flash 器件都受位交换现象的困扰, 而 NAND 发生的概率比 NOR 大, 因此 NAND 厂商推荐使用 NAND 的时候最好使用 ECC(Error Checking and Correcting), 汉明码是最简单的 ECC。

ecctype 字段表明了 EC 的类型, 包括 MTD_ECC_NONE(不支持自动 ECC)、MTD_ECC_RS_DiskOnChip(DiskOnChip 上自动 ECC)、MTD_ECC_SW(Toshiba & Samsung 设备 SW ECC)。

mtd_info 中的 read()、write()、read_ecc()、write_ecc()、read_oob()及 write_oob()是 MTD 设备驱动要实现的主要函数, 在 NOR Flash 的驱动代码中几乎看不到 mtd_info 的成员函数, 即这些成员函数对于 Flash 芯片驱动是透明的, 这是因为 Linux 在 MTD 的下层实现了针对 NOR Flash 和 NAND Flash 的通用 mtd_info 成员函数。

13.3.2 mtd_table 结构体

mtd_table 结构体用来存放 mtd_info 的指针, 是多个 mtd_info 结构体的集合。mtdcore.c 中定义了 MTD 设备数组:

```
static struct mtd_info * mtd_table[MAX_MTD_DEVICES];
```

这是一个指向 mtd_info 结构体的静态数组指针, 最多可以有 MAX_MTD_DEVICES (默认定义为 16)个设备, 每个 MTD 分区也算一个 MTD 设备。

13.3.3 mtd_part 结构体

mtd_part 结构体用于描述分区,定义在 drivers/mtd/mtdpart.c 文件中,mtd_info 结构体成员用于描述本分区,会加入到 mtd_table 中,其大部分成员由主分区 mtd_part->master 决定,各种函数也指向主分区的相应函数,而主分区(其大小涵盖所有分区)不作为一个 MTD 原始设备加入 mtd_table,以下是 mtd_part 结构体的源代码。

```
struct mtd_part {
    struct mtd_info mtd;           //分区的信息
    struct mtd_info *master;       //该分区的主分区
    uint64_t offset;              //该分区的偏移地址
    struct list_head list;
};
```

13.3.4 mtd_partition 结构体

mtd_partition 在 MTD 原始设备层调用 add_mtd_partitions()时传递分区信息用,定义在 include/linux/mtd/partitions.h 文件中,以下是该结构体的代码。

```
struct mtd_partition {
    char * name;                  /* 标识字符串 */
    uint64_t size;                /* 分区大小 */
    uint64_t offset;             /* 主 MTD 空间内的偏移 */
    uint32_t mask_flags;         /* 掩码标志 */
    struct nand_ecclayout * ecclayout;
};
```

13.3.5 map_info 结构体

map_info 结构体主要是针对 NOR Flash 的结构体,定义在 include/linux/mtd/map.h 文件中,NOR Flash 驱动的核心就是定义 map_info 结构体,指定了 NOR Flash 的基址、位宽、大小等信息及 Flash 的读写函数。该结构体对于 NOR Flash 驱动至为关键,甚至 NOR Flash 驱动的代码本质可以认作是根据 map_info 探测芯片的过程,以下是其源代码。

```
struct map_info {
    const char * name;
    unsigned long size;
    resource_size_t phys;
#define NO_XIP (-1UL)

    void __iomem * virt;          //虚拟地址
    void * cached;

    int swap;
```

```

int bankwidth;                                /* 总线宽度 */

#ifdef CONFIG_MTD_COMPLEX_MAPPINGS
map_word (* read)(struct map_info *, unsigned long);
void (* copy_from)(struct map_info *, void *, unsigned long, ssize_t);

void (* write)(struct map_info *, const map_word, unsigned long);
void (* copy_to)(struct map_info *, unsigned long, const void *, ssize_t);
#endif

/* 缓存的虚拟地址 */
void (* inval_cache)(struct map_info *, unsigned long, ssize_t);

/* set_vpp() 函数是可重写的,其状态包括可用(enable)与不可用(disable) */
void (* set_vpp)(struct map_info *, int);

unsigned long pfow_base;
unsigned long map_priv_1;
unsigned long map_priv_2;
void * fldrv_priv;
struct mtd_chip_driver * fldrv;
};

```

13.4 驱动相关函数

13.4.1 add_mtd_device 函数

add_mtd_device 函数用来注册 MTD 设备,定义在 drivers/mtd/mtdcore.c 文件中,是 add_mtd_partitions 函数的一个底层调用,如果 Flash 不分区,则可以在驱动代码中直接调用此函数。

13.4.2 del_mtd_device 函数

del_mtd_device 函数用来注销 MTD 设备,定义在 drivers/mtd/mtdcore.c 文件中,是 del_mtd_partitions 函数的一个底层调用,如果 Flash 不分区,则可以在驱动代码中直接调用此函数。

