

独辟蹊径品内核：
Linux
内核源代码导读

李云华 编著



独辟蹊径品内核： Linux 内核源代码导读

较新的内核版本：本书使用的内核版本为2.6.24。

独特的写作手法：本书在讨论“How”的基础上，力求进一步探究“Why”。

“授人以渔”的写作宗旨：Linux内核处于飞速发展中，任何资料都无法覆盖内核的方方面面。本书以笔者学习过程的疑问和经验为基础，融会贯通于各个章节，毫无保留地就如何学习内核，如何分析内核进行了大量且大胆的探讨，从而力求体现本书的写作宗旨——“授人以渔”。



作者简介：

李云华，是一名内核技术的狂热爱好者，长期从事操作系统内核、计算机网络、设备驱动程序、以及嵌入式系统方面的开发和研究。拥有丰富的设备驱动开发、网络优化、内核及驱动移植、嵌入式系统构建等方面的开发经验。对Windows内核驱动及Linux内核驱动均有丰富的开发经验及心得体会。

上架建议：Linux



ISBN 978-7-121-08515-4



9 787121 085154 >

定价：65.00元



责任编辑：高洪霞
责任美编：李玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

独辟蹊径品内核：
Linux
内核源代码导读

李云华 编著

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

本书根据最新的 2.6.24 内核为基础。在讲述方式上，本书注重实例分析，尽量在讨论“如何做”的基础上，深入讨论为什么要这么做，从而实现本书的写作宗旨：“授人以渔”。在内容安排上，本书包含以下章节 x86 硬件基础；基础知识；Linux 内核 Makefile 分析；Linux 内核启动；内存管理；中断和异常处理；系统调用；信号机制在类 UNIX 系统中；时钟机制；进程管理；调度器；文件系统；常用内核分析方法。

本书适合初、中级 Linux 用户、从事内核相关开发的从业人员，也可以作为各类院校相关专业的教材及 Linux 培训班的教材，也可作为 Linux 内核学习的专业参考书。

读者可登录 www.broadview.com.cn，下载本书源代码。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

独辟蹊径品内核：Linux 内核源代码导读 / 李云华编著. —北京：电子工业出版社，2009.8

ISBN 978-7-121-08515-4

I. 独… II. 李… III. Linux 操作系统 IV. TP316.89

中国版本图书馆 CIP 数据核字（2009）第 038405 号

责任编辑：高洪霞

印 刷：北京东光印刷厂

装 订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：31 字数：804 千字

印 次：2009 年 8 月第 1 次印刷

印 数：4000 册 定价：65.00 元

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

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

服务热线：(010) 88258888。

前 言

几乎每一个操作系统内核的学习者在初学阶段都会感觉到难以入门。这是由于内核涉及到知识面非常广泛，需要学习者从根本上掌握大量的知识，这包括：程序编译，链接，装载的细节，操作系统理论，计算机系统体系结构，数据结构与算法，深厚的 C/汇编语言编程功底。如此相对较高的门槛常常令很大一部分初学者望而却步。那么是不是一定要先学好以上的各门知识后才能学习内核呢？事实上大部分学习者在学习以上各门知识都会遇到同样的问题，因为知识是一个网状结构。所以重要的不是先去学会什么知识，而是学会如何学习，学会在自己掌握的知识体系上提出问题，学会思考，进而坚持不懈的解决心中的疑问。笔者从学完 C/C++ 语言开始，由于 C/C++ 的示例程序都是在命令行下的，于是常常想如何才能编写出视窗程序，学习了 MFC，但是同样想不通诸如 WM_CHAR, WM_LBUTTONDOWN 的消息从何而来，带着 MFC 中诸多疑问，笔者开始学习 Windows SDK 程序开发，在这个学习过程中感觉对 MFC 的认识更加深入了，但同时又有新的问题想不通，于是进而学习 Windows DDK，之后开始学习操作系统内核。在这个过程中，笔者也遇到过数不尽的疑问，但是都是需要的时候再补充相关知识。因此初学者要明白，学习并不需要等到“万事具备”了才可以开始。需要的是保持好奇心，养成思考的习惯，树立解决问题的决心。很多读者渴望寻找好的入门教材，也常常有人问看什么书才能进步的快，但是当他们看了别人推荐的书却没有取得同样的收获，这是为什么呢？笔者认为，读书有以下几种境界：

1. 面对书上讲到的某个知识点，不能接合自己掌握的知识提出疑问¹，仅仅知识死记书本上的东西。这种状态就算学到最高境界，也仅仅只是能把书本上的知识点完好的记下来在脑海中形成孤立的知识节点。
2. 面对书本上讲到的某个知识点，能接合自己掌握的知识提出疑问，但是大多数时候没有探索精神，仅仅局限于到其他书籍或者请教别人来排除心中的疑问。脑海中的知识形成了简单网状结构，但由于探索能力长期得不到锻炼，综合自己的知识去分析和解决问题的能力十分有限。
3. 面对书上讲到的某个知识点，能接合自己掌握的知识提出疑问，并且能根据问题补充

¹比如：如何才能根据自己掌握的知识来解释或者推理出新看到的知识点，或者找到要解释推理出新知识还欠缺的知识点在哪方面。

相关必要的知识，不断综合分析各知识点的关系，提出各种假设和验证排除的方法并亲自验证，解决不了问题决不罢休。如能经过长期锻炼，其脑海中的知识点形成复杂的网状结构，综合分析能力必将加强。

4. 根据自己掌握的知识，提出全新的问题，并始终坚持找到答案为止。这种境界需要渊博的知识作为基础。

因此，不要还没学内核就被吓倒，说了这么多看似和内核无关的东西，就是要从先排除读者的心理担忧，树立正确的态度，重要的不是学会什么，而是学会学习。确定自己处于哪一种学习境界，然后通过学习某项具体的知识把自己提升到更高的境界。在现实生活中我们不难发现，能力强的学什么都又好又快。其根本原因在于他们处于更高的学习境界，并形成了良性循环！

有很多的人都渴望学习操作系统内核，但是内核涉及到的知识非常广泛，因此很多人半途而废，许多人往往抱怨没有好的书籍，教材。实际上，对于同一本书籍，不同的读者收获也是不同的，这取决于他们的态度和学习方法。笔者建议，在读书的时候，一定要以自己心中的疑问作为主线，而不要没有任何疑问就死记书本上的知识。

如何使用本书

笔者认为对于任何知识的学习，首先是以自我为中心，任何书籍资料都是用来解答读者心中的疑问的，因此在你阅读一本书时，首先要明确自己的疑问是什么？这可以是一个非常梗概的问题，例如：“Linux 内核是什么？”；也可以是一个非常细节的问题，例如：“按下键盘上的 A，到屏幕上显示出字符 A 的内部原理”。当你有了来自内心深处经过独立思考的疑问后，阅读对你来说是一种享受，一种乐趣。来自内心的疑问，经过不断的综合分析，缜密的推理，坚持不懈的查阅和求索，之后拨开迷雾见天日喜悦只有经过才能体会。虽然本书是一本很厚的书，但是这不是畏惧的理由，也不要因为它厚，就给自己下一个决心，制定一个阅读计划，几个月要读完本书。学习是主动探求的过程，而不是被动接受，在这个过程中，有太多的东西，不是谁可以计划出来的。例如：在笔者学习内核之初，看到大量的传言，读完《Understanding the Linux Kernel》，读完《Linux 内核情景分析》… 就可以成为“高手”了。于是笔者常常捧着厚厚的书，寻思着自己什么时候可以读完，然而有时好几天也前进不了几页，免不了感慨自己今生将与“高手”无缘，但是又心有不甘，于是囫囵吞枣的“快速”前进，但是越前进，就越感觉到艰难。“欲速则不达”这个道理人人都懂，但是在切身体会之前，人人都会犯这个错误。在经历了很长一段曲折和郁闷之后，笔者摆脱了“书”的束缚，完全以自己的疑问为中心，例如在读到中断处理时，由于知识不够全面，于是丢开内核的书籍，阅读了大量的计算机体系结构方面的资料，同样计算机体系结构的书籍也很厚，但是我也没有想过要把它们读完，这时只捡中断相关的读，之后再来读内核的书籍，发现自己原理懂了，但是具体到理解代码时，就迷糊了，于是有补充 GCC 内嵌汇编，C 代码编译到汇编代码的相关知识，反复试验等等。这个过程很慢，但是积累到最后，笔者发现自己读的非常快，甚至可以不读了，因为很多地方，只要读到前面的，就领悟了作者后面想要说什么了。

至今，我仍然没有完成当初为了成为“高手”而制定下的“宏伟”目标，因为我没有完整的读完《Understanding the Linux Kernel》、《Linux 内核情景分析》或《Linux 内核完全剖析》等等这类传说中“惊世骇俗”之作中的任何一本。但是笔者却从这些著作中受益匪浅。

现在，你应该知道要如何使用本书了吧？那就是不要拘泥如任何教条。虽然本书经笔者从初学到现在的心得体会以及相关笔记和资料整理而成，初学者的大量疑问都能在本书中找到答案。但是每个人都是独一无二的，笔者希望任何一个读者能综合利用本书和其它相关资料寻找你自己的答案。多问一点为什么，多一点假设，多一点思考，多一点推

理，多一点试验，多一点坚持。最后，你会感慨原来传说中的任何“秘籍”都是“浪得虚名”，因为读完它，你不一定能成为“高手”，而“高手”却不需要读完它。能否成为“高手”的决定性因素取决于你的学习方法和学习态度，而好的“秘籍”仅仅只是催化剂。

由于笔者水平有限，纰漏之处在所难免，因此希望读者能够指正。笔者的联系方式是：Addylee2004@163.com. 另外对本书有任何问题，意见或建议，勘误等等，欢迎前来<http://www.osplay.org>与笔者进行讨论。

目 录

第 1 章 x86 硬件基础	1
1.1 保护模式	1
1.1.1 分页机制	1
1.1.2 分段机制	7
1.2 系统门	13
1.3 x86 的寄存器	14
1.4 典型的 PC 系统结构简介	16
第 2 章 基础知识	18
2.1 AT&T 与 Intel 汇编语法比较	18
2.2 gcc 内嵌汇编	20
2.3 同步与互斥	25
2.3.1 原子操作	25
2.3.2 信号量	27
2.3.3 自旋锁	29
2.3.4 RCU 机制	35
2.3.5 percpu 变量	39
2.4 内存屏障	41
2.4.1 编译器引起的内存屏障	41
2.4.2 缓存引起的内存屏障	44
2.4.3 乱序执行引起的内存屏障	47
2.5 高级语言的函数调用规范	49
第 3 章 Linux 内核 Makefile 分析	52
3.1 Linux 内核编译概述	52
3.2 内核编译过程分析	54
3.3 内核链接脚本分析	62

第 4 章 Linux 内核启动	65
4.1 BIOS 启动阶段	65
4.2 实模式 setup 阶段	67
4.3 保护模式 startup_32	77
4.4 内核启动 start_kernel()	84
4.5 内核启动时的参数传递	90
4.5.1 内核参数处理	91
4.5.2 模块参数处理	95
第 5 章 内存管理	99
5.1 内存地址空间	99
5.1.1 物理内存地址空间	99
5.1.2 虚拟地址空间	101
5.2 内存管理的基本数据结构	104
5.2.1 物理内存页面描述符	104
5.2.2 内存管理区	106
5.2.3 非一致性内存管理	108
5.3 内存管理初始化	109
5.3.1 bootmem allocator 的初始化	109
5.3.2 页表初始化	115
5.3.3 内存管理结构的初始化	118
5.4 内存的分配与回收	127
5.4.1 伙伴算法	127
5.4.2 SLUB 分配器	138
第 6 章 中断与异常处理	152
6.1 中断的分类	152
6.2 中断的初始化	156
6.2.1 异常初始化	156
6.2.2 中断的初始化	160
6.2.3 中断请求服务队列的初始化	167
6.3 中断与异常处理	171
6.3.1 特权转换与堆栈变化	171
6.3.2 中断处理	172
6.3.3 异常处理	177

6.4 软件中断与延迟函数	180
6.4.1 softirq	180
6.4.2 tasklet	185
6.5 中断与异常返回	187
6.6 中断优先级回顾	191
6.7 关于高级可编程中断控制器	192
6.7.1 APIC 初始化	193
第 7 章 信号机制	199
7.1 信号机制的管理结构	200
7.2 信号发送	204
7.3 信号处理	210
第 8 章 系统调用	220
8.1 Libc 和系统调用	220
第 9 章 时钟机制	226
9.1 clocksource 对象	227
9.1.1 clocksource 概述	227
9.1.2 clocksource 初始化	228
9.2 tickless 机制	232
9.2.1 tickless 由来	232
9.2.2 clock event device 对象概述	234
9.2.3 clock event device 对象的初始化	236
9.3 High-Resolution Timers	247
9.3.1 High-Resolution Timers 管理结构	247
9.3.2 High-Resolution Timers 初始化	252
9.3.3 High-Resolution Timers 操作	258
9.4 时钟中断处理	268
9.4.1 时钟维护	276
9.4.2 进程时间信息统计	281
9.5 软件定时器	283
9.5.1 基本管理结构	283
9.5.2 初始化	284
9.5.3 注册与过期处理	287

第 10 章 进程管理	295
10.1 进程描述符	296
10.1.1 进程状态	297
10.1.2 进程标识	299
10.1.3 进程的亲缘关系	300
10.1.4 进程的内核态堆栈	301
10.1.5 进程的虚拟内存布局	302
10.1.6 进程的文件信息	305
10.2 进程的建立	306
10.2.1 建立子进程的 task_struct 对象	308
10.2.2 子进程的内存区域	315
10.2.3 子进程的内核态堆栈	323
10.2.4 0 号进程的建立	325
10.3 进程切换	327
10.4 进程的退出	331
10.4.1 do_exit 函数	331
10.4.2 task_struct 结构的删除	334
10.4.3 通知父进程	335
10.5 do_wait()函数	338
10.6 程序的加载	344
第 11 章 调度器	351
11.1 早期的调度器	351
11.2 CFS 调度器的虚拟时钟	353
11.3 CFS 调度器的基本管理结构	357
11.4 CFS 调度器对象	359
11.5 CFS 调度操作	360
11.5.1 update_curr()函数	360
11.5.2 scheduler_tick()函数	362
11.5.3 put_prev_task_fair()函数	364
11.5.4 pick_next_task()函数	366
11.5.5 等待和唤醒操作	368
11.5.6 nice 系统调用	373
第 12 章 文件系统	376
12.1 Ext2 的磁盘结构	376
12.2 Ext2 的内存结构	385

12.3 虚拟文件系统的管理结构	387
12.3.1 文件系统对象	388
12.3.2 VFS 的超级块	389
12.3.3 VFS 的 inode 结构	400
12.3.4 VFS 的文件对象	406
12.3.5 VFS 的目录对象	409
12.3.6 VFS 在进程中的文件结构	412
12.4 文件系统的挂载	413
12.5 路径定位	425
12.6 文件打开与关闭	441
12.7 文件读写	449
12.7.1 缓冲区管理	449
12.7.2 文件读写操作分析	456
第 13 章 常用内核分析方法	471
13.1 准确定位同名宏及结构体	471
13.2 准确定位同名函数	473
13.3 利用 link map 文件定位全局变量	474
13.4 准确定位函数调用线索	476
13.5 SystemTap 在代码分析中的使用	479

第1章 x86 硬件基础

如果你是一个 Linux 内核初学者，你一定常常遇到：保护模式，分段机制，分页机制，段地址，线性地址，中断门，调用门，局部描述符，全局描述符，等等这样的名词。这些概念常常把初学者弄得“云里来，雾里去”。你会常常感慨，Intel 为什么要设计这么复杂的概念呢？仅仅是一个地址机制，就常常让初学者打开书本，发现自己会算了，可是一旦关上书本，换个例子，又迷惑了。

事实上这些机制的背后都有它的缘由，任何一个复杂的设计都是由一个简单的设计发展起来的，当简单的设计满足不了实际需要时，就会一步一步地革新，直到问题被圆满地解决。因此理解一个“复杂”的东西的最好方式不是去记住它，而是要从最简单的地方入手，一步一步地推敲，简单的设计在现实中会遇到什么问题？又该如何解决这个问题？再联系到你现在要理解的复杂的例子，慢慢地建立起一条完整的线索，这样知识不会出现断层，你也不需要一次跨越一个鸿沟，一切都水到渠成。例如：在学习保护模式中的地址机制时，你一定会感觉为什么要这么复杂呢？实模式不是很简单吗？既然实模式简单，那么不妨想想，如果使用实模式会遇到什么问题呢？把这些问题都一一列出来，再结合现有机制，你就会很自然的理解为什么需要这么做了。本章将试图让读者看到这些概念背后的“为什么”。

1.1 保护模式

1.1.1 分页机制

内存按字节编址，每个地址对应一个字节的存储单元，早期的程序直接使用物理地址。在单任务操作系统时代，物理内存被划分为两部分，一部分地址空间由操作系统使用，另外一部分由应用程序使用。到了多任务时代，由于程序中的全局变量，起始加载地址是在链接期决定的¹。如果直接使用物理地址，则很可能有多个起始地址一致的应用程序需要同时被加载运行，这就需要把冲突的程序加载到另外的地址上去，然后重新修正程序中的所有相关的全局符号的地址。然而在早期的计算机系统上内存容量十分有限，即便是通过重定

¹不能理解这句话的读者需要补充程序编译、链接以及加载方面的知识，可以通过学习 Windows 平台下的 PE 文件格式，或者 Linux 下的 ELF 文件格式，来补充这部分知识。《Windows 环境下 32 位汇编语言程序设计》以及《Linkers & Loaders》是两本不错的教材。

位解决了加载地址冲突的问题，由于内存大小的限制，能够同时加载运行的程序仍然十分有限。而在多任务系统上，某些进程在部分时间内处于等待状态，于是人们很自然地想到，当内存不够的时候把处于等待状态的进程换入磁盘，腾出一些内存空间来加载新程序。这又带来了新的问题：每次腾出来的空间地址可不是固定不变的，这就意味着把磁盘上的内容加载进来的时候，又要重新修正程序中的相关地址，在每次换入换出的过程中要不断地修正相关程序的地址。而且还有一个更为严重的问题：假设进程 A 出现一个错误，对某个物理地址进行了写入操作，恰好这个地址又属于进程 B，当进程 B 被调度运行的时候，必然会出现错误。很难想象一个软件产品的 BUG 却导致用户抱怨另外一个软件产品。于是虚拟内存技术发展起来。在虚拟内存中，程序代码中访问的不再是物理地址，而是虚拟地址。

以 32 位系统为例，每一个进程有 4GB 的虚拟地址空间，每个进程中有一个表，它记录着每个虚拟地址对应的物理地址是多少，这样当程序加载的时候，可以先分配好物理内存，然后把物理内存的地址填入这个表里面，这样进程之间互不影响。假设程序 A 和 B 都是要求在地址 BASE 处加载(程序中使用的都是虚拟地址。)，由于每个进程都有 4GB 的私有虚拟地址空间，因此两个进程没有加载冲突。操作系统分配的物理地址分别是 A1 和 B1，然后 A1 和 B1 起始的物理内存地址分别被填入两个进程虚拟地址映射表中，从而建立虚拟地址和物理地址的一一映射关系。当进程 A 访问虚拟地址 BASE+X 的时候，由于 MMU 的硬件支持，硬件自动查找进程 A 的地址映射表，从而访问到物理地址为 A1+X 的内存单元。同理，当进程 B 访问虚拟地址为 BASE+X 的时候，MMU 自动查找进程 B 的地址映射表，从而访问到 B1+X 的内存单元。当然，实际上的虚拟地址机制比这个复杂得多，但是在对它有了总体认识之后，再来学习一个实际的例子就要简单得多了。接下来就以 32 位的 X86 系统为例，进一步介绍虚拟内存机制。

每个进程拥有 4GB 的虚拟地址空间，每个字节的虚拟地址可以通过地址映射表映射到一个字节的物理地址上面去。因此这个映射表本身必然要占据很大的内存空间，如何设计映射表成为问题的关键。如果在虚拟地址映射表中为每一个字节建立映射关系，那么映射 4GB 的虚拟地址需要 $2^{30} \times 4B$ (32 位系统地址为 4Byte)的内存。可见简单的一一填表映射是不能满足现实要求的。为了要减少虚拟地址映射表项占用的内存空间，所有操作系统都采用了页式管理。把物理内存划分为 4KB, 8KB 或者 16KB 大小的页，这样每个页面在虚拟地址映射表中仅仅占用 4Byte 的内存。以 4KB 的页大小为例，4GB 的虚拟地址空间有 2^{20} 个页面，那么映射 4GB 空间的映射表仅仅需要 $2^{20} \times 4B$ (32 位系统地址为 4Byte)的内存。

其映射原理如图1.1所示：程序要访问的地址是 0x12345A10，CPU 中的 MMU 首先找到这个进程的虚拟地址映射表，其起始物理地址为 0x10000000。在 4KB 页大小的情况下，4GB 虚拟地址空间含有 2^{20} 个页面，只需要 20 位就可以表示 2^{20} 的大小了，所以虚拟地址的高 20 位 0x12345 作为虚地址映射表中的索引，在 32 位系统上虚地址映射表中的每一项是 4 个字节，所以 MMU 根据地址 $0x10000000 + 0x12345 * 4$ 取得虚拟地址 0x12345A10 对应的物理页面起始地址为 0x54321000，该地址的低 12 位总是为 0，这是由于每一个 4KB 大小的物理页面总是在 4KB 的边界上对齐的。而虚地址 0x12345A10 中的低 12 位被用做

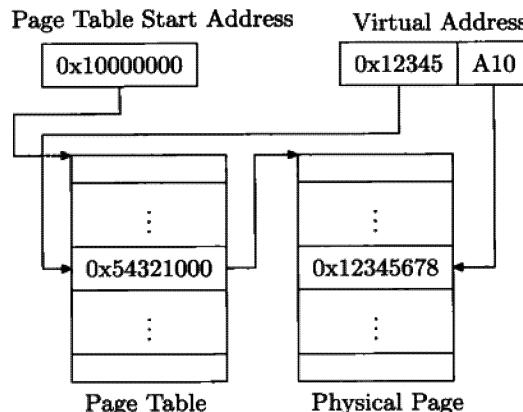


图 1.1 虚拟地址映射

页内偏移量，最终虚地址 `0x12345A10` 对应的物理地址为 `0x54321000+A10`，而 CPU 访问到的内容是 `0x12345678`。

由于页大小为 4KB，虚地址表中的表项低 12 位总是为 0，因此可以把低 12 位用来做标识位。例如把第 0 位用做存在位，当第 0 位为 1 时表示该页面在物理内存中，反之表示该页面不在物理内存中。假设一个进程要占用 10MB 的内存空间，在进程初始化的时候，虚地址映射表初始化为 0，在内存不足的情况下，系统只分配了 5MB 的内存，这 5MB 内存的物理地址被填入到映射表中，同时表项中最低位被设置成 1，当进程访问到另外 5MB 的虚地址的时候，MMU 在查表时发现最低位为 0，于是触发一个缺页中断，这个时候，系统缺页中断处理例程再分配内存页面，同时更新相应的映射表项，之后程序就可以正常运行了。

同理，还可以把一位划分出来作为读写位，如果对一个地址进行写入操作，MMU 在查表的时候会根据其读写位判断是否允许写入。几乎每一个程序员都知道访问 `NULL` 指针时，一定会出错，那么操作系统是如何捕捉到这个错误的呢？地址 `0(NULL)` 实实在在地对应了内存中的一个物理内存地址，如何根据一个地址来判断指针是不是合法的呢？各个操作系统都保证在一个进程中虚拟地址从 0 开始的某一段区域是不映射的，其页表项为 0。例如 Windows 中把 0~64K 的地址区域划分到 `NULL` 指针区而不被映射。因此访问这部分地址的时候必然会触发缺页中断，这个时候操作系统就可以判断出这个地址是否落在 `NULL` 指针区内。否则无论像 `malloc` 这一类的函数返回的指针是 0 还是其他的值，都无法判断分配成功或是失败。

另外 Windows, Linux 等操作系统都有一个文件映射技术，它可以把一个大小远超出系统物理内存的文件映射到进程的虚地址空间 X 起始处，之后程序访问通过数组的方式，`X[0], X[1]...X[n]` 来访问文件的第 0 到 n 个字节。由于物理内存小于文件大小，所以内核只读入一部分的文件内容，并建立映射，当访问到没有被映射的部分 `X[m]` 的时候就会触发缺

页中断，这个时候中断处理例程会再分配一定的物理内存页面，然后把它映射到 $X[m]$ 上，同时从磁盘读入文件对应的内容到该处。这个过程对程序来说是透明的，程序根本不用关心系统物理内存到底有多大。还可以通过把同一物理页面映射到不同的进程的虚拟地址空间中去来实现内存共享，无论各个进程映射的虚拟地址是相同还是不同的，访问到的都是同一个物理页面。在操作系统中有一个重要的 Copy-On-Write 机制，多个进程共享同一片内存，这片内存的读写位被设置成 0，当某个进程对其写入的时候，触发中断，在中断处理程序中，把要写入的相关页面复制一份，之后该进程单独使用这些内存，而进程间仍然共享其他的内存。通过这些例子读者可以看到虚拟内存、虚拟地址、进程的虚地址空间及 MMU 之间的区别和联系。

虚地址到物理的转换过程是由硬件自动完成的。由于虚地址映射表是进程私有的，因此各个进程的虚地址映射表被放在不同的物理内存中，而且每个进程都必须把这个表的起始物理地址告诉 MMU，这就是 Page Table Start Address 的作用，很容易想到这个起始地址必须是物理地址。前面说过虚地址映射表大小为 4MB，而虚地址的高 20 位是这个表中的索引，这意味着这 4MB 空间必须作为进程的必备资源在启动的时候一次分配，而且这 4MB 的内存必须在物理地址上是连续的。这可不是一个好消息，想想虚拟内存的设计理念就是要在一个小内存系统上运行尽可能多的程序，而这个限制将导致一个配备 $64M^2$ 内存的系统也运行不了几个进程。考虑到一个进程不需要同时访问到 4GB 的内存，因此映射表也可以被分散开来，像物理页面那样在需要的时候再分配映射。于是两级，三级甚至四级³页表的概念被提出来。

图1.2是32位X86系统的两级页表结构。地址映射表分为两级，第一级被称为页目录表，第二级为页表，每个进程的页目录表起始物理地址由 CPU 中的寄存器 CR3 指定，而虚拟地址的最高 10 位将作为页目录的索引， $CR3 + (\text{页目录索引} \times 4)$ 就可以得到 PDE⁴。PDE 的高 20 位指定了一个页表的起始地址，低 12 位是一些标致位，同时虚拟地址的中间 10 位被用做页表的索引，从而得到 PTE⁵。PTE 的高 20 位指定了一个 4KB 页面的起始地址，而虚拟地址的最低 12 位被用做页内偏移量，从而访问到虚拟地址指定的内存单元。最高 10 位用做页目录的索引，每一项 4 个字节，所以页目录大小正好为 4KB，1024 项，每一项指向一个页表，每个页表又有 1024 项指向对应的 4KB 页，总共可以映射的内存就是 $1024 \times 1024 \times 4KB$ ，也就是 4GB。这样的话，一个进程的页表可以被分散开来，在页目录索引时如果发现页表没有被映射，就可以通过缺页中断来分配并建立新的映射。

两级页表虽然能解决进程一次要连续分配 4MB 页表的问题，但是每次访问内存都需要两次查表才得到物理地址最后访问到指定的内存，这样一来降低了系统内存的访问速度，为此 CPU 内部设置了最近存取的页面的缓存，被称为 TLB，程序给出虚拟地址后 CPU 先到 TLB 中查找，如果 TLB 没有命中再访问两级页表，从而极大地提高了访问速度。

²在早期配备 64MB 内存已经算是大内存系统了。

³三级，四级页表一般被应用在 64 位系统上。

⁴即 Page Directory Entry，页目录项。

⁵即 Page Table Entry，页表项。

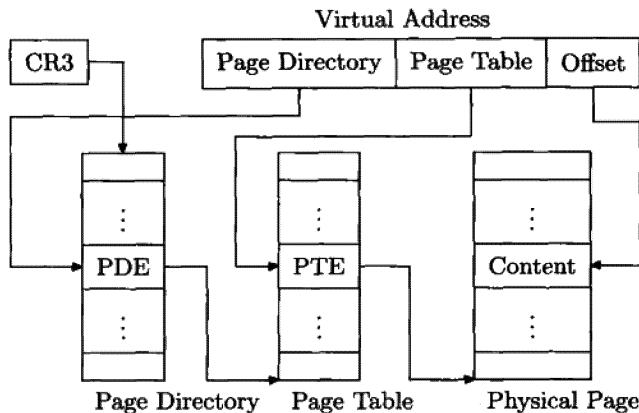


图 1.2 32 位 X86 系统两级页表结构

通过前面的讨论可以看到，每一个进程都有独立的页表，进程总是通过查本进程页表获取物理地址的，因此无法直接修改其他进程的内存。这样一来，进程就被保护起来而不受其他进程的影响了。这是保护模式的一个重要特征，但是这样的保护还是不够的。我们知道几乎每一个进程都要通过操作系统提供的接口执行一些操作系统内核提供的代码。例如进程通过 `read` 系统调用读取文件内容，而具体读取文件的代码则是操作系统内核提供的，同时内核的这些代码也需要使用一些数据，这部分代码和数据必须映射到每一个进程的地址空间中，但是如果任何一个进程有意或无意地修改了这部分代码或数据的话，那么其后果是严重的。

一种由硬件支持的进程权限级别就是用来解决这个问题的，多数构架的 CPU 提供 4 个级别，0 代表最高级别，3 则是最低级别。然而多数操作系统的设计只使用了其中两个级别，内核运行在级别 0 上，而应用程序运行在级别 3 上，人们通常把它们称做 ring0 和 ring3，当 ring3 的代码试图访问 ring0 的代码或者数据的时候，硬件自动进行权限检查，只有通过检查的才能访问成功。于是进程有两个执行环境，应用层和内核层，应用层程序在 ring3 执行程序自己的代码，而只能通过特殊的途径⁶进入 ring0 执行内核代码。有了这个保护，ring3 的代码无法直接修改 ring0 的代码和数据，而进程进入 ring0 时执行的那些代码都是内核提供的，内核保证这部分代码不会有错误，也不会恶意修改数据或代码，从而保证了系统的健壮性。除此之外，CPU 的指令也被分类，某些特权指令只能在 ring0 的级别上执行。这样保证某些重要的资源不能被应用程序随意修改，例如中断向量表的起始地址。分页保护和权限保护机制被称为保护模式，而之前的 8086 不具备这两种机制，被称为实模式。

图1.3是 X86 页表，页目录中的一些常用标志位，操作系统通过设置这些标志位来控制硬件实现前面介绍的保护模式的功能。其中页表项和页目录项大致相同。更加详细的信息请参看《Intel Architecture Software Developer's Manual Volume 3: System Programming》。

⁶如系统调用。

31	12 11	9 8	7	6	5	4	3	2	1	0
Base Address	Avail	G	PS	D	A	PCD	PWT	U/S	R/W	P

图 1.3 页表项结构

- 第 0 位是 Present 位，在页目录项中，该位为 0 的话说明指定的页表不在物理内存中，页表项中该位为 0 的话，说明对应的物理页面不在物理内存中，当系统访问到 Present 位为 0 的页表或页目录项的时候将触发缺页异常，异常处理例程会根据需要分配物理内存，把磁盘上的相关内容调入内存并建立好页表/页目录项，然后返回继续执行。
- 第 1 位是 Read/Write 位，为 0 表示对应的页面只读。第 2 位是 User/Supervisor 位，为 0 表示特权级页面，否则表示普通权限。U/S 为和 R/W 位组合对页面进行保护，其

