HZ BOOKS
华章科技

Linux/Unix
技术丛书

Linux内核探秘

深入解析文件系统和设备驱动的架构与设计

高剑林 著

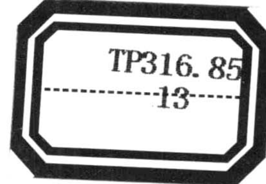
Explore Linux Kernel

In-depth Analysis of Architecture and Design about File System and Device Driver

- 腾讯资深Linux内核专家10余年工作经验结晶，业界多位专家联袂推荐，Linux内核工程师和驱动开发工程师的必读之作
- 从工业需求角度另辟蹊径，注重效率和实用性，将Linux内核分为基础部分和应用部分以及内核架构和内核实现两个维度，对Linux内核的文件系统、设备驱动的架构设计与实现原理进行了深入分析

5.85

机械工业出版社
China Machine Press


TP316.85
13

Linux内核探秘

深入解析文件系统和设备驱动的架构与设计

Explore Linux Kernel
In-depth Analysis of Architecture and Design about File System and Device Driver

高剑林 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Linux内核探秘：深入解析文件系统和设备驱动的架构与设计 / 高剑林著 .—北京：机械工业出版社，2013.12
（Linux/UNIX技术丛书）

ISBN 978-7-111-44585-2

Ⅰ. L… Ⅱ. 高… Ⅲ. Linux操作系统 Ⅳ. TP316.89

中国版本图书馆CIP数据核字（2013）第256030号

版权所有・侵权必究
封底无防伪标均为盗版
本书法律顾问 北京市展达律师事务所

本书从工业需求角度出发，注重效率和实用性，是帮助内核研发及调试、驱动开发等领域工程师正确认识并高效利用Linux内核的难得佳作！作者是腾讯公司资深的Linux内核专家和存储系统专家，在该领域工作和研究的10余年间，面试了数百位Linux内核工程师，深知学习Linux内核过程中经常遇到的困惑，以及在工作中容易犯的错误。基于这些原因作者撰写了本书。本书出发点和写作方式可谓独辟蹊径，将Linux内核分为两个维度，一是基础部分和应用部分，二是内核架构和内核实现，将两个维有机统一，深入分析了Linux内核的文件系统、设备驱动的架构设计与实现原理。

全书在逻辑上分为三部分：第一部分（第1 ~ 2章）首先将内核层划分为基础层和应用层，讲解了基础层包含的服务和数据结构，以及应用层包含的各种功能，然后对文件系统的架构进行了提纲挈领的介绍，为读者学习后面的知识打下基础；第二部分（第3 ~ 9章）从设备到总线到驱动，逐步深入，剖析了设备的总体架构、为设备服务的特殊文件系统sysfs、字符设备和input设备、platform总线、serio总线、PCI总线、块设备的实现原理和工作机制；第三部分（第10 ~ 13章）对文件系统的读写机制进行了深入分析，最后通过一个真实文件系统ext2，复习本书所有知识点。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：白 宇
北京市荣盛彩色印刷有限公司印刷
2014年1月第1版第1次印刷
186mm × 240 mm ・ 14.5印张
标准书号：ISBN 978-7-111-44585-2
定 价：59.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

前　言

为什么要写这本书

目前市面上关于 Linux 内核方面的书籍可以分为两类，一类是学院派书籍，其中比较有名的包括《深入理解 Linux 内核》(ULK）等；一类是国内特有的培训教材。大体而言，学院派的书籍体系一般很完整，在广度和深度上都有完善的阐述。可惜也正因为它的完整、复杂和庞大，使得阅读学院派的书籍往往是个艰巨的任务。ULK 这本书已经有八百页的体量，还有很多细节知识没有讲述，而早期的《内核情景分析》一书更是达到上千页的体量。以至于业界公认内核的学习曲线最陡峭，学习难度最大。

而本书是从工业界角度出发，为工业界使用而写。比较关注计算机科学方面进展的工程师，应可以意识到计算机科学和计算机工业是两个不同的领域。前者注重创新和理论完备，后者注重效率和实用。从效率和实用的角度，需要在降低学习难度的基础上提供相对体系化的结构，这就必须对庞大复杂的内核进行分解和抽取，这也正是本书试图将内核分解为基础层和应用层的原因。