13.4.3 add_mtd_partitions 函数

add_mtd_partitions() 对每一个新建分区建立一个新的 mtd_part 结构体,将其加入 mtd_partitions 中,并调用 add_mtd_device() 将此分区作为 MTD 设备加入 mtd_table。成功则返回 0,如果分配 mtd_part 时内存不足,则返回-ENOMEM。

add_mtd_partitions() 在 drivers/mtd/mtdpart.c 文件中定义,声明为: int add_mtd_partitions(struct mtd_info * master, struct mtd_partition * parts, int nbparts), 以下是其

源代码。

```
int add_mtd_partitions(struct mtd_info * master,
                      const struct mtd_partition * parts,
                      int nbparts)
{
    struct mtd_part * slave;
    uint64_t cur_offset = 0;
    int i;

    printk(KERN_NOTICE "Creating %d MTD partitions on \"%s\":\n", nbparts, master->name);

    for (i = 0; i < nbparts; i++) {
        slave = allocate_partition(master, parts + i, i, cur_offset);
        if (IS_ERR(slave))
            return PTR_ERR(slave);

        mutex_lock(&mtd_partitions_mutex);
        list_add(&slave->list, &mtd_partitions);
        mutex_unlock(&mtd_partitions_mutex);

        add_mtd_device(&slave->mtd);

        cur_offset = slave->offset + slave->mtd.size;
    }

    return 0;
}
```

add_mtd_partitions()中新建的 mtd_part 需要依赖传入的 mtd_partition 参数对其进行初始化。

13.4.4 del_mtd_partitions 函数

del_mtd_partitions()的作用是针对 mtd_partitions 上的每一个分区,定义在 drivers/mtd/mtdpart.c 文件中,如果主分区是 master(参数 master 是被删除分区的主分区),则将它从 mtd_partitions 和 mtd_table 中删除并释放掉,函数会调用 del_mtd_device()。

del_mtd_partitions()函数在 drivers/mtd/mtdpart.c 文件中的源代码如下。

```
int del_mtd_partitions(struct mtd_info * master)
{
    struct mtd_part * slave, * next;
    int ret, err = 0;

    mutex_lock(&mtd_partitions_mutex);
    list_for_each_entry_safe(slave, next, &mtd_partitions, list)
```

```

        if (slave->master == master) {
            ret = del_mtd_device(&slave->mtd);
            if (ret < 0) {
                err = ret;
                continue;
            }
            list_del(&slave->list);
            free_partition(slave);
        }
        mutex_unlock(&mtd_partitions_mutex);

    return err;
}

```

13.4.5 do_map_probe 函数

do_map_probe() 用来探测 Flash 是否得到 mtd_info, 定义在 drivers/mtd/chips/chipreg.c 文件中, 在 include/linux/mtd/map.h 下也有声明, 函数原型为:

```
struct mtd_info *do_map_probe(const char *name, struct map_info *map);
```

第一个参数为探测的接口类型, 常见的调用方法如下:

- (1) do_map_probe("cfi_probe", &xxx_map_info);
- (2) do_map_probe("jedec_probe", &xxx_map_info);
- (3) do_map_probe("map_rom", &xxx_map_info);

do_map_probe() 根据传入的参数 name, 通过 get_mtd_chip_driver() 得到具体的 MTD 驱动, 调用与接口对应的 probe() 函数探测设备, 以下是其源代码。

```

struct mtd_info *do_map_probe(const char *name, struct map_info *map)
{
    struct mtd_chip_driver *drv;
    struct mtd_info *ret;

    drv = get_mtd_chip_driver(name);

    if (!drv && !request_module("%s", name))
        drv = get_mtd_chip_driver(name);

    if (!drv)
        return NULL;

    ret = drv->probe(map);

```

/* 完成计数处理, 它有可能是一个仅具有探测的模块, 用于处理实际驱动的代码

```

    */
    module_put(drv->module);

    return ret;
}

```

13.5 Nor 型 Flash 驱动实例

本实例实现了一个简单的 Nor 型 Flash 驱动的开发,其功能是对申请的一段 Nor 型 Flash 实现简单的擦除和读写操作。

13.5.1 Nor 型 Flash 驱动设计流程

在 Linux 系统中,实现了针对 cfi、jedec 等接口的通用 Nor 驱动,这一层的驱动直接面向 mtd_info 的成员函数,使得 Nor 的芯片级驱动变得十分简单,只需要定义具体的内存映射情况结构体 map_info,并使用指定接口类型调用 do_map_probe() 函数。MTD、通用 NorFlash 的层次结构如图 13.3 所示。