表 1.1 R/W 与 U/S 的组合保护

U/S	R/W	级别 0	级别 3
0	0	读/写	不允许访问
0	1	读/写	不允许访问
1	1	读/写	读
1	1	读/写	读/写

规则如表1.1所示。其中第一行表示当一个页面的 U/S, R/W 位都为 0 的时候，运行在 ring0 级别的进程可以对该页进行读写访问，而运行在 ring3 的进程既不能对该页进行读取访问，也不能进行写入访问。

- 第 3 位是 Write-through 位，1 表示写直达，当对这个页面写入时，要立即写到内存中，为 0 的时候，写到缓存中，在必要的时候再一次性刷缓存。
- 第 4 位是 Page-level cache disable 位，当该位为 1 的时候，表示对应的页禁用缓存，对页目录项而言表示对应的页表不缓存。
- 第 5 位是 Accessed 位，当一个页面/页表被初始载入内存的时候，该位初始化为 0，之后当对该页面进行访问(读/写)后，该位被置 1。
- 第 6 位是 Dirty 位，当一个内存页面被初始载入内存的时候，该位为 0，之后软件对该页进行写入操作的时候该位被置为 1。这是一个很重要的标志，比如在物理内存不够的情况下，系统把一部分内存写到磁盘上，之后在需要的时候再读出来，之后再次写入磁盘的时候，可以根据页面的 Dirty 标志来判定哪个页面要写入，而没被修改的页面就不需要再次写入磁盘了。
- 第 7 位是 Page Size 位，在页目录中，该位为 0 表示一个物理页面大小是 4K，为 1 的时候表示一个物理页面是 4M。4M 页面的时候，没必要使用两级页表，所以高 10 位

表示页目录的索引，而低 22 位表示 4M 中的偏移。在页表中的该位没被使用，总是为 0。

- 第 8 位是 Global 位，当进程切换的时候要改变页目录地址 CR3 寄存器，于是 CPU 内部的 TLB 内容将被丢弃，而对于一些各个进程都要共享访问的页面，可以通过在页表中设置该位来阻止对应页面的 TLB 被丢弃。页目录中的该位被忽略。
- 第 9~11 位是保留给系统软件自由使用的。
- 高 20 位是页表/页面的起始地址。

1.1.2 分段机制

仅仅是分页机制就能够满足虚拟内存管理和保护模式的要求了，现在某些构架的处理器完全不使用分段机制，如 MIPS。段机制实际上是 x86 的一个历史遗留问题，Linux 内核也是尽量不使用段机制⁷。Intel 在 16 位处理器时代，由于 16 位地址线只支持 64KB 的内存寻址，但是 64KB 的内存显得太少了。于是提出了分段机制，地址由段基址和偏移组成，用 BASE:OFFSET 表示，而物理地址由 $\text{BASE} \ll 4 + \text{OFFSET}$ 形成。BASE 和 OFFSET 都是 16 位，OFFSET 是段内偏移，因此一个段最多有 64KB，而 1MB 内存最多有 16 个段，如果使用两个 16 位的地址分别作为高 16 位和低 16 位地址的话，一共是 32 位，可以最大寻址 4GB 内存，但是从当时的软硬件条件来看，没有必要这么做。

随着操作系统的发展，提出了保护模式和虚拟内存管理，Intel 在分段机制的基础上实现保护模式和虚拟内存管理。后来更为优越的分页机制逐步占据了主流，但是 Intel 采用分页机制的同时，出于兼容的目的而保留了分段机制。但是 16 位的段寄存器已经不能作为 32 位系统的段基址了，于是段寄存器被用做选择子，而段基址以及段的一些其他属性被存放在一片内存中，被称为段描述符表。为什么不直接把段寄存器也扩充为 32 位或者更长的呢？因为 x86 允许不使用分页机制的同时，也能实现保护模式和虚拟内存管理的需要。因此段的基本信息除了段基址外，还需要类似分页机制中页表项中的一些标志位，于是出现了段描述符表。表中的每一项占 8 字节，它定义了一个段的基本属性。描述符格式如图1.4所示。

- Segment Base 0-31 位表示一个段的起始地址。
- Segment Limit 0-19 位表示一个段的最大长度。
- Byte5 的最低位 A 为 0，表示本段还从未被访问过，为 1 表示本段已经被访问过。
- Byte5 中的 ED 位表示段的增长方向，ED=0 时，段地址增长方向是向上的，也就是说从 Base 开始到 Base+Limit 这段区间是属于本段的。ED=1 时，表示段的地址增长方

⁷建议读者不要花费太多的时间和精力在臃肿的 Intel 段机制上面

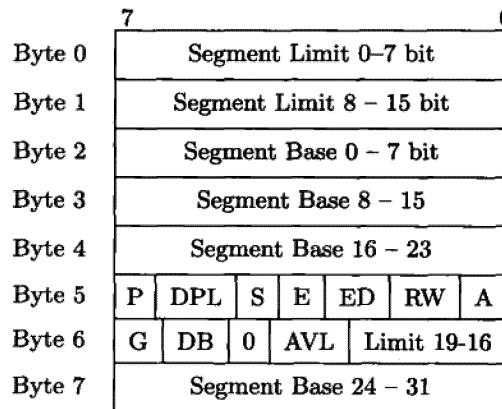


图 1.4 段描述符

向是向下的，也就是说本段的地址区间从 Base 开始到 Base-Limit 结束。通常堆栈段是向下增长的。

- Byte5 中的 E 位表示可执行位，当 E=1 时，表示代码段(可执行)，E=0 表示数据段。
- Byte5 中 RW 为读写位，在数据段中(E=0)，RW=0 表示该段不能被写入，RW=1 表示本段可以被写入。在代码段中(E=1)，表示代码段，这时 ED=0 表示忽略特权级别，ED=1 时表示遵守特权级别，RW=0，段不可读，RW=1，可读。
- Byte5 的第 4 位 S=1，表示代码段或者数据段，堆栈段也是数据段。S=0，表示系统段，例如中断门，调用门等(见第1.2节)。
- Byte5 的 DPL 占两位(Descriptor Privilege Level)，00~11 分别表示访问本段所需要的特权级别，只有级别大于等于 DPL 中指定的权限才能访问本段。0 是最大权限。
- Byte5 的 P 位为 0 表示本段不在内存中，当段中的内存被换入磁盘时可以设置为 0，当访问这样的段时将触发中断。
- Byte6 的第 4 位保留给系统软件使用。
- Byte6 的第 5 位是保留位总是为 0。
- Byte6 的第 6 位，DB=0，表示默认地址和操作数是 16 位，DB=1，表示默认地址和操作数是 32 位。
- Byte6 的第 7 位 G=0，表示 20 位的段界限的单位是字节，这样一个段的最大长度是 1M，G=1 表示 20 位段界限的单位是 4K，这样段的最大长度是 4G。

可以看到，上面许多属性的作用和页表项中重复，实际上在 x86 上抛开分页机制，也能够实现保护模式的各种功能。系统中每一个段都由一个段描述符来表示，这些描述符依次存放在描述符表中，系统中描述符表分全局描述符表和局部描述符表：

- 全局描述符表 GDT，CPU 内部寄存器 GDTR 的高 32 位指向该表的起始地址，低 16 位表示全局描述符表的界限，界限以字节为单位。低 16 位可以表示 65536 个字节，而每一个段描述符表项有 8 个字节，因此全局描述符表最多有 8192 项。汇编指令 LGDT 把一个 48 位的内存操作数加载到 GDTR 寄存器中，而 SGDT 把 GDTR 保存到一个 48 位的内存操作数中。
- 局部描述符表 LDT，由 LDTR 寄存器指定。由于 Intel 的设计本意是每个进程有自己的局部描述符表，因此描述符表的位置不是固定的，它需要随着进程的切换而切换，可能是考虑到进程 A 切换到进程 B 的时候，A 把自己的 LDTR 保存在自己的进程地址空间中，之后切换到进程 B，再切换回 A 的时候，由于保护模式下的地址空间隔离，进程 B 不能恢复 A 的 LDTR，所以进程的 LDTR 必须保存在全局共享的内存中。于是 Intel 把每一个局部描述符表的首地址放到全局描述符表中，这样通过 16 位的 LDTR 寄存器在 GDT 的索引得到一个段描述符表项，该描述符的基地址部分指定了一个局部描述符表的起始地址，在这个局部描述符表中又有许多局部描述符表项。其访问过程如图 1.5 所示。16 位的寄存器 LDTR 是一个 GDT 中的索引，实际的 LDT 由 GDT 中的一个 64 位的表项来描述。这样一来每次访问 LDT 表都要两次查表，为了节省开销，LDTR 寄存器包括软件可见的 16 位和软件不可见的 64 位，LLDT 指令的 16 位操作数作为索引，在 GDT 中找到 64 位的一个表项，然后把它加载到 LDTR 不可见的部分。

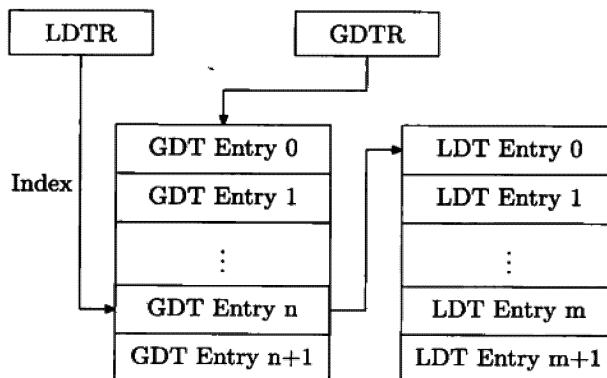


图 1.5 局部描述符表访问示意图

在 32 位 CPU 上，段寄存器 CS, DS, ES, SS, FS, GS⁸仍然是 16 位的，被用做全局

⁸其中 FS 和 GS 是后来增加的。

段描述符表和局部段描述符表的索引，被称为段选择子，其格式如图1.6所示。

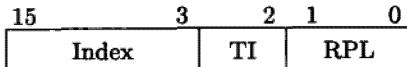


图 1.6 段选择子

其中第 0 位和第 1 位表示 Request Privilege Level，0-3 分别代表 3 个不同的请求级别，最高 13 位是索引位，可以表示 8192 项，当 TI=1 时，表示从局部描述符表中选择相应的描述符，TI=0 时，从全局描述符表中选择相应的描述符。这样当 CPU 通过 DS:Offset 访问内存的过程是：

- (1) 如果 TI=0，则根据 GDTR 寄存器指定的首地址找到全局描述符表，再利用 DS 的高 13 位作为索引找到对应的描述符表项，在通过相应的权限检查之后，从表项中取出 32 位的段地址，最后再加上 Offset 从而形成线性地址，如果分页机制开启，该地址还要经过分页机制处理，最后才得到物理地址。
- (2) 如果 TI=1，情况类似，先根据 GDTR 寄存器指定的首地址找到全局描述符表，再利用 LDTR 的 16 位软件可见部分作为索引，找到一个表项，然后从该表项中取出 32 位的基址，再根据这个基址找到局部描述符表，然后把 DS 的高 13 位作为索引在该表中找到一个表项，取出基址再加上 Offset 最后得到线性地址。实际上在执行 LLDT 指令把 16 位的操作数加载到 LDTR 软件可见部分的同时，就从这个全局描述符表中取出了对应的 64 位表项加载到 LDTR 软件不可见部分的缓存中，所以访问的时候就可以根据不可见部分直接找到局部描述符表了，这里是为了强调 LDT 和 GDT 的关系。

这里段描述符号中的 DPL(见图1.4)和段选择子中的 RPL 都是用来做权限检查的，段寄存器 CS 和 SS 中的 RPL 是两个特殊的权限级别，它们代表进程的当前权限级别，又被称为 CPL(Current Privilege Level)。进程在执行指令时，目标操作数中的段选择子中的 RPL 代表请求级别。例如，设当前进程的 CS 和 SS 中的 RPL 位为 0，它表示该进程的当前级别(CPL)为 Ring0，当执行 MOV DS:Offset, 0 这条指令时，操作数 DS 中的 RPL 位表示想要获取对该段的访问权限级别，这里假设为 1，而 DS 选择子选中的描述符中的那个 DPL(见图1.4)则代表访问该段所需要的权限，假设为 3。这样权限检查的依据就是，一个权限为 CPL 的进程，试图以 RPL 权限去访问一个要求 DPL 权限的段，在这个例子中，当前权限(CPL)为 0，试图以权限(RPL)为 1 的身份，去访问一个最小权限要求(DPL)为 3，这个检查是可以通过的。

除此之外，系统中还包括一个特殊的段---任务状态段 TSS，任务状态是在硬件级别支持多进程管理而设置的。从 CPU 的角度来看，进程上下文包括一组 CPU 的寄存器的值，当进程切换的时候，CPU 会把当前进程的寄存器保存到一片内存中，然后再把新进程的寄存器值从内存中加载到 CPU 中，这片内存被称为 TSS，其格式如图1.7所示。

31	15	1 0
I/O Map Base Address	0	T 100
0	LDT Segment Selector	
0	GS	
0	FS	
0	DS	
0	SS	
0	CS	
0	ES	
	EDI	
	ESI	
	EBP	
	ESP	
	EBX	
	EDX	
	ECX	
	EAX	
	EFLAGS	
	EIP	
	CR3	
0	SS2	
	ESP2	
0	SS1	
	ESP1	
0	SS0	
	ESP0	4
0	Previous Task Link	0

图 1.7 任务状态段

每一进程都有一个 TSS 段来保存 CPU 上下文，每一个这样的 TSS 段也占用 GDT 中的一个描述表项。描述符的格式如图1.8所示。硬件通过 TSS 提供进程切换机制，TR 寄存器⁹指向当前进程的 TSS。当通过段间跳转指令 JMP Seg:Offset 或者 CALL Seg:Offset 的时候，如果段寄存器的高 13 位选择子选中的描述符项是一个 TSS 段的时候，硬件先根据 TR

⁹TR 寄存器和 LDTR 类似，软件可见部分是全局描述符表中的一个索引。

寄存器找到当前进程的 TSS，把当前进程上下文保存到 TSS 中，然后根据 JMP 或者 CALL 指令中的段寄存器的选择子找到目标进程的 TSS，再把新的 TSS 加载到 CPU 中，这样就完成了进程切换。在图1.8任务段描述符中，Byte5 中的 S 位为 1 的时候，表示这是一个系统段描述符，而 Byte5 的 0~3 位被称为 Type 位，表示系统段的类型。10B1 表示这是一个 TSS 段描述符，其中 B 位为 1 表示忙。通过为当前正在执行的任务设置忙标志可以避免任务嵌套切换(从任务 A 切换到任务 A 本身。)。

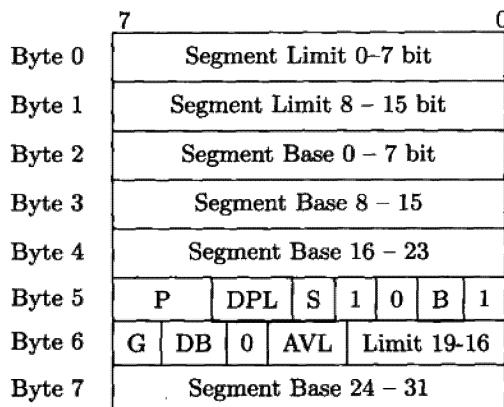


图 1.8 任务状态段描述符

其中 Type 位所表示的其他类型如表1.2所示，其中任务门、调用门、中断门和陷阱门请参见1.2节。

表 1.2 系统段的类型

类型号	定义	类型号	定义
0	未定义(Intel 保留)	8	未定义
1	有效的 286TSS	9	有效的 386TSS
2	LDT	A	386TSS 忙
3	286TSS 忙	B	未定义
4	286 调用门	C	386 调用门
5	任务门	D	未定义
6	286 中断门	E	386 中断门
7	286 陷阱门	F	386 陷阱门

在图1.7中，除了当前 SS:ESP 外还可以看到 SS0:ESP0, SS1: ESP1 和 SS2: ESP2 这三组堆栈指针，这是为什么呢？前面说过 CPU 有 4 个级别，分别是 ring0~ring3，为了避免相互影响，进程在不同级别使用的是独立的堆栈，假设进程从 ring3 切换到 ring0，CPU 从 TSS 中取出 SS0:ESP0，同时把 ring3 的 SS:ESP 保存在新的 ring0 的堆栈中，当从 ring0 返回

到 ring3 的时候，CPU 根据 ring0 堆栈的值恢复 ring3 的 SS:ESP。只有从低级别向高级别切换的时候，才从 TSS 中取出高级别的堆栈指针，而从高级别向低级别切换的时，是从高级别的堆栈中恢复低级别的堆栈指针，所以 TSS 中有 SS0:ESP0-SS2:ESP2 这 3 对堆栈指针。

1.2 系统门

前面说到描述符表项除了有段描述符之外，还有系统段描述符，当段描述符中的 S 位为 0 的时候表示这个一个系统段描述符。系统段描述符号包括中断门、陷阱门、任务门、以及调用门。中断门和陷阱门用来描述一个中断例程，他们的入口地址依次存放在中断描述符表中。48 位的 IDTR 寄存器的高 32 位指向这个表，低 16 位表示该表的界限，lidtr 指令用来加载一个 48 位的操作数到 IDTR 寄存器。中断门和陷阱门如图1.9所示。其中 16 位

	7	0	
Byte 0	Offset 0-7 bit		
Byte 1	Offset 8 - 15 bit		
Byte 2	Selector 0 - 7 bit		
Byte 3	Selector 8 - 15 bit		
Byte 4	0	0	Reserved
Byte 5	P	DPL	S Type
Byte 6	Offset 16 - 23 bit		
Byte 7	Offset 24 - 31 bit		

图 1.9 中断门 & 陷阱门描述符

的选择子和 32 位的 Offset 偏移地址指定了一个中断处理例程的地址。Type 指示了门的类型，其组合见表1.2。陷阱门和中断门的格式类似，外部中断的处理例程的描述符项被称为中断门，而异常处理例程的描述符则被称为陷阱门。严格来说，中断是由外部设备向 CPU 发出的，CPU 经过地址译码，取指令，指令译码，指令执行等流水步骤，在最后会检查 CPU 中断请求线是否有信号，所以中断是一个异步事件，CPU 在执行指令期间，外部设备随时可以发出中断请求。而异常是一个同步事件，比如除 0 错误，是 CPU 自己在指令期触发的。另外还可以通过执行 CPU 指令 int x 触发软件异常。当发生中断/异常的时候，CPU 利用中断/异常号到中断向量表中索引得到门描述符，然后调用描述符指定的处理例程。

调用门和中断门类似，只是其中 Reserve 字段被用来表示 Param Count，表示调用门的参数的个数(以 4 字节为单位)。调用门可以用来实现低级别的权限向高级别权限切换，也可以用来实现系统调用。前面说过，不同权限级别使用不同的堆栈，而系统调用的时候，如果需要传递参数的话，就需要把当前低级别要传递的参数拷贝到高级别的堆栈中取，而 Param Count 指定了拷贝参数的数量。当执行 CALL Seg:Offset 或者 JMP Seg:Offset 等转移

指令的时候，如果段选择子 Seg 选中的是一个调用门，CPU 会转向调用门指定的入口地址 +Offset 的地方执行，Offset 一般为 0。调用门在实际中没有被使用。Linux 中的系统调用是通过 int 0x80 来实现系统调用的，这是一个陷阱门。

任务门是用来实现任务切换的，注意在图1.8中描述的任务段描述符是段描述的一种，可以通过它来实现任务切换，而这里的任务门是门描述符的一种，不要把两者搞混淆了。

	7	0		
Byte-0	Reserved			
Byte-1	Reserved			
Byte-2	TSS Segment Selector 0 - 7 bit			
Byte-3	TSS Segment Selector 8 - 15 bit			
Byte-4	Reserved			
Byte-5	P	DPL	0	Type
Byte-6	Reserved			
Byte-7	Reserved			

图 1.10 任务门描述符

任务门的格式如图1.10所示。其中 16 位 TSS Selector 是 TSS 段选择子，当 CALL, JMP 或者 IRET 等转移指令执行的时候，如果转移目标 Seg:Offset 指向的是一个任务门，CPU 将从任务门中取出 16 位的选择子，然后利用这个选择子的高 13 位作为全局描述符表的索引，找到对应的任务段描述符，然后从中取出任务状态段 TSS 的地址，最后切换到该 TSS 指定的任务。任务门在实际中也没有被使用。

1.3 x86 的寄存器

x86 包括 32 位的通用寄存器 EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI; 16 位的段寄存器 CS, DS, SS, ES, FS, GS; 32 位的标志寄存器 EFLAGS; 32 位的指令指针寄存器 EIP; 32 位的控制寄存器 CR0, CR1, CR2, CR3; 48 位的全局描述符表寄存器 GTDR, 中断描述符表寄存器 IDTR; 16 位的局部描述符表寄存器 LDTR, 任务状态描述符表寄存器 TR; 32 位的调试寄存器 DR0~DR7。其中大部分寄存器在前面已经介绍过了。

31	4	3	2	1	0
PG		ET	TS	EM	MP

图 1.11 CR0 寄存器

图1.11是CR0寄存器。

- 第 0 位是保护模式允许位(Protection Enable), 当 PE 为 1 的时候 CPU 进入保护模式, PE 为 0 时 CPU 工作在实模式。
- 第 1 位是协处理器监控位(Monitor Coprocessor), MP=1 表示有协处理器, MP=0 表示没有协处理器。
- 第 2 位位是模拟协处理器位(Emulate Coprocessor), 当没有协处理器时, CPU 如果遇到浮点指令, 产生协处理器无效中断, 中断处理例程将用软件方式模拟协处理器。
- 第 3 位是任务转换位(Task Switched), 当任务转换完成后该位被置 1。
- 第 4 位是处理器扩展类型控制位(Processor Extension Type), 为 1 表示协处理器是 32 位的 80387, 0 表示协处理器是 16 位的 80287。
- 最高位是分页允许位(Paging Enable), 置 1 表示开启分页机制, 0 表示关闭分页机制。

CR1 保留没有被使用, CR2 是页异常线性地址寄存器。前面说过当 CPU 访问一个内存地址的时候, 如果该地址没有对应的物理页面被映射到页表中, 那么会触发缺页异常, 同时 CPU 会把这个内存地址放到 CR2 中, 这样异常处理程序就可以根据 CR2 寄存器的值进一步处理该异常。在调试寄存器中 DR0~DR3 这四个是内存断点寄存器, DR4,DR5 保留, DR6 是断点状态寄存器, DR7 是断点控制标志寄存器。CPU 每次执行内存相关的指令时会检测 DR7 中的标志, 然后把相关指令的地址和 DR0~DR3 做比较, 如果相等, 则触发调试异常, DR7 中规定了对相关地址的监控要求, 比如是读还是写, 是一个字节, 两个字节或者 4 个字节。设 DR0=X, 当 DR7 中规定读监控两个字节时, CPU 读取 X+1 的地址也会触发异常。CR3 寄存器的高 20 位用来保存页目录基址, 不同进程拥有不同的地址空间就是通过切换 CR3 来实现的, 关于页目录请参阅 1.1.1 节。

标志寄存器 EFLAGS 如图 1.12 所示: 其中大部分标志位和 16 位的 8086 一样, 这里的第 12, 13 位 IOPL 是特权指令许可位, 当执行 IN, OUT, CLI 等 IO 指令的时候, CPU 会比较指令段寄存器 CS 中的 CPL¹⁰, 只有当 CPL 大于等于 IOPL(数字上小于)时, IO 指令才能正确执行。第 17 位 WM 位是虚拟 8086 位, 如果该位被设置, CPU 工作在虚拟 8086 模式下。

31	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	0	CF	

图 1.12 EFLAGS 标志寄存器

¹⁰CS 中的 RPL 位代表当前进程的权限级别 CPL。

1.4 典型的PC系统结构简介

图1.13是一个典型的PC系统的简要框图，其中Intel 945和Intel 82801分别代表传统的北桥和南桥。也就是人们常说的主板芯片组。

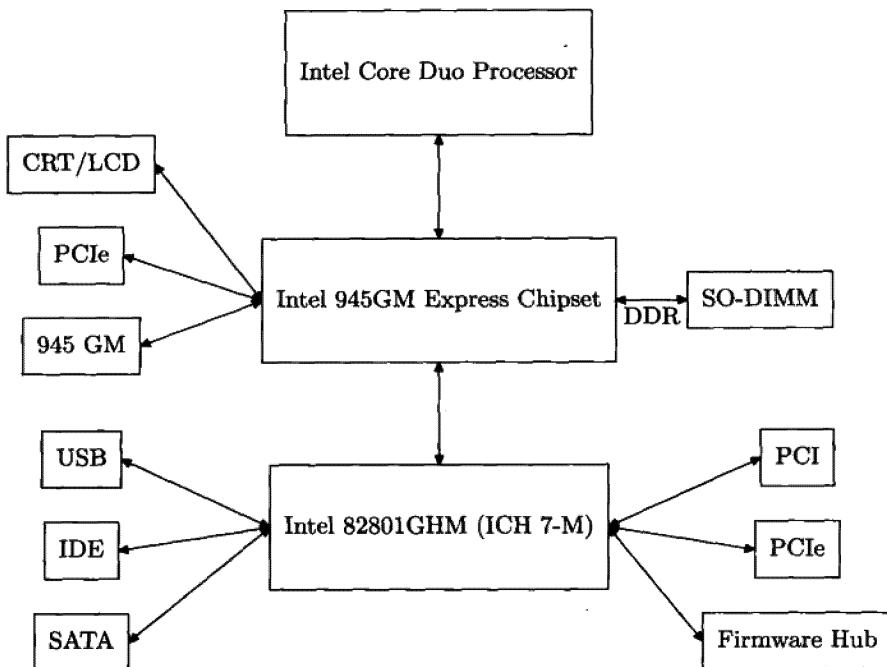


图 1.13 典型 PC 系统简要框图示例

图中的945GM主要集成显卡和内存控制器，被称为Graphics and Memory Controller Hub(GMCH)，82801GHM就是通常说的南桥，即I/O Controller Hub(ICH)。北桥主要控制相对高速的设备，而南桥控制速度相对较低的IO设备。南/北桥上面有大量的配置寄存器，例如北桥上面的某个寄存器可以配置内存的起始地址和结束地址，由于外部设备也有板载存储空间，因此当CPU从地址总线发出一个寻址请求时，北桥会根据配置寄存器确定这个地址是落在内存的存储空间中，还是在南桥上面的外部设备的存储空间中，从而实现正确的寻址。理解这一点非常重要，因为这是以后理解外部设备的地址配置的基础，因此在这里特别强调这一点。例如：把地址X写入到某个PCI设备的基址寄存器后，当CPU发出对X的寻址请求时，北桥芯片经过比较，判断出这个地址落在是PCI地址空间，于是寻址请求转发到南桥，同样南桥再转发到PCI总线，PCI总线上的各个PCI设备芯片都会把总线上的地址和基址寄存器中的值进行比较，只有相匹配的设备才会做出应答。当然对于某些新的CPU，它的内存控制器是集成在CPU中的，那么CPU内部也有同样的判断。通常所说的32位地址总线的系统上支持的最大物理内存为4GB，其实这4GB指的是内存

地址空间，所以系统中支持的内存肯定不能是 4GB，因为一部分地址空间需要留给外部设备的板载内存。而 4GB 的内存限制对许多服务器来说是不够的，所以 Intel 在奔腾 3 之后把地址总线扩充到 36 位，被称为 PAE。

在南桥中集成了大量的 I/O 设备相关的控制器，其中包括许多资料中介绍的 8259A 可编程中断控制器，当然也包括了 Advanced Programmable Interrupt Controller(APIC)。

下面通过 `lspci` 可以看到 Intel 945 和 Intel 82801 的系统中的信息。

```
1 00:00.0 Host bridge: Intel Corporation Mobile
2   945GM/PM/GMS, 943/940GML and 945GT Express
3   Memory Controller Hub (rev 03)
4 00:02.0 VGA compatible controller: Intel Corporation Mobile
5   945GM/GMS, 943/940GML Express Integrated Graphics
6   Controller (rev 03)
7 00:02.1 Display controller: Intel Corporation Mobile
8   945GM/GMS/GME, 943/940GML Express Integrated
9   Graphics Controller (rev 03)
10 00:1b.0 Audio device: Intel Corporation 82801G (ICH7 Family)
11   High Definition Audio Controller (rev 02)
12 00:1d.0 USB Controller: Intel Corporation 82801G (ICH7 Family)
13   USB UHCI Controller #1 (rev 02)
14 00:1d.1 USB Controller: Intel Corporation 82801G (ICH7 Family)
15   USB UHCI Controller #2 (rev 02)
16 00:1d.2 USB Controller: Intel Corporation 82801G (ICH7 Family)
17   USB UHCI Controller #3 (rev 02)
18 00:1d.3 USB Controller: Intel Corporation 82801G (ICH7 Family)
19   USB UHCI Controller #4 (rev 02)
20 00:1d.7 USB Controller: Intel Corporation 82801G (ICH7 Family)
21   USB2 EHCI Controller (rev 02)
22 00:1e.0 PCI bridge: Intel Corporation 82801
23   Mobile PCI Bridge (rev e2)
24 00:1f.0 ISA bridge: Intel Corporation 82801GBM (ICH7-M)
25   LPC Interface Bridge (rev 02)
26 00:1f.2 IDE interface: Intel Corporation 82801GBM/GHM
27   (ICH7 Family) SATA IDE Controller (rev 02)
28 00:1f.3 SMBus: Intel Corporation 82801G (ICH7 Family)
29   SMBus Controller (rev 02)
30   .....
```

第2章 基础知识

操作系统内核是一门综合性很强的学科，正因为如此，通过学习内核，可以极大地提高计算机综合技能。然而，如果没有掌握一定的基础知识，常常会感觉“四处碰壁”。本章对软件方面常见的基础知识进行总结，从而为进一步的内核学习打下基础。

2.1 AT&T 与 Intel 汇编语法比较

任何一个操作系统的源代码中总是少不了汇编语言，因此汇编语言是学习操作系统必备的基础知识，然而目前国内的教学大多是在 Windows 平台上进行的，因此大多数读者熟悉的是 Intel 的汇编语法，但是在 Linux 内核代码中使用的却是 AT&T 的汇编。本节对这两种汇编进行对比，让已经熟悉 Intel 汇编语法的读者快速掌握 AT&T 的汇编。

1. 前缀

在 Intel 汇编语法中，寄存器和立即数都没有前缀，但是在 AT&T 的汇编语法中，寄存器的前缀为“%”，而立即数的前缀为“\$”。两种格式的区别如下例所示：

代码片段 2.1 Intel 与 AT&T 汇编格式比较

```
# EAX <= 8
MOV EAX, 8 (Intel)
movl $8, %eax (AT&T)

# EAX <= EBX
MOV EAX, EBX (Intel)
movl %EBX, %EAX (AT&T)
```

2. 操作数方向

Intel 汇编和 AT&T 的操作数方向相反，Intel 汇编中的第一个操作数为目的操作数，而第二个操作数为源操作数，而 AT&T 汇编语法中，第一个数为源操作数，第二个数为目的操作数。从上面的例子可以看出它们之间的区别。

3. 操作数位宽

Intel 汇编中，由特定的字符指定操作数的位宽，例如"BYTE PTR", "WORD PTR", "DWORD PTR"来表示。在 AT&T 汇编中，由操作码最后一个字符来指定操作数的位宽，b,w,l 分别代表 8 位，16 位，32 位。下面的例子说明了它们的区别：