这些年来，笔者先后面试过上百位内核工程师，组织过多次讲座或者交流会议，和国内多家公司的一流工程师有过深入交流。总体而言，国内内核应用和开发的水平处于非常低的水平，这一方面表现在理解内核的技术人员在国内总体上不多，即使是专业的内核的工程师，对内核的一些基本问题理解不清甚至理解错误的也不在少数；另一方面是大多数人认为内核在工作中用处不大，很难发挥价值。

针对第一个问题，笔者做过调查问卷，通过调查发现，公认学习内核最大的问题就是内核代码的难懂和跳跃。从一个函数跳到另一个函数，然后又跳到下一个函数，对执行的逻辑难以理解。跳跃超过三次，基本就难以继续，只能放弃。第二个问题和第一个问题强相关。因为了解不够系统，很难形成整体的内核执行逻辑。而实际工作中碰到的问题总是千变万化，个人了解的一块未必能碰到。比如一个文件系统只读问题，是内核 VFS 层的问题？是文件系统自身？还是块设备或者硬盘的问题？如果不能形成清晰的视图，就很难有针对性的调试和改进。

按照方法论的观点，通常人类的学习过程是从易到难、从部分到整体、从已知到未知。而对内核的学习有其特殊之处，内核几乎是九十度的学习曲线，极难找到入门的路径，更别说快速流畅地阅读内核代码了。从那时起，笔者开始对内核进行整理，希望能找到一条学习的路径，在不断探索过程中，逐渐形成一份文档，然后通过一些培训活动验证了其有效，最终形成了本书。

本书可以归纳为两个思路。一个是对内核代码的分类。笔者把内核分为基础部分和应用部分。内核中的内存管理、任务调度和中断异常处理归为基础部分。而文件系统、设备管理和驱动归为应用部分。打开一份完整的内核代码统计一下，应用部分占了绝大多数，庞大复杂，但冗余很多，很多代码具有相似性；而基础部分则是短小精悍。应用部分经常要调用基础部分提供的内存管理、任务调度等服务。为了快速理解基础部分，首先要整理基础部分的服务，理解在内核中如何使用基础部分的服务。

第二个思路是把内核分为内核架构和内核实现。内核架构是内核中通用的、具普遍性的层次，比如块设备、字符设备、总线、文件系统的 VFS 层等。理解了内核架构，就对内核有了整体上的掌握，就能了解内核设计者的思路，进而快速流畅地阅读内核代码。但即使理解了内核架构，也还有很多具体问题要攻克。比如驱动中一个寄存器的使用、设备链路状态如何检测、文件系统如何使用 barrier I/O、同步和异步 I/O 的区别等。这是需要开发人员仔细研读和琢磨的。本书试图归纳整理出内核的常用架构层，这些架构层具有举一反三的作用，它们构成了 Linux 内核的骨架。

发展到今天，内核已经非常庞大和复杂。本书希望通过一些架构层次代码的分析，结合简单的例子，帮助读者理解内核的整体框架。当碰到内核问题或者需要加入某些内核功能或者修改某些实现时，可以迅速流畅地阅读相关代码，确定自己的方案，而不至于茫然无措。而对于细节的实现，则需要程序员根据自己的需求来设计。

关于内核版本，本书用的是 2.6.18 版。内核有一套自己的不兼容策略，不同内核版本之间经常不能编译，至于函数消失和数据结构修改更是家常便饭。所以我们只能选择一个版本作为基础。

阅读内核代码前的准备：下载一份完整的内核，Linux 内核的官方网站是 http://www.kernel.org，这里可以下载到各个版本的内核；再准备一个好的代码阅读软件，因为内核代码经常要前后关联阅读，所以需要具有代码工程管理的软件，强烈推荐 source insight，这是国内应用很广的一个软件。