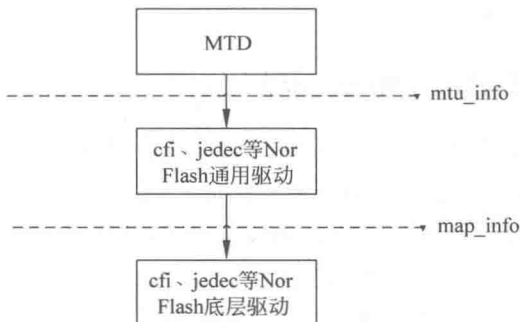


图 13.3 MTD、通用 NOR Flash 的层次结构

Nor Flash 驱动在 Linux 中的实现非常简单,如图 13.4 所示,主要的工作如下:

(1) 定义 map_info 的实例,初始化其中的成员,根据目标板的情况为 name、size、bankwidth 和 phys 赋值。

(2) 如果 Flash 要分区,则定义 mtd_partition 数组,将实际电路板中 Flash 分区信息记录其中。

(3) 以 map_info 和探测的接口类型(例如“cfi_probe”、“jedec_probe”等)为参数调用 do_map_probe(),探测 Flash 得到 mtd_info。

(4) 在模块初始化时以 mtd_info 为参数调用 add_mtd_device()或以 mtd_info、mtd_partition 数组及分区数为参数调用 add_mtd_partitions()注册设备或分区。在这之前可以调用 parse_mtd_partitions()查看 Flash 上是否已有分区信息,并将查看出的分区信息通过

add_mtd_partitions()注册。

(5) 在模块卸载时调用第 4 步函数的“反函数”删除设备或分区。

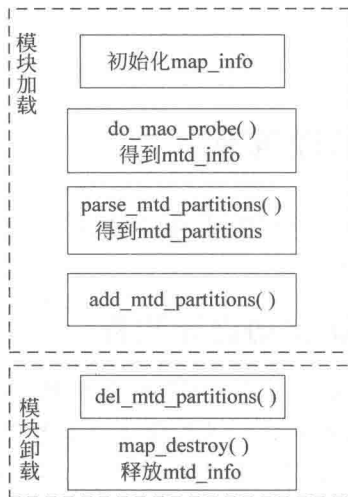


图 13.4 Nor Flash 驱动在 Linux 中的实现流程

13.5.2 Nor 型 Flash 驱动详细设计

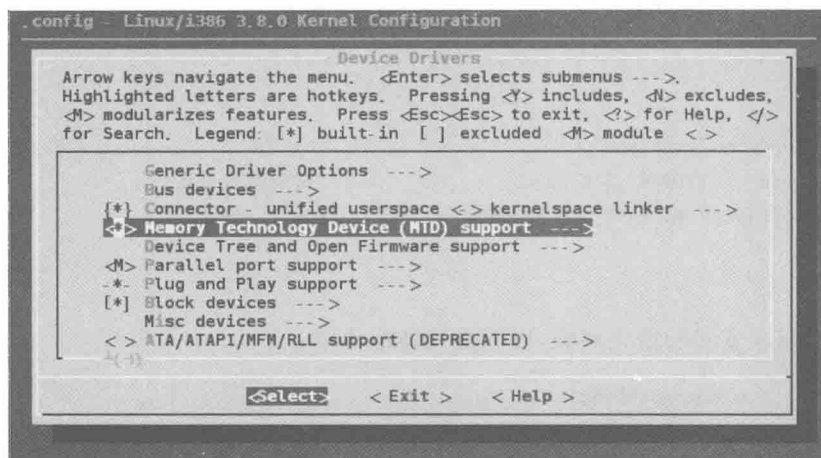
为了使系统能够支持 MTD,在 make menuconfig 对内核进行配置的时候将 Memory Technology Device(MTD)support、Direct char device access to MTD devices 和 Caching block device access to MTD devices 3 个选项勾选上,提供系统对 MTD 的支持和 MTD 对字符设备、块设备的支持,如图 13.5 所示。