代码片段 2.2 Intel 与 AT&T 汇编格式比较

```
MOV AL, BYTE PTR BAR  (Intel)
movb %al, BAR (AT&T)
```

4. 间接寻址方式

Intel 和 AT&T 的间接寻址格式如下所示：

代码片段 2.3 Intel 与 AT&T 汇编格式比较

-
- 1 SEGREG:[BASE+INDEX*SCALE+DISP] (Intel)
 - 2 segreg:disp(base, index, scale) (AT&T)
-

上面的例子是间接寻址的通用形式，为了方便理解，我们举一个例子对这个通用形式进行解释，假设有下面这样的一个结构体数组：

```
struct test {
    int a;
    int b;
};

struct test bar[10];
```

假设现在要访问数组第 6 项中的成员 b，使用默认的段寄存器，那么它对应的汇编代码如下：

代码片段 2.4 Intel 与 AT&T 汇编格式比较

```
# BASE 表示数组基地址，对应本例中的 bar。
# INDEX 为数组索引，对应本例中的第 6 项，就是 51。
# SCALE 是结构体的大小，本例中大小为 8 个字节。
# DISP 是结构体内的偏移量，本例中 b 的偏移量为 4。
MOV EAX, DWORD PTR [bar+5*8+4] (Intel)
movl 4(bar,5,8), %eax (AT&T)
```

从这里可以看出，Intel 语法相对直观，通用形式中的 SCALE,INDEX 等都可以为空，例如以下是常见的形式。

¹注意：数组下标从 0 开始。

代码片段 2.5 Intel 与 AT&T 汇编格式比较

```
MOV EAX, DWORD PTR [ebp+20h] (Intel)
movl 0x20(%ebp), %eax, (AT&T)
```

2.2 gcc 内嵌汇编

Linux 内核源代码中，许多 C 代码中嵌入了汇编语句，这是通过关键字 `asm` 来实现的。它的形式如下所示：

代码片段 2.6 gcc 内嵌汇编示例

```
1 static inline unsigned long native_read_cr2(void)
2 {
3     unsigned long val;
4     asm volatile("movl %%cr2,%0\n\t" : "=r" (val));
5     return val;
6 }
```

其中 `asm` 表示汇编指令的开始，由于 `gcc` 在编译优化阶段，可能会调整指令的顺序，关键字 `volatile` 阻止 `gcc` 对这里的内嵌汇编指令进行优化。另外在内核代码中常常还看到 `_asm_`, `_volatile_`，它们的作用和 `asm`, `volatile` 是一样的，为什么要两个作用相同的关键字呢？这是由于在 C 标准制定之初，并没有 `asm` 和 `volatile` 这两个关键字，后来由于实际需要，加入了这两个关键字时，考虑到已有的 C 代码中，可能存在变量名为 `asm` 或 `volatile` 的情况，为了兼容这种情况，所以又加入了 `_asm_` 和 `_volatile_` 这两个关键字。

`gcc` 内嵌汇编的通用形式如下所示：

代码片段 2.7 Intel 与 AT&T 汇编格式比较

```
1 asm volatile (assembler template : output : input: clobber);
```

其中 `assembler template` 为汇编指令部分，例如上例中的“`movl %%cr2, %0`”。`output` 是输出部分，`input` 表示输入部分，`clobber` 表示被修改的部分。汇编指令中的数字和前缀表示样板操作数，例如 `%0`, `%1` 等，用来依次指代后面的输出部分，输入部分等样板操作数。由于这些样板操作数使用了%，因此寄存器前面要加两个%，例如上例中的 `%%cr2`。`output` 和 `input` 分别是输出部分和输入部分，`clobber` 是损坏部分。

为什么要这样呢？这是由于汇编指令中常常需要使用寄存器，但是寄存器是由 `gcc` 在编译时分配的，由于访问寄存器要比访问内存快，因此当 `gcc` 对一个代码块进行优化时，通常把空闲的寄存器分配给被频繁访问的变量。假设有以下代码块：

代码片段 2.8 C 代码示例

```
1 int f()
```

```

2 {
3     ....
4     int i;
5     int total = 0;
6     for (i = 0; i < 100; i++) {
7         total += i;
8     }
9     ...
10    return total;
11 }

```

在上面的代码块中，`i` 和 `total` 都是被频繁访问的变量，因此 `gcc` 可能会为它们分配两个空闲的寄存器，例如把 `EAX` 和 `ECX` 分别分配给 `i` 和 `total`，这样在整个 `for` 循环只需要对寄存器进行操作。如果在 `for` 循环之前所有的寄存器已经分配出去了，此时就没有空闲的寄存器，`gcc` 可能需要插入一条指令把某个寄存器的值写回到对应的内存变量中(和前面的 `EAX` 对应于堆栈中的一个内存变量 `i` 是一样的道理)，然后再把这个寄存器分配给变量 `i`。需要注意的是，这里讨论的是有这种可能。因此程序员在任何时候都不知道哪个寄存器是空闲的。

我们再来看一个例子，汇编指令 `bts n, ADDR`，可以把地址为 `ADDR` 的内存单元的第 `n` 位设置为 1。其中 `ADDR` 是一个内存变量，而 `n` 必须是立即数或寄存器。如果要在 C 代码中之间嵌入这条汇编指令，而 `n` 是通过计算得到的结果(`n` 不是立即数)，这时就需要腾出一个寄存器来。而程序员不能预测到 `gcc` 在编译时对寄存器的分配情况，所以 `gcc` 的内嵌汇编提供一个模板，用来指导 `gcc` 该如何处理。下面是在 C 代码中内嵌 `btsl` 指令的示例。

代码片段 2.9 gcc 内嵌汇编代码示例

```

1 static inline void set_bit(int nr, void *addr)
2 {
3     asm("btsl %1,%0" : "+m" (*(u32 *)addr) : "Ir" (nr));
4 }

```

在这个例子中输出部分为 `addr`，输入部分为 `nr`，没有损坏部分。修饰符"`+m`"限定指针 `addr` 指向一个可读写的内存单元，"`Ir`"指定 `nr` 必须是一个立即数，或者是一个寄存器变量。当某段代码调用 `set_bit()` 函数时，如果参数 `nr` 是一个立即数或者是一个寄存器，那么就满足这个限制条件。但是如果调用 `set_bit()` 函数时传递的参数 `nr` 在内存中，这时候 `gcc` 必须分配一个寄存器，并且插入一条指令把内存中的值加载到这个寄存器中。如果没有空闲的寄存器，那么 `gcc` 就要插入一条指令把某个寄存器保存到对应的内存中，腾出这个寄存器来保存参数 `nr` 的值，最后再插入一条指令恢复这个寄存器的值。

在指令部分中的`%0` 表示 `addr` 对应的操作数，`%1` 表示 `nr` 对应的操作数。现在假设某段代码调用 `set_bit()` 函数时传递的参数 `nr` 内存变量，`gcc` 在代码生成阶段，生成类似下面这

样的代码²:

代码片段 2.10 汇编代码

```

1 # 腾出 EAX 寄存器。
2 pushl %eax;
3 movl nr, %eax;
4
5 # 假设 (ebp-8) 就是指针 addr 指向的内存单元地址。
6 bts %eax, -8 (ebp);
7
8 # 恢复 EAX 寄存器。
9 popl %eax

```

通过这个例子就可以理解 gcc 内嵌汇编复杂的原因了，在理解这个例子后，再来看 gcc 的内嵌汇编，也就简单得多了。在 gcc 内嵌汇编中，常用的限制符如表2.1所示。

现在来看一个相对复杂的例子，

代码片段 2.11 gcc 内嵌汇编代码示例

```

1 #define mov_blk(src, dest, numwords)
2 __asm__ __volatile__ (
3     "cld\n\t"
4     "rep\n\t"
5     "movsl"
6     :
7     : "S" (src), "D" (dest), "c" (numwords)
8     : "%ecx", "%esi", "%edi"
9 )

```

我们知道，串操作指令 `movs` 可以把 ESI 寄存器指向的内存块复制到 EDI 指向的内存块中，复制的字节数由 ECX 寄存器指定。所以执行 `rep movsl` 之前需要初始化 ESI,EDI 和 ECX 寄存器，但是在上面这段代码的汇编指令部分中，并没有看到设置这几个寄存器的汇编指令。这是由 gcc 自动完成的。

在输入参数部分 "S"(src) 指定参数 src 必须保存到 ESI 寄存器中，"D"(dest) 指定参数 dest 必须保存到 EDI 中，"c"(numwords) 指定参数 numwords 必须保存到 ECX 中。在 gcc 的代码生成阶段，在调用 `mov_blk()` 这个宏的代码块中，如果参数 src,dest,numwords 都没有分配到对应的寄存器中，那么 gcc 在展开汇编指令部分最前面插入指令，保存相应的寄存器，并且把参数 src,dest,numwords 复制到对应的寄存器中。如果在此之前正好某个参数已经被 gcc 分配到对应的寄存器中，那么就不需要插入这样的汇编指令。

在损坏部分，"%ecx","%esi","%edi" 说明在汇编指令部分会改变这几个寄存器的值，这样如果之前这几个寄存器被分配给其他的变量，在代码前面 gcc 会插入汇编指令保存这几个

²注意这个例子的目的是为了说明原理，实际试验时不一定能够得到一模一样的结果。

寄存器，在代码的最后，gcc 又会插入代码恢复这几个寄存器的值。

表 2.1 gcc 内嵌汇编限制符

限制符	说明
a	对应的变量必须在 EAX 寄存器中
b	EBX
c	ECX
d	EDX
s	ESI
D	EDI
q	EAX, EBX, ECX, EDX 中的任何一个
r	EAX, EBX, ECX, EDX, ESI, EDI 中的任何一个
A	EAX:EDX 组合成一个 64 位的操作数
m	操作数必须是内存中的变量
o	操作是内存变量，并且对操作数的寻址方式为基址加一个偏移量
v	操作数是内存变量，但是寻址方式为基址，没有偏移量
g	操作数可以是内存变量，立即数，EAX,EBX,ECX 或者 EDX
I	操作数是 0~31 的立即数(用于 32 位的移位操作)
J	操作数是 0~63 的立即数(用于 64 位的移位操作)
K	操作数必须是 0xFF
L	操作数必须是 0xFFFF
M	操作数是 0,1,2 或 3
N	操作数可以是 0-255 中的任何一个数(用于 in/out 指令)
f	操作数是浮点寄存器
t	第一个浮点寄存器
u	第二个浮点寄存器
=	操作数是只写的(用于输出)
+	操作数是可读写的(用于输入输出)
&	表示在汇编代码前，对应的操作数会被输入部分修改
memory	用在损坏部分中，表示内存被修改了

现在来看一个更复杂的例子：

代码片段 2.12 gcc 内嵌汇编代码示例

```
1 static __always_inline
2 void * __memcpy(void * to,
3                  const void * from,
4                  size_t n)
5 {
6     int d0, d1, d2;
```

```

7   __asm__ __volatile__(  

8     "rep ; movsl\n\t"  

9     "movl %4,%ecx\n\t"  

10    "andl $3,%ecx\n\t"  

11    "jz 1f\n\t"  

12    "rep ; movsb\n\t"  

13    "1:"  

14    : "&c" (d0), "&D" (d1), "&S" (d2)  

15    : "0" (n/4), "g" (n), "1" ((long) to), "2" ((long) from)  

16    : "memory");  

17    return (to);  

18 }

```

这个例子和上个例子一样，也是要先设置好 ESI,EDI 和 ECX，在输出部分，%0 和 d0 结合，要求使用 ECX(c)寄存器，%1 和 d1 结合，要求使用 EDI(D)寄存器，%2 和 d2 结合，要求使用 ESI(S)寄存器。在输入部分，“0”(n/4)，限制符“0”表示(n/4)和%0 使用同一个寄存器，也就是输出部分指定的 ECX。同理，to 和%1 使用同一个寄存器，from 和%2 使用同一个寄存器。“g”(n)和%4 结合，其中“g”表示 n 可以放在通用寄存器或者内存中。

gcc 在代码的最前面会插入指令，把(n/4)保存到 ECX 中，输出部分限制符前面的 = 号表示这个操作数是可写入的，用于输出，也就是在代码的最后，要把 ECX 的值保存到 d0 中，& 号表示这个操作数会被输入部分修改。同理把 to,from 分别保存到 EDI,ESI 寄存器中之后，rep movsl 指令每次从 ESI 指向的内存块复制 4 个字节到 EDI 指向的内存块，直到 ECX 为 0 时结束(n/4 次)。最后如果字节数不是 4 的整数倍，就还需要移动 $n \bmod 4$ 个字节。

这段代码中同样改变了 ECX,EDI 和 ESI 寄存器，可是为什么在损坏部没有列出这 3 个寄存器呢？它不会影响其他的代码块吗？这是因为在这个函数中，定义了 3 个局部变量 d0,d1,d2，并且要求 ECX,EDI,ESI 分别和 d0,d1,d2 结合，这样在调用这个函数的代码块中，如果这 3 个寄存器不是空闲的话，gcc 会插入汇编指令，保存这 3 个寄存器，把它们腾出来分配给 d0,d1,d2，同理在代码的最后，gcc 会插入指令恢复它们。从这里可以看出内嵌汇编的灵活之处了，当然这是建立在深刻理解的基础上的。

那么损坏部分的“memory”又表示什么呢？假设在调用 __memcpy() 的代码块中，gcc 把 EAX 寄存器分配给某个内存变量 i，如果 i 位于 to 指向的内存块中，在调用 __memcpy() 之后的代码块中，由于这片内存已经被改变了，所以要通知 gcc，内存被修改了，这样 gcc 会插入指令，刷新寄存器分配关系，例如在 __memcpy() 之后，重新把内存变量 i 加载到 EAX 寄存器中。

2.3 同步与互斥

在多任务操作系统中，多个进程按照不可预测的顺序运行，因为多个进程之间常常存在相互制约或者相互依赖的关系。这些关系可以被划分为同步和互斥关系。

1. 同步

同步是指多个进程之间的依赖关系，例如两个进程，其中一个为生产者，另外一个为消费者，消费者的输入来自生产者的输出，那么消费者必须等待生产者把数据准备好之后，才能进行处理。

2. 互斥

互斥是指多个进程的制约关系，在多任务操作系统中，某些资源在同一时刻只允许一个进程使用，这一类资源被称为临界资源。当多个进程要求使用临界资源时，它必须等待，直到临界资源空闲。例如当两个进程都要求使用一个打印机时，其中一个必须等到另外的进程使用完成，否则打印出来的结果将会交错在一起。

2.3.1 原子操作

假设使用一个全局变量 `printer` 表示可用的打印机数量，这个变量被初始设置为 1，进程按照下面的方式申请使用打印机。

代码片段 2.13 示例代码

```
1 int request_printer()
2 {
3     while (printer == 0)
4         wait();
5     printer--;
6     .....
7 }
```

当 `printer` 为 0 时，说明别的进程正在使用打印机，于是当前进程进入等待状态，当进程被唤醒时，如果测试到 `printer` 不为 0，就跳出 `while` 循环，然后执行 `printer--`。但是这段代码中有一个隐蔽的同步问题，这个问题需要从 `printer--` 的汇编代码才能看出，在 x86 平台上，`printer--` 可能会被编译成下面的汇编指令。

代码片段 2.14 示例代码

```
1 # printer--
2 movl  printer %eax
3 decl  %eax
4 movl  %eax  printer
```

由于 CPU 在每条指令流水阶段的最后会进行中断检查，因此上面 3 条指令之间都可能发生中断(见第6章)，假设在第2行发生中断，而在中断处理例程中唤醒了另外一个进程，中断返回后调度另外一个进程运行，而另外一个进程也要申请使用打印机，这个进程检测到 `printer` 为 1，所以它获得了打印机的使用权，当这个进程对打印机的使用还未结束时，内核可能会调度前一个进程运行，此时该进程从第3行继续运行，于是两个进程同时获得了打印机的使用权。

这个问题是由于 `printer--` 不是原子操作引起的。为了解决这个问题，在 x86 平台提供了 `dec` 和 `inc` 指令，这两条指令可以直接对内存进行减 1 或者加 1 操作。因此说上面的指令可以直接编译为 `decl printer`，这样在单 CPU 系统上，这就是一个原子操作了。但是在多处理器系统上，上述问题依然存在，因为在 CPU 内部，`decl printer` 还是由几条微指令组成，它还是需要首先把 `printer` 的值从内存读取到 CPU 中的某个程序员不可见的寄存器中，然后执行减一操作，最后再写入内存，只不过在这个过程中间，CPU 不会进行中断检查，因此对单 CPU 来说它是原子操作。但是对多 CPU 系统来说，情况就不一样了，例如在它写回之前，另外一个 CPU 可能也会去读取 `printer` 的值，此时还是会发生在前面的情况。为此 x86 处理器提供了 `lock` 前缀，来避免这个问题。其汇编指令如下：

代码片段 2.15 示例代码

```
1 # printer--
2 lock decl printer
```

`lock` 前缀告诉 CPU，在执行当前指令期间锁住内存总线，这样在 `decl` 操作的微指令执行期间，如果另外的 CPU 访问 `printer`，由于得不到总线仲裁的许可，在 `decl` 操作完成之前，不会访问到 `printer` 内存变量，因此它保证了在多处理器上的原子性。现在我们来看看内核中原子操作的实例。

代码片段 2.16 示例代码

```
1 typedef struct { int counter; } atomic_t;
2
3 static __inline__ void atomic_dec(atomic_t *v)
4 {
5     __asm__ __volatile__(
6         LOCK_PREFIX "decl %0"
7         : "+m" (v->counter)
8     );
9 }
```

其中 `v->counter` 是一个可读写的内存变量，在单 CPU 平台上 `LOCK_PREFIX` 展开为空，在 x86 的多 CPU 平台上，`LOCK_PREFIX` 展开为 `lock`。表2.2是 Linux 内核中常用的原子操作函数。

表 2.2 内核常用的原子操作函数

操作函数	说明
atomic_add()	原子加
atomic_sub()	原子减
atomic_sub_and_test()	原子减，如果结果为 0，返回 1，否则返回 0
atomic_inc()	原子加 1
atomic_dec()	原子减 1
atomic_dec_and_test()	原子减 1，如果结果为 0，返回 1，否则返回 0
atomic_inc_and_test()	原子加 1，如果结果为 0，返回 1，否则返回 0
atomic_add_negative()	原子加，如果结果为负数(溢出了)，返回 1，否则返回 0
atomic_add_return()	原子加，并返回结果
atomic_sub_return()	原子减，并返回结果

2.3.2 信号量

信号量(semaphore)是建立在原子操作的基础上的，它是由荷兰学者 E.W.Dijkstra 提出来的。信号量实际上是一个整型变量，有两个最基本的原子操作，就是操作系统理论教材中常说的 P(Prolagen)操作和 V(Verhogen)操作，又被称为 PV 原语。这两个单词是荷兰语，在 Linux 内核中，人们使用 down 和 up 来表示。我们还是以上一节的打印机的例子来说明 PV 操作。假设只有一个打印机，因此信号量 printer 被初始设置为 1。当进程需要使用打印机时，进行 P(down)操作，该操作定义如下：

代码片段 2.17 示例代码

```

1  printer--;
2  if (printer >= 0) {
3      /* 使用打印机。*/
4  } else {
5      /*
6      * 把当前进程设置为阻塞状态，并且加入到信号量 printer 的等待队列。
7      * 然后调度其他进程运行。
8      */
9 }
```

当一个进程在使用打印机结束时，必须进行 V(up)操作，定义如下：

代码片段 2.18 示例代码

```

1  printer++;
2  if (printer <= 0) {
3      /* 从信号量 printer 的等待队列中唤醒一个进程。*/
4 }
```

从这里可以看出，信号量的初值表示资源的可用数量，当它为0时，说明供求恰好满足，当它小于0时，说明供小于求，当它大于0时反映了供大于求。例如当信号量为-2时，说明等待队列中有两个进程在等待使用打印机。

在Linux内核中，信号量定义如下：

代码片段 2.19 节自 include/asm/semaphore_32.h

```

1 struct semaphore {
2     /* 信号量。*/
3     atomic_t count;
4
5     /* 等待该信号量的进程个数。*/
6     int sleepers;
7
8     /* 该信号量的等待队列。*/
9     wait_queue_head_t wait;
10 };

```

Linux内核中的P操作对应于函数down(), 定义如下：

代码片段 2.20 节自 include/asm/semaphore_32.h

```

1 static inline void down(struct semaphore * sem)
2 {
3     might_sleep();
4     __asm__ __volatile__(
5         "# 在多CPU平台下使用lock前缀，锁住内存总线。
6         "# atomic down operation\n\t"
7         LOCK_PREFIX "decl %0\n\t"
8
9         # 如果sem->count >=0 就跳转到标号2处。
10        "jns 2f\n"
11        "\tlea %0,%eax\n\t"
12        "call __down_failed\n"
13        "2: "
14        : "+m" (sem->count)
15        :
16        : "memory", "ax");
17 }

```

在这个函数中，%0对应于sem->count，如果sem->count减1后，结果不小于0，则说明down操作成功完成。否则调用__down_failed，它最终将调用__down()把当前进程设置为等待状态，并把当前进程加入到该信号量的等待队列，调度其他进程来运行。当某个进程释放相应的资源时调用up(), 定义如下：

代码片段 2.21 节自 include/asm/semaphore_32.h

```

1 static inline void up(struct semaphore * sem)
2 {
3     __asm__ __volatile__(
4         "# 在多 CPU 平台下使用 lock 前缀，锁住内存总线。
5         "# atomic up operation\n\t"
6         LOCK_PREFIX "incl %0\n\t"
7
8         # 如果 sem->count 加 1 大于 0，就跳转到标号 1。
9         "jg 1f\n\t"
10        "lea %0,%%eax\n\t"
11
12        "call __up_wakeup\n"
13        "1: "
14        : "+m" (sem->count)
15        :
16        : "memory", "ax");
17 }

```

在这段代码中，如果 `sem->count` 加 1 的结果小于等于 0，就说明该信号量的等待队列中不为空，因此调用 `__up_wakeup()` 从等待队列中唤醒相应的进程。

2.3.3 自旋锁

在前面的原子操作中，使用 `lock` 前缀能够避免 CPU 在执行单条指令时被打扰，然而当面对成块指令时就无能为力了。我们考虑下面这样一段代码：

代码片段 2.22 示例代码

```

1 struct _foo {
2     int tag;
3     int a;
4     int b;
5     int c;
6 };
7
8 #define SIZE 10
9 struct _foo foo[SIZE];
10
11 int clean_by_tag(int tag)
12 {
13     for (int i = 0; i < SIZE; i++) {
14         if (foo[i].tag == tag) {

```

```

15     foo[ i ].a = 0;
16     foo[ i ].b = 0;
17     foo[ i ].c = 0;
18 }
19 }
20 }
```

由于 CPU 在每条指令结束时都会进行中断检查，而中断产生具有不确定性。现在假设在单处理器系统上，一个进程在执行到第15行时，发生了一个中断，此时进入中断处理函数，而在中断处理函数中，中断处理结束时，内核可能调度另外一个进程运行(见第6章)，如果恰好这个进程需要访问上面要修改的数组成员，那么它将得到错误的数据，这些数据的一部分是旧的，另外一部分是新的。在某些场合，这将会导致严重的问题。为此要保证一个代码块的原子性，最好的办法是在第15行之前关闭中断，在第17行之后再开启中断。在 x86 中，可以通过 cli 指令清除标志寄存器中的中断允许位，在使用 sti 指令开启中断前，CPU 执行完一条指令后，就不会进行中断检查，因此可以免受中断打扰。由于关闭中断后，系统的外部设备可能得不到响应，因此这就要求进程关中断的时间必须是短暂的，并且开启中断前，进程不得进入睡眠状态。

通过关中断的方式可以使 CPU 在执行某些“临界”代码块中免受打扰，但是 cli 只能关闭当前 CPU 的中断，因此在多 CPU 系统中，还是无法避免这个问题，因为当其中一颗 CPU 关闭中断后，执行上面的代码的同时，另外一颗 CPU 可能也会访问同一个数据，在这种情况下同样会得到错误的结果。虽然可以通过信号量机制来防止其他 CPU 在同时访问同一个数据，但是由于下面的理由，信号量显得不太合适。

- (1) 信号量引起的进程切换消耗相对较大，显得不划算。由于这类“临界”代码的特征是执行时间非常短暂，也就是说 CPU 执行这段代码的时间，远远小于进程切换的消耗，因此在这种情况下，还不如让另外一个 CPU 进入“忙等待”状态，直到临界代码操作完成。
- (2) 由于信号量可能引起的进程切换，但是在某些环境下，是不允许进程切换的，例如在中断环境中(见第6章)。

因此在这种情况下，当一个进程不能进入临界区时，最好的办法是让 CPU 进入忙等待状态。其基本原理就是设置一个锁变量，用来保护临界区代码，当 CPU 进入临界区之前，检查锁的状态，如果已经上锁，则当前 CPU 执行一个空循环反复检测锁的状态，直到其他的 CPU 解锁。由于在测试期间，CPU 处于忙等待的“自旋”状态，因此把这种机制称为自旋锁。在 Linux 内核中，自旋锁的类型为 spinlock_t，其定义如下：

代码片段 2.23 节自 include/linux/spinlock_types.h

```

1 typedef struct {
2     raw_spinlock_t raw_lock;
```

```

3 #if defined(CONFIG_PREEMPT) && defined(CONFIG_SMP)
4     unsigned int break_lock;
5 #endif
6 #ifdef CONFIG_DEBUG_SPINLOCK
7     unsigned int magic, owner_cpu;
8     void *owner;
9 #endif
10 #ifdef CONFIG_DEBUG_LOCK_ALLOC
11     struct lockdep_map dep_map;
12 #endif
13 } spinlock_t;

```

为了方便讨论，我们假设没有配置 CONFIG_PREEMPT 及 CONFIG_DEBUG_XXX 标志，所以 spinlock_t 就有一个 raw_lock 成员，其类型为 raw_spinlock_t，定义如下：

代码片段 2.24 raw_spinlock_t 定义

```

1 #ifdef CONFIG_SMP
2     typedef struct {
3         unsigned int slock;
4     } raw_spinlock_t;
5 #else
6     typedef struct {} raw_spinlock_t;
7 #endif

```

在单 CPU 的情况下，raw_spinlock_t 为空，这个结构定义在 include/linux/spinlock_types_up.h 文件中。在多 CPU 的情况下，raw_spinlock_t 结构包含了一个 slock 成员，slock 将记录锁的状态。

当使用自旋锁时，首先需要用 spin_lock_init 来初始化，spin_lock_init 是一个宏，其定义如下：

代码片段 2.25 节自 include/linux/spinlock.h

```

1 # define spin_lock_init(lock) \
2     do { *(lock) = SPIN_LOCK_UNLOCKED; } while (0)

```

由于涉及自旋锁的调试，SPIN_LOCK_UNLOCKED 是一个套了好几层的宏，这里我们略去中间的宏，最终 SPIN_LOCK_UNLOCKED 展开如下：

代码片段 2.26 SPIN_LOCK_UNLOCKED 宏

```

1 #ifdef CONFIG_SMP
2     lock.raw_lock = 1;
3 #else
4     lock.raw_lock = {};
5 #endif

```

在配置了 CONFIG_SMP 的情况下，就把 raw_lock 成员初始化为 1，否则由于 raw_lock 为空，所以在里面的初始化过程中，什么也不用做。

申请和释放自旋锁分别由 spin_lock_irq() 和 spin_unlock_irq() 完成，spin_lock_irq() 首先关闭中断，然后测试锁的状态，如果不可用就进入忙等待状态。这也是一个套了好多层的宏，我们略去中间步骤，结果如下：

代码片段 2.27 节自 kernel/spinlock.c

```

1 #define spin_lock_irq(lock)      __spin_lock_irq(lock)
2
3 void __lockfunc __spin_lock_irq(spinlock_t *lock)
4 {
5     local_irq_disable();
6     preempt_disable();
7     spin_acquire(&lock->dep_map, 0, 0, __RET_IP__);
8     LOCK_CONTENDED(lock, __raw_spin_trylock, __raw_spin_lock);
9 }
10
11 /* LOCK_CONTENDED 其实就是调用 __raw_spin_lock 函数。 */
12 #define LOCK_CONTENDED(_lock, try, lock)
13     lock(_lock)

```

preempt_disable() 的作用是关闭进程抢占，由于中断或系统调用之后，可能会调度其他的进程运行(例如当前进程时间片用完，或者有一个拥有更高优先级的进程已经进入了就绪状态)，preempt_disable() 关闭调度器的这个功能，从而保证当前进程在执行临界区代码的过程中不会被其他进程干扰。

local_irq_disable() 的作用是关闭中断，而 LOCK_CONTENDED 最终将调用 __raw_spin_lock() 函数。在 x86 平台上，local_irq_disable() 最终调用 native_irq_disable()，定义如下：

代码片段 2.28 节自 include/asm-x86/irqflags_32.h

```

1 static inline void native_irq_disable(void)
2 {
3     asm volatile("cli": : :"memory");
4 }

```

这其实就是一条关中断的汇编指令，前面我们说过，在单 CPU 的系统中，进入临界区代码只需要关闭中断就可以了，而多 CPU 的系统中，则需要测试自旋锁的状态。所以 __raw_spin_lock() 也有两个版本：

代码片段 2.29 节自 include/asm/spinlock_32.h

```

1 # define __raw_spin_lock(lock)      __raw_spin_lock(&(lock)->raw_lock)

```

```
2
3 #ifdef CONFIG_SMP
4 /* 多 CPU, 测试锁的状态, 有可能进入自旋状态。*/
5 static inline void __raw_spin_lock(raw_spinlock_t *lock)
6 {
7     asm volatile("\n1:\t"
8         /* 锁住内存总线, lock->slock--。*/
9         "LOCK_PREFIX " ; decb %0\n\t"
10        /*
11         * 如果 lock->slock-- >= 0, 说明成功获取到了这个自旋锁,
12         * 跳转到标号 3(第28行), 成功退出。
13         */
14     "jns 3f\n"
15
16     /* 否则进入自旋状态, 执行 nop 指令, 并测试 lock->slock 是否为 0. */
17     "2:\t"
18     "rep; nop\n\t"
19     "cmpb $0,%0\n\t"
20
21     /* 如果 lock->slock <=0, 就跳转到标号 2(第17行), 继续“自旋”。*/
22     "jle 2b\n\t"
23     /*
24      * 否则, lock->slock > 0, 就说明别的 CPU 释放了该自旋锁,
25      * 就跳转到标号 1(第7行), 尝试开锁。
26      */
27     "jmp 1b\n"
28     "3:\n\t"
29     : "+m" (lock->slock) : : "memory");
30 }
31
32 #else
33 /* 单 CPU 展开为一个空操作。*/
34 # define __raw_spin_lock(lock) do ( (void)(lock); ) while (0)
35 #endif
```

从上面可以看出, 当一个 CPU 获取自旋锁失败时, 这个 CPU 就在循环中做无用功, 等待其他的 CPU 释放自旋锁。正是这个原因, 所以要求持有自旋锁的时间必须尽可能短暂, 并且持有自旋锁时不能进入睡眠状态。

类似地, `spin_unlock_irq()`展开的结果如下:

代码片段 2.30 节自 `include/linux/spinlock.h`

```
1 # define spin_unlock_irq(lock) \
```

```

2 do { \
3     __raw_spin_unlock(&(lock)->raw_lock); \
4     __release(lock); \
5     local_irq_enable(); \
6 } while (0)

```

在单 CPU 系统中，`spin_unlock_irq()`只需要打开中断就可以了，因此 `__raw_spin_unlock()` 是一个空操作，开中断由 `native_irq_enable()` 函数完成，定义如下：