另外，本书已经假设读者能编译和安装模块，并且具有计算机基本结构的知识。此外，有一台已安装了 Linux 系统的计算机或者虚拟机，并且经常实战练习。

由于笔者水平有限，而且从架构层次分析内核代码，可用来参考的资料很少，希望广大读者能多提意见，共同推进中国的技术水平。

任何书籍都不能替代读者自己对内核实际代码的研究和学习。但如果没有书籍，浩如云烟的内核代码让有志学习者茫然，而低效率地一点点啃代码也会浪费大量的时间。书籍的作用是带领读者入门，读者需要尽快转入自我学习阶段，对需要的部分代码自行分析和研究。

试读结束，需要全本PDF请购买 www.ertongbook.com

读者对象

本书适合以下读者阅读参考：

- ❑大专院校在校学生；
- ❑对系统内核感兴趣，有志于从事内核研发的人员；
- ❑从事驱动开发的工程师；
- ❑从事操作系统内核方面工作的工程师；
- ❑负责 Linux 系统调试和优化的技术人员。

如何阅读本书

本书将整个内核分为基础层和应用层。这种划分大大减少了阅读内核的难度，但是仍然需要对基础层有全面正确的理解。本书第 1 章介绍了内核的基础层，读者应该多做一些实践练习，才能加深理解。

第 2 章是本书提纲挈领的一章，重点介绍了文件系统的基础知识。文件系统在应用层的位置非常重要，因此只有掌握了文件系统的重要概念、理解了基本的操作过程才能为整体理解内核打下良好的基础。

第 3 ~ 9 章是关于设备的章节。建议读者结合具体的设备，从设备到总线再到驱动，逐步深入。本书的章节安排遵循从易到难、代码结合实例的方式，相信读者可以比较顺畅地阅读并理解。

第 10 ~ 13 章，再次对文件系统的读写和内核通用块层进行阐述。阅读的过程中，读者若能结合实际做一些小的程序，则可以帮助迅速提升能力。比如自己实现内核的 I/O 路径或者实现一个模拟的块设备系统，实践中应用才是能力提升的最佳途径。

勘误和支持

由于笔者的水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。书中的全部源文件可以从华章网站⊖下载。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱 easyblue99@hotmail.com，期待能够得到你们的真挚反馈。

致谢

感谢 IBM 公司的小强、中兴的老谢、百度的子旬、索尼的大 A 以及淘宝、Intel、微软、搜狐、

⊖ 参见华章网站 www.hzbook.com。——编辑注

新浪等公司的各位朋友。与你们的一次次讨论澄清了很多概念和问题，使我受益良多。

感谢机械工业出版社华章公司的编辑白宇，是你认真审核每一章的内容，提高了书稿的质量。编辑是个清苦的职业，远远谈不上令人羡慕，但不论多少生活的烦恼，白女士总是耐心修改，职业精神令人敬佩。

感谢我的亲人们，特别要感谢我的母亲，她是中国核工业的早期建设者，在艰难中养育了三个孩子，谨以此书献给我的母亲杨玉芳女士。感谢我的妻子和女儿，你们是我永远的动力之源。