在 drivers/mtd/chips/目录下创建一个 flash.c 文件,作为程序的源文件,接下来叙述代码的编写过程。

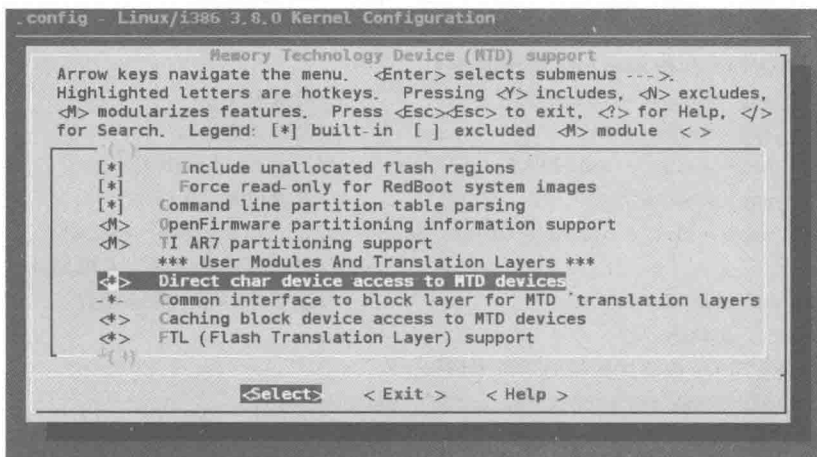
首先,声明头文件,以下是其具体代码。

```

#include <linux/module.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/device.h>
#include <linux/platform_device.h>
#include <linux/mtd/mtd.h>
#include <linux/mtd/map.h>
#include <linux/mtd/partitions.h>
#include <linux/mtd/physmap.h>
#include <asm/io.h>
  
```



(a)



(b)

图 13.5 内核配置选项

接着,声明 `del_mtd_partitions` 和 `add_mtd_partitions` 函数,以及 `mtd_info`、`map_info` 和 `mtd_partition` 结构体,以下是其源代码。

```
extern int add_mtd_partitions(struct mtd_info * master,
const struct mtd_partition * parts, int nbparts);
extern int dev_mtd_partitions(struct mtd_info * master);
static struct map_info * flash_nor_map;           //声明一个 map_info 对象
static struct mtd_info * flash_nor_mtd;          //声明一个 mtd_info 对象
static struct mtd_partition flash_nor_parts[] = {
[0] = {
    .name = "bootloader_nor",
```

```

        .size = 0x00040000,
        .offsize = 0,
    },
    [1] = {
        .name = "root_nor",
        .size = MTDPART_SIZ_FULL,           //剩余所有分区
        .offset = MTDPART_OFS_APPEND,       //紧接一分区
    }
};

```

然后编写模块安装初始化函数, 以下是其详细代码。

```

static int __init nor_init(void)
{
    /* 分配一个 map_info 结构体 */
    s3c_nor_map = kzalloc(sizeof(struct map_info), GFP_KERNEL);

    /* 设置上面分配的 map_info 结构体 */
    s3c_nor_map->name = "s3c_nor";           //设置名字
    s3c_nor_map->phys = 0;                   //设置物理地址
    s3c_nor_map->size = 0x1000000;           //设置 Nor Flash 大小
    s3c_nor_map->bankwidth = 2;              //设置位宽, 为 16 位
    s3c_nor_map->virt = ioremap(s3c_nor_map->phys, s3c_nor_map->size);
                                                //将物理地址映射为虚拟地址
    simple_map_init(s3c_nor_map);            //设置读写和擦除等函数
    printk("use cfi_probe\n");

    /* 调用 Nor Flash 协议层提供的函数来识别 */
    s3c_nor_mtd = do_map_probe("cfi_probe", s3c_nor_map);
    if (!s3c_nor_mtd)
    {
        printk("use jedec_probe\n");
        s3c_nor_mtd = do_map_probe("jedec_probe", s3c_nor_map);
    }

    if (!s3c_nor_mtd)
    {
        iounmap(s3c_nor_map->virt);
        kfree(s3c_nor_map);
        return -EIO;
    }

    /* 添加分区 */
    add_mtd_partitions(s3c_nor_mtd, s3c_nor_parts, 2);
    return 0;
}

```


编写模块退出函数,以下是其源代码。

```
static void __exit nor_exit(void)
{
    del_mtd_partitions(s3c_nor_mtd);
    iounmap(s3c_nor_map->virt);
    kfree(s3c_nor_map);
}
```

最后绑定模块安装初始化和退出函数并声明模块许可证,以下是其代码及注释。

```
/* 绑定模块安装初始化及退出函数 */
module_init(nor_init);
module_exit(nor_exit);
MODULE_LICENSE("GPL");           //声明模块许可证
```

修改 drivers/mtd/chips/目录下的 Makefile 文件,将代码 obj-\$(CONFIG_MTD_FLASH) += flash.o 加入文件中,以下是其源代码。

```
#
#linux/drivers/chips/Makefile
#

obj-$(CONFIG_MTD) += chipreg.o
obj-$(CONFIG_MTD_CFI) += cfi_probe.o
obj-$(CONFIG_MTD_CFI_UTIL) += cfi_util.o
obj-$(CONFIG_MTD_CFI_STAA) += cfi_cmdset_0020.o
obj-$(CONFIG_MTD_CFI_AMDSTD) += cfi_cmdset_0002.o
obj-$(CONFIG_MTD_CFI_INTELEXT) += cfi_cmdset_0001.o
obj-$(CONFIG_MTD_GEN_PROBE) += gen_probe.o
obj-$(CONFIG_MTD_JEDEC_PROBE) += jedec_probe.o
obj-$(CONFIG_MTD_RAM) += map_ram.o
obj-$(CONFIG_MTD_ROM) += map_rom.o
obj-$(CONFIG_MTD_ABSENT) += map_absent.o
obj-$(CONFIG_MTD_FLASH) += flash.o
```