代码片段 2.31 节自 include/asm-x86/irqflags_32.h

```

1 static inline void native_irq_enable(void)
2 {
3     asm volatile("sti": : : "memory");
4 }

```

在多 CPU 系统中，`spin_unlock_irq()` 还需要把 spinlock 设置为开锁状态。

代码片段 2.32 节自 include/asm/spinlock_32.h

```

1 static inline void __raw_spin_unlock(raw_spinlock_t *lock)
2 {
3     /* lock->slock = 1 */
4     asm volatile("movb $1,%0" : "+m" (lock->slock) :: "memory");
5 }

```

从上面的分析中，我们看到 `spin_lock_irq()` 和 `spin_unlock_irq()` 保护临界区代码不受打扰。然而申请自旋锁时需要关闭中断，释放自旋锁时又要开启中断，这里带来了一个潜在的问题。现在假设某段代码在调用 `spin_lock_irq()` 之前需要关闭中断，之后获取了某个自旋锁，在释放自旋锁时，`spin_unlock_irq()` 开启了中断，然而该段代码此时可能根本不允许开启中断。因此我们需要在调用申请自旋锁时保存当时的中断许可情况，在释放自旋锁时恢复它，而不是盲目地开启中断。`spin_lock_irqsave` 和 `spin_unlock_irqrestore` 就用来完成这个工作，定义如下：

代码片段 2.33 节自 include/linux/spinlock.h

```

1 #define spin_lock_irqsave(lock, flags) \
2     flags = __spin_lock_irqsave(lock)
3
4 #define spin_unlock_irqrestore(lock, flags) \
5     __spin_unlock_irqrestore(lock, flags)

```

由于中断许可位位于 CPU 的标志寄存器中，因此 `spin_lock_irqsave()` 在获取自旋锁之前，把标志寄存器的值保存到 `flags` 中，而 `spin_unlock_irqrestore()` 在释放自旋锁之后，根据 `flags` 恢复标志寄存器。这是通过下面的两个函数完成的。

代码片段 2.34 节自 include/asm-x86/irqflags_32.h

```

1 static inline unsigned long native_save_f1(void)
2 {
3     unsigned long f;
4     /* 把标志寄存器的值保存到 f 中。*/
5     asm volatile("pushfl ; popl %0": "=g" (f): /* no input */);
6     return f;
7 }
8
9 static inline void native_restore_f1(unsigned long f)
10 {
11     /* 根据 f 恢复标志寄存器。*/
12     asm volatile("pushl %0 ; popfl": /* no output */
13                 : "g" (f)
14                 : "memory", "cc");
15 }

```

除此之外，内核中还有许多自旋锁的操作函数，例如 `read_lock_irqsave()`/`read_unlock_irq()` 等，在理解了前面讨论的知识的基础上，其它的也是很容易理解的，读者可自行分析。

2.3.4 RCU 机制

自旋锁能够很好地对临界资源进行保护，但是当 CPU 处于自旋状态时实际上是在做“无用功”，因此我们需要尽量避免实用自旋锁。在实际应用中，对于某些关键数据结构而言，读取操作的次数远远超过修改操作的次数。典型的例子就是内核中的路由表，每当收到一个网络数据包时，内核路由模块都需要读取路由表，来判断如何处理这个网络数据包，但是却很少修改路由表。事实上，在一个多 CPU 系统中，当多个 CPU 同时读取路由表时，并不会带来任何问题。但是我们不得不考虑到写入操作带来的同步问题，例如在图2.1中，CPU1 正在遍历链表，同时当 CPU1 通过 a 的 next 指针引用到 b 时，CPU2 却把 b 删除了，这样 CPU1 就引用到一个无效的内存地址。因此我们必须对读写操作进行加锁保护，但是这必然会造成性能的浪费，例如：当 CPU1 读取链表时，CPU2 也要进行读取操作，在传统的锁机制中，CPU2 必须等待 CPU1 操作完成后，才能“开锁”。在读取操作次数远大于写入操作的情况下，带来了不必要的消耗。为了解决这个问题，人们专门为“读多写少”设计了 RCU 机制，它的基本原理就是读操作不加锁，写操作必须加锁。

在图2.1中，对链表中的数据结构进行读取操作时不需要任何额外的加锁操作，现在同时有一个写操作需要删除 b，由于写操作不确定当前是否有其他进程正在引用 b，或即将通过 `b->next` 引用 c，或即将通过 `b->prev` 引用 a，因此在第二步中，写操作仅仅把 `a->next` 调整为 c，同时 `c->prev` 调整为 a，这样后来的读操作就不能再“见到 b”，而对于在删除操作之前就引用 b 的进程而言，这并不会有什么影响，同时它还可以通过 `b->next` 或者 `b->prev`

引用到 c 或 a。同时写操作者需要注册一个回调函数，将来在适当的时机由这个回调函数来删除 b，如图2.1中的第3步所示。

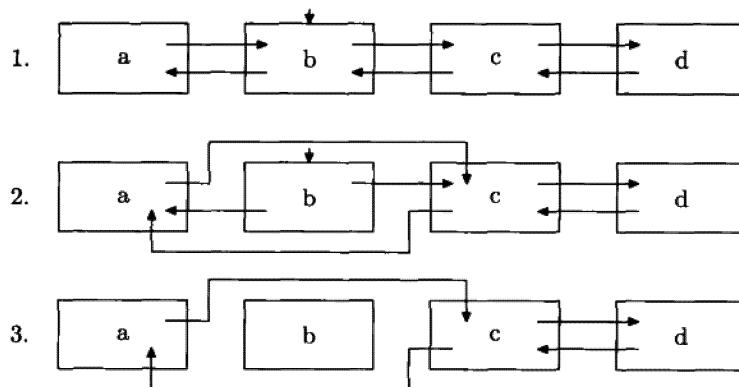


图 2.1 RCU 机制中删除操作

那么什么是适当的时机呢？RCU 机制做了一个规定，进行 RCU 读取操作时，在操作结束之前，读操作进程不能被抢占。在这个前提下，我们就可以确定什么是适当的时机了，那就是在一个 RCU 回调函数被注册之后，所有 CPU 都至少完成过一次进程切换后，就可以安全地执行这个回调函数了。在图2.1第二步进行之后，如果系统中的每一个 CPU 都至少进行了一次进程切换，这就意味着没有进程在引用 b 了，因为在链表中，后来的进程不可能“见到” b，同时之前使用 b(如果有的话)的进程也肯定是用完了。现在我们来看看 RCU 的相关操作函数：

代码片段 2.35 (节自 `include/linux/rcupdate.h`)

```

1 #define rcu_read_lock() \
2     do { \
3         preempt_disable(); \
4         __acquire(RCU); \
5         rCU_read_acquire(); \
6     } while(0)
7
8 #define rCU_read_unlock() \
9     do { \
10        rCU_read_release(); \
11        __release(RCU); \
12        preempt_enable(); \
13    } while(0)

```

在上面的代码中，`rcu_read_lock()`和`rcu_read_unlock()`宏展开后分别为`preempt_disable()`和`preempt_enable()`，它们的作用是禁止或开启抢占功能。其他的都是空操作。`__acquire(RCU)`和

`_release(RCU)`是给 sparse³用的。而 `rcu_read_acquire()`和 `rcu_read_release()`用于调试目的。另外还有一对函数 `rcu_read_lock_bh()`和 `rcu_read_unlock_bh()`，这里的 `bh`是指软中断，对于这两个函数来说，如果没有进行进程切换，但是软中断处理完成，也可以处理由 `call_rcu_bh()`注册的回调函数。在 RCU 机制中，`bh` 系列函数和非 `bh` 的原理是一样的，因此这里我们主要介绍非 `bh` 系列函数。

每一个 CPU 都有一个 `rcu_data` 用于保存该 CPU 的 RCU 回调函数，回调函数的注册由 `call_rcu()`来完成，定义如下：

代码片段 2.36 (节自 kernel/rcupdate.c)

```

1 void fastcall call_rcu(struct rcu_head *head,
2                         void (*func)(struct rcu_head *rcu))
3 {
4     unsigned long flags;
5     struct rcu_data *rdp;
6
7     /* 设置 RCU 的回调函数。*/
8     head->func = func;
9     head->next = NULL;
10    local_irq_save(flags);
11    /* 获得当前 CPU 的 rcu_data。*/
12    rdp = &__get_cpu_var(rcu_data);
13    /* 把 RCU 回调函数加入到队列中。*/
14    *rdp->nxttail = head;
15    rdp->nxttail = &head->next;
16    /*
17     * 如果回调函数太多，就尝试请求调度，在进程切换之后，
18     * 就可以执行这些回调函数了。
19     */
20    if (unlikely(++rdp->qlen > qhimark)) {
21        rdp->blimit = INT_MAX;
22        force_quiescent_state(rdp, &rcu_ctrlblk);
23    }
24    local_irq_restore(flags);
25 }
```

当 CPU 经历一次进程切换时，会调用 `rcu_qsctr_inc()`函数标记这一动作。在时钟中断发生时，中断会检查 RCU 回调函数队列。

代码片段 2.37 (节自 kernel/timer.c)

```
1 void update_process_times(int user_tick)
```

³一个专门针对 Linux 内核，比 gcc 还要严格的语法检测器，可以在编译内核时传递 C=n 来开启该功能，例如 make C=2 bzImage。

```

2 {
3     struct task_struct *p = current;
4     int cpu = smp_processor_id();
5
6     /* Note: this timer irq context must be accounted for as well. */
7     account_process_tick(p, user_tick);
8     run_local_timers();
9     /* 如果当前cpu的RCU回调函数队列非空。*/
10    if (rcu_pending(cpu))
11        rCU_check_callbacks(cpu, user_tick);
12    scheduler_tick();
13    run_posix_cpu_timers(p);
14 }

```

如果cpu的RCU回调函数队列非空，rcu_check_callbacks()就会处理安排一个tasklet(参见6.4节)来对回调函数进行处理。

代码片段 2.38 (节自 kernel/rcupdate.c)

```

1 void rCU_check_callbacks(int cpu, int user)
2 {
3     if (user || (idle_cpu(cpu) &&
4             !in_softirq()) &&
5             hardirq_count() <= (1 << HARDIRQ_SHIFT))) {
6         rCU_qsctr_inc(cpu);
7         rCU_bh_qsctr_inc(cpu);
8     } else if (!in_softirq())
9         rCU_bh_qsctr_inc(cpu);
10    /* 调度一个tasklet来执行RCU链上的回调函数。*/
11    tasklet_schedule(&per_cpu(rcu_tasklet, cpu));
12 }

```

现在我们来看一个例子：

代码片段 2.39 (节自 net/ipv4/devinet.c)

```

1 static inline void inet_free_ifa(struct in_ifaddr *ifa)
2 {
3     call_rcu(&ifa->rcu_head, inet_rcu_free_ifa);
4 }
5
6 static void inet_rcu_free_ifa(struct rcu_head *head)
7 {
8     struct in_ifaddr *ifa = container_of(head, struct in_ifaddr, rcu_head);
9     if (ifa->ifp)

```

```

10     in_dev_put(ifa->if_a_ifa);
11     kfree(ifa);
12 }

```

在上面这个例子中 `inet_free_ifa()` 并没有执行真正的释放操作，它调用 `call_rcu()` 注册了一个回调函数 `inet_rcu_free_ifa()`，将来这个回调函数来执行真正的释放操作。

另外内核中还有大量的 RCU 的相关函数，在明白 RCU 机制之后，这些代码都很容易理解，在此不再一一列举。

2.3.5 percpu 变量

相对于自旋锁来说，RCU 减小了临界区同步的开销，但是并不是所有应用都满足 RCU 的前提条件。例如在 CPU 的调度队列中，假设多个 CPU 共享同一个调度队列，各个 CPU 需要频繁对调度队列中的关键数据结构进行操作，而且读操作次数并不会远远小于写操作次数。因此必须使用自旋锁来保护。事实上内核中进程切换非常频繁，各个 CPU 都需要频繁地访问调度队列。在这种情况下，某些 CPU 由于获取不到自旋锁进入忙等待状态。为了提高性能，最好的办法就是各个 CPU 使用各自的运行队列，这样在访问时就不需要使用锁来保护，当某个 CPU 的就绪进程个数为 0 或大于某个阈值时，启动负载均衡机制来平衡各个 CPU 的运行队列。我们把这样的变量称为 percpu 变量。

通常在程序中定义的全局变量，编译后这些变量位于执行文件的数据区，可执行文件加载之后，这些变量被复制到对应的内存中，源程序中的变量名用于对变量进行寻址。而 percpu 变量的关键就是：要求根据 CPU 的个数，在内存中生成多份拷贝，并且能够根据变量名和 CPU 编号，正确地对各个 CPU 的变量进行寻址。可以通过下面的方法来定义一个 percpu 变量。

代码片段 2.40 percpu 变量定义

```

1 DEFINE_PER_CPU(long, var) = 0;

```

上面的代码定义了一个类型为 `long`，名字为 `var` 的 percpu 变量，它的初始值为 0。其中 `DEFINE_PER_CPU` 定义如下：

代码片段 2.41 节自 `include/asm/percpu_32.h`

```

1 #define DEFINE_PER_CPU(type, name) \
2     __attribute__((__section__(".data.percpu"))) \
3     __typeof__(type) per_cpu_##name

```

根据上面的例子，在这里 `__typeof__(type)` 得到的就是 `long`，而 `per_cpu_##name` 得到的就是 `per_cpu_var4`，并且通过 `__section__` 把这个变量放到名为 `.data.percpu` 的数据

⁴两个 ## 表示字符连接操作，这是由编译器完成的。

段(见第3.3节)。因此在编译时所有的 percpu 变量都会统一的放到.data.percpu 数据段。当内核启动后，会根据检测到的 cpu 个数从数据段.data.percpu 为各个 CPU 复制一份单独的拷贝。start_kernel()函数(见第4章)会调用 setup_per_cpu_areas()来完成这个工作，setup_per_cpu_areas()定义如下：

代码片段 2.42 节自 init/main.c

```

1 static void __init setup_per_cpu_areas(void)
2 {
3     unsigned long size, i;
4     char *ptr;
5     /* 取 CPU 的数量。 */
6     unsigned long nr_possible_cpus = num_possible_cpus();
7
8     /* Copy section for each CPU (we discard the original) */
9     size = ALIGN(PERCPU_ENOUGH_ROOM, PAGE_SIZE);
10    /* 分配内存。 */
11    ptr = alloc_bootmem_pages(size * nr_possible_cpus);
12    /*
13     * __per_cpu_start 和 __per_cpu_end 分别是数据段
14     * .data. percpu 的起始地址和结束地址。
15     */
16    for_each_possible_cpu(i) {
17        /*
18         * 保存每一个 CPU 的 percpu 变量的起始地址和 __per_cpu_start 的
19         * 差值，将来通过 __per_cpu_offset[i] 来定位各个 CPU 的变量。
20         */
21        __per_cpu_offset[i] = ptr - __per_cpu_start;
22
23        /* 把 .data. percpu 复制到 ptr 指向的内存。 */
24        memcpy(ptr, __per_cpu_start,
25               __per_cpu_end - __per_cpu_start);
26        /* 调整内存指针。 */
27        ptr += size;
28    }
29 }

```

在上面的代码中，为每一个 CPU 复制一份.data.percpu 的数据，这个动作发生在运行期，那么在编译期源程序如何访问呢？内核定义了几个宏来实现对 percpu 变量的访问，这里以 per_cpu()为例来介绍，其定义如下：

代码片段 2.43 节自 include/asm-x86/percpu_32.h

```
1 #define per_cpu(var, cpu) (*({ \
```

```

2   extern int simple_identifier_##var(void); \
3   RELOC_HIDE(&per_cpu_##var, __per_cpu_offset[cpu]); })

```

`per_cpu()`相对来说比较复杂，让我们还是使用前面的实例来解释，假设现在访问前面的例子中的 1 号 CPU 的 `var` 变量，访问方式为 `per_cpu(var, 1)`，由于 `percpu` 变量 `var` 在定义时由 `DEFINE_PER_CPU` 展开后得到的名字是 `per_cpu_var`，因此 `RELOC_HIDE` 的第一个参数就是编译时 `per_cpu_var` 的地址，而第二个参数 `__per_cpu_offset[cpu]` 是在 `setup_per_cpu_areas()` 函数中计算出来的，它表示运行时复制的 `percpu` 数据区首地址和编译时确定的首地址的差值，这两个值相加就得到了正确的 `percpu` 的地址。其中 `RELOC_HIDE` 返回的就是这两个值相加的结果，定义如下：

代码片段 2.44 节自 `include/linux/compiler-gcc.h`

```

1 #define RELOC_HIDE(ptr, off)
2 ({ unsigned long __ptr;
3     __asm__ ("": "=r"(__ptr) : "0"(ptr));
4     (typeof(ptr)) (__ptr + (off)); })

```

可能这样还不太直观，我们进一步来实际计算一下这个地址，假设编译时 `percpu` 变量的数据段起始地址为 `A`，变量 `var` 在数据段中的偏移为 `c`，地址为 `A+c`，在运行时检测到了 3 颗 CPU，因此分配了一片内存，并且复制了 3 份 `percpu` 变量的数据段，这 3 份的起始地址分别为 `X`，`Y`，`Z`。同时 `__per_cpu_offset[i]` 中分别保存 `X-A`，`Y-A`，`Z-A`，因此当访问 `percpu` 变量 `var` 的时候，`RELOC_HIDE` 传递的地址为 `A+c`，现在访问第一个 CPU 的变量，因此 `RELOC_HIDE` 的定位方式为 $(Y-A)+(A+c)$ ，所以得到的地址为 `Y+c`。

2.4 内存屏障

由于编译器的优化和缓存的使用，导致对内存的写入操作不能及时地反映出来，也就是说当完成对内存的写入操作之后，读取出来的有可能是旧的内容。我们把这种现象称为内存屏障(Memory Barrier)。

2.4.1 编译器引起的内存屏障

首先让我们来看一个例子，假设有下面这样一段代码：

代码片段 2.45 内存屏障示例代码

```

1 int flag = 0;
2
3 void wait()
4 {

```

```

5   while (flag == 0) {
6     sleep(1000);
7   }
8   .....
9 }
10
11 void wakeup()
12 {
13   flag = 1;
14 }
```

由于编译器的优化，当 gcc 发现 sleep() 函数内部不会修改 flag 变量时，它可能把某个寄存器分配给内存变量 flag，于是上面的代码编译后可能是这个样子(为了尽量直观易懂，这个例子中采用了 C 和汇编代码结合的方式来说明这个问题)。

代码片段 2.46 内存屏障示例代码

```

1 void wait()
2 {
3   movl flag, %edx;
4
5   while (%eax == 0) {
6     sleep(1000);
7   }
8   .....
9 }
```

在这个例子中 gcc 为了优化代码，把 EDX 分配给内存变量 flag，这样可以减少内存访问的次数。假设现在 flag 为 0，线程进入睡眠状态，当它被唤醒时，会再次判断 EDX 的值。在这种情况下，就算另外一个线程在某个时候调用了 wakeup() 把 flag 设置为 1，这个睡眠的线程仍然不能跳出 while 循环。由此可见编译器的优化带来了副作用。即便是在单 CPU 的系统上，也会出现问题。好在我们可以使用 volatile 来避免这种情况，因此上面对 flag 变量的定义可以修改为：

代码片段 2.47 内存屏障示例代码

```
1 volatile int flag = 0;
```

这里关键字 volatile 的作用是要避免编译器的优化，这样编译器就不会把某个寄存器分配给 flag。编译后的代码就是，每一次对 flag 的访问都是通过内存访问来进行的，从而避免了这个问题。

另外 volatile 常常用于外部设备 IO 寄存器访问，考虑下面的例子：

代码片段 2.48 内存屏障示例代码

```

1 /* 假设 0x80 为某一个外部设备寄存器的地址。*/
2 volatile unsigned int *p_status = 0x80;
3
4 while (*p_status != ERROR) {
5     do_something();
6 }
```

在这个例子中，由于指针 `p_status` 指向外部设备的某个寄存器，而外部设备随时有可能改变这个寄存器的值，因此也要通过 `volatile` 阻止编译器优化。

通过使用 `volatile` 可以避免编译器在优化时把寄存器分配给不必要的内存变量，从而保证对内存的修改立即反映到相关进程。但是在某些情况下，由于涉及的变量比较多，如果把每一个变量声明为 `volatile` 显得很烦琐。因此内核中使用另外一个方式来避免编译器优化引起的副作用。其代码如下：

代码片段 2.49 节自 include/linux/compiler-gcc.h

```

1 #define barrier() __asm__ __volatile__("": : : "memory")
```

这条汇编指令指令部分、输出部分、输入部分都为空，唯独损坏部分为"memory"，它告诉 gcc 内存已经被修改了。当 gcc 遇到这条指令时，gcc 会插入必要的指令重新刷新内存和它对应的寄存器。那么这个 `barrier()` 如何使用呢？我们来看内核中的一个实际例子。

代码片段 2.50 节自 kernel/sched.c

```

1 static inline void
2 context_switch(struct rq *rq,
3                 struct task_struct *prev,
4                 struct task_struct *next)
5 {
6     struct mm_struct *mm, *oldmm;
7
8     prepare_task_switch(rq, prev, next);
9     mm = next->mm;
10    oldmm = prev->active_mm;
11    .....
12    /* Here we just switch the register state and the stack. */
13    switch_to(prev, next, prev);
14    barrier();
15    finish_task_switch(this_rq(), prev);
16 }
```

在内核中 `context_switch()` 负责从当前进程切换到另外一个进程环境中，但是由于当前进程需要修改某些内核数据结构，这些修改需要及时地反映到另外一个进程中。因此在这

里插入 barrier(), 这样 gcc 会采取必要的措施, 以保证内存变量和对应的寄存器的一致性。需要注意的是, 这个动作是在编译期发生的。

2.4.2 缓存引起的内存屏障

硬件设计者根据程序的局部性原理在 CPU 中集成了缓存, 由于 CPU 对缓存的访问速度远远大于对内存的访问速度, 因此缓存的出现, 极大的提高了 CPU 的性能。但是也不可能避免地带来了新的问题。那就是运行期的内存屏障。

我们还是先以一个例子来介绍单 CPU 系统中存在的问题, DMA 机制使得外部设备和 CPU 都能够操作内存。现在假设一块网卡通过 DMA 收发数据。在接收过程中, 网卡通过 DMA 操作把数据传送到内存中, 然后 CPU 从这片内存区域中读取接收到的数据, 如果之前这片内存区域的一部分在缓存中, 那么 CPU 读到的数据就是缓存中的旧数据。在发送过程中, CPU 把待发送的数据写入内存, 然后对网卡写入“发送”命令, 网卡通过 DMA 操作发送数据时, 由于 CPU 写入的数据可能保存在缓存中, 因此网卡实际发送出去的数据很可能是错误的。因此需要一种机制, 当外部设备的 DMA 操作结束时, 需要通知 CPU 其内部缓存的对应区域已经无效。而 CPU 发动 DMA 操作时, 在向外部设备发送启动命令前, 需要把缓存中的内容回写到内存中。这个任务可以通过软件实现, 也可以通过硬件实现。目前大部分精简指令集(RISC)构架的 CPU 使用软件方式实现, 其基本原理是 CPU 提供特殊的指令支持, 当设备完成 DMA 操作后, CPU 开始接手处理时运行该指令(例如在网卡的接收中断处理函数中), 使缓存无效, 这样 CPU 就会从内存中读入最新的数据。当把内存中的数据准备好, 在启动外部设备 DMA 操作之前, 执行该指令(例如在网卡发送函数中), 把缓存的内容回写到内存。而在 x86 构架的 CPU 上, 采用的是硬件的实现方式。它采用的方法又被称为总线监测技术(Bus Watching), 基本原理是: CPU 和外部设备访问内存时都必须经过总线仲裁, 现在使用一个专门的硬件模块记录在缓存中的内存区域, 当外部设备对内存进行写入操作时, 该模块判断这个区域是否位于 CPU 的缓存中, 如果是则使缓存无效, 当外部设备读取内存时, 如果这个内存区域在缓存中, 并且缓存已经被 CPU 修改, 就把缓存内容回写到内存。因此在 x86 的单 CPU 平台上, 软件不需要考虑内存一致性的问题。然而在多 CPU 的平台上, 情况变得复杂。

在多 CPU 的系统中, 由于各个 CPU 内部都有独立的缓存, 而缓存大小远远小于内存大小, 因此缓存和内存是一对多的关系。也就是说一个缓存行对应多个内存区域。在这种情况下, 可能 CPU1 对内存的修改还没有回写到内存中, 而 CPU2 读取到旧的数据。即使 CPU1 及时地把修改的内容回写到内存之中, 但是由于 CPU2 在此之前已经把对应的内存区域读取到缓存之中, 因此 CPU2 仍然从缓存中读取到旧的数据。

假设内存内存地址 A, B 都对应缓存行 X, 缓存行大小为 L, 现在来考虑下面的代码:

代码片段 2.51 内存屏障示例代码

```
1 struct wk *insert(long data)
```

```
2 {
3     struct wk *p;
4     p = kmalloc(sizeof(*p), GFP_ATOMIC);
5     spin_lock(&mutex);
6     p->next = head.next;
7     p->data = data;
8     smp_wmb();
9     head.next = p;
10    spin_unlock(&mutex);
11 }
12
13 struct wk *search(long data)
14 {
15     struct el *p;
16     p = head.next;
17     while (p != &head) {
18         smp_rmb();
19         if (p->data == data) {
20             return (p);
21         }
22         p = p->next;
23     };
24     return (NULL);
25 }
```

kmalloc 分配的结构和 head 都被保存在缓存中，假设 head.next 的内存地址为 A，进程在调用 insert() 函数之后，对内存地址 B 进行访问，由于 A 和 B 对应的缓存行都是 X，因此 CPU1 把缓存 X 回写到内存 A，同时把 B 的内容加载到缓存 X 中。此时从外部来看，对内存的实际写入操作与源程序的顺序不一致，因为第9行的写入操作已经到达内存，但是第6至7行的写入操作还在 CPU1 的缓存中。此时如果 CPU2 上的进程调用 search() 对链表进行遍历，可能出现以下两种情况：

- 添加到链表中的内容不在 CPU2 的缓存中，此时 CPU2 完全从内存中加载，由于在 insert() 函数中 head->next 已经指向了 p，但是内存结构 p 却在缓存中，而没有被及时地回写到内存，因此该 CPU 将访问到错误的数据，更为严重的是读取到的 p->next 是一个错误的指针。
- 添加到链表中的内容在该 CPU 的缓存中，此时缓存命中，就算之前 insert() 函数的修改已经被全部回写到内存中。这个 CPU 读取到的也是旧的数据。

因此对某些关键的数据结构进行修改时，如果其他的 CPU 有可能对这些数据进行访问，需要一种机制来保证内存的一致性，Linux 内核把这个机制称为“内存屏障”。当 CPU

对数据关键数据结构进行修改时，为了保证这个修改立即反映到其他 CPU，因此需要调用 `smp_wmb()`，它立即把修改过的缓存回写到内存中。当其他 CPU 访问某个关键数据结构时，由于这个数据结构可能已经位于该 CPU 的缓存中，但是内存可能被其他 CPU 修改过了，因此需要调用 `smp_rmb()` 使缓存无效，这样就可以确保访问到最新的内容。`smp_wmb()` 和 `smp_rmb()` 定义如下：

代码片段 2.52 节自 `include/asm-x86/system_32.h`

```

1 #ifdef CONFIG_SMP
2 #define smp_mb()    mb()
3 #define smp_rmb()   rmb()
4 #define smp_wmb()   wmb()
5
6 /* 没有配置 SMP 的情况下，就仅仅设置编译器的内存屏障。*/
7 #else
8 #define smp_mb()    barrier()
9 #define smp_rmb()   barrier()
10 #define smp_wmb()   barrier()
11#endif

```

对于不同的 CPU，`smp_wmb()` 和 `smp_rmb()` 的实现是不同的，由于采用了总线监测技术，因此 x86 架构可以保证写入操作的 cache 和内存的一致性，假设 CPU1 对某个内存地址进行写入操作，即使结果保存在 CPU 的缓存中，而没有及时地反映到内存，现在 CPU2 对内存进行读取，只要对应的内存地址之前没有在 CPU2 的缓存中（CPU2 由于缓存没有命中，因此通过内存总线对该地址进行了一次读取操作），在这种情况下，总线监测模块能够保证 CPU2 读取到最新的值。因此对于关键数据结构的写入操作后，软件不需要额外考虑内存屏障的问题，但是在这个例子中，当进行读取操作时，如果对应的内存之前就被保存到 CPU2 的缓存中，由于这个读操作不需要经过内存总线，因此总线监测模块就显得无能为力了，所以在读取操作之前 CPU2 必须使自己的缓存无效。在 x86 中，有许多指令能够做到这一点，例如：带有 lock 前缀的指令，`cpuid`, `iret` 等指令。因此内核中 `mb()`, `rmb()`, `wmb()` 定义如下：

代码片段 2.53 节自 `include/asm-x86/system_32.h`

```

1 /*
2  * For now, "wmb()" doesn't actually do anything, as all
3  * Intel CPU's follow what Intel calls a *Processor Order*,
4  * in which all writes are seen in the program order even
5  * outside the CPU.
6  *
7  * I expect future Intel CPU's to have a weaker ordering,
8  * but I'd also expect them to finally get their act together
9  * and add some real memory barriers if so.
10 */

```

```

11
12 #define mb() alternative("lock; addl $0,0(%esp)",  

13           "mfence", X86_FEATURE_XMM2)
14 #define rmb() alternative("lock; addl $0,0(%esp)",  

15           "lfence", X86_FEATURE_XMM2)
16 #define wmb() alternative("lock; addl $0,0(%esp)",  

17           "sfence", X86_FEATURE_XMM)

```

上面的例子中，使用 lock 前缀来使缓存无效，esp+0 对 esp 没有影响。虽然在 Intel 的 CPU 中，软件不需要考虑写入操作的内存屏障问题，但是由于作者考虑到 Intel 将来的 CPU 可能会需要 wmb()，所以这里 wmb() 也使用了 lock 前缀来使缓存无效(多虑了？)。关于 alternative() 中的其他指令的作用，我们在下一节介绍。

2.4.3 乱序执行引起的内存屏障

为了追求更高的系统性能，超标量已经成为处理器的主流技术。超标量的本质是，一个 CPU 拥有多条独立的流水线，它拥有多个取指、译码、运算等部件，因此可以同时在不同的流水线上执行多条指令。而传统的标量处理器只有一条流水线。为了说明这个问题，我们考虑下面的简化代码：

代码片段 2.54 内存屏障示例代码

```

1 .....
2 addl (memA), %eax
3 movl (memB), %ecx
4 movl (memC), %edx
5 .....