目录

前　言

第 1 章　内核的基础层和应用层 ……1

1.1 内核基础层提供的服务 ……1

1.1.1 内核中使用内存 ……2

1.1.2 内核中的任务调度 ……2

1.1.3 软中断和 tasklet ……3

1.1.4 工作队列 ……4

1.1.5 自旋锁 ……5

1.1.6 内核信号量 ……5

1.1.7 原子变量 ……5

1.2 内核基础层的数据结构 ……6

1.2.1 双向链表 ……6

1.2.2 hash 链表 ……6

1.2.3 单向链表 ……7

1.2.4 红黑树 ……7

1.2.5 radix 树 ……7

1.3 内核应用层 ……8

1.4 从 Linux 内核源码结构纵览内核 ……9

1.5 内核学习和应用的四个阶段 ……10

1.6 本章小结 ……11

第 2 章　文件系统 ……12

2.1 文件系统的基本概念 ……12

2.1.1 什么是 VFS ……13

2.1.2 超级块 super_block ……13

2.1.3 目录项 dentry ……14

2.1.4 索引节点 inode ……15

2.1.5 文件 ……17

2.2 文件系统的架构 ……17

2.2.1 超级块作用分析 ……17

2.2.2 dentry 作用分析 ……18

2.2.3 inode 作用分析 ……20

2.2.4 文件作用分析 ……21

2.3 从代码层次深入分析文件系统 ……21

2.3.1 一个最简单的文件系统 aufs ……22

2.3.2 文件系统如何管理目录和文件 ……26

2.3.3 文件系统的挂载过程 ……38

2.3.4 文件打开的代码分析 ……42

2.4 本章小结 ……59

第 3 章　设备的概念和总体架构 ……60

3.1 设备的配置表 ……60

3.2 访问设备寄存器和设备内存 ……61

3.3 设备中断和 DMA ……61

3.4 总线对设备的扫描 ……62

3.5 设备驱动管理……………………62
3.6 本章小结……………………62

第4章 为设备服务的特殊文件系统 sysfs……………………63

4.1 文件和目录的创建……………………63
4.1.1 sysfs 文件系统的初始化……64
4.1.2 sysfs 文件系统目录的创建…64
4.1.3 普通文件的创建……………………68
4.2 sysfs 文件的打开操作……………………69
4.2.1 real_lookup 函数详解……………………70
4.2.2 为文件创建 inode 结构……70
4.2.3 为 dentry 结构绑定属性……71
4.2.4 调用文件系统中的 open 函数……………………72
4.3 sysfs 文件的读写……………………74
4.3.1 读文件的过程分析……………………74
4.3.2 写文件的过程分析……………………75
4.4 kobject 结构……………………76
4.4.1 kobject 和 kset 的关系……………………76
4.4.2 kobject 实例：总线的注册…77
4.5 本章小结……………………79

第5章 字符设备和 input 设备……………………80

5.1 文件如何变成设备……………………80
5.1.1 init_special_inode 函数……80
5.1.2 def_chr_fops 结构……………………81
5.2 input 设备的注册……………………82
5.2.1 主从设备号……………………83
5.2.2 把 input 设备注册到系统……84
5.2.3 设备区间的登记……………………85
5.2.4 注册字符设备……………………86
5.2.5 打开 input 设备……………………87
5.3 input 设备架构……………………88
5.3.1 注册 input 设备的驱动……88
5.3.2 匹配 input 管理的设备和驱动……………………89
5.3.3 注册 input 设备……………………90
5.4 本章小结……………………92

第6章 platform 总线……………………93

6.1 从驱动发现设备的过程……………………93
6.1.1 驱动的初始化……………………93
6.1.2 注册驱动……………………94
6.1.3 为总线增加一个驱动……95
6.1.4 驱动加载……………………95
6.1.5 遍历总线上已经挂载的设备…96
6.2 从设备找到驱动的过程……………………98
6.2.1 注册设备和总线类型……98
6.2.2 注册设备的资源……………………99
6.2.3 增加一个设备对象……………………100
6.3 本章小结……………………102

第7章 serio 总线……………………103

7.1 什么是总线适配器……………………103
7.2 向 serio 总线注册设备……………………103
7.2.1 注册端口登记事件……………………104
7.2.2 遍历总线的驱动……………………106
7.2.3 注册 input 设备……………………109
7.3 虚拟键盘驱动……………………110
7.3.1 键盘驱动的初始化……………………110
7.3.2 与设备建立连接……………………111

7.3.3 启动键盘设备 111
7.3.4 输入设备和主机系统之间的事件 112
7.4 键盘中断 112
7.4.1 q40kbd 设备的中断处理 113
7.4.2 serio 总线的中断处理 113
7.4.3 驱动提供的中断处理 113
7.5 本章小结 116

第 8 章 PCI 总线 117

8.1 深入理解 PCI 总线 117
8.1.1 PCI 设备工作原理 117
8.1.2 PCI 总线域 118
8.1.3 PCI 资源管理 118
8.1.4 PCI 配置空间读取和设置 119
8.2 PCI 设备扫描过程 120
8.2.1 扫描 0 号总线 120
8.2.2 扫描总线上的 PCI 设备 121
8.2.3 扫描多功能设备 124
8.2.4 扫描单个设备 125
8.2.5 扫描设备信息 125
8.3 本章小结 128