然后,修改 drivers/mtd/chips/目录下的 Kconfig 函数,改变配置,在文件中加入代码 config MTD_FLASH tristate “==== My Flash =====”。

之后,进入到内核根目录,有的机器是在 /usr/src/linux/目录下,执行 makeconfig 命令,在配置选项里选择上自定义的“==== My flash =====”这一项,如图 13.6 所示。

在完成配置之后,依次执行 make 命令、make modules_install 命令及 make install 命令,执行完成后,自定义的 Flash 驱动就成功加载到内核中了。

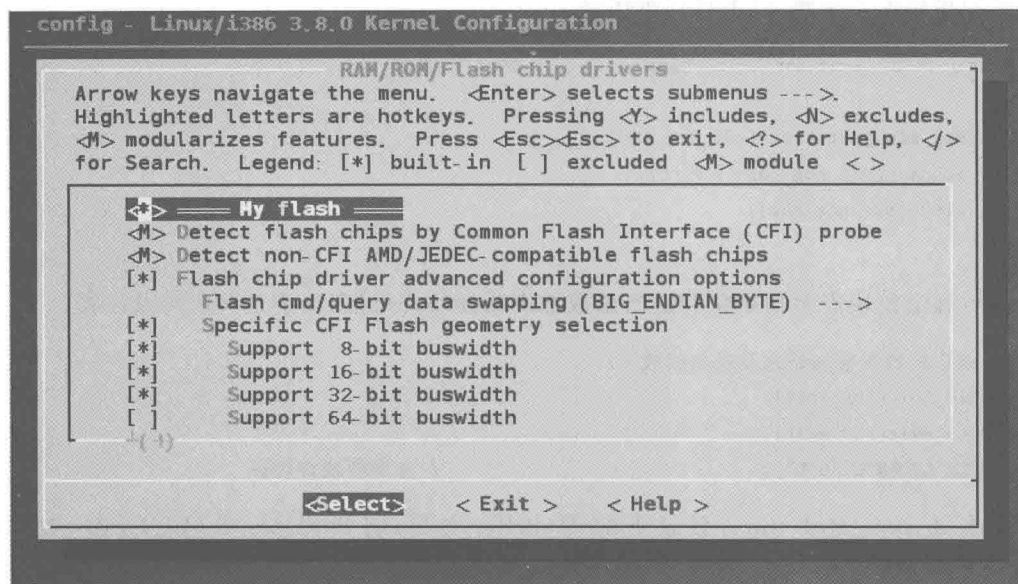


图 13.6 重新编译内核配置选项

13.6 Nand 型 Flash 驱动实例

13.6.1 Nand 型 Flash 设备驱动设计步骤

有了 MTD 层,完成 1 个 Nand Flash 驱动在 Linux 中的工作量很小,如图 13.7 所示,主要工作如下:

(1) 如果 Flash 要分区,则定义 mtd_partition 数组,将实际电路板中 Flash 分区信息记录其中。

(2) 在模块加载时,为每一个 chip(主分区)分配 mtd_info 和 nand_chip 的内存,根据目标板 nand 控制器的特殊情况,初始化 nand_chip 中的 hwcontrol()、dev_ready()等成员函数(如果不初始化,将会使用 nand_base.c 中的默认函数),填充 mtd_info,并将其成员 priv 指向 nand_chip。

(3) 以 mtd_info 为参数,调用 nand_scan()函数探测 Nand Flash 的存在。nand_scan()函数从 Flash 芯片中读取参数,填充相应的 nand_chip 成员。