```

我们假设在一个双发射的超标量处理器上面执行这一段代码，该处理器可以同时取两条指令，然后同时译码，并根据译码结果提交到不同的运算部件上执行。由于在上面的代码中，第一条指令可能要比第二条指令占用更多的执行周期，因此第二条指令先于第一条指令完成。在完成第二条指令的流水线中，可能接着执行第三条指令。这说明在超标量处理器中，CPU 对内存的操作实际顺序和指令中的顺序不一致。在超标量处理器中，不仅仅是对内存操作乱序，可能指令也是乱序执行的，也就是说前面的指令还在执行，后面的指令可能已经结束。实际上超标量在提升性能的同时，也有很多限制，如下面的代码所示：

代码片段 2.55 内存屏障示例

```

1 movl (memA), %eax
2 movl %eax, %ecx

```

在第一条指令结束前，第二条指令是不允许执行的(因为第二条指令的输入来自第一条指令的输出)，这个限制被称为相关性。一共有 5 种相关性，它们分别是：真实数据相关

性(true data dependency), 过程相关性(procedural dependency), 资源冲突(resource conflicts), 输出相关性(output dependency), 反相关性(antidependency)。如何避免这些相关性带来的问题, 是超标量处理器研究的一个热门话题。目前已经存在不少方法来解决相关性的问题, 其中包括软件方法和硬件方法。感兴趣的读者可以参考计算机体系结构的相关资料进一步获取相关知识⁵。

我们还是回到内存乱序的话题, 考虑下面的代码:

代码片段 2.56 内存屏障示例

```

1 struct wk *insert(long data)
2 {
3     struct wk *p;
4     p = kmalloc(sizeof(*p), GFP_ATOMIC);
5     spin_lock(&mutex);
6     p->next = head.next;
7     p->data = data;
8     smp_wmb();
9     head.next = p;
10    spin_unlock(&mutex);
11 }
```

假设由于某种原因导致第9行对应的汇编指令, 先于第6行和第7行完成, 而此时另外一个CPU正在通过head访问链表, 由于head.next已经指向p, 但是p的内容却还没有更新, 因此这个CPU将访问到错误的数据。在这种情况下, 必须保证第6行和第7行对内存的写入操作完成后, 才能执行第9行的指令。在x86平台上由专门的汇编指令来完成这个工作, 分别是lfence, sfence, mfence。它们的原理都是停止流水线停, 保证相关操作确实已经完成。作用如下:

lfence 当CPU遇到lfence指令时, 停止相关流水线, 直到lfence之前对内存进行读取操作的指令全部完成。

sfence 当CPU遇到sfence指令时, 停止相关流水线, 直到sfence之前对内存进行写入操作的指令全部完成。

mfence 当CPU遇到mfence指令时, 停止相关流水线, 直到mfence之前对内存进行读取和写入操作的指令全部完成。

Intel是从Pentium 4开始引入超标量技术的, 所以lfence, sfence, mfence是Pentium 4及后续处理器特有的指令, 因此不能在Pentium 4之前的处理器上执行这些指令, 而且在Pentium 4之前的CPU中也不可能因指令乱序执行而引起内存屏障, 因此只需要使用上一节介绍的lock前缀, 来避免缓存引起的内存屏障就可以了。

⁵关于超标量带来的相关性及其避免的话题, 可以参考《计算机体系结构: 量化研究方法》一书。

代码片段 2.57 节自 include/asm-x86/system_32.h

```

1 #define mb() alternative("lock; addl $0,0(%esp)",  

2                               "mfence", X86_FEATURE_XMM2)  

3 #define rmb() alternative("lock; addl $0,0(%esp)",  

4                               "lfence", X86_FEATURE_XMM2)  

5 #define wmb() alternative("lock; addl $0,0(%esp)",  

6                               "sfence", X86_FEATURE_XMM)

```

内核启动时会根据检测到的 CPU 来决定执行 alternative() 的哪一条指令，如果是 Pentium 4 之前的 CPU 就执行 lock addl \$0,0(%esp)，否则就执行 xfence 指令(这里的 xfence 表示 mfence, lfence, sfence)。

2.5 高级语言的函数调用规范

函数(Function)是高级语言的一个重要概念，有时又被称为过程(Procedure)或者例程(Routing)。我们知道在函数调用过程中，局部变量是在堆栈中分配的，参数也是通过堆栈传递的。考虑下面这样一个函数：

代码片段 2.58 示例代码

```

1 int f1(int arg1, int arg2, int arg3)  

2 {  

3     .....  

4     return 1;  

5 }

```

当通过 f1(1, 2, 3)来调用这个函数时，编译器需要有一种“约定”，调用者如何通过堆栈来传递参数，被调用者如何获取参数，当调用结束后，堆栈中的参数不再有用，又该如何清除堆栈中的参数，把堆栈还原到调用前的模样，子程序如何把返回值传递到父函数。在 Linux 内核中，常用的调用规范主要有 C 和 Fastcall，除此之外 Pascal 也是一种常见的调用规范。它们的特点如下：

1. C

调用者的参数从右到左入栈，最后由调用者负责清除堆栈中的参数，返回值通过 EAX 寄存器返回，如果返回值超过 32 位或者是浮点数，则通过 EDX:EAX 返回。

2. fastcall

通过寄存器传递参数，这也是它被命名为 fastcall 的原因，因为访问寄存器的开销小于访问内存，但是由于 x86 平台的寄存器数量有限，多余的参数仍然通过堆栈传递。返回值通过 EAX 或者 EDX:EAX 传递。在 gcc 中，默认前两个参数(从左到右)通过 ECX 和 EDX 传递，其余的通过堆栈传递。但是可以通过 __attribute__((regparm(3))) 来

控制需要通过寄存器传递参数的个数，这里 `regparm(3)` 意味着前 3 个参数需要通过寄存器传递，默认为 `EAX,ECX,EDX`。

3. Pascal

参数入栈顺序为从左到右，被调用者清除堆栈中的参数，返回值通过 `EAX` 或 `EDX:EAX` 传递。

如果由被调用者清除堆栈中的参数，那么编译器会在子函数代码块的末尾插入一条汇编指令 `ret $n` 来清除堆栈中的参数，`n` 表示参数的总字节数。如果是由调用者清除堆栈中的参数，那么编译器将在父函数体内(调用子函数之后)的代码中插入 `addl $n, %esp` 来清除堆栈中的参数。由调用者负责清除堆栈中的参数有一个好处，那就是它可以支持变参函数，像 `printf`, `printk` 这一类函数是变参函数的典型代表，它们的参数的个数是在运行时决定的。而由被调用者清除堆栈中的参数则不能支持变参函数，因为子函数在编译期不能预知调用者会压入多少个参数，从而无法生成 `ret $n` 这条指令。但是相对而言，它的优点是可以减小代码的体积，因为如果由调用者来清除堆栈中的参数，那么在代码中每当调用一个子函数之后，都会出现一条 `addl $n, %esp` 这样的指令。

在内核启动、中断处理、系统调用、进程切换等相关模块都需要从汇编代码中调用 C 函数。因此理解函数调用规范是学习这部分代码的基本知识。因此我们以 C 调用规范为例，来介绍具体的操作过程。现在假设以下面的代码调用前面的函数 `f1()`。

代码片段 2.59 示例代码

```
ret = f1(1, 2, 3);
```

编译后的代码类似于下面这个样子：

代码片段 2.60 示例代码

```
1 pushl $3
2 pushl $2
3 pushl $1
4 call  f1
5 movl  %eax, ret
6 addl  $0xc, %esp
```

在执行 `call f1` 之后，该进程的堆栈如图2.2所示。

从图2.2中可以看出，堆栈中首先是压入的参数(从左到右入栈)，之后是函数的返回地址，然后是函数的局部变量。当函数返回后，堆栈的这一部分内容被清除(`ESP` 寄存器指向调用前的堆栈顶端)。因此如果需要在汇编代码中，调用一个 C 函数，只需要按照它的调用规范，传递参数，清除参数，以及获取返回值就可以了，将来我们在内核分析中会看到内核代码中的具体实例。

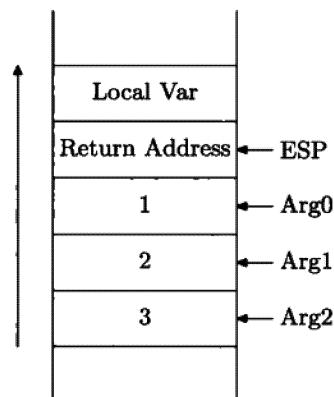


图 2.2 函数调用的堆栈变化示意图

第3章 Linux 内核 Makefile 分析

Makefile 维护着程序编译链接的相关规则，分析 Makefile 可以从总体上把握程序的结构。在 Linux 内核学习过程中，我们将看到熟悉 Linux 内核 Makefile 将为源码分析带来极大的方便。在分析过程中涉及许多 Makefile 的知识，可以参考《GNU Make 中文手册》。

3.1 Linux 内核编译概述

Linux 2.6 内核引入了 kbuild 机制，通过 make menuconfig 配置把配置信息保存在.config 文件中，当.config 文件被改变之后，再重新编译内核，Kbuild 能够保证只进行最小化的编译。Linux 内核的编译系统主要包括以下文件：

- 交互配置工具，这些文件是编译生成的可执行文件，它负责提供内核编译过程中的交互，并把用户配置交互的结果保存到.config 文件。比如用户输入 make menuconfig 命令时，主 Makefile 会编译出 mconf 的文件，mconf 会根据内核根目录下的.config 初始化配置界面，在配置结束的时候，把配置结果保存到.config 文件中。
- Kconfig 文件，位于各个子目录下。这个文件必须符合 kconfig language 规范，它定义了交互配置时的菜单等信息。例如当输入 make menuconfig 命令的时候，会运行 mconf arch/x86/Kconfig，这个 Kconfig 定义了出现在 menuconfig 中的所有菜单项。

代码片段 3.1 节自 net/Kconfig

```
.....
# 关键字 menu 定义了一个菜单项目。
menu "Networking"

# config NET 定义了一个配置项 NET，如果 Networking support
# 被选中的话，相应的 CONFIG_NET=y 就被保存到 config 文件中。
# help 后面是该菜单项对应的帮助提示。
config NET
    bool "Networking support"
    ---help---
    help message
```

```
.....
# source 把另外一个 Kconfig 文件包含进来。
source "net/Kconfig"
.....
# 另外 kconfig language 还有许多其他关键字,
# 但是都比较简单, 在此不再一一列举。
```

- **.config** 文件, 这是内核配置文件, 由配置工具生成.config 文件, 该文件由 CONFIG_XXX=Z 组成, 其中 Z 可以是 y, m, 空, 数字或者字符串。其中 Z 为 y 表示把对应的模块编译进内核; m 表示把对应模块编译成可加载模块; 空代表不编译进内核; 其他的可以传递到内核代码做参数。其中 y, m 和空是最常用的选项。
- **scripts/Makefile.***, 这些 Makefile 定义了各种编译选项和编译规则, 例如 gcc, ld 的参数等。具体的各个 Makefile 的主要作用见下面的分析。
- 顶层 Makefile, 这个文件接受 make xxx¹命令, 然后根据 xxx 做相应的操作, 最主要操作是编译出内核文件 vmlinux 和相关模块文件。
- **kbuild Makefiles**, 分布在各个子目录下的 Makefile, 它并不符合 GNU Makefile 的语法, 为了区别于通常意义上的 Makefile, 所以被称为 kbuild Makefile。它的内容形如 obj-\$(CONFIG_XXX) += Z 或者 obj-\$(CONFIG_XXX) += Z.o。这里 obj-\$(CONFIG_XXX) 被替换成 obj-y, obj-m 或者 obj-²。Kbuild 会根据 obj-y+=z.o 把对应目录下的 z.c 文件编译成 z.o, 对于 obj-y+=Z/, Kbuild 会读取 Z/ 目录下的 Makefile, 这依然是一个 Kbuild Makefile, 通过这个 Kbuild Makefile, Kbuild 就可以对子目录 Z 进行递归处理了。对于 obj-m 的情况是类似的。
- ***.cmd** 文件, 把一个.c 文件编译成.o 文件后, 如果再次编译的时候, Make 会根据.o 和.c 文件的修改时间来判断是否需要再次编译该.c 文件。当修改了.config 文件或者编译的命令行参数或者依赖关系后, 再次编译内核, make 根据修改时间判断是否需要重新编译是不够的, 因为编译参数可能改变了, 为了解决这个问题, 当编译链接每一个文件的时候, kbuild 把编译时的依赖关系和编译链接的命令保存在对应的.cmd 文件中。下一次编译的时候, 除了对比文件的修改时间外, 还要根据.cmd 文件保存的历史编译参数和依赖关系, 来判断是否需要进行重新编译。例如编译后生成的 init/.main.o.cmd 文件如下:

代码片段 3.2 节自 init/.main.o.cmd

```
# 这是编译 main.o 时的命令行参数。
cmd_init/main.o := gcc -m32 -Wp,-MD,init/.main.o.d -nostdinc
```

¹在本章中的 make xxx, 如果没有特别说明 xxx 将泛指 menuconfig, bzImage, vmlinux 等。

²当 CONFIG_XXX 为空时, 被替换成 obj-。

```

.....
-c -o init/main.o init/main.c
# 这是 main.o 的依赖文件列表,
# 如果该列表中的任何一个文件的修改时间晚于 main.o,
# 则需要重新编译 main.o.
deps_init/main.o := \
    init/main.c \
    $(wildcard include/config/x86/local/apic.h) \
    $(wildcard include/config/acpi.h)
.....
# 这里表示 main.o 依赖上面的变量$(deps_init/main.o)指定的文件。
init/main.o: $(deps_init/main.o)

$(deps_init/main.o):

```

以后随着分析的深入，我们将看到 kbuild 是如何处理上面这些变量的。

3.2 内核编译过程分析

通常编译 x86 的平台命令是 make bzImage，打开主 Makefile 却找不到 bzImage，于是在主 Makefile 中添如下代码：

代码片段 3.3 none

```

.....
bzImage:
@echo bzImage
.....

```

再次输入 make bzImage，得到这样的提示：arch/x86/Makefile_32:140: warning: overriding commands for target 'bzImage'。打开 arch/x86/Makefile_32：

代码片段 3.4 节自 arch/x86/Makefile_32

```

1 .....
2 head-y := arch/x86/kernel/head_32.o arch/x86/kernel/init_task.o
3 .....
4 all: bzImage
5
6 # KBUILD_IMAGE specify target image being built
7 KBUILD_IMAGE := $(boot)/bzImage
8 zImage zlilo zdisk: KBUILD_IMAGE := arch/x86/boot/zImage
9

```

```

10 zImage bzImage: vmlinux
11   $(Q) $(MAKE) $(build)=$(boot) $(KBUILD_IMAGE)
12   $(Q) mkdir -p $(objtree)/arch/i386/boot
13   $(Q) ln -fsn ../../x86/boot/bzImage
14     $(objtree)/arch/i386/boot/bzImage
15 .....

```

可以看到 bzImage 依赖 vmlinux，这里先放下 bzImage 后面的那一串命令，伪目标 vmlinux 位于主 Makefile 中：

代码片段 3.5 节自 Makefile

```

1 .....
2 # 定义了一些要编译的子目录。
3 init-y      := init/
4 drivers-y    := drivers/ sound/
5 net-y       := net/
6 libs-y      := lib/
7 core-y      := usr/
8 .....
9 # 把目录最后的/号去掉，把所有目录列表赋值给 vmlinux-dirs。
10 vmlinux-dirs := $(patsubst %,%,$(filter %,
11           $(init-y) $(init-m) $(core-y) $(core-m) $(drivers-y)
12           $(drivers-m) $(net-y) $(net-m) $(libs-y) $(libs-m)))
13 .....
14 vmlinux-init := $(head-y) $(init-y)
15 vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
16 vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
17 vmlinux-lds  := arch/$(SRCARCH)/kernel/vmlinux.lds
18 .....
20 # 伪目标 vmlinux 依赖的项目。
21 vmlinux: $(vmlinux-lds) $(vmlinux-init)
22   $(vmlinux-main) $(kallsyms.o) vmlinux.o FORCE
23 ifdef CONFIG_HEADERS_CHECK
24   $(Q) $(MAKE) -f $(srctree)/Makefile headers_check
25 endif
26 ifdef CONFIG_SAMPLES
27   $(Q) $(MAKE) $(build)=samples
28 endif
29   $(call vmlinux-modpost)
30   $(call if_changed_rule,vmlinux__)
31   $(Q) rm -f .old_version
32 .....

```

```

33     vmlinux.o: $(vmlinux-lds) $(vmlinux-init)
34             $(vmlinux-main) $(kallsyms.o) FORCE
35     $(call if_changed_rule, vmlinux-modpost)
36     .....
37 # 都依赖$(vmlinux-dirs)。
38 $(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds):
39         $(vmlinux-dirs) ;
40     .....
41 PHONY += $(vmlinux-dirs)
42 $(vmlinux-dirs): prepare scripts
43         $(Q)$(MAKE) $(build)=$@
44     .....
45 PHONY += FORCE
46 FORCE:

```

这里的第10行中，filter 和 patsubst 是 Makefile 的内部函数，分别用来过滤和替换。% 在这里表示匹配任意字符，filter %, \$(ARGS) 表示把\$(ARGS)中以/号结尾的项保留，其他的删除。实际目的是把不是目录的项从列表中删除。patsubst %,%,\$(ARGS) 表示把\$(ARGS)中每一项末尾的/符号删除。%/替换成%的效果是删除末尾的"/"号。

第21行中，如果 vmlinux 的依赖项目都没有被修改过的话，则后面的命令不会被执行，但是即便是没有修改过任何源文件，还需要检查链接 vmlinux 的命令行参数是否与之前的编译一致，如果不一致的话，需要重新链接。这就是 if_changed_rule, vmlinux_ 的作用，它的原理在后面很快会谈到。为了让 make 在 vmlinux 依赖的项目都没有发生变化的情况下，也执行下面的命令，于是定义了一个空的伪目标 FORCE：，该伪目标没有任何依赖也没有任何命令，但是 make 认为 vmlinux 中的 FORCE 这个依赖被成功更新，所以会执行 vmlinux 下面的命令。在 PHONY 中的项目被称为伪目标。例如：

代码片段 3.6 伪目标示例代码

```

1 PHONY += clean
2 clean:
3     -rm -f *.o

```

如果没有'PHONY'，当输入 make clean 时，如果恰好在当前目录下存在名为 clean 的文件，make 会把这个名字为 clean 的文件当成目标文件，并且在第2行中，目标文件 clean 没有任何依赖，于是没有必要执行第3行的命令。所以使用伪目标 PHONY 来告诉 make 不要到当前目录下检测 clean 文件，因为 make 的目标 clean 不是一个真正的目标文件。

第42行，这里略去对 prepare 和 scripts 这两个依赖的分析，对于\$(vmlinux-dirs)中的每一个目录执行\$(Q)\$(MAKE) \$(build)=\$@。Makefile 中\$@ 代表目标，我们将在后面讨论\$(Q)和\$(build)的作用。如果当前的目标是 init 目录，则执行的命令是 make -f scripts-/Makefile.build obj=init。可以看出 scripts 目录中 Makefile.build 文件的作用是对一个指定的

3.2 内核编译过程分析

目录进行编译。

现在来看看 scripts 目录下面的 Makefile.build 文件。

代码片段 3.7 节自 scripts/Makefile.build

```

1  # obj 是调用的时候通过 obj=xxx 传递下来的目录参数。
2  src := $(obj)
3  .....
4  # include 前面的-号表示，如果文件不存在就忽略。
5  -include include/config/auto.conf
6  include scripts/Kbuild.include
7
8  # src 是$(obj)传进来的目录名，$(srctree)是内核源码的绝对路径。
9  kbuild-dir := $(if $(filter /%, $(src)), $(src), $(srctree)/$(src))
10
11 # 把要编译的目录下面的 Makefile 包含进来,
12 # 还记得前面提到的 Kbuild Makefile 吗?
13 kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),
14                 $(kbuild-dir)/Kbuild, $(kbuild-dir)/Makefile)
15 include $(kbuild-file)
16 .....
17 include scripts/Makefile.lib
18 .....
19 ifneq (${strip $(lib-y) $(lib-m)
20         $(lib-n) $(lib-)}), )
21 lib-target := $(obj)/lib.a
22 endif
23
24 ifneq (${strip $(obj-y) $(obj-m) $(obj-n) $(obj-) $(lib-target)}), )
25 builtin-target := $(obj)/built-in.o
26 endif
27 .....
28 # 主要的依赖。
29 __build: $(if ${KBUILD_BUILTIN}, $(builtin-target)
30             $(lib-target) ${extra-y}) $(if ${KBUILD_MODULES},
31             $(obj-m) ${subdir-y}) $(always)
32             @:
33 .....
34 PHONY += ${subdir-y}
35 ${subdir-y}:
36     $(Q) $(MAKE) $(build)=@0
37 .....
38 $(builtin-target): $(obj-y) FORCE

```

```

39 $(call if_changed,link_o_target)
40 .....
41 $(obj)/%.o: $(src)/%.c FORCE
42   $(call cmd,force_checks)
43   $(call if_changed_rule,cc_o_c)

```

第9行, `$(filter %,$(src))`把`$(src)`中/开头的部分删除, 如果`$(src)`是一个绝对路径的话(以/开头)则返回空, 那么这个 if 将把`$(src)`赋值给变量 `kbuild-dir`, 否则要在`$(src)`前面加上内核源码的绝对路径赋值给 `kbuild-dir`。

第29行, 如果是编译内核, 则模块的 `obj-m` 项被排除在依赖项之外, 反之亦然。`subdir-y.m` 定义在 `scripts/Makefile.lib` 中, 由于第13行把对应子目录下的 Makefile 包含进来了。该 Makefile 又被称为 Kbuild Makefile, 其中形如 `obj-y=Z` 或者 `obj-m=Z` 的内容, 只要是以/结尾的都作为目录存放在 `subdir-y.m` 中。

第35行对每一个子目录递归调用 `scripts/Makefile.build` 文件进行子目录的编译。

第38行, `builtin-target` 依赖所有的 `obj-y` 项, 在 Kbuild Makefile 中形如 `obj-y=Z.o` 的项由第41行把对应的.c 编译成.o 文件。注意这里并不是目录下面所有的.c 文件都会被编译成.o, 而只有在 Kbuild Makefile 中用 `obj-y` 指定的那些项, 才会出现在第38行的依赖项中, 才会被编译。当一个目录下必要的文件都编译好后, 第38下面的链接命令会把这些.o 文件链接成 `built-in.o`。

第42行和43行类似, 这里以`$(call if_changed_rule,cc_o_c)`为例。`if_changed_rule` 定义在 `scripts/Kbuild.include` 中:

代码片段 3.8 节自 scripts/Kbuild.include

```

1 .....
2 # Escape single quote for use in echo statements
3 escsq = $(subst ${squote},'\${squote}',$1)
4 .....
5 # Find any prerequisites that is newer
6 # than target or that does not exist.
7 # PHONY targets skipped in both cases.
8 any-prereq = $(filter-out $(PHONY),$?)
9   $(filter-out $(PHONY) $(wildcard $^),$^)
10 .....
11 ifneq ($(KBUILD_NOCMDDEP),1)
12 # Check if both arguments has same arguments.
13 # Result is empty string if equal.
14 # User may override this check using make KBUILD_NOCMDDEP=1
15 arg-check = $(strip $(filter-out $(cmd_$(1)),
16   $(cmd_@)) $(filter-out $(cmd_@), $(cmd_$(1)))) )
17 endif

```

```

18
19 # Usage: $(call if_changed_rule,foo)
20 # Will check if $(cmd_foo) or any of the prerequisites changed,
21 # and if so will execute $(rule_foo).
22 if_changed_rule = $(if $(strip $(any-prereq)
23             $(arg-check)), @set -e; $(rule_$(1)))

```

第22行中，`strip` 的作用是去除字符串开头和结尾的空格，`$(any-prereq)`和`$(arg-check)`分别根据前一次编译保存的.cmd 文件来检测依赖项或者编译命令行参数是否发生了变化。如果有变化则返回不为空，于是执行下面的两条命令。

在第8行 `any-prereq` 中，`$?`表示比目标还要新的依赖文件列表，`$^` 表示目标的所有依赖文件。首先把伪目标从`$?`和`$^` 中删除，否则前面提到的 FORCE 总是出现在这个列表中。如果`$?`返回非空的话，意味着有比目标还新的依赖文件。`$(wildcard $^)`在当前目录下匹配所有的依赖文件，再用 `filter-out` 把匹配到的文件列表从整个`$^` 列表中删除，如果不为空的话，则说明某些依赖文件不存在，这是为某些依赖文件需要在编译时生成的情况而准备的。

在第15行的 `arg-check` 中，`$(1)`表示第一个参数，比如调用 `if_changed_rule cc_o_c`，则`$(1)`表示 `cc_o_c`。`$(cmd_cc_o_c)`是这一次编译的命令行参数。`$@` 在 Makefile 中表示目标文件。以编译 `init/main.o` 为例，在 `arg-check` 中的`$(cmd_$@)`表示`$(cmd_init/main.o)`，而在 `init/.main.o.cmd` 文件中 `cmd_init/main.o` 保存着上次编译的参数。所以如果 `arg-check` 不为空的话，说明编译参数发生了变化，那么就执行`$(rule_cc_o_c)`。主 Makefile 中通过下面的方式把.cmd 文件包含进来：

代码片段 3.9 节自 Makefile

```

1 .....
2 # read all saved command lines
3 targets := $(wildcard $(sort $(targets)))
4 cmd_files := $(wildcard *.cmd $(foreach f,$(targets),
5           $(dir $(f)).$(notdir $(f)).cmd))
6
7 ifneq ($(cmd_files),)
8   $(cmd_files): ;
9   # Do not try to update included dependency files
10  include $(cmd_files)
11 endif
12 .....

```

`rule_cc_o_c` 是在 `Makefile.build` 中定义的：

代码片段 3.10 节自 scripts/Makefile.build

```

1 .....
2 cmd_cc_o_c = $(CC) $(c_flags) -c -o $(@D)/.tmp_$(@F) $<

```

```

3 .....
4 define rule_cc_o_c
5   $(call echo-cmd,checksrc) \
6   $(cmd_checksrc) \
7   $(call echo-cmd,cc_o_c) \
8   $(cmd_cc_o_c); \
9   $(cmd_modversions) \
10  scripts/basic/fixdep $(depfile) $@ \
11      $(call make-cmd,cc_o_c)' > $(dot-target).tmp; \
12      rm -f $(depfile); \
13      mv -f $(dot-target).tmp $(dot-target).cmd
14 endef

```

其中 `checksrc` 是调用一个 `sparse` 程序对内核源程序进行语法检查, `gcc` 在编译的时候会进行语法检查, 但是 Linus Torvalds 开发的 `sparse` 可以专门对内核源程序进行更严格的检查。可以通过 `make C=2 bzImage` 来启用该功能。`echo-cmd` 是在编译过程中输出命令的, 默认输出是简略模式, 比如 `CC init/main.o`。可以通过 `make V=1 bzImage` 来开启详细输出模式, 这个时候在屏幕上将看到整个 `gcc` 编译的命令行参数。真正的命令由 `$(cmd_cc_o_c)` 来执行, 它被定义在2行中。第10行的 `fixdep` 以及后面的几条命令用来把依赖关系和命令行参数保存到对应的.cmd 文件中。类似地, 可以看到当一个目录下的.o 都编译好之后, 会把这些.o 链接到该目录下的 `built-in.o` 文件中, 最后把这些 `built-in.o` 文件链接到一起生成内核文件 `vmlinux`。

上面的分析, 虽然略过了许多的细节, 但是 `Makefile` 本身也很复杂, 涉及许多 `Makefile`, `shell` 等细节。考虑到个人的基础、思考习惯等差异的不同, 很难做到面面俱到。这里只能给出一个引子, 当读者遇到迷惑的地方, 就需要自己动手动脑了。在分析过程中, 笔者主要使用的方法有, 使用 `echo` 命令输出相关变量的值; 使用 `Makefile` 的函数 `error` 来观察一个 `ifdef` 是否成立; 使用目标重定义根据 `make` 的警告信息来定位一个目标的位置; 使用 `find`, `grep`, `make -d --debug` 等命令来收集信息等。这里再举一个例子, 如果看不懂前面提到的 `(Q)(MAKE) $(build)=$@`, 可以通过使用 `echo $(Q)` 和 `echo $(build)` 来查看它的值, 然后通过 `grep` 找到变量 `build` 定义在哪里。在主 `Makefile` 中, `Q` 是这样定义的:

代码片段 3.11 节自 `Makefile`

```

1 .....
2 ifdef V
3   ifeq ("$(origin V)", "command line")
4     KBUILD_VERBOSE = $(V)
5   endif
6 endif
7 ifndef KBUILD_VERBOSE
8   KBUILD_VERBOSE = 0

```

```
9 endif
10 .....
11 ifeq ($${KBUILD_VERBOSE},1)
12     quiet =
13     Q =
14 else
15     quiet=quiet_
16     Q = @
17 endif
18 .....
```

当输入 make V=1 xxx 的时候，会输出 make 过程中的详细信息，读者可以通过输入 make xxx 和 make V=1 xxx 看出区别。origin 是 makefile 的一个内置函数，它判定出 Makefile 中的一个变量是在哪里定义的，如果在 shell 中输入 make V=1 vmlinux，origin 将返回"command line"；如果这当前 shell 中的环境变量中定义了 V，在 shell 中输入 make vmlinux，origin 返回"environment"。这里主要是保证 V=1 来自命令行，才定义 KBUILD_VERBOSE，否则如果当前 shell 的环境变量中有一个 V=1，用户输入 make xxx 命令也会看到详细的信息。如果 KBUILD_VERBOSE=1，则 Q 设置为 @，否则设置为空。再通过查 Makefile 手册，或者自己在 Makefile 中做一个试验看看 @ls 和 ls 的区别就明白了。

我们都知道 vmlinux 是内核的执行文件，可是在 x86 系统上我们通常使用的是 bzImage，那么 bzImage 和 vmlinux 有什么关系呢？编译好的 bzImage 文件位于 arch/x86/boot/bzImage³ 目录下，通过分析 Makefile，编译输出信息及相关的.cmd 文件，我们很容易找到答案。

1. 使用 objcopy 把内核源文件根目录下的 vmlinux 转变成二进制文件 vmlinux.bin，该文件位于 arch/x86/boot/compressed/ 目录下，vmlinux 文件是 ELF 文件格式，objcopy 通过把一个 ELF 文件加载到内存中，然后再把内存中的二进制镜像赋值出来得到 vmlinux.bin。由于 ELF 文件加载过程需要根据 ELF 文件的要求做相应的处理，通过 objcopy 处理后，boot loader 就可以避免这样的处理。
2. 使用 arch/x86/boot/compressed/relocs 命令把 vmlinux 的重定位信息提取出来保存到 vmlinux.relocs 文件中。vmlinux 是一个 ELF 文件，当一个 ELF 文件被加载的时候，如果不能加载到该 ELF 文件指定的基地址上，那么该 ELF 文件中的重定位符号需要做相应的调整。通常 ELF 文件的加载器会根据文件中的重定位表处理这些信息，但是通常 boot loader 并不支持直接处理 ELF 格式，为了使内核能够在不同的地址上被加载，这里把重定位信息提取出来。读者可以在了解 ELF 文件格式的基础上分析 arch/x86/boot/compressed/relocs.c 文件。
3. 把 vmlinux.relocs 和 vmlinux.bin 合并得到 vmlinux.bin.all。可以看到为了处理重定位信息，

³2.6.24 之前的 bzImage 位于 arch/i386/boot/ 目录下。

vmlinux.relocs 增加了内核的大小。可以在配置的时候取消 CONFIG_RELOCATABLE，这样内核只能够在固定的内存地址上加载。

4. vmlinux.bin.all 被 gzip 压缩得到 vmlinux.bin.gz，将来内核在启动过程中，会对这个文件进行解压缩，这样可以减小内核文件的体积。
5. 压缩后的 vmlinux.bin.gz 文件被链接成 piggy.o 文件，这是一个 ELF 文件，通过 objdump -h piggy.o 看到链接后的 piggy.o 文件中只有一个数据段。另外也可以通过 piggy.o 的链接脚本看到这一点。
6. arch/x86/boot/compressed/ 目录下的 head_32.o, misc_32.o 和 piggy.o 被链接成 vmlinux。vmlinux 也是一个 ELF 文件。通过对 head_32.S 和 misc_32.c 的分析，可以知道这两个文件处理前面提到的重定位及内核运行时的解压缩工作。
7. 利用 objcopy 把 arch/x86/boot/compressed/ 目录下的 vmlinux 文件转换成二进制的 vmlinux 文件，保存在 arch/x86/boot/ 目录下。
8. 利用 build 工具把 arch/x86/boot/ 目录下的 setup.bin 和 vmlinux.bin 拼接成 bzImage，通过对 arch/x86/boot/tools/ 目录下的 build.c 文件的分析可以发现，build 的主要功能就是把这两个文件合并。而 setup.bin 是 arch/x86/boot/ 目录下的相关文件编译得到的。它也是利用 objdump 从 setup.elf 文件转变得到的，其主要功能是在内核启动阶段对平台相关的硬件进行初始化，并利用平台的 BIOS 获取必要的硬件信息。vmlinux 文件是平台无关的。可以看到 bzImage 由两部分组成，我们将在第4章看到这两部分各自的作用。

在以后的内核分析中，会常常用到从 Makefile 分析中获取到的有用的信息。例如：利用 ctags 或者 lxr 阅读源代码的时候，时常遇到的一个问题是在多个文件中有同名的宏定义而无法区分，可以利用 .cmd 文件中的命令来修改 gcc 参数，通过 -E 参数告诉 gcc 只进行预处理，从预处理输出文件可以分析出源文件中的宏定义出自哪个文件。还可以通过 -S 得到汇编文件。对于多个同名函数也是一样的，可以利用链接过程的 -M 参数分析出函数出自哪一个源文件。在以后的内核分析中，我们将通过实例来展示了解 Makefile 所带来的好处。

3.3 内核链接脚本分析

命令行工具 ld 把多个 .o 文件链接成一个文件，在链接的过程中除了受命令行参数的控制之外，它的最主要指挥者是链接脚本。ld 有一个默认的链接脚本，可以通过 ld --verbose 看到。在多数情况下，默认的链接脚本能够处理普通的 c 程序的链接，默认的链接脚本并不能满足内核的需要。在这种情况下，可以在链接时通过 -T 参数为 ld 指定一个链接脚本。根据内核根目录下的 vmlinux.cmd 文件的提示，可以发现 vmlinux 的链接脚本是 arch/x86/kernel/vmlinux.lds，这个文件是有 vmlinux_32.lds.S 宏处理后的到的，这里直接分析宏处理后的 vmlinux.lds 更为直接。该文件的主要内容片断如下：

代码片段 3.12 节自 arch/x86/kernel/vmlinux.lds

```

1 .....
2 # 规定了链接输出的目标文件格式。
3 OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
4 # 规定了目标文件的运行平台。
5 OUTPUT_ARCH(i386)
6 # 目标文件的入口地址。
7 ENTRY(phys_startup_32)
8 .....
9 # 目标文件代码段，数据段读，写和可执行属性。
10 PHDRS {
11   text PT_LOAD FLAGS(5); /* R_E */
12   data PT_LOAD FLAGS(7); /* RWE */
13   note PT_NOTE FLAGS(0); /* ____ */
14 }
15 SECTIONS
16 {
17   # . 表示当前地址计数。这里是 0xC0100000。
18   . = 0xC0000000 + ((0x100000 +
19                     (0x100000 - 1)) & ~(0x100000 - 1));
20   phys_startup_32 = startup_32 - 0xC0000000;
21   .text.head : AT(ADDR(.text.head) - 0xC0000000) {
22     _text = .; /* Text and read-only data */
23     *(.text.head)
24   } :text = 0x9090
25   /* read-only */
26   .text : AT(ADDR(.text) - 0xC0000000) {
27   .....