第 9 章 块设备 129

9.1 块设备的架构 129
9.1.1 块设备、磁盘对象和队列 129
9.1.2 块设备和通用磁盘对象的绑定 130
9.1.3 块设备的队列和队列处理函数 131
9.2 块设备创建的过程分析 132
9.2.1 nbd 驱动的初始化 132
9.2.2 为通用磁盘对象创建队列成员 133
9.2.3 将通用磁盘对象加入系统 134
9.3 块设备文件系统 135
9.3.1 块设备文件系统的初始化 135
9.3.2 块设备文件系统的设计思路 136
9.4 块设备的打开流程 136
9.4.1 获取块设备对象 137
9.4.2 执行块设备的打开流程 140
9.5 本章小结 142

第 10 章 文件系统读写 143

10.1 page cache 机制 143
10.1.1 buffer I/O 和 direct I/O 143
10.1.2 buffer head 和块缓存 143
10.1.3 page cache 的管理 144
10.1.4 page cache 的状态 145
10.2 文件预读 146
10.3 文件锁 146
10.4 文件读过程代码分析 147
10.5 读过程返回 161
10.6 文件写过程代码分析 162
10.7 本章小结 169

第 11 章 通用块层和 scsi 层 170

11.1 块设备队列 170
11.1.1 scsi 块设备队列处理函数 170
11.1.2 电梯算法和对象 171
11.2 硬盘 HBA 抽象层 172

11.3 I/O 的顺序控制 ······ 173
11.4 I/O 调度算法 ······ 173
11.4.1 noop 调度算法 ······ 173
11.4.2 deadline 调度算法 ······ 174
11.5 I/O 的处理过程 ······ 178
11.5.1 I/O 插入队列的过程分析 ··· 178
11.5.2 I/O 出队列的过程分析 ······ 186
11.5.3 I/O 返回路径 ······ 194
11.6 本章小结 ······ 203

第 12 章 内核回写机制 ······ 204

12.1 内核的触发条件 ······ 204
12.2 内核回写控制参数 ······ 204
12.3 定时器触发回写 ······ 205
12.3.1 启动定时器 ······ 205
12.3.2 执行回写操作 ······ 207
12.3.3 检查需要回写的页面 ······ 208
12.3.4 回写超级块内的 inode ······ 209
12.4 平衡写 ······ 213
12.4.1 检查直接回写的条件 ······ 214
12.4.2 回写系统脏页面的条件 ··· 215
12.4.3 检查计算机模式 ······ 216
12.5 本章小结 ······ 216

第 13 章 一个真实文件系统 ext2 ······ 217

13.1 ext2 的硬盘布局 ······ 217
13.2 ext2 文件系统目录树 ······ 218
13.3 ext2 文件内容管理 ······ 219
13.4 ext2 文件系统读写 ······ 219
13.5 本章小结 ······ 219

试读结束，需要全本PDF请购买 www.ertongbook.com

第 1 章
内核的基础层和应用层

前言中提到，内核分为内核基础层和内核应用层。这既有对整个操作系统软件架构的分析和理解，也有现实应用情况的支持。

操作系统对应用软件提供了统一的编程接口，操作系统的系统调用是稳定的、向下兼容的，但是在内核中，并不提供这种稳定且兼容的保证。实际上，同样的代码在不同的内核版本经常可能编译失败。内核的这种开发模式，造成了学习内核时版本众多而且不稳定的特点，也大大增加了学习的困难。

在长期对内核代码的分析和应用中，笔者注意到一个事实：内核中提供了大量的软件基础设施。这些基础设施既包括内核中对内存的使用，对进程调度的控制，也包括自旋锁、信号量等内核提供的同步函数，同时还包括内核提供的数据结构，比如链表、hash 链表、红黑树等。这些软件基础设施如同操作系统提供的系统调用一样，是理解内核代码和编写内核代码的基础。而这些软件基础设施在各个内核版本中基本是稳定的。