(4) 如果要分区,则以 mtd_info 和 mtd_partition 为参数调用 add_mtd_partitions(),添加分区信息。

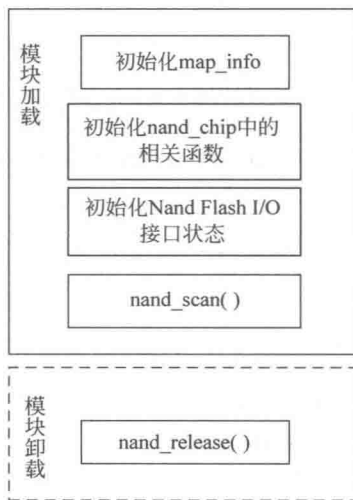


图 13.7 Nand 型 Flash 驱动

13.6.2 Nand 型 Flash 驱动实现

Nand 型 Flash 驱动实现, 以下是其源代码。

```
#define CHIP_PHYSICAL_ADDRESS ...
#define NUM_PARTITIONS 2
static struct mtd_partition partition_info[] = //定义 partition 数组
{
    {
        .name = "Flash partition 1", .offset = 0, .size = 8 * 1024 * 1024
    },
    {
        .name = "Flash partition 2", .offset = MTDPART_OFS_NEXT, .size = MTDPART_SIZ_FULL
    },
};

int __init board_init(void) //完成芯片初始化工作, 包括分配内存,
                           //根据特殊情况选择实现函数
{
    struct nand_chip *this;
    int err = 0;
    /* 为 MTD 设备结构体和 nand_chip 分配内存 */
    board_mtd = kmalloc(sizeof(struct mtd_info) + sizeof(struct nand_chip), GFP_KERNEL);
    if(!board_mtd)
    {
        printf("Unable to allocate NAND MTD device structure.\n");
        err = -ENOMEM;
    }
}
```

```

        goto out;
    }
    /* 初始化结构体 */
    memset((char *)board_mtd, 0, sizeof(struct mtd_info) + sizeof(struct nand_chip));
    /* 映射物理地址 */
    baseaddr = (unsigned long) ioremap(CHIP_PHYSICAL_ADDRESS, 1024);
    if(!baseaddr)
    {
        printk("Ioremap to access NAND chip failed\n");
        err = -EIO;
        goto out_mtd;
    }

    /* 获得私有数据(nand_chip)指针 */
    this = (struct nand_chip *)(&board_mtd[1]);
    /* 将 nand_chip 赋予 mtd_info */
    board_mtd->priv = this;

    /* 设置 Nand Flash 的 I/O 基地址 */
    this->IO_ADDR_R = baseaddr;
    this->IO_ADDR_W = baseaddr;
    /* 硬件控制函数 */
    this->hwcontrol = board_hwcontrol;
    /* 从数据手册获知命令延迟时间 */
    this->chip_delay = CHIP_DEPENDEND_COMMAND_DELAY;
    /* 初始化设备 ready 函数 */
    this->dev_ready = board_dev_ready;
    this->eccmode = NAND_ECC_SOFT;

    if(nand_scan(board_mtd, 1))                                //扫描以确定设备的存在
    {
        err = -ENXIO;
        goto out_ior;
    }
    add_mtd_partitions(board_mtd, partition_info, NUM_PARTITIONS);    //添加分区
    goto out;
    out_ior: iounmap((void *)baseaddr);
    out_mtd: kfree(board_mtd);
    out: return err;
}

//下面是选择初始化的成员函数
static void __exit board_cleanup(void)
{
    /* 释放资源,注销设备 */
    nand_release(board_mtd);
    /* unmap 物理地址 */
    iounmap((void *)baseaddr);

```

```

    /* 释放 MTD 设备结构体 */
    kfree(board_mtd);
}

/* GPIO 方式的硬件控制 */
static void board_hwcontrol(struct mtd_info *mtd, int cmd)    //hwcontrol()是芯片特定的
                                                            //硬件控制函数
{
    /* 判断控制命令的类型 */
    switch(cmd)
    {
        case NAND_CTL_SETCLE:
            break;
        case NAND_CTL_CLRCLE:
            break;
        case NAND_CTL_SETALE:
            break;
        case NAND_CTL_CLRALE:
            break;
        case NAND_CTL_SETNCE:
            break;
        case NAND_CTL_CLRNCE:
            break;
    }
}

/* 返回设备 ready 状态 */
static int board_dev_ready(struct mtd_info *mtd)
{
    return xxx_read_ready_bit();
}

```

参考文献

- [1] 曹国辉. 深入理解嵌入式 Linux 设备驱动程序[M]. 北京:电子工业出版社,2012.
- [2] 陈莉君,张琼声,张宏伟,译. 深入理解 LINUX 内核[M]. 北京:电力出版社,2007.
- [3] 张光建. 嵌入式 Linux 驱动程序开发实例教程[M]. 北京:清华大学出版社,2012.
- [4] 陈学松. 深入 Linux 设备驱动程序内核机制[M]. 北京:电子工业出版社,2012.
- [5] 宋宝华,何昭然,史海滨,吴国成,译. 精通 Linux 设备驱动程序开发[M]. 北京:人民邮电出版社,2010.
- [6] 魏永明,耿岳,钟书毅,译. Linux 设备驱动程序(第三版)[M]. 北京:电力出版社,2006.
- [7] 刘森. 嵌入式系统接口设计与 Linux 驱动程序开发[M]. 北京:北京航空航天大学出版社,2006.
- [8] Venkateswaran S. Essential Linux Device Drivers[M]. Prentice Hall Press, 2008.
- [9] 陈明计,陈渝. ARM 嵌入式 Linux 系统构建与驱动开发范例[M]. 北京:北京航空航天大学出版社,2006.
- [10] 宋宝华. Linux 设备驱动开发详解[M]. 北京:人民邮电出版社,2008.
- [11] 吴国伟等. Linux 内核分析与高级编程[M]. 北京:清华大学出版社,2012.
- [12] 李俊. 嵌入式 Linux 设备驱动开发详解[M]. 北京:人民邮电出版社,2008.
- [13] 郑强等. Linux 驱动开发入门与实战[M]. 北京:清华大学出版社,2011.