```

第7行的 ENTRY 定义了链接输出文件的入口点，从这一点我们可以发现内核的入口是 phys_startup_32。第18行符号.定义了当前的地址计数，链接器把多个文件的代码、数据段等链接成一个文件，输出文件中的符号需要有一个相对的起始地址。

第18行把起始地址设置为 0xC0100000，这是由于进入保护模式后，内核的物理起始地址是 0，它被映射的起始虚拟地址是 0xC0000000，而保护模式的代码被加载到 1M 起始的地方，所以这里链接输出的起始地址设置为 0xC0100000。

第20行，定义了一个符号变量，它的值是 phys_startup_32-0xC0000000，startup_32 是在 arch/x86/kernel/head_32.S 文件中定义的，它是代码段中的第一个符号，从内核源代码根目录下的.vmlinux.cmd 文件中可以看到 head_32.o 链接命令行中的第一个文件，所以 startup_32 就等于 0xC0100000。phys_startup_32 就是 0x100000，也就是 1M。

在第7行 ENTRY 指定的入口为 phys_startup_32 也就是 0x100000，这是由于保护模式下的内核代码被加载到物理地址为 1M 开始的地方，内核执行到这里的时候，保护模式的相关配置还没有完成，所以入口就是 1M。利用 objdump -f vmlinux 和 readelf -l vmlinux 都可以看到 vmlinux 的 vmlinux 的入口地址是 0x100000。另外如果在编译内核时选中了 Compile the kernel with debug info，在这里可以使用 addr2line -e vmlinux 0xC0100000，得知 startup_32 是在 arch/x86/kernel/head_32.S:83 定义的。把.vmlinux.cmd 文件中的 ld 命令复制出来，再加上-M 输出 linkmap 文件，然后在 linkmap 文件中搜索 startup_32，也可以得知 startup_32 位于 arch/x86/kernel/head_32.o，再看看.head_32.o.cmd 就可以知道 startup_32 位于 head_32.S 文件。

第21行，.text.head 定义了链接输出 ELF 文件的一个节(Section)，这个节的名字就是.text.head。同时符号 _text 指定了代码节的起始位置。大括号里面的 *(.text.head) 表示把 ld 命令行中指定的所有输入文件的.text.head 节都合并到输出文件的.text.head 节中。关键字 AT 规定了这个节的加载地址(LVA)，ADDR(.text.head) 表示.text.head 的起始地址，第18行到21行没有规定任何输出，所以这里还是 0xC0100000。这里 AT(0x100000) 表示这个节将被加载到 0x100000 的地方。.号指定的是内存中的虚拟地址(MVA)，如果不使用 AT 指定加载地址的，默认的加载地址就是 MVA。这里需要弄清楚 LVA 和 MVA 的区别，MVA 主要影响到指令中访问的全局符号的地址，而 LVA 指定了一个节被加载到内存的什么地方。每个节都有对齐的要求，:text = 0x9090 表示对齐空出来的部分用 0x9090 填充。

vmlinux 的链接脚本还提供了其他的有用信息，我们将在以后具体的内核分析再来继续挖掘这些信息。到此为止，我们已经找到了 vmlinux 的入口的代码位置，但是根据前面的分析我们知道 arch/x86/boot/setup.bin 被合并到 vmlinux 的最前面，通过.setup.elf.cmd 我们知道 setup.elf 的链接脚本是 setup.ld，该脚本中 ENTRY 指定的入口是 _start，使用 ld -M 参数可以知道 _start 位于 arch/x86/boot/header.S 中。内核被 boot loader 加载后，将跳转到 _start 处，进入内核继续执行。

第4章 Linux 内核启动

Linux 内核代码复杂、庞大，让人感觉难以入手，正是因为它的复杂性，任何一本教材都会把相关的内容进行分类讲解，例如中断处理，文件系统，等等。然而在阅读相关章节时，你是不是常常想弄明白某个相关的数据结构是在什么时候建立的？是在什么时候初始化的？本章从 BIOS 启动开始，对内核的启动部分进行逐步介绍，这样可以为读者建立一个初步的、整体的认识。读者在进一步的学习过程中，可以根据本章的介绍迅速找到相关内容的初始化部分。

4.1 BIOS 启动阶段

CPU 在上电初始化时，指令寄存器 CS:EIP 总是被初始化为固定的值，这就是 CPU 复位后的第一条指令的地址。断电后内存中的内容就丢失了，所以这一条指令必须保存在“非易失”的存储器中。此类存储器包括 ROM, PROM, EPROM, Nor Flash 等。早期的 BIOS 存放在只读存储器中，非常不方便修改。现在 EEPROM 和 Nor Flash 都能够通过电的方式来进行擦除和编程写入，所以通常升级 BIOS 就是利用 BIOS 芯片的电可擦除编程特性。对于 32 位地址总线的系统来说，4GB 的物理地址空间至少被划分为两个部分，一部分是内存的地址空间，另外一部分地址空间用于对 BIOS 芯片存储单元进行寻址。除此之外，随着系统外部设备的增加以及设备本身的板载存储空间的增加，16 位 8086 处理器拥有的 64KB 的 IO 地址空间早已不够(通过 in/out 汇编指令来访问的 I/O 端口。)，实际上 4GB 的物理内存地址空间还有一部分用于外部设备的板载存储空间的寻址。 $x86$ 复位后工作在实模式下，该模式下 CPU 的寻址空间为 1MB。CS:IP 的复位值是 FFFF:0000，物理为 FFFF0。主板的设计者必须保证把这个物理地址映射到 BIOS 芯片上，而不是 RAM 上。

早期的 IBM PC 地址空间映射如图4.1所示。其中高 256KB 的只读存储空间映射到 BIOS 芯片中，中间的 128KB VVDR 映射到视频卡的存储空间，屏幕上面的像素点受该区域控制，剩下的 640KB 映射到 RAM 上面。可以看出对于硬件系统的设计者来说，物理地址空间也是一种资源，而这里所说的映射就是以硬件方式对物理地址资源的分配。图4.1所示的 640KB 的 RAM 是 BIOS 设计者自由使用的区域，如何使用取决于 BIOS 软件的设计者。CPU 执行 BIOS 代码对系统进行必要的初始化，并在物理地址 0 开始的 1KB

内存中建立实模式下的中断向量表，随后的一部分内存被用来保存 BIOS 在启动阶段检测到的硬件信息。另外 BIOS 代码在执行期还需要使用随后的一部部分内存。最后 BIOS 会根据配置把引导设备的第一个扇区加载到物理地址 0x07C00 的地方，然后跳转到这里继续执行。通常这是 Boot Loader 的代码，Boot Loader 接着把内核加载到内存中。前面说过 arch/x86/boot/tools/build 工具把 setup 和 vmlinu 合成一个 bzImage。setup 是实模式的代码，vmlinu 是保护模的代码。

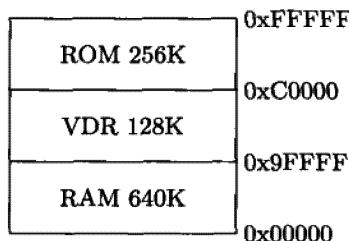


图 4.1 IBM PC 地址映射空间

如图4.2所示，BIOS 把 Boot Loader 加载到 0x07C00 的地方然后跳转到这里继续执行，之后 Boot Loader 会把实模式代码 setup 加载到 0x07C00 之上的某个地址上，其中 setup 的前 512 字节是一个引导扇区，现在这个引导扇区的作用并不是用来引导系统，而是为了兼容及传递一些参数。之后 Boot Loader 会跳转到 setup 的入口点，通过前面对链接脚本 arch/x86/boot/setup.ld 的分析(见第3.3节)，我们知道这个入口点是_start。

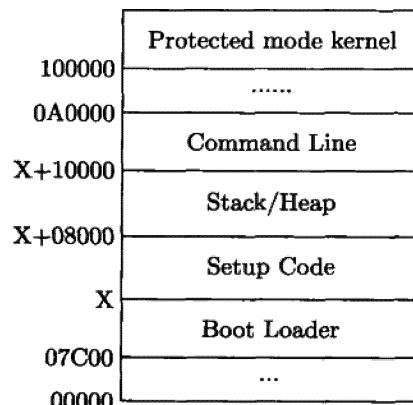


图 4.2 BIOS 启动时的内存布局

4.2 实模式 setup 阶段

setup 用于体系统结构相关的硬件初始化工作，在 arch 目录中的各个系统结构的平台相关都有类似功能的代码。在 32 位的 x86 平台中，setup 的入口点是 arch/x86/boot/header.S 中的 _start。

代码片段 4.1 节自 arch/x86/boot/header.S

```
1      .code16
2      section ".bstext", "ax"
3
4      .global bootsect_start
5 bootsect_start:
6
7      # Normalize the start address
8      ljmp    $BOOTSEG, $start2
9
10     start2:
11     movw    %cs, %ax
12     movw    %ax, %ds
13     movw    %ax, %es
14     movw    %ax, %ss
15     xorw    %sp, %sp
16     sti
17     cld
18
19     movw    $bugger_off_msg, %si
20
21 msg_loop:
22     lodsb
23     andb    %al, %al
24     jz      bs_die
25     movb    $0xe, %ah
26     movw    $7, %bx
27     int     $0x10
28     jmp     msg_loop
29     .....
30
31     .section ".bsdata", "a"
32     bugger_off_msg:
33     .ascii  "Direct booting from floppy is no longer supported\r\n"
34     .ascii  "Please use a boot loader program instead.\r\n"
35     .ascii  "\n"
36     .ascii  "Remove disk and press any key to reboot . . .\r\n"
```

```

36     .byte 0
37
38     .section ".header", "a"
39
40     .globl hdr
41 hdr:
42 setup_sects:    .byte SETUPSECTS
43 root_flags:      .word ROOT_RDONLY
44 syssize:         .long SYSSIZE
45 ram_size:        .word RAMDISK
46 vid_mode:        .word SVGA_MODE
47 root_dev:        .word ROOT_DEV
48 boot_flag:       .word 0xAA55
49
50     # offset 512, entry point
51     .globl _start
52 _start:
53     # Explicitly enter this as bytes, or the assembler
54     # tries to generate a 3-byte jump here, which causes
55     # everything else to push off to the wrong offset.
56     .byte 0xeb          # short (2-byte) jump
57     .byte start_of_setup-1f
58 l:
59     .....
60 code32_start: # here loaders can put a different
61     # start address for 32-bit code.
62 #ifndef __BIG_KERNEL__
63     .long 0x1000          # 0x1000 = default for zImage
64 #else
65     .long 0x100000         # 0x100000 = default for big kernel
66#endif

```

从第48行可以看出，第一个引导扇区以 0x55AA 结尾，现在的内核都需要由 Boot Loader 来加载。如果是由 BIOS 直接加载这个扇区，上面这段代码就会打印出错信息。setup_sects 在第一个扇区内的偏移为 0x1F1，数据结构 hdr 用来传递一些信息给之后运行的代码。细心的读者可能会有疑问，从文件开始到第48行的 0xAA55 似乎没有 512 字节？这要从链接脚本 setup.ld 说起。

代码片段 4.2 节自 arch/x86/boot/setup.ld

```

1     .....
2     # 入口点。
3 ENTRY(_start)

```

```

4
5 SECTIONS
6 {
7     # 从 0 偏移开始存放上面的 bstext 和 bsdata 段。
8     . = 0;
9     .bstext      : { *(.bstext) }
10    .bsdata       : { *(.bsdata) }
11
12    # header 段从 497(0x1F1) 开始。
13    . = 497;
14    .header      : { *(.header) }
15    .....
16 }

```

在第13行通过.**.=497** 指定了 header 这个段的起始地址是 0x1F1(497)，读者可以从前面代码片段4.1header.S 中第38行的.section ".header"开始，数一数后面的字节数，填充到这里正好 512 个字节。从 0x1F1 偏移开始的这些是传递给内核的参数，有几个是很常用的，其中 setup_sects 表示该扇区之后的实模式代码占用几个扇区。后面的 code32_start 表示 32 位保护模式代码的入口点，也就是 bzImage 的第二部分的入口地址。在图4.2中，Boot Loader 把 setup 加载到 X 处后，DS 寄存器设置为 X，再跳转到 DS:200 把控制权交到 _start，这里是硬编码一条跳转指令，它跳转到 start_of_setup 处。

代码片段 4.3 节自 arch/x86/boot/header.S

```

1 .....
2     .section ".inittext", "ax"
3 start_of_setup:
4 #ifdef SAFE_RESET_DISK_CONTROLLER
5 # Reset the disk controller.
6     movw    $0x0000, %ax          # Reset disk controller
7     movb    $0x80, %dl           # All disks
8     int     $0x13
9 #endif
10
11 # Force %es = %ds
12     movw    %ds, %ax
13     movw    %ax, %es
14     cld
15 .....
16 # Invalid %ss, make up a new stack
17     movw    $end, %dx
18     testb   $CAN_USE_HEAP, loadflags
19     jz      1f

```

```

20    movw    heap_end_ptr, %dx
21    .....
22 # We will have entered with %cs = %ds+0x20, normalize %cs so
23 # it is on par with the other segments.
24    pushw    %ds
25    pushw    $6f
26    lretw
27 6:
28    .....
29
30 # Zero the bss
31    movw    $__bss_start, %di
32    movw    $_end+3, %cx
33    xorl    %eax, %eax
34    subw    %di, %cx
35    shrw    $2, %cx
36    rep; stosl
37 # Jump to C code (should not return)
38    call    main
39    .....

```

程序的起始地址在链接脚本中被设置为 0，如果 `setup` 被加载到其他地方(起始地址不为 0 的地方)，那么指令里面访问的全局符号都会有重定位的问题。由于 Boot Loader 跳转到上面这段代码的时候，把 DS 设置为 `setup` 加载的基地址，而第17行访问 `_end` 默认是用数据段寄存器 DS 的，所以不会有重定位的问题。但是在38行转移指令用的 CS 寄存器，而符号 `main` 的地址是在链接期决定的，现在加载的基地址改变了，那肯定会出现问题了。所以事先在第24行把 CS 设置为和 DS 一样，这样就解决了重定位的问题了。代码的最后跳转到了 `arch/x86/boot/main.c` 中的 `main` 函数中继续执行。

代码片段 4.4 节自 `arch/x86/boot/main.c`

```

1 void main(void)
2 {
3     /* First, copy the boot header into the "zeropage" */
4     copy_boot_params();
5
6     /* End of heap check */
7     if (boot_params.hdr.loadflags & CAN_USE_HEAP) {
8         heap_end = (char *) (boot_params.hdr.heap_end_ptr
9                             +0x200-STACK_SIZE);
10    } else {
11        /* Boot protocol 2.00 only, no heap available */
12        puts("WARNING: Ancient bootloader,

```

```
13         some functionality may be limited!\n");
14     }
15
16     /* Make sure we have all the proper CPU support */
17     if (validate_cpu()) {
18         puts("Unable to boot - please use a
19             kernel appropriate for your CPU.\n");
20         die();
21     }
22
23     /* Tell the BIOS what CPU mode we intend to run in. */
24     set_bios_mode();
25
26     /* Detect memory layout */
27     detect_memory();
28
29     /* Set keyboard repeat rate (why?) */
30     keyboard_set_repeat();
31
32     /* Set the video mode */
33     set_video();
34
35     /* Query MCA information */
36     query_mca();
37
38     /* Voyager */
39 #ifdef CONFIG_X86_VOYAGER
40     query_voyager();
41 #endif
42
43     /* Query Intel SpeedStep (IST) information */
44     query_ist();
45
46     /* Query APM information */
47 #if defined(CONFIG_APM) || defined(CONFIG_APM_MODULE)
48     query_apm_bios();
49 #endif
50
51     /* Query EDD information */
52 #if defined(CONFIG_EDD) || defined(CONFIG_EDD_MODULE)
53     query_edd();
54 #endif
```

```

55  /* Do the last things and invoke protected mode */
56  go_to_protected_mode();
57 }

```

这个 main 函数先(调用 copy_boot_params 函数)把位于第一个扇区的参数复制到 boot_params 变量中, boot_params 位于 setup 的数据段。其后是一些设置和获取系统硬件参数有关的代码。笔者建议除非特别需要, 否则不要花费过多的精力去分析这些代码。例如 detect_memory() 通过 BIOS 调用查询系统内存信息。虽然可以通过查阅 BIOS 相关资料得知如何获取内存信息。但是这里有许多历史因素, 而且通过 BIOS 获取内存信息也只是一种方法, 在不同的平台上方法是不一样的, 重点需要分析的是这些信息收集好之后如何传递给内核, 这样在嵌入式平台开发过程中, 就可以根据自己平台的系统设计, 把相关信息传递给内核。笔者在初学阶段花费了过多的时间精力在 BIOS 调用、A20 等这些历史问题的细节上面, 这些东西既不是“真理”也不是设计上的“精华”, 等到恍然大悟时才后悔莫及, 当然这只是笔者的个人建议。

main 函数中收集到的信息全部存放在 setup 的数据段中的 boot_params 中, 结构体 boot_params 是在 include/asm-x86/bootparam.h 内定义的。

代码片段 4.5 节自 include/asm-x86/bootparam.h

```

1  /* The so-called "zeropage" */
2  struct boot_params {
3      /*0x000*/ struct screen_info screen_info;
4      /*0x040*/ struct apm_bios_info apm_bios_info;
5      /*0x054*/ __u8 _pad2[12];
6      /*0x060*/ struct ist_info ist_info;
7      /*0x070*/ __u8 _pad3[16];
8      /*0x080*/ __u8 hd0_info[16]; /* obsolete! */
9      /*0x090*/ __u8 hd1_info[16]; /* obsolete! */
10     /*0x0a0*/ struct sys_desc_table sys_desc_table;
11     /*0x0b0*/ __u8 _pad4[144];
12     /*0x140*/ struct edid_info edid_info;
13     /*0x1c0*/ struct efi_info efi_info;
14     /*0x1e0*/ __u32 alt_mem_k;
15     /*0x1e4*/ __u32 scratch; /* Scratch field! */
16     /*0x1e8*/ __u8 e820_entries;
17     /*0x1e9*/ __u8 eddbuf_entries;
18     /*0x1ea*/ __u8 edd_mbr_sig_buf_entries;
19     /*0x1eb*/ __u8 _pad6[6];
20     /*0x1f1*/ struct setup_header hdr; /* setup header */
21     __u8 _pad7[0x290-0x1f1-sizeof(struct setup_header)];
22     /*0x290*/ __u32 edd_mbr_sig_buffer[EDD_MBR_SIG_MAX];
23     /*0x2d0*/ struct e820entry e820_map[E820MAX];

```

```

24 /*0xcd0*/ __u8 _pad8[48];
25 /*0xd00*/ struct edd_info eddbuf[ EDDMAXNR ];
26 /*0xee0*/ __u8 _pad9[276];
27 } __attribute__((packed));

```

其中比较重要的有 `setup_header` 和 `e820_map`, `e820_map` 我们以后再探讨, 这里先看看 `setup_header`。它的定义和 `boot_params` 结构在同一个文件。

代码片段 4.6 节自 `include/asm-x86/bootparam.h`

```

1 struct setup_header {
2     /* 实模式代码 setup 镜像的扇区个数。 */
3     __u8 setup_sects;
4     __u16 root_flags;
5     __u32 syssize;
6     __u16 ram_size;
7     #define RAMDISK_IMAGE_START_MASK 0x07FF
8     #define RAMDISK_PROMPT_FLAG    0x8000
9     #define RAMDISK_LOAD_FLAG      0x4000
10    __u16 vid_mode;
11    __u16 root_dev;
12    __u16 boot_flag;
13    __u16 jump;
14    __u32 header;
15    __u16 version;
16    /*
17     * realmode_swtch 用来设置 Hook 函数,
18     * setup 在进入保护模式前调用该函数。
19     */
20    __u32 realmode_swtch;
21    __u16 start_sys;
22    __u16 kernel_version;
23
24    /* Boot Loader 的类别, 不同的 Boot Loader 有一个唯一的编号。 */
25    __u8 type_of_loader;
26    __u8 loadflags;
27    #define LOADED_HIGH (1<<0)
28    #define KEEP_SEGMENTS (1<<6)
29    #define CAN_USE_HEAP   (1<<7)
30    __u16 setup_move_size;
31
32    /* 保护模式代码的入口。 */
33    __u32 code32_start;
34    /*

```

```

35     * Boot Load 把 Ramdisk 镜像文件加载到内存中,
36     * 并设置该参数指向内存的起始地址。
37     */
38     __u32 ramdisk_image;
39     /* Ramdisk 镜像文件的大小。*/
40     __u32 ramdisk_size;
41     __u32 bootsect_kludge;
42     __u16 heap_end_ptr;
43     __u16 _pad1;
44
45     /* Boot Loader 存放内核参数的地址。*/
46     __u32 cmd_line_ptr;
47     __u32 initrd_addr_max;
48     __u32 kernel_alignment;
49     __u8 relocatable_kernel;
50     __u8 _pad2[3];
51     __u32 cmdline_size;
52     __u32 hardware_subarch;
53     __u64 hardware_subarch_data;
54 } __attribute__((packed));

```

现在来看看 copy_boot_params 函数。

代码片段 4.7 节自 arch/x86/boot/main.c

```

1 static void copy_boot_params(void)
2 {
3     struct old_cmdline {
4         u16 cl_magic;
5         u16 cl_offset;
6     };
7     const struct old_cmdline * const oldcmd =
8         (const struct old_cmdline *)OLD_CL_ADDRESS;
9
10    BUILD_BUG_ON(sizeof(boot_params) != 4096);
11    /* 第二个参数是在 header.S 中。*/
12    memcpy(&boot_params.hdr, &oldcmd->cl_offset, sizeof(hdr));
13
14    if (!boot_params.hdr.cmd_line_ptr &&
15        oldcmd->cl_magic == OLD_CL_MAGIC) {
16        /* Old-style command line protocol. */
17        u16 cmdline_seg;
18
19        /* Figure out if the command line falls in the region of memory

```

```

20     that an old kernel would have copied up to 0x90000... */
21     if (oldcmd->cl_offset < boot_params.hdr.setup_move_size)
22         cmdline_seg = ds();
23     else
24         cmdline_seg = 0x9000;
25
26     boot_params.hdr.cmd_line_ptr =
27     (cmdline_seg << 4) + oldcmd->cl_offset;
28 }
29
30 printf("kernel command line: %s at 0x%x\n",
31        boot_params.hdr.cmd_line_ptr - (ds()<<4),
32        boot_params.hdr.cmd_line_ptr);
33 }
```

这里主要是把 header.S 中的 hdr 结构中的参数复制到 boot_params 的 hdr 中，Boot Loader 可以向内核传递参数，它把参数放在适当的位置，并且把内核参数的指针存放到 hdr.cmd_line_ptr 中。最后一行是笔者增加的，读者可以把这一行添加到内核中，之后编译运行看看效果。需要注意的是这个指针是 32 保护模式下的线性地址，在后面我们将看到进入保护模式后，启动阶段的临时段基址为 0，实模式下的代码默认会用 DS<<4+Offset 来访问，所以这里在 printf 中传递的参数可以把这个指针减去 ds()<<4 访问到正确的命令行参数。如果读者不能理解这行代码，就需要及时补充相关基础知识了。

main 函数的最后是通过调用 go_to_protected_mode() 进入保护模式。

代码片段 4.8 节自 arch/x86/boot/pm.c

```

1 void go_to_protected_mode(void)
2 {
3     /* Hook before leaving real mode, also disables interrupts */
4     realmode_switch_hook();
5
6     /* Move the kernel/setup to their final resting places */
7     move_kernel_around();
8
9     /* Enable the A20 gate */
10    if (enable_a20()) {
11        puts("A20 gate not responding, unable to boot...\n");
12        die();
13    }
14
15    /* Reset coprocessor (IGNNE#) */
16    reset_coprocessor();
17}
```

```

18 /* Mask all interrupts in the PIC */
19 mask_all_interrupts();
20
21 /* Actual transition to protected mode... */
22 setup_idt();
23 setup_gdt();
24 protected_mode_jump(boot_params.hdr.code32_start,
25                      (u32)&boot_params + (ds() << 4));
26 }

```

`realmode_switch_hook()`在进入保护模式前调用 `hdr` 中的 `realmode_swch` Hook 函数。Boot Loader 可以利用 `realmode_swch` 抓住在实模式下执行某些代码的最后机会。`setup_gdt` 在全局描述符表中建立临时的数据段和代码供保护模式使用，其段基址为 0。`boot_params.hdr.code32_start` 保存着 32 位保护模式下的代码入口，Boot Loader 把 `bzImage` 的第二部分搬到合适的地方，再设置该参数。`boot_params` 是 `setup` 镜像的数据段中的参数。读者不妨思考一下，为什么这里要加上`(ds() << 4)`。可以参考代码片段4.7中的第30行，笔者添加的打印命令行参数的相关解释。`protected_mode_jump` 最后开启 CR0 寄存器的 PE 位进入保护模式，跳转到 `code32_start` 继续执行，读者可以做一个试验，把 `code32_start` 的值打印出来结合图4.2思考一下。

代码片段 4.9 节自 `arch/x86/boot/pmjump.S`

```

1 /*
2  * void protected_mode_jump(u32 entrypoint, u32 bootparams);
3  */
4 protected_mode_jump:
5     # edx = bootparams, eax = entrypoint.
6     # esi 指向 boot_params.
7     # Pointer to boot_params table
8     movl %edx, %esi
9     # Patch ljmp instruction
10    movl %eax, 2f
11
12    movw $__BOOT_DS, %cx
13    # Per the 32-bit boot protocol
14    xorl %ebx, %ebx
15    # Per the 32-bit boot protocol
16    xorl %ebp, %ebp
17    # Per the 32-bit boot protocol
18    xorl %edi, %edi
19
20    movl %cr0, %edx
21    # Protected mode (PE) bit

```

```

22    orb $1, %dl
23    movl %edx, %cr0
24    # 这个跳转用于清除流水线队列。
25    # Short jump to serialize on 386/486
26    jmp lf
27    1:
28
29    movw %cx, %ds
30    movw %cx, %es
31    movw %cx, %fs
32    movw %cx, %gs
33    movw %cx, %ss
34
35    # Jump to the 32-bit entrypoint
36    # ljmp opcode
37    .byte 0x66, 0xea
38    2: .long 0 # offset
39    .word __BOOT_CS # segment

```

第37行是硬编码的跳转指令，其目标跳转地址是第10行通过 eax 传递进来的入口参数改写的。调用的时候是 C 代码，而这里是汇编代码，需要确定 C 代码的参数和汇编代码中的寄存器的对应关系。这里可以通过.cmd 文件的命令行参数和 gcc 的-S 参数来分析。另外如果要用 objdump 来反汇编分析，则需要指定其指令是 8086，其命令是 objdump -d -M i8086 arch/x86/boot/setup.elf。

4.3 保护模式 startup_32

实模式的 protected_mode_jump 后，跳出了 bzImage 的第一部分，Boot Loader 默认把第二部分放在 0x100000 处，但是在允许重定位内核的情况下这个地址可以是变化的。这个入口是 startup_32，内核代码中有两个 startup_32，一个位于 arch/x86/kernel/head_32.S，另外一个位于 arch/x86/boot/compressed/head_32.S。这里跳转的入口是哪一个呢？在对 bzImage 和 vmlinux 关系的分析中，我们得知 arch/x86/kernel/head_32.S 是编译到内核 vmlinux 中，vmlinux 被 objcopy 处理再经过压缩，最后被当成数据段的内容和 arch/x86/boot/compressed/head_32.o 链接。所以先执行的是 compressed/head_32.S 中的 startup_32。它把数据段中的压缩内核解压缩后，最后还是要跳转到 arch/x86/kernel/head_32.S 中的 startup_32。

代码片段 4.10 节自 arch/x86/kernel/head_32.S

```

1   .section .text.head, "ax", @progbits
2 ENTRY(startup_32)
3   .....