现实情况提供了另一方面的支持。学习的动力来自于应用，传统的操作系统教科书全面，但也很少有人能完全读懂并且结合代码进行实战应用。大多数程序员在工作中应用到内核的部分，绝大多数是设备驱动，而讲操作系统的书多数不会关注到设备驱动层面。除了设备驱动之外，内核中文件系统也有较多的应用。

要做到快速流畅地阅读内核代码，前提是了解内核中的软件基础设施。这些知识使用范围很广，分布在内核代码的各个部分，如果不了解，在内核代码的理解上就容易出现障碍。

1.1　内核基础层提供的服务

操作系统通常提供的服务是内存管理、进程管理、设备管理和文件系统。本书将内存管理、进程管理以及其他内核提供的基础软件设施，比如工作队列、tasklet 以及信号量和自旋锁都作为内核的基础层。本书并不分析和探讨这些基础层的原理和实现，本章只介绍如何使用这些基础软件设施。

1.1.1 内核中使用内存

简单说，内核提供了两个层次的内存分配接口。一个是从伙伴系统分配，另一个是从slab系统分配。伙伴系统是最底层的内存管理机制，提供页式的内存管理，而slab是伙伴系统之上的内存管理，提供基于对象的内存管理。

从伙伴系统分配内存的调用是alloc_pages，注意此时得到的是页面地址，如果要获得能使用的内存地址，还需要用page_address调用来获得内存地址。

如果要直接获得内存地址，需要使用__get_free_pages。__get_free_pages其实封装了alloc_pages和page_address两个函数。

alloc_pages申请的内存是以页为单元的，最少要一个页。如果只是申请一小块内存，一个页就浪费了，而且内核中很多应用也希望一种对象化的内存管理，希望内存管理能自动地构造和析构对象，这都很接近面向对象的思路了，这就是slab内存管理。

要从slab申请内存，需要创建一个slab对象，使用kmem_cache_create创建slab对象。kmem_cache_create可以提供对象的名字和大小、构造函数和析构函数等，然后通过kmem_cache_alloc和kmem_cache_free来申请和释放内存。

内核中常用的kmalloc其实也是slab提供的对象管理，只不过内核已经构建了一些固定大小的对象，用户通过kmalloc申请的时候，就使用了这些对象。

一个内核中创建slab对象的例子如代码清单1-1所示。

代码清单 1-1 创建 slab 对象

```
bh_cachep = kmem_cache_create("buffer_head",
                sizeof(struct buffer_head), 0,
                (SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD),
                init_buffer_head,
                NULL);
```

创建一个slab对象时指定了slab对象的大小，用以下代码申请一个slab对象：

```
struct buffer_head *ret = kmem_cache_alloc(bh_cachep, gfp_flags);
```

内核中还有一个内存分配调用：vmalloc。vmalloc的作用是把物理地址不连续的内存页面拼凑为逻辑地址连续的内存区间。

理解了以上几个函数调用之后，阅读内核代码的时候就可以清晰内核中对内存的使用方式。

1.1.2 内核中的任务调度

内核中经常需要进行进程的调度。首先看一个例子，如代码清单1-2所示。

代码清单 1-2 使用 wait 的任务调度

```
#define wait_event(wq, condition)
do {
    if (condition)
```

```
        break;
    __wait_event(wq, condition);
} while (0)

#define __wait_event(wq, condition)
do {
    DEFINE_WAIT(__wait);

    for (;;) {
        prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
        if (condition)
            break;
        schedule();
    }
    finish_wait(&wq, &__wait);
} while (0)
```

上文定义了一个 wait 结构，然后设置进程睡眠。如果有其他进程唤醒这个进程后，判断条件是否满足，如果满足，删除 wait 对象，否则进程继续睡眠。

这是一个很常见的例子，使用 wait_event 实现进程调度的实例在内核中很多，而且内核中还实现了一系列函数，简单介绍如下。

- ❑wait_event_timeout：和 wait_event 的区别是有时间限制，如果条件满足，进程恢复运行，或者时间到达，进程同样恢复运行。
- ❑wait_event_interruptible：和 wait_event 类似，不同之处是进程处于可中断的睡眠。而 wait_event 设置进程处于不可中断的睡眠。两者区别何在？可中断的睡眠进程可以接收到信号，而不可中断的睡眠进程不能接收信号。
- ❑wait_event_interruptible_timeout：和 wait_event_interruptible 相比，多个时间限制。在规定的时间到达后，进程恢复运行。
- ❑wait_event_interruptible_exclusive：和 wait_event_interruptible 区别是排他性的等待。