随着物联网和人工智能的发展，Linux将更多地应用于嵌入式设备中，这对Linux内核中各种驱动的设计和实现也提出了更高的要求。

Linux内核版本不断升级，其设备管理方式也发生变化，内核提供的设备管理的关键数据结构和函数也产生变化，尤其是随着新的硬件体系结构变化和新型外围设备的出现，内核设备管理也随之不断变化。本书基于最新的Linux 3.8.13 内核，通过13章内容（包括Linux内核、驱动开发基础、驱动开发实例），全面深入地论述了Linux设备驱动开发的全方位技术……

主要内容

- Linux内核设备管理方式
- Linux驱动开发基础
- Linux字符设备驱动开发
- Linux内核中断机制
- Linux块设备驱动开发
- Linux网络设备驱动开发
- Linux MMC/SD驱动开发
- Linux USB驱动开发
- Linux I2C总线设备驱动
- Linux PCI总线设备驱动
- Linux输入设备驱动
- Linux Flash驱动开发

清华大学出版社数字出版网站

WQBook  书文局泉
www.wqbook.com



[General Information]

书名=深入理解Linux驱动程序设计

作者=吴国伟，姚琳，毕成龙编著

页数=192

SS号=13880224

DX号=

出版日期=2015.11

出版社=北京清华大学出版社

封面

书名

版权

前言

目录

第1章 Linux内核组成和机制

1.1 Linux内核版本与发展

1.1.1 Linux操作系统的诞生

1.1.2 Linux内核版本的变迁

1.2 Linux内核编译

1.2.1 获取内核源码

1.2.2 内核源码树

1.2.3 编译内核

1.3 Linux内核组成

1.4 Linux内核机制

1.4.1 内核启动过程

1.4.2 模块机制

第2章 Linux内核设备管理方式

2.1 devfs设备文件系统

2.2 sysfs文件系统

2.3 udev设备文件系统

2.4 主要数据结构

2.4.1 kobject

2.4.2 ktype

2.4.3 kset

2.4.4 三者关系

2.5 热插拔设备管理机制

2.5.1 热插拔事件流程

2.5.2 涉及的模块

2.5.3 关键驱动函数

第3章 Linux驱动开发基础

3.1 同步机制

3.1.1 内核同步机制分类

3.1.2 自旋锁与信号量的比较

3.2 make及makefile

3.2.1 makefile文件

3.2.2 编写makefile文件

- 3.2.3 make命令
- 3.3 调试方法
 - 3.3.1 printk
 - 3.3.2 /proc文件系统
 - 3.3.3 调试器及相关工具
- 第4章 Linux字符设备驱动开发
 - 4.1 关键数据结构
 - 4.2 接口函数部分内核代码分析
 - 4.3 字符设备驱动设计
 - 4.3.1 字符设备驱动设计场景描述
 - 4.3.2 字符设备驱动设计过程
- 第5章 Linux内核中断机制
 - 5.1 中断
 - 5.2 中断处理
 - 5.2.1 注册中断处理程序
 - 5.2.2 编写中断处理程序
 - 5.3 中断上半部与下半部的对比
 - 5.4 中断下半部
 - 5.5 BH机制与任务队列机制
 - 5.6 软中断
 - 5.6.1 软中断的实现
 - 5.6.2 软中断的使用
 - 5.7 tasklet
 - 5.7.1 tasklet的实现
 - 5.7.2 tasklet的使用
 - 5.8 工作队列
 - 5.8.1 工作队列的实现
 - 5.8.2 工作队列的使用
- 第6章 Linux块设备驱动开发
 - 6.1 块设备管理机制
 - 6.1.1 块设备基本概念
 - 6.1.2 块设备在Linux中的结构
 - 6.2 块设备关键数据结构
 - 6.2.1 gendisk数据结构
 - 6.2.2 block_device_operations数据结构
 - 6.2.3 request数据结构
 - 6.2.4 request_queue数据结构