```

```
4 /* Set segments to known values. */
5 # 加载全局描述符表以及初始化段选择子。
6 1: lgdt boot_gdt_descr - __PAGE_OFFSET
7 movl ${__BOOT_DS},%eax
8 movl %eax,%ds
9 movl %eax,%es
10 movl %eax,%fs
11 movl %eax,%gs
12 2:
13 /*
14 * Clear BSS first so that there are no surprises...
15 */
16 # BSS 是未初始化变量区，把这个区初始化为 0。
17 cld
18 xorl %eax,%eax
19 movl $__bss_start - __PAGE_OFFSET,%edi
20 movl $__bss_stop - __PAGE_OFFSET,%ecx
21 subl %edi,%ecx
22 shr1 $2,%ecx
23 rep ; stosl
24 /*
25 * Copy bootup parameters out of the way.
26 * Note: %esi still has the pointer to the real-mode data.
27 * With the kexec as boot loader, parameter segment might
28 * be loaded beyond kernel image and might not even be
29 * addressable by early boot page tables. (kexec on panic case).
30 * Hence copy out the parameters before initializing page tables.
31 */
32 movl $(boot_params - __PAGE_OFFSET),%edi
33 movl $(PARAM_SIZE/4),%ecx
34 cld
35 rep
36 movsl
37 movl boot_params - __PAGE_OFFSET + NEW_CL_POINTER,%esi
38 andl %esi,%esi
39 jz 1f # No command line
40 movl $(boot_command_line - __PAGE_OFFSET),%edi
41 movl $(COMMAND_LINE_SIZE/4),%ecx
42 rep
43 movsl
44 1:
```

由于在链接脚本中的起始地址是 0xC0100000，如果直接访问其中的符号，其地址必然是在 0xC0100000 之上。现在还没有启用分页机制，所以访问内核中的符号都要减去 `_PAGE_OFFSET`，也就是 0xC0100000。`boot_gdt_descr` 被定义在同一个文件中：

代码片段 4.11 节自 `arch/x86/kernel/head_32.S`

```

1 boot_gdt_descr:
2     .word _BOOT_DS+7
3     .long boot_gdt - _PAGE_OFFSET
4
5 # 数据段和代码段的基地址都是 0.
6 ENTRY(boot_gdt)
7     .fill GDT_ENTRY_BOOT_CS, 8, 0
8
9 /* kernel 4GB code at 0x00000000 */
10    .quad 0x00cf9a000000ffff
11
12 /* kernel 4GB data at 0x00000000 */
13    .quad 0x00cf92000000ffff

```

在前面代码片段4.10中的第32行开始，这里的 `boot_params` 并不是第一部分 `setup` 中的 `boot_params`，这个 `boot_params` 定义在 `arch/x86/kernel/setup_32.c` 文件中，它被链接到 `vmlinux` 的数据段中，寄存器 `esi` 指向 `setup` 中的 `boot_params` 参数。这里把 1MB 以下的 `setup` 保存的参数复制到 `vmlinux` 数据区的 `boot_params` 结构中。同样内核命令行参数也在代码片段4.10的第40行中被复制到 `init/main.c` 中的 `boot_command_line` 中。

接下来的工作就是初始化页表，为开启分页机制做好准备。

代码片段 4.12 节自 `arch/x86/kernel/head_32.S`

```

1 .....
2 page_pde_offset = (_PAGE_OFFSET >> 20);
3
4 default_entry:
5     movl $(pg0 - _PAGE_OFFSET), %edi
6     movl $(swapper_pg_dir - _PAGE_OFFSET), %edx
7     /* 0x007 = PRESENT+RW+USER */
8     movl $0x007, %eax
9 10:
10    /* Create PDE entry */
11    leal 0x007(%edi), %ecx
12    /* Store identity PDE entry */
13    movl %ecx, (%edx)
14    /* Store kernel PDE entry */
15    movl %ecx, page_pde_offset(%edx)

```

```
16    addl $4,%edx
17    movl $1024, %ecx
18 11:
19    stosl
20    addl $0x1000,%eax
21    loop 11b
22
23    /*
24     * End condition: we must map up to and including
25     * INIT_MAP_BEYOND_END
26     * bytes beyond the end of our own page tables;
27     * the +0x007 is the attribute bits
28     */
29    leal (INIT_MAP_BEYOND_END+0x007) (%edi),%ebp
30    cmpl %ebp,%eax
31    jb 10b
32    movl %edi,(init_pg_tables_end - __PAGE_OFFSET)
33
34    .....
35    /*
36     * Enable paging
37     */
38    movl $swapper_pg_dir-__PAGE_OFFSET,%eax
39    /* set the page table pointer.. */
40    movl %eax,%cr3
41    movl %cr0,%eax
42    orl $0x80000000,%eax
43    /* ..and set paging (PG) bit */
44    movl %eax,%cr0
45    /* Clear prefetch and normalize %eip */
46    ljmp $__BOOT_CS,$1f
47 1:
48    /* Set up the stack pointer */
49    lss stack_start,%esp
50
51    .....
52 .data
53 ENTRY(stack_start)
54     .long init_thread_union+THREAD_SIZE
55     .long __BOOT_DS
56     .....
```

`swapper_pg_dir` 是 `arch/x86/kernel/head_32.S` 定义的，它是内核的页目录的起始地址，而 `pg0` 是在链接脚本 `vmlinux.lds` 中定义的，它位于内核数据段结束的地方。这里把物理地址 0 开始的一段内存同时映射到虚拟地址空间 `0x00000000` 和 `0xC0000000` 开始的地方。前面提到内核使用的是从 `0xC0000000` 开始大小为 1G 的虚拟地址空间，这里为什么要把 0 开始的物理地址同时映射到 `0x00000000` 呢？这是因为当开启分页机制后，在执行以某一个符号为转移目标的指令之前¹，`EIP` 还是使用物理地址取指令。在上面的第44行执行之后，分页机制被开启，但是 `EIP` 中还是物理地址，取下面的一条指令(`EIP+4`)会去查页表，所以把 `0x00000000` 开始的内存也临时地做相应的映射就不会有问题了。

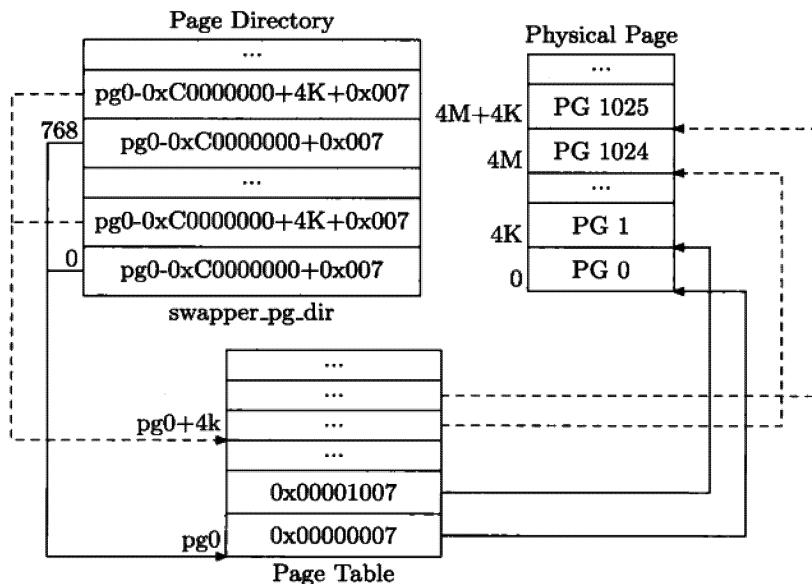


图 4.3 启动时建立的页面映射

第4行开始，用于建立如图4.3所示的页表映射关系。`pg0__PAGE_OFFSET` 得到页表的起始物理地址并存放到 `edi` 寄存器，`swapper_pg_dir__PAGE_OFFSET` 得到页目录的起始物理地址并存放到 `edx` 寄存器，`0x007` 是低 12 位标志。`leal 0x007(%edi), %ecx` 把页表的起始物理地址加上低 12 位标志的页表项存入 `ecx` 寄存器。`movl %ecx, (%edx)` 把第一个页表项存入第一个页目录中，`movl %ecx, page_pde_offset(%edx)` 把第一个页表存入第 768 个页目录中。所以虚拟地址 `0x00000000` 和 `0xC0000000` 的高 10 位在页目录中索引得到的页表地址是相同的。`mov $4, %edx`，将 `edx` 寄存器指向页目录中的下一项，为下一次做好准备。之后，`movl $1024, %ecx, stosl, addl $0x1000, %ecx` 把页表中的第 `0,1,2...1023` 项分别指向物理地址为 `0,4K,8K ... 4M` 的物理页面。一个目录项可以映射 `4M` 的内存，如果不够的话，`jb 10b` 跳回去再建立

¹ 符号的地址是在编译链接期决定的，内核中的符号起始偏移是 `0xC0100000`，所以执行到以某个符号为目标的转移指令后，`EIP` 就会被更新为正确的虚拟地址。

一个新的页表项目，再映射 4M。从这里可以看出来，访问 0 地址是不会有错误的，因为为虚拟地址 0 建立了映射，这样就检测不到 NULL 指针引用错误了。所以在后面我们可以看到，这个映射很快就会被撤销，在内核中虚拟地址空间从 0 开始的一段区域是不映射，专门用于 NULL 指针检测的。开启分页机制前，要准备好页表映射多大的空间呢？整个内核镜像 vmlinux 肯定是需要映射，划分了 1G 的虚拟地址空间，将来要把更多的物理内存映射到这个虚拟地址空间上来，所以用于映射这 1G 虚拟地址空间的页表也要提前建立好映射。否则开启分页机制后，内核要映射其他的物理页面时，需要填写页表项，所以进入保护模式前必须保证页表项本身的内存能够通过页机制访问到。所以 INIT_MAP_BEYOND_END 定义为 BOOTBITMAP_SIZE + (PAGE_TABLE_SIZE + ALLOCATOR_SLOP)*PAGE_SIZE_asm，PAGE_SIZE_asm 是页大小，这里 PAGE_TABLE_SIZE 就是用做页表的大小，定义为 1024，它能够映射 4G 的地址空间；ALLOCATOR_SLOP 定义为 4，保险起见多映射几个。进入 start_kernel 后，内核要临时设立一个启动时的内存管理机制 bootmem_allocator，BOOTBITMAP_SIZE 是 bootmem_allocator 需要使用的空间大小。读者请自己思考一下，为什么这里 INIT_MAP_BEYOND_END 中没有包括 vmlinux 的大小呢？（提示：请接合 vmlinux.lds 中对 pg0 的定义）。

第49行使用 lss 指令为内核设置新的堆栈指针，可以看到堆栈段选择子 ss 为 _BOOT_DS，和数据段在同一个段，栈顶指针 esp 为 init_thread_union + THREAD_SIZE，THREAD_SIZE 默认是 8K。考虑到堆栈是向低地址增长，因此堆栈的底部是 init_thread_union，它是在 arch/x86/kernel/init_task.c 文件中定义的：

代码片段 4.13 节自 arch/x86/kernel/init_task.c

```
union thread_union init_thread_union
__attribute__((__section__(".data.init_task")))=
{ INIT_THREAD_INFO(init_task) };
```

堆栈本身被设置在 vmlinux 镜像的.data.init_task 段中，它是在 vmlinux.lds 中定义的：

代码片段 4.14 节自 arch/x86/kernel/vmlinux.lds

```
. = ALIGN((8192)); /* init_task */
.data.init_task : AT(ADDR(.data.init_task) - 0xC0000000) {
*(.data.init_task)
}
```

以后我们将会看到内核会利用这个堆栈“手工”建立一个进程，最终在这个“手工”建立的进程环境中调用 kernel_thread() 创建 init 进程。

最终建立的虚拟地址到物理地址的映射关系如图4.4所示。当然在允许内核重定位的情况下，图4.4中的 vmlinux 不一定是从 1M 的地方开始的。在设置好页目录和页表后，把页表结束的物理地址保存到 init_pg_tables_end 变量中，这个变量是在 arch/x86/kernel/setup_32.c 中定义的，以后内核会用到这个变量，它记录了内核目前使用到的最大物理内存地址。最后通过置位 CR0 寄存器中的 PE 位进入分页模式。

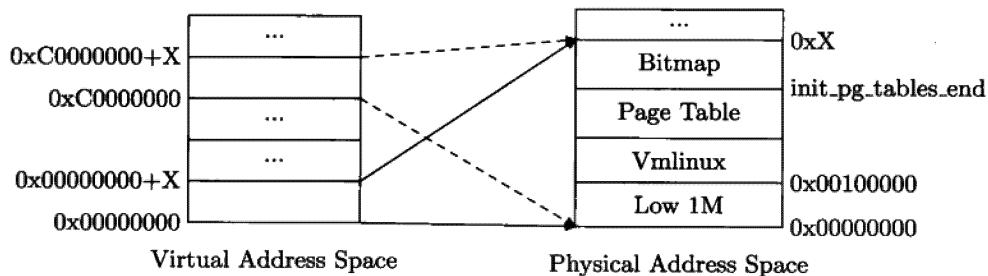


图 4.4 启动时虚拟地址和物理地址映射关系

页表初始化完成后，就需要对中断进行初始化了，首先，内核在 arch/x86/kernel/traps_32.c 文件中定义了一个中断描述符表，一共 256 项，每一项 8 个字节，其定义如下：

代码片段 4.15 节自 arch/x86/kernel/traps_32.c

```
struct desc_struct idt_table[ 256]
__attribute__((__section__(".data.idt"))) = { { 0, 0}, };
```

idt_table 位于内核数据段中的.data.idt 段中，被初始化为 0。现在，需要把对应的中断处理函数的首地址填充到 idt_table 中，在启动过程之初，所有的中断处理程序都被临时设置为 ignore_int。

代码片段 4.16 节自 arch/x86/kernel/head_32.S

```

1 setup_idt:
2   lea ignore_int,%edx
3   movl $__KERNEL_CS << 16,%eax
4   /* selector = 0x0010 = cs */
5   movw %dx,%ax
6   /* interrupt gate - dpl=0, present */
7   movw $0x8E00,%dx
8
9   lea idt_table,%edi
10  mov $256,%ecx
11  rp_sidt:
12  movl %eax, (%edi)
13  movl %edx, 4(%edi)
14  addl $8,%edi
15  dec %ecx
16  jne rp_sidt
17
18  .....
19  /* 跳转到 start_kernel 处继续执行。 */
```

```
20    jmp start_kernel
```

有了中断描述符表后，还需要把这个表的内存地址告诉 CPU，因此在 `head_32.S` 文件中定义了 `idt_descr`。

代码片段 4.17 节自 `arch/x86/kernel/head_32.S`

```
idt_descr:
    # idt contains 256 entries
    .word IDT_ENTRIES*8-1 // 段界限IDT
    .long idt_table      // 基地址IDT
```

然后通过 `lidt idt_descr` 指令，把 `idt_descr` 加载到 `IDTR` 寄存器中，现在 CPU 就可以在中断时，根据 `IDTR` 寄存器找到中断描述符表，然后利用中断号在描述符表中索引到中断处理程序的地址。最后在第20行跳转到 `start_kernel()` 函数。

4.4 内核启动 `start_kernel()`

`start_kernel` 是内核初始化的主要函数，在这个函数中负责对内核的各个模块进行初始化，包括内存，中断等，在这里我们先对 `start_kernel` 进行总体上的介绍，对某个具体模块初始化的介绍放到相关的具体章节中。

代码片段 4.18 节自 `init/main.c`

```
1 asmlinkage void __init start_kernel(void)
2 {
3     char * command_line;
4     extern struct kernel_param __start__param[], __stop__param[];
5
6     .....
7     /* 打印 Linux 内核版本信息。 */
8     printk(linux_banner);
9     /* 根据 BIOS 和 Boot Loader 传递的参数收集系统硬件信息。 */
10    setup_arch(&command_line);
11    .....
12    /* 复制参数。 */
13    setup_command_line(command_line);
14    .....
15    /* 初始化进程调度模块。 */
16    sched_init();
17    .....
18    /* 内存管理相关的初始化。 */
19    build_all_zonelists();
```

```
20    page_alloc_init();  
21  
22    /* 在终端上打印内核参数。 */  
23    printk(KERN_NOTICE "Kernel command line: %s\n",  
24            boot_command_line);  
25  
26    /* 处理内核参数。 */  
27    parse_early_param();  
28    parse_args("Booting kernel",  
29                static_command_line,  
30                __start__param,  
31                __stop__param - __start__param,  
32                &unknown_bootoption);  
33    .....  
34    /* 中断处理初始化。 */  
35    trap_init();  
36    init_IRQ();  
37    .....  
38    /* 软中断初始化。 */  
39    softirq_init();  
40    timekeeping_init();  
41    /* 系统时钟相关初始化。 */  
42    time_init();  
43    /* OProfile 系统性能分析模块的初始化。 */  
44    profile_init();  
45    .....  
46    /* 控制台初始化。 */  
47    console_init();  
48    .....  
49    /* 内存管理相关的初始化。 */  
50    mem_init();  
51    kmem_cache_init();  
52    .....  
53    /* 最后初始化其他的。 */  
54    rest_init();  
55 }
```

这里各个初始化函数我们将在后面的相关章节进行讨论。`rest_init()`主要调用`kernel_thread()`建立两个内核进程继续对内核进行初始化。

代码片段 4.19 节自 `init/main.c`

```
1 static void noinline __init_refok rest_init(void)  
2 {
```

```

3  /* 建立 kernel_init 内核线程。*/
4  kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
5  .....
6
7  /* 建立 kthreadd 内核线程。*/
8  pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
9  kthreadd_task = find_task_by_pid(pid);
10 unlock_kernel();
11 /*
12  * The boot idle thread must execute schedule()
13  * at least once to get things moving:
14 */
15 init_idle_bootup_task(current);
16 preempt_enable_no_resched();
17 schedule();
18 preempt_disable();
19
20 /* Call into cpu_idle with preempt disabled */
21 cpu_idle();
22 }

```

kernel_init 将继续初始化内核，并建立 init 进程。

代码片段 4.20 节自 init/main.c

```

1 static int __init kernel_init(void * unused)
2 {
3     .....
4     /* do_basic_setup 中最后主要调用 do_initcalls.*/
5     do_basic_setup();
6     .....
7     init_post();
8 }
9
10 static void __init do_initcalls(void)
11 {
12     initcall_t *call;
13     .....
14
15     for (call = __initcall_start; call < __initcall_end; call++) {
16         ktime_t t0, t1, delta;
17         char *msg = NULL;
18         char msgbuf[40];
19         int result;

```

```
20      /* 打印调用的函数名。*/
21      if (initcall_debug) {
22          printk("Calling initcall 0x%p", *call);
23          print_fn_descriptor_symbol(":%s()", (unsigned long)*call);
24          printk("\n");
25          t0 = ktime_get();
26      }
27
28
29      /* 调用具体的初始化函数。*/
30      result = (*call)();
31      .....
32  }
33  .....
34 }
35
36 static int __init init_post(void)
37 {
38     .....
39     /* 打开终端。*/
40     if (sys_open((const char __user *)
41                 "/dev/console", O_RDWR, 0) < 0)
42         printk(KERN_WARNING "Warning:
43             unable to open an initial console.\n");
44
45     (void) sys_dup(0);
46     (void) sys_dup(0);
47
48     /* 执行在启动阶段用 init=x 指定的内核命令。*/
49     if (ramdisk_execute_command) {
50         run_init_process(ramdisk_execute_command);
51         printk(KERN_WARNING "Failed to execute %s\n",
52               ramdisk_execute_command);
53     }
54     /*
55      * We try each of these until one succeeds.
56      *
57      * The Bourne shell can be used instead of init if we are
58      * trying to recover a really broken machine.
59      */
60     if (execute_command) {
61         run_init_process(execute_command);
```

```

62     printk(KERN_WARNING "Failed to execute %s. Attempting "
63             "defaults...\n", execute_command);
64 }
65 /*
66  * 运行 init 进程, init 是 Linux 系统上其他进程的父进程,
67  * 到这里内核启动就完成了。
68  * /etc/inittab 文件是 init 进程的配置文件,
69  * 系统可以配置 inittab 文件来启动需要的进程。
70 */
71 run_init_process ("/sbin/init");
72 run_init_process ("/etc/init");
73 run_init_process ("/bin/init");
74 run_init_process ("/bin/sh");
75
76 /* 有内核编译经验的读者对这个 panic 一定很熟悉。*/
77 panic("No init found. Try passing init= option to kernel.");
78 }

```

在 do_initcalls 函数中, for 循环中从 __initcall_start 开始, 逐一调用里面的函数指针, 直到 __initcall_end 结束, 这两个符号是在 arch/x86/kernel/vmlinux.lds 中定义的。

代码片段 4.21 节自 arch/x86/kernel/vmlinux.lds

```

.initcall.init : AT(ADDR(.initcall.init) - 0xC0000000) {
    __initcall_start = .;
    *(.initcall0.init)
    *(.initcall0s.init)
    *(.initcall1.init)
    *(.initcall1s.init)
    *(.initcall2.init)
    *(.initcall2s.init)
    *(.initcall3.init)
    *(.initcall3s.init)
    *(.initcall4.init)
    *(.initcall4s.init)
    *(.initcall5.init)
    *(.initcall5s.init)
    *(.initcallrootfs.init)
    *(.initcall6.init)
    *(.initcall6s.init)
    *(.initcall7.init)
    *(.initcall7s.init)
    __initcall_end = .;
}

```

可以看到这些函数指针是各个文件中的.initcallx.init 段合并而成的，链接的时候数字 x 越小的越靠前。这些段将被合并到 vmlinux 的.initcall.init 段中，虽然可以在启动时传入命令行内核命令行参数 initcall_debug=1 在运行时打印出调用的函数名，但是笔者认为深入了解源程序的语法、编译、链接、及目标文件格式之间的联系在对程序的领悟和分析来说是非常重要的。这里我们使用 objdump -h vmlinux 得到的输出如下：

代码片段 4.22 initcall.init 数据段

sections:						
Idx	Name	Size	VMA	LMA	File off	Align
18	.initcall.init	00000029c	c037daf4	0037daf4	0027eaf4	2**2

其中 Size 指示这个段的大小，File Off 说明它在文件中的偏移。现在可以通过 hexdump 打印出.initcall.init 段中的数据，然后根据 System.map 文件就可以查出 do_initcalls 依次调用的函数。可以通过下面的 shell 脚本来实现。

代码片段 4.23 shell 命令

```

1#!/bin/bash
2addrs='hexdump -s 0x0027eaf4 -n $((16#29c)) -e '1/4 "%x\n"' vmlinux'
3for i in $addrs
4do
5  do cat System.map | sed -n "/$i/p";
6done

```

在笔者机器上的一次输出片断如下：

```

1 c035d2f0 t init_cpufreq_transition_notifier_list
2 c035f110 t net_ns_init
3 c034a480 t cpufreq_tsc
4 c034f050 t reboot_init
5 .....
6 c0359000 t kobject_uevent_init
7 c03591e0 t pcibus_class_init
8 c0359820 t pci_driver_init
9 c0359b10 t tty_class_init
10 c035a540 t vtconsole_class_init
11 .....

```

这些函数指针是编译的时候放入内核文件的数据段的，内核代码中通过下面的宏把一个函数指针放入.initcall.init 段中。

代码片段 4.24 节自 include/linux/init.h

```

#define pure_initcall(fn) __define_initcall("0", fn, 0)

#define core_initcall(fn) __define_initcall("1", fn, 1)
#define core_initcall_sync(fn) __define_initcall("1s", fn, 1s)
#define postcore_initcall(fn) __define_initcall("2", fn, 2)
#define postcore_initcall_sync(fn) __define_initcall("2s", fn, 2s)
#define arch_initcall(fn) __define_initcall("3", fn, 3)
#define arch_initcall_sync(fn) __define_initcall("3s", fn, 3s)
#define subsys_initcall(fn) __define_initcall("4", fn, 4)
#define subsys_initcall_sync(fn) __define_initcall("4s", fn, 4s)
#define fs_initcall(fn) __define_initcall("5", fn, 5)
#define fs_initcall_sync(fn) __define_initcall("5s", fn, 5s)
#define rootfs_initcall(fn) __define_initcall("rootfs", fn, rootfs)
#define device_initcall(fn) __define_initcall("6", fn, 6)
#define device_initcall_sync(fn) __define_initcall("6s", fn, 6s)
#define late_initcall(fn) __define_initcall("7", fn, 7)
#define late_initcall_sync(fn) __define_initcall("7s", fn, 7s)

#define __initcall(fn) device_initcall(fn)
/*
 * 函数 func 最后都被展开成 __initcall_func_id。
 * 这些函数的地址最后被放到.initcall##level##.init 段中。## 表示字符连接。
 */
#define __define_initcall(level,fn,id) \
    static initcall_t __initcall_##fn##id __attribute_used__ \
    __attribute__((__section__(".initcall" level ".init")))=fn

```

如果希望 `do_initcalls()` 调用某一个初始化函数，就需要使用这些宏。比如上面的 `postcore_initcall` 函数就是使用 `postcore_initcall(pci_driver_init)` 实现的。在添加的时候需要注意的是不同 `_initcall` 宏的调用顺序。

4.5 内核启动时的参数传递

内核命令行参数在启动阶段由 `startup_32` 复制到 `init/main.c` 的 `boot_command_line` 数组中，最后由函数 `start_kernel()` 对参数进行解析处理。

代码片段 4.25 节自 `init/main.c`

```

1 asmlinkage void __init start_kernel(void)
2 {
3     char * command_line;
4     extern struct kernel_param __start__param[], __stop__param[];

```

```

5     .....
6     setup_arch(&command_line);
7     printk("code32: 0x%lx\n", boot_params.hdr.code32_start);
8
9     /* 复制参数到 saved_command_line 和 static_command_line 中。 */
10    setup_command_line(command_line);
11    .....
12    /* 打印内核参数。*/
13    printk(KERN_NOTICE "Kernel command line: %s\n",
14           boot_command_line);
15    parse_early_param();
16    parse_args("Booting kernel",
17               static_command_line,
18               __start__param,
19               __stop__param - __start__param,
20               &unknown_bootoption);
21    .....
22 }

```

上面的第15行和第16行分别用于处理内核参数和编译进内核的模块参数。下面我们将分别讨论这两种参数处理的区别。

4.5.1 内核参数处理

在启动阶段，用户可以向内核传递各种参数，那么内核是如何接收这些参数，进行相应的处理的呢？从上面的第15行和第16行可以看出内核参数首先由 `parse_early_param()` 进行处理。

代码片段 4.26 节自 `init/main.c`

```

1 void __init parse_early_param(void)
2 {
3     static __initdata int done = 0;
4     static __initdata char tmp cmdline[COMMAND_LINE_SIZE];
5
6     if (done)
7         return;
8
9     /* All fall through to do_early_param */
10    strlcpy(tmp cmdline, boot_command_line, COMMAND_LINE_SIZE);
11    parse_args("early options",
12               tmp cmdline,
13               NULL, 0,

```

```

14         do_early_param);
15     done = 1;
16 }
17
18 extern struct obs_kernel_param __setup_start[], __setup_end[];
19
20 static int __init do_early_param(char *param, char *val)
21 {
22     struct obs_kernel_param *p;
23
24     for (p = __setup_start; p < __setup_end; p++) {
25         if ((p->early && strcmp(param, p->str) == 0) ||
26             (strcmp(param, "console") == 0 &&
27              strcmp(p->str, "earlycon") == 0)) {
28             if (p->setup_func(val) != 0)
29                 printk(KERN_WARNING
30                         "Malformed early option '%s'\n",
31                         param);
32         }
33     }
34     /* We accept everything at this stage. */
35     return 0;
36 }

```

在 parse_early_param() 函数中，主要调用 parse_args() 分析传递进来的命令行参数。把参数字符串 var1=value1 分解出来，用指针 param 指向命令行参数的名字 var1，指针 val 指向对应的命令行参数的值 value1。最后使用 param 和 var 作为参数调用 do_early_param() 函数。内核中可以接收命令行参数的模块需要使用宏 __setup() 来注册相关处理函数。每一个注册的项目在内核中使用一个 obs_kernel_param 结构来表示，该结构定义如下：

代码片段 4.27 节自 include/linux/init.h

```

1 struct obs_kernel_param {
2     /* 参数的名字。*/
3     const char *str;
4     /* 参数对应的处理函数指针。*/
5     int (*setup_func)(char *);
6     /* 是否有必要在内核启动的初期处理该参数。*/
7     int early;
8 };

```

内核中每一个可以处理的参数都在 .init.setup 段中占有一项 obs_kernel_param 结构。函数 do_early_param() 就是根据参数名字遍历所有的 obs_kernel_param 结构，并使用参数

value1 调用对应的处理函数 `setup_func()`。宏 `_setup` 通过在 `.init.setup` 段中添加一个这样的结构来注册一个可处理的参数。宏的定义如下：

代码片段 4.28 节自 `include/linux/init.h`

```

1 #define __setup(str, fn)
2     __setup_param(str, fn, fn, 0)
3
4 /*
5  * 启动初期需要处理的参数。许多参数是不需要在启动初期处理的,
6  * 因为这个时候许多内核模块的初始化还没完成。
7 */
8 #define early_param(str, fn) \
9     __setup_param(str, fn, fn, 1)
10
11 #define __setup_param(str, unique_id, fn, early) \
12     static char __setup_str##unique_id[] \
13         __initdata __aligned(1) = str; \
14     static struct obs_kernel_param __setup##unique_id \
15         __attribute_used__ \
16         __attribute__((__section__(".init.setup"))) \
17         __attribute__((aligned(sizeof(long))))) \
18     = { __setup_str##unique_id, fn, early }
19
20 #define __setup_null_param(str, unique_id) \
21     __setup_param(str, unique_id, NULL, 0)

```

在第12行，用一个数组指针指向参数的名字 `str`。这个变量放在 `.init.data2` 段。这个段的内存启动结束后会被释放，但是没关系，因为它的值在随后的18行将被复制到 `obs_kernel_param` 结构的实例中。而这个实例是在第14行定义的，并且由第16行指定这个结构放到 `.init.setup` 段中。在链接脚本 `vmlinux.lds` 中指定把所有 `.o` 文件中的 `.init.setup` 段都合成到一起。`__setup_start` 和 `__setup_end` 也是在链接脚本 `vmlinux.lds` 中定义的，它们分别指向该段开始和结束的地方。

代码片段 4.29 节自 `arch/x86/kernel/vmlinux.lds`

```

1 .init.setup : AT(ADDR(.init.setup) - 0xC0000000) {
2     __setup_start = .;
3     *(.init.setup)
4     __setup_end = .;
5 }

```

² `__initdata` 展开为 `__attribute__((__section__(".init.data"))))`。

现在来看一个实例，我们知道在内核启动的时候可以通过传递 init=/bin/bash 来绕过口令登录界面，内核是如何处理这个参数的呢？

代码片段 4.30 节自 init/main.c

```
1  /* 注册 init=X 的内核命令行参数处理函数为 init_setup. */
2  __setup("init=", init_setup);
3
4  /* init_setup 函数使用全局变量 execute_command 指向该参数的值。 */
5  static int __init init_setup(char *str)
6  {
7      unsigned int i;
8
9      execute_command = str;
10     /*
11      * In case LILO is going to boot us with default command line,
12      * it prepends "auto" before the whole cmdline which makes
13      * the shell think it should execute a script with such name.
14      * So we ignore all arguments entered _before_ init=... [ MJ]
15      */
16     for (i = 1; i < MAX_INIT_ARGS; i++)
17         argv_init[i] = NULL;
18     return 1;
19 }
20
21 /*
22  * 内核启动的最后阶段在 init_post 函数中判断如果 execute_command 不为 NULL,
23  * 则使用 run_init_process 函数建立一个进程来运行由 execute_command 指定的命令。
24  */
25 static int __init __attribute__((noinline)) init_post(void)
26 {
27     .....
28     /*
29      * We try each of these until one succeeds.
30      *
31      * The Bourne shell can be used instead of init if we are
32      * trying to recover a really broken machine.
33      */
34     if (execute_command) {
35         run_init_process(execute_command);
36         printk(KERN_WARNING "Failed to execute %s. Attempting "
37               "defaults...\n", execute_command);
38 }
```

```
39     run_init_process("/sbin/init");
40     run_init_process("/etc/init");
41     run_init_process("/bin/init");
42     run_init_process("/bin/sh");
43
44     panic("No init found. Try passing init= option to kernel.");
45 }
```