注意

何谓排他性的等待？有一些进程都在等待队列中，当唤醒的时候，内核是唤醒所有的进程。如果进程设置了排他性等待的标志，唤醒所有非排他性的进程和一个排他性进程。

1.1.3　软中断和 tasklet

Linux 内核把对应中断的软件执行代码分拆成两部分。一部分代码和硬件关系紧密，这部分代码必须关闭中断来执行，以免被后面触发的中断打断，影响代码的正确执行，这部分代码放在中断上下文中执行。另一部分代码和硬件关系不紧密，可以打开中断执行，这部分代码放在软中断上下文执行。

需要指出的是，这种划分是一种粗略、大概的划分。中断是计算机系统的宝贵资源，关闭中断意味着系统不响应中断，这常常是代价高昂的。所以为了避免关闭中断的不利影响，

即使在中断上下文中，也有很多代码的执行是打开中断的。而在软中断上下文，甚至进程上下文的内核代码中，有的时候也是需要关闭中断的。哪些地方需要关闭中断，而哪些地方又可以打开中断，需要仔细地考虑，既要尽可能打开中断以防止关闭中断的不利影响，又要在需要的时候关闭中断以避免出现错误。

Linux 内核定义了几个默认的软中断，网络设备有自己的发送和接收软中断，块设备也有自己的软中断。为了方便使用，内核还定义了一个 tasklet 软中断。tasklet 是一种特殊的软中断，同一时刻一个 tasklet 只能有一个 CPU 执行，不同的 tasklet 可以在不同的 CPU 上执行。这和软中断不同，软中断同一时刻可以在不同的 CPU 并行执行，因此软中断必须考虑重入的问题。

内核中很多地方使用了 tasklet。分析一个例子，代码如下所示：

```
DECLARE_TASKLET_DISABLED(hil_mlcs_tasklet, hil_mlcs_process, 0);
tasklet_schedule(&hil_mlcs_tasklet);
```

上面的例子首先定义了一个 tasklet，它的执行函数是 hil_mlcs_process。当程序中调用 tasklet_schedule，会把要执行的结构插入到一个 tasklet 链表，然后触发一个 tasklet 软中断。每个 CPU 都有自己的 tasklet 链表，内核会根据情况确定在何时执行 tasklet。

可以看到，tasklet 使用起来很简单，本节只需要了解在内核如何使用即可。

1.1.4 工作队列

工作队列和 tasklet 相似，都是一种延缓执行的机制。不同之处是工作队列有自己的进程上下文，所以工作队列可以睡眠，也可以被调度，而 tasklet 不可睡眠。代码清单 1-3 是工作队列的例子。

代码清单 1-3 工作队列

```
INIT_WORK(&ioc->sas_persist_task,
  mptsas_persist_clear_table,
  (void *)ioc);
schedule_work(&ioc->sas_persist_task);
```

使用工作队列很简单，schedule_work 把用户定义的 work_struct 加入系统的队列中，并唤醒系统线程去执行。那么是哪个系统线程执行用户的 work_struct 呢？实际上，内核初始化的时候，就要创建一个工作队列 keventd_wq，同时为这个工作队列创建内核线程（默认是为每个 CPU 创建一个内核线程）。

内核同时还提供了 create_workqueue 和 create_singlethread_workqueue 函数，这样用户可以创建自己的工作队列和执行线程，而不用内核提供的工作队列。看内核的例子：

```
kblockd_workqueue = create_workqueue("kblockd");
int kblockd_schedule_work(struct work_struct *work){
    return queue_work(kblockd_workqueue, work);
}
```

kblockd_workqueue 是内核通用块层提供的工作队列，需要由 kblockd_workqueue 执行的工作就要调用 kblockd_schedule_work，其实就是调用 queue_work 把 work 插入到

kblockd_workqueued 的任务链表。

create_singlethread_workqueue 和 create_workqueue 类似，不同之处是，像名字揭示的一样，create_singlethread_workqueue 只创建一个内核线程，而不是为每个 CPU 创建一个内核线程。