- 6.2.5 bio数据结构
- 6.3 块设备驱动设计函数
 - 6.3.1 块设备驱动注册与注销函数
 - 6.3.2 块设备驱动打开与关闭函数
 - 6.3.3 块设备驱动ioctl、read和write函数
 - 6.3.4 块设备驱动的请求函数
- 6.4 Ramdisk块设备驱动实例
 - 6.4.1 Ramdisk块设备驱动实例分析
 - 6.4.2 Ramdisk块设备驱动实例测试
- 第7章 Linux网络设备驱动开发
 - 7.1 网络设备
 - 7.1.1 网络系统分层结构
 - 7.1.2 网络设备管理
 - 7.2 NAPI机制
 - 7.3 关键数据结构
 - 7.4 内核提供的网络设备驱动设计函数
 - 7.4.1 alloc_netdev
 - 7.4.2 register_netdev
 - 7.4.3 ether_setup
 - 7.4.4 unregister_netdev
 - 7.5 网络设备驱动开发实例
 - 7.5.1 snull_init_module函数
 - 7.5.2 snull_init函数
 - 7.5.3 相关操作函数
- 第8章 Linux MMC/SD驱动开发
 - 8.1 MMC子系统基本架构
 - 8.2 关键数据结构
 - 8.2.1 基本数据结构
 - 8.2.2 基本数据结构主要成员及关系
 - 8.3 MMC/CD卡驱动实例
 - 8.3.1 MMC/SD卡设备驱动设计场景
 - 8.3.2 MMC/SD卡设备驱动实例实现
- 第9章 LinuxUSB驱动开发
 - 9.1 USB设备管理机制
 - 9.1.1 USB与串口
 - 9.1.2 USB设备属性拓扑结构管理机制
 - 9.1.3 USB设备逻辑组织管理机制

- 9.2 USB驱动关键数据结构分析
- 9.3 USB设备驱动函数及其使用说明
 - 9.3.1 客户端驱动管理
 - 9.3.2 USB设备配置和管理
 - 9.3.3 主机控制器的管理
 - 9.3.4 协议控制命令集和数据传输管理
- 9.4 USB设备驱动开发实例
 - 9.4.1 实例开发场景设计
 - 9.4.2 USB设备驱动开发实例的实现
 - 9.4.3 驱动测试分析
- 第10章 Linux I2C总线设备驱动
 - 10.1 Linux总线驱动及I2C总线
 - 10.1.1 Linux总线驱动设计过程
 - 10.1.2 I2C总线的工作原理与应用
 - 10.1.3 总线基本操作
 - 10.2 Linux I2C体系结构
 - 10.2.1 Linux的I2C体系结构组成
 - 10.2.2 Linux I2C关键数据结构
 - 10.3 Linux I2C核心
 - 10.4 Linux I2C总线驱动
 - 10.4.1 I2C适配器驱动加载与卸载
 - 10.4.2 I2C总线通信方法
 - 10.5 Linux I2C设备驱动
 - 10.5.1 Linux I2C设备驱动模块加载与卸载
 - 10.5.2 Linux I2C设备驱动的数据传输
 - 10.5.3 Linux i2c-dev.c文件分析
 - 10.6 Linux I2C驱动实例——EEPROM
 - 10.6.1 初始化
 - 10.6.2 探测设备
 - 10.6.3 检查适配器的功能
 - 10.6.4 访问设备
 - 10.6.5 其他函数
- 第11章 Linux PCI总线设备驱动
 - 11.1 PCI总线设备
 - 11.1.1 PCI总线
 - 11.1.2 PCI设备
 - 11.2 PCI设备驱动结构

11.3 PCI设备驱动实例

- 11.3.1 PCI设备驱动程序基本框架
- 11.3.2 初始化设备模块
- 11.3.3 打开设备模块
- 11.3.4 数据读写和控制信息模块
- 11.3.5 中断处理模块
- 11.3.6 释放设备模块
- 11.3.7 卸载设备模块

第12章 Linux输入设备驱动

- 12.1 Linux输入子系统结构
- 12.2 输入设备驱动核心数据结构分析
- 12.3 Linux输入设备驱动实例
 - 12.3.1 输入设备驱动流程
 - 12.3.2 USB鼠标驱动编写实例

第13章 Linux Flash驱动开发

- 13.1 Flash存储器
- 13.2 Linux MTD系统层次结构
- 13.3 关键数据结构
 - 13.3.1 mtd_info结构体
 - 13.3.2 mtd_table结构体
 - 13.3.3 mtd_part结构体
 - 13.3.4 mtd_partition结构体
 - 13.3.5 map_info结构体
- 13.4 驱动相关函数
 - 13.4.1 add_mtd_device函数
 - 13.4.2 del_mtd_device函数
 - 13.4.3 add_mtd_partitions函数
 - 13.4.4 del_mtd_partitions函数
 - 13.4.5 do_map_probe函数
- 13.5 Nor型Flash驱动实例
 - 13.5.1 Nor型Flash驱动设计流程
 - 13.5.2 Nor型Flash驱动详细设计
- 13.6 Nand型Flash驱动实例
 - 13.6.1 Nand型Flash设备驱动设计步骤
 - 13.6.2 Nand型Flash驱动实现

参考文献

封底