1.1.5 自旋锁

自旋锁用来在多处理器的环境下保护数据。如果内核发现数据未锁，就获取锁并运行；如果数据被锁，就一直旋转（其实是一直反复执行一条指令）。之所以说自旋锁用在多处理器环境，是因为在单处理器环境（非抢占式内核）下，自旋锁其实不起作用。在单处理器抢占式内核的情况下，自旋锁起到禁止抢占的作用。

因为被自旋锁锁着的进程一直旋转，而不是睡眠，所以自旋锁可以用在中断等禁止睡眠的场景。自旋锁的使用很简单，请参考下面的代码例子。

```
spin_lock(shost->host_lock);
shost->host_busy++;
spin_unlock(shost->host_lock);
```

1.1.6 内核信号量

内核信号量和自旋锁类似，作用也是保护数据。不同之处是，进程获取内核信号量的时候，如果不能获取，则进程进入睡眠状态。参考代码如下：

```
down(&dev->sem);
up(&dev->sem);
```

内核信号量和自旋锁很容易混淆，所以列出两者的不同之处。

- ❑内核信号量不能用在中断处理函数和 tasklet 等不可睡眠的场景。
- ❑深层次的原因是 Linux 内核以进程为单位调度，如果在中断上下文睡眠，中断将不能被正确处理。
- ❑可睡眠的场景既可使用内核信号量，也可使用自旋锁。自旋锁通常用在轻量级的锁场景。即锁的时间很短，马上就释放锁的场景。

1.1.7 原子变量

原子变量提供了一种原子的、不可中断的操作。如下所示：

```
atomic_t            mapped;
```

内核提供了一系列的原子变量操作函数，如下所示。

- ❑atomic_add：加一个整数到原子变量。
- ❑atomic_sub：从原子变量减一个整数。
- ❑atomic_set：设置原子变量的数值。
- ❑atomic_read：读原子变量的数值。

1.2 内核基础层的数据结构

内核使用的数据结构有双向链表、hash 链表和单向链表，另外，红黑树和基树（radix 树）也是内核使用的数据结构。实际上，这也是程序代码中通常使用的数据结构。

container 是 Linux 中很重要的一个概念，使用 container 能实现对象的封装。代码如下所示：

```
#define container_of(ptr, type, member) ({
    const typeof( ((type *)0)->member ) *__mptr = (ptr);
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

这个方法巧妙地实现了通过结构的一个成员找到整个结构的地址。内核中大量使用了这个方法。

1.2.1 双向链表

list 是双向链表的一个抽象，它定义在 /include/linux 目录下。首先看看 list 的结构定义：

```
struct list_head {
struct list_head *next, *prev;
};
```

list 库提供的 list_entry 使用了 container，通过 container 可以从 list 找到整个数据对象，这样 list 就成为了一种通用的数据结构：

```
#define list_entry(ptr, type, member)
 container_of(ptr, type, member)
```

内核定义了很多对 list 结构操作的内联函数和宏。

- LIST_HEAD：定义并初始化一个 list 链表。
- list_add_tail：加一个成员到链表尾。
- list_del：删除一个 list 成员。
- list_empty：检查链表是否为空。
- list_for_each：遍历链表。
- list_for_each_safe：遍历链表，和 list_for_each 的区别是可以删除遍历的成员。
- list_for_each_entry：遍历链表，通过 container 方法返回结构指针。

1.2.2 hash 链表

hash 链表和双向链表 list 很相似，它适用于 hash 表。看一下 hash 链表的头部定义：

```
struct hlist_head {
struct hlist_node *first;
};
```

和通常的 list 比较，hlist 只有一个指针，这样就节省了一个指针的内存。如果 hash 表非常庞大，每个 hash 表头节省一个指针，整个 hash 表节省的内存就很可观了。这就是内核中专门定义 hash list 的原因。

试读结束，需要全本PDF请购买 www.ertongbook.com