4.5.2 模块参数处理

使用 insmod 或者 modprobe 安装一个模块时，可以向模块传递参数。对于直接编译进内核的模块，则不需要使用 insmod 或者 modprobe 命令来安装的。在这种情况下，参数该如何传递呢？和内核参数类似，在启动时可以通过内核命令行来向模块传递参数。因此内核启动过程中，同样要对模块的参数进行处理。这就是 parse_args("Booting kernel", static_command_line, __start_param, __stop_param - __start_param, &unknown_bootoption) 的作用。处理方式和内核参数类似，__start_param 是一个 kernel_param 类型的数组，它们位于 __param 段中。kernel_param 结构定义如下：

代码片段 4.31 节自 include/linux/moduleparam.h

```
1 struct kernel_param {
2     /* 参数名称。*/
3     const char *name;
4     /* 参数出现在/sys 目录下的文件权限。*/
5     unsigned int perm;
6     /* 设置参数的函数指针。*/
7     param_set_fn set;
8     /* 读取该参数的函数指针。*/
9     param_get_fn get;
10    /* 参数信息。*/
11    union {
12        void *arg;
13        const struct kparam_string *str;
14        const struct kparam_array *arr;
15    };
16};
```

模块可以使用宏 module_param() 添加一个 kernel_param 实例到 __start_param 数组中。例如 AMD PCnet32 网卡驱动模块 pcnet32.c 中使用 module_param(debug, int, 0) 告诉内核，该模块接收一个 int 型的 debug 参数。参数 debug 是一个 int 型的变量，参数 int 用于说明这个参数的类型是 int，内核会在 sys 文件系统中建立 /sys/modules/pcnet32/parameters/debug

文件，第三个参数用于说明这个文件的权限，0 表示不需要在 sys 文件系统中创建文件。宏 module_param 会在 __param 数据段中添加一个 kernel_param 数据结构，name 指向字符串“debug”，perm 设置成对应的权限，函数指针 set、get 分别指向内核默认的设置函数 param_set_int 和 param_get_int，而 arg 则指向模块中的这个 int 类型的 debug 变量。每一个需要接收参数的模块都需要使用 module_param 宏添加一个 kernel_param 到内核的 __param 段，__start__param 和 __stop__param 在链接脚本 vmlinux.lds 中定义的，分别指向 __param 的开头和结尾。parse_arg 就是根据传递进来命令行参数到 __start__param 数组中寻找对应的参数处理函数。module_param 在 moduleparam.h 中：

代码片段 4.32 节自 include/linux/moduleparam.h

```

1 #define module_param(name, type, perm) \
2     module_param_named(name, name, type, perm)
3
4 #define module_param_named(name, value, type, perm) \
5     param_check_##type(name, &(value)); \
6     module_param_call(name, param_set_##type, \
7                         param_get_##type, &value, perm); \
8     __MODULE_PARM_TYPE(name, #type)

```

其中 param_check_##type() 用于检测参数一致性检查，它的设计非常巧妙，它的定义如下：

代码片段 4.33 节自 include/linux/moduleparam.h

```

1 .....
2 #define param_check_byte(name, p) \
3     __param_check(name, p, unsigned char)
4 .....
5 #define param_check_int(name, p) __param_check(name, p, int)
6
7 /* 所有 param_check_xxx 都被定义为 __param_check. */
8 #define __param_check(name, p, type) \
9     static inline type *__check_##name(void) { return(p); }
10
11 /* __param_check(debug, debug, int) 展开： */
12 static inline __attribute__((always_inline))
13 int *__check_debug(void)
14 {
15     return(&(debug));
16 }
17
18 /* __param_check(debug, debug, byte) 展开： */
19 static inline __attribute__((always_inline))

```

```

20 byte * __check_debug(void)
21 {
22     return(&(debug));
23 }

```

可以看到 `param_check_##type` 是如何对参数合法性进行检查的。在 `pcnet32` 模块中，变量 `debug` 被定义为 `int` 类型，然后调用宏 `module_param(debug, int, 0)`。从第11行可以看出，`param_check_##type` 生成一个返回值为 `int` 类型的指针，而函数体中返回变量 `debug` 的地址，在编译的时候，不会有错误。而如果变量 `debug` 被定义为 `byte` 类型，`module_param(debug, byte, 0)` 生成的代码如第18行所示，这样在编译期会导致编译错误。这个宏展开得到的只是一个函数体，没有任何地方调用这个函数，由于编译器的代码优化，这个函数体不会出现在目标代码中。

参数检查通过后，`kernel_param` 结构的实例是在 `module_param_call()` 创建的，这个宏定义如下：

代码片段 4.34 节自 `__module_param_call`

```

1 /*
2  * This is the fundamental function for registering
3  * boot/module parameters.  perm sets the visibility
4  * in sysfs: 000 means it's not there, read bits mean
5  * it's readable, write bits mean it's writable.
6 */
7
8 #define __module_param_call(prefix, name, set, get, arg, perm) \
9     static int __param_perm_check_##name \
10    __attribute__((unused)) = \
11        BUILD_BUG_ON_ZERO((perm) < 0 || \
12        (perm) > 0777 || ((perm) & 2)); \
13 static const char __param_str_##name[] = prefix #name; \
14 static struct kernel_param const __param_##name \
15    __attribute_used__ \
16    __attribute__((unused, __section__("__param"), \
17        aligned(sizeof(void *)))) = { \
18        __param_str_##name, perm, set, get, { arg } } \
19
20 /*
21  * Force a compilation error if condition is true,
22  * but also produce a result (of value 0 and type size_t),
23  * so the expression can be used e.g. in a structure
24  * initializer (or where-ever else comma expressions
25  * aren't permitted).
26 */

```

```
27  
28 /* 本例中条件 (perm) < 0 || (perm) > 0777 || ((perm) & 2)被传递给 e. */  
29 #define BUILD_BUG_ON_ZERO(e) (sizeof(char[1 - 2 * !(e)])) - 1
```

我们希望对 `module_param` 的第三个参数 `perm` 进行严格的检查，例如权限参数不能为负数。遗憾的是编译器无法直接完成这样的检查。宏 `_param_perm_check_##name` 的巧妙之处就在于它确实能够在编译期检测出这样的错误。关键是 `BUILD_BUG_ON_ZERO` 中的 `sizeof`，如果条件 `e` 不为 0，则两次³取反后得到 1，而 `sizeof(char[-1])` 会导致编译器报错。可以看出这么多的工作都是为了尽量在模块的编译期检查出可能存在的错误，避免在加载期出错对内核造成不必要的破坏，从而保证内核的健壮性。

当一切都准备妥当后，第14行定义了一个 `kernel_param` 结构，并且利用 `_attribute_` 把这个结构安排在 `_param` 段中。对于常用的数据类型内核定义了其 `set` 和 `get` 函数，如 `param_set_int`, `param_get_int` 等。这些函数是由 `kernel/params.c` 中的 `STANDARD_PARAM_DEF` 宏生成的，这里把这些宏的分析留给读者。值得一提的是，`param_set_xxx`, `param_get_xxx` 是不能直接在代码中找到的，然而 `linker map` 文件能够帮助我们轻而易举地定位到这些函数出自 `kernel/params.c`，从而定位到 `STANDARD_PARAM_DEF` 宏。

³两次取反是为了保证无论 `e` 为何值，其结果非 0 即 1。

第5章 内存管理

在内核中，任何一个子模块都离不开内存，内存是其他模块的基础。实际上从内核启动，甚至从设计 CPU 及主板之初，就要考虑内存管理的相关内容。因此我们首先对内存管理的相关内容进行介绍，为下一步的学习打下基础。本章首先在内核启动的基础上，逐步介绍内核如何从开机状态建立起内存管理的相关数据结构，例如页表映射的建立过程，物理页面描述符，等等。之后在这个基础上，对常见的内存管理机制及在内核中的实现进行深入分析，包括：内存区，非一致性内存管理，伙伴算法，SLUB¹算法，等等。

5.1 内存地址空间

5.1.1 物理内存地址空间

系统启动之后，由 `detect_memory()` 通过 BIOS 系统调用获取到内存信息(见第4.2节)，这些信息存放在 `boot_params` 结构中，该结构的内容在进入保护模式后被复制到 `arch/x86/kernel/setup_32.c` 文件的 `boot_params` 中(见第4.3节)，相关结构定义如下：

代码片段 5.1 节自 `arch/x86/kernel/setup_32.c`

```
#define E820MAX 128

struct boot_params {
    ...
    /* e820entry 数组中的有效项目的个数。 */
    __u8 e820_entries;
    ...
    /* 数组中的每一项代表一个内存块。 */
    struct e820entry e820_map[E820MAX];
    ...
}

/* 内存 */
```

¹SLAB 算法的替代者。

```

#define E820_RAM      1
/* 只读存储区域，如 ROM, NOR FLASH 等。 */
#define E820_RESERVED  2
#define E820 ACPI     3
#define E820_NVS       4

/* e820entry 的定义。 */
struct e820entry {
    __u64 addr; /* start of memory segment */
    __u64 size; /* size of memory segment */
    __u32 type; /* type of memory segment */
} __attribute__((packed));

```

从这里可以看到内存被分成了许多块，变量 `e820_entries` 保存了总的块数；而 `e820entry` 结构中的 `addr` 表示一块内存的起始地址；`size` 表示这块内存的大小；`type` 则说明这块内存类型。宏 `E820_RAM` 和 `E820_RESERVED` 分别表示内存和只读存储空间。内存为什么要分成许多块呢？在第4章中我们得知，物理地址对于硬件设计者来说是来说是一种逻辑上的资源，物理内存地址空间的一部分用于 BIOS 存储空间寻址，另外的部分用于主存以及外设的板载存储空间寻址。表5.1是 Intel ICH7 的部分地址空间分配²。

表 5.1 Intel ICH7 芯片物理地址空间

Memory Range	Target	Dependency/Comments
0000 0000h - 000D FFFFh 0010 0000h - TOM (Top of Memory)	Main Memory	TOM registers in Host controller
000E 0000h - 000E FFFFh	Firmware Hub	Bit 6 in Firmware Hub Decode Enable register is set
000F 0000h - 000F FFFFh	Firmware Hub	Bit 7 in Firmware Hub Decode Enable register is set
FEC0 0000h - FEC0 0100h	I/O APIC inside Intel ICH7	
.....

从表5.1中可以看出，物理内存地址空间被分成两部分，中间空出 128KB 的地址空间分配给 Firmware，从 1MB 到 TOM 开始这部分空间又划分给内存，TOM 是北桥(Host Bridge)芯片中的一个可配置的寄存器，这个寄存器中保存着最大内存地址。这主要是为了 16 位模式低 1MB 的兼容。Firmware Hub 上面连接的是 BIOS 芯片。内核启动过程中 `setup_arch` 函数会根据 `e820` 信息打印物理内存地址信息，在笔者的 2GB 内存，Intel ICH7 芯片的电脑打印如下：

²出自《Intel I/O Controller Hub 7 Family》第 6 章 Register and Memory Mapping。

代码片段 5.2 物理内存地址空间

```
BIOS-provided physical RAM map:
BIOS-e820: 0000000000000000 - 000000000009f800 (usable)
BIOS-e820: 000000000009f800 - 00000000000a0000 (reserved)
BIOS-e820: 00000000000dc000 - 0000000000100000 (reserved)
BIOS-e820: 0000000000100000 - 000000007f686000 (usable)
BIOS-e820: 000000007f686000 - 000000007f700000 (ACPI NVS)
....
```

其中 **usable** 表示可用的内存，**reserved** 则表示只读的存储区域。

5.1.2 虚拟地址空间

同样的，虚拟地址对于操作系统设计者来说也是一种资源。32位系统的虚拟地址空间是4GB。Linux设计者把4GB中最高1GB划分给内核空间程序使用，0~3GB划分给用户空间程序使用³。每一个用户态程序可以使用独立的0~3GB地址空间，共享1GB的内核空间。

内核中1GB的映射⁴如图5.1所示：物理地址0~896MB的这一部分内存被映射到3GB~3GB+896MB的虚拟地址上。这个区域被称作固定映射区，物理地址和虚拟地址相差3GB。内核定义几个相关的宏：

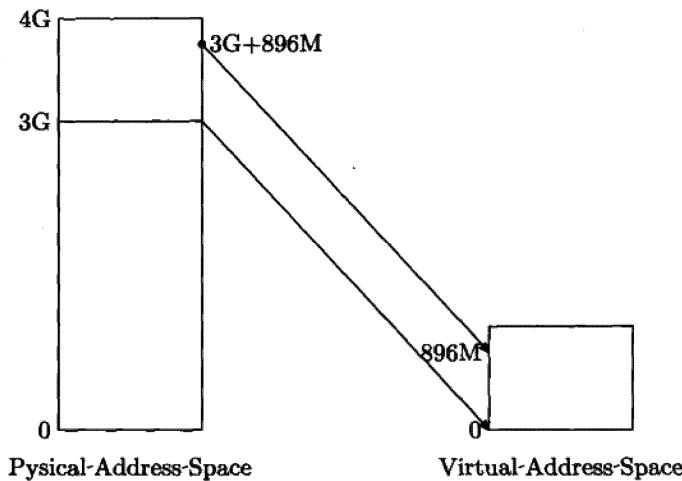


图 5.1 内核空间地址映射

³这是默认配置，在本书针对的 2.6.24 的内核中，内核空间和用户空间的虚拟地址可以在 make menuconfig 中配置。但是不管如何划分，其原理是一样的，这里我们就对默认配置 3GB/1GB 的划分方式进行讨论。

⁴把某个物理地址映射到 RAM 或者是 ROM，是通过硬件设计实现的；而某个虚拟地址映射到某个物理地址上，这是通过页表来实现的。

代码片段 5.3 节自 include/asm/page_32.h

```

1 /* CONFIG_PAGE_OFFSET 默认是 3GB. */
2 #ifdef __ASSEMBLY__
3 #define __PAGE_OFFSET CONFIG_PAGE_OFFSET
4 #else
5 #define __PAGE_OFFSET ((unsigned long)CONFIG_PAGE_OFFSET)
6 #endif
7
8 #define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
9
10 #define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
11
12 unsigned int __VMALLOC_RESERVE = 128 << 20;
13 #define VMALLOC_RESERVE ((unsigned long)__VMALLOC_RESERVE)
14 #define MAXMEM (-__PAGE_OFFSET-__VMALLOC_RESERVE)

```

其中 `__pa` 把固定映射区的一个虚拟地址转换为物理地址，转换方式很简单，就是用虚拟地址减去 `PAGE_OFFSET`(3GB)就可以了。`__va` 用于返回一个物理地址对应的虚拟地址。慢着，既然这么简单就可以计算出虚拟地址到物理地址的转换，何必在第1章中费了那么大的力气去介绍页表虚拟地址转换呢？那是 CPU 的 MMU 的计算方法，软件要为 MMU 设置页表，就必须了解它的规则。而这里是 Linux 内核的固定映射关系。比如给出一个物理地址 X，内核需要为它建立页表，有了页表 CPU 才能按照它(CPU&MMU)的规则进行访问，可是在确定虚拟地址前是无法填写页表的。因此首先通过 $3GB+X$ 来得到它的虚拟地址，页表项的内容页就确定下来了。

为什么只映射 896MB 呢？剩下 128MB 的虚拟地址空间做什么？那是保留给 `vmalloc` 使用的。我们知道，没有足够的连续物理页面的情况下，`vmalloc` 仍然可以利用页表的映射关系分配虚拟地址连续的内存。试想有 1GB 连续的物理内存被映射到起始地址为 3GB 的虚拟地址空间上，假设现在还剩下 100M 的不连续物理内存，由于采用固定映射，剩下的 100MB 虚拟地址空间也是不连续的。这样就无法分配到 100M 虚拟地址连续的内存。所以保留出来 128MB 的虚拟地址空间专门用于 `vmalloc`。`vmalloc` 也不能完全用满这 128MB 的虚拟地址空间，必须保留一些虚拟地址空间用来检测错误。例如：内核中两个模块分别使用 `vmalloc` 分配了两块内存，为了最大化地利用 128MB 的虚拟地址空间，假设这两个模块通过 `vmalloc` 分配到连续的虚拟地址，当其中一个模块越界写入的时候，系统无法捕捉到这个错误。所以它们之间必须最少空出一个页面的虚拟地址空间用于错误检测。与 NULL 指针检测类似，把这个页面的页表项的存在位设置为 0(见第1.1.1节)，当某个模块试图访问的时候会触发缺页中断，在缺页中断中可以检测出这一类的错误。

内核中对 1GB 的虚拟地址空间的分配如图5.2所示。`PAGE_OFFSET` 是内核空间起始虚拟地址，`high_memory` 变量指向最大的物理地址对应的虚拟地址，如果物理内存超过 896MB，`high_memory` 就是 $3GB+896MB$ 。也就是说内核空间能够直接访问的物理内存就

是 896MB。这显然是不可接受的。32 位系统的寻址能力是 4GB，这里仅仅是虚拟地址空间不够，如果物理内存大于 896MB 而小于 4GB，我们可以从保留的 128M 虚拟地址空间中划分出一部分用于映射 896MB 之上的这部分物理内存，这样内核就可以访问到全部的 4GB 的内存。如果物理内存大于 4GB，这就需要使用 PAE 了，它支持 36 位的地址宽度，可以最大寻址 64GB。通常内核把低 896MB 称为 Low Memory，而 896MB 之上的称为高端内存(High Memory)。用于映射高端内存的虚拟地址空间有限，如果对高端内存访问的过于频繁，需要不断地腾出这一部分虚拟地址的页表，所以高端内存虚拟地址空间又被划分为两部分，一部分为临时映射区，频繁访问的那部分为固定的映射，PKMAP_BASE 就是指向这个固定映射区起始虚拟地址。系统启动过程中，我们可以看到内核打印出来的虚拟地址空间分配的关系。下面是笔者 2GB 内存的机器上启动信息的片断：

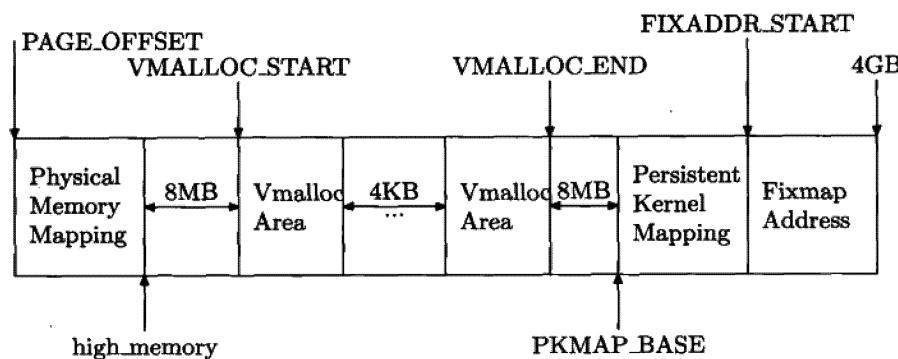


图 5.2 1GB 内核虚拟地址空间分配

代码片段 5.4 虚拟内存地址空间

```
Memory: 2058164k/2087448k available (2015k kernel code, 28160k reserved,
915k data, 364k init, 1169944k highmem)
virtual kernel memory layout:
fixmap : 0xffff4d000 - 0xfffff000 ( 712 kB)
pkmap : 0xff800000 - 0xffc00000 (4096 kB)
vmalloc : 0xf8800000 - 0xff7fe000 ( 111 MB)
lowmem : 0xc0000000 - 0xf8000000 ( 896 MB)
.init : 0xc03e3000 - 0xc043e000 ( 364 kB)
.data : 0xc02f7e86 - 0xc03dce84 ( 915 kB)
.text : 0xc0100000 - 0xc02f7e86 (2015 kB)
```

随着外部设备功能的增强，其板载存储空间也日益增加，这些芯片使用的板载存储空间也要占用一部分地址空间。大部分外部设备暂用地址是软件可配置的，但是也有一部分设备地址是固定的，例如 intel ich7 IO/APIC 使用的地址是 FEC0 0000h - FEC0 0100h(见表5.1)，这个地址位于高端内存区，但是需要经常访问，所以必须把它映射到高端内存的固

定映射区，这个区位于 PKMAP_BASE 的最后，前面说到，物理地址 0~1GB 被映射到虚拟地址 0~3GB，但是 APIC 的物理地址是一个例外，其物理地址在 0~1GB 之外，却必须要映射到虚拟地址 0~3GB 之中。所以这个区域被称为 Fixmap Address，FIXADDR_START 就是这个区的起始地址。

5.2 内存管理的基本数据结构

5.2.1 物理内存页面描述符

系统启动之后，就要根据 e820entry 数组建立基本的内存管理结构。内存的基本管理单位是页，从物理地址 0 开始，依次给每一个页面从 0 开始的一个编号，这个编号被称为 PFN，可以看出 PFN 左移 12 位就可以得到页面的起始物理地址，同样可以根据一个物理地址计算出对应的 PFN。有了 PFN 后，系统还要为每一个物理页面建立一个 page 的数据结构，它记录着一个物理页面的基本信息，结构体 page 是在 include/linux/mm_types.h 文件中定义的：

代码片段 5.5 节自 include/linux/mm_types.h

```
/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page, though if it is a pagecache page, rmap structures can tell us
 * who is mapping it.
 */
struct page {
    /* Atomic flags, some possibly updated asynchronously */
    unsigned long flags;

    /* Usage count, see below. */
    atomic_t _count;
    /*
     * Count of ptes mapped in mms, to show when page
     * is mapped & limit reverse map searches.
     */
    union {
        atomic_t _mapcount;
        /* SLUB: Nr of objects */
        unsigned int inuse;
    };
    union {
```

```

    struct {
    /*
     * Mapping-private opaque data: usually used for buffer_heads
     * if PagePrivate set; used for swp_entry_t if PageSwapCache;
     * indicates order in the buddy system if PG_buddy is set.
     */
    unsigned long private;
    /*
     * If low bit clear, points to inode address_space, or NULL.
     * If page mapped as anonymous memory, low bit is set, and
     * it points to anon_vma object: see PAGE_MAPPING_ANON below.
     */
    struct address_space *mapping;
    };

#ifndef NR_CPUS >= CONFIG_SPLIT_PTLOCK_CPUS
    spinlock_t ptl;
#endif

/* SLUB: Pointer to slab */
struct kmem_cache *slab;
/* Compound tail pages */
struct page *first_page;
};

union {
    /* Our offset within mapping. */
    pgoff_t index;
    /* SLUB: freelist req. slab lock */
    void *freelist;
};

/* Pageout list, eg. active_list protected by zone->lru_lock !*/
struct list_head lru;

#if defined(WANT_PAGE_VIRTUAL)
    /* Kernel virtual address (NULL if not kmapped, ie. highmem) */
    void *virtual;
#endif /* WANT_PAGE_VIRTUAL */
};

```

内核启动后为每一个物理页面建立一个这样的结构，这些结构存储在什么地方呢？前面说过内核进入 start_kernel 前所使用的物理内存从 1M 开始到 init_pg_tables_end 结束(见第4.3节)。进入 start_kernel 后内核首先建立一个最基本的内存管理器，叫做 bootmem_allocator。它为内存建立一个位图保存在 init_pg_tables_end 的下一个页面边界开始的地方，第 N 个 Bit 表示第 N 个物理页面的使用状态，1 表示占用，0 表示空闲。之后内

核需要内存就向 bootmem_allocator 申请。bootmem_allocator 的管理方式很简单，它根据页位图找到一个空闲的页面，分配该页面同时更新页位图的状态。用于存储 page 结构的内存就是向 bootmem_allocator 申请的。需要注意的是，系统在进入 start_kernel 前建立好的页表映射的终点是页面位图的位置(见第4.3节)，页面位图之上的内存不能直接访问，所以内核需要利用进入保护模式前保留的页表空间尽快映射更多的页面。每一个物理页面都有一个 page 结构，包括保存 page 结构本身的那些页面。之后系统会根据页面位图设置 page 的状态，最后系统回收页面位图占用的内存，bootmem_allocator 分配器被替代。mem_mep 是一个 page 结构的指针数组，利用 PFN 作为数组索引就可以得到对应页面的 page 描述符。

5.2.2 内存管理区

建立好基本的 page 结构，内核就可以对内存进行更好的管理了。然而内核并不能一律统一对待这些页面，例如对于外部设备来说，MMU 单元是不可见的，这些外部设备就只能使用物理地址进行 DMA 操作了，但是某些总线上的设备的地址总线宽度不足 32 位，因此这些设备不具备对高地址区域内存寻址的能力。所以默认把内存划分成了 3 个区，低 16M 位于 ZONE_DMA 区，16M 到 896M 位于 ZONE_NORMAL 区，896M 以上是 ZONE_HIGHMEM 区。MAX_NR_ZONES 定义为 3。其中结构 zone 定义在 include/linux/mmzone.h 文件中：

代码片段 5.6 节自 include/linux/mmzone.h

```

struct zone {
    /*
     * 当内存不足时，会调度 kswapd 脱出一些内存到磁盘的交换分区上，
     * 这样的 IO 操作会引起进程阻塞，某些进程在不可阻塞的执行路径下申请内存。
     * 例如在中断处理中，或者持有自旋锁时，这时进程应该在申请内存时指定
     * GFP_ATOMIC 标志标明不允许阻塞，如果内存不足，直接返回失败。
     * 为了减少这样的失败，每个 zone 中专门保留了一些页面用于处理 GFP_ATOMIC
     * 的情况，pages_min 就是这个保留的页面数量。
     */
    unsigned long pages_min, pages_low, pages_high;
    unsigned long    lowmem_reserve[MAX_NR_ZONES];

#ifdef CONFIG_NUMA
    int node;
    unsigned long    min_unmapped_pages;
    unsigned long    min_slab_pages;
    struct per_cpu_pageset  *pageset[NR_CPUS];
#else
    struct per_cpu_pageset  pageset[NR_CPUS];
#endif
    /* free areas of different sizes */
}

```

```
spinlock_t      lock;
#endif CONFIG_MEMORY_HOTPLUG
    seqlock_t    span_seqlock;
#endif
    struct free_area  free_area[MAX_ORDER];
#endif CONFIG_SPARSEMEM
/*
 * Flags for a pageblock_nr_pages block. See pageblock-flags.h.
 * In SPARSEMEM, this map is stored in struct mem_section
 */
unsigned long   *pageblock_flags;
#endif /* CONFIG_SPARSEMEM */

ZONE_PADDING(_pad1_)

/* Fields commonly accessed by the page reclaim scanner */
spinlock_t      lru_lock;
struct list_head active_list;
struct list_head inactive_list;
unsigned long    nr_scan_active;
unsigned long    nr_scan_inactive;
/* since last reclaim */
unsigned long    pages_scanned;
/* zone flags, see below */
unsigned long    flags;

/* Zone statistics */
atomic_long_t    vm_stat[NR_VM_ZONE_STAT_ITEMS];

int prev_priority;
ZONE_PADDING(_pad2_)
wait_queue_head_t * wait_table;
unsigned long    wait_table_hash_nr_entries;
unsigned long    wait_table_bits;

/* Discontig memory support fields. */
struct pglist_data *zone_pgdat;
/* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
unsigned long    zone_start_pfn;

/* total size, including holes */
unsigned long    spanned_pages;
```

```

/* amount of memory (excluding holes) */
unsigned long present_pages;
const char *name;
} __cacheline_internodealigned_in_smp;

```

zone 中的这些结构用于维护属于本 zone 的全部 page 结构的状态，具体成员的作用我们在后面讲述。不同的 page 属于不同的区，程序在申请内存时会指定从哪一个区中分配。

5.2.3 非一致性内存管理

非一致内存又被称为 NUMA⁵。简化的图5.3可以用来说明这样的系统结构，图中 CPU 访问各个内存模块的消耗不一致，所以称为非一致内存节点。我们希望尽可能为每个 CPU 分配到消耗比较小的一致内存。

假设 CPU1 上某个进程需要分配 5 个页面，首先检测 Memory1 能否满足，如果 Memory1 只剩下 3 个页面，则从 Global Memory 分配完整的 5 个页面。因此在 zone 之上增加了一层节点管理结构，图5.3中的每一块内存被称为一个内存节点，用结构 pd_data_t 来表示，该结构定义如下：

代码片段 5.7 节自 include/linux/mmzone.h

```

typedef struct pglist_data {
    /* 属于该节点的 zone. */
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];
    /* 该节点 zone 的个数。*/
    int nr_zones;
#ifndef CONFIG_FLAT_NODE_MEM_MAP
    /* 该节点的 page 结构数组。*/
    struct page *node_mem_map;
#endif
    /* 由启动时的 bootmem allocator 使用，该结构包含了页位图的首地址。*/
    struct bootmem_data *bdata;
#ifndef CONFIG_MEMORY_HOTPLUG
    spinlock_t node_size_lock;
#endif
    /* 该节点第一个页面的编号 PFN.*/
    unsigned long node_start_pfn;
    /* total number of physical pages */
    unsigned long node_present_pages;
    /* total size of physical page range, including holes */
    unsigned long node_spanned_pages;

```

⁵既 Non-Uniform Memory Access.

```
/* node 编号。*/
int node_id;
wait_queue_head_t kswapd_wait;
struct task_struct *kswapd;
int kswapd_max_order;
} pg_data_t;
```

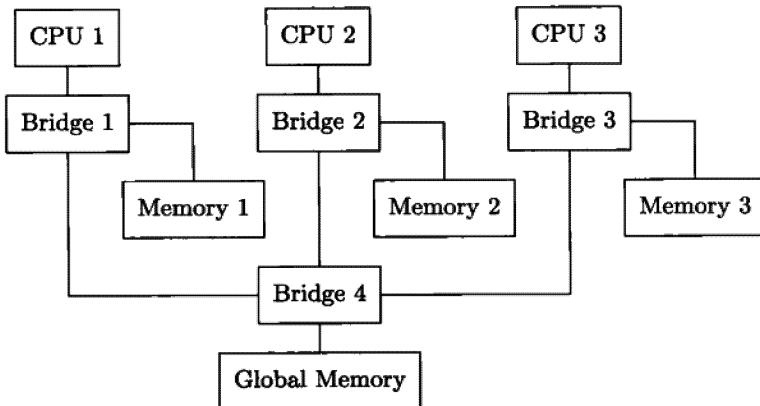


图 5.3 非一致内存结构

内核中所有的内存节点 `pg_data_t` 存放在全局变量 `pgdat_t` 中。然而 X86 是 UMA 结构，实际上没有必要使用内存节点这一层，但是为了各种体系结构的代码一致性，x86 还是使用了内存节点，`contig_page_data` 结构是内核中的唯一一个 `pg_data_t` 结构。

通过上面的讨论，你一定被页面、区、内存节点的概念搞糊涂了，这三者的关系如图5.4所示。其中 `pg_data_t` 是 X86 中唯一的一个内存节点。在 NUMA 体系中可能有多个内存节点，每个节点的内存分为三个区，每个区中包含一定数量的物理页面。

5.3 内存管理初始化

5.3.1 bootmem allocator 的初始化

在第4章我们看到，启动过程中对内存的分配就是从物理地址 0 开始依次向上增长，没有任何的管理方式，启动之后需要尽快初始化内存管理模块。由于内存管理模块需要建立必要的管理结构，这些管理结构也需要占用内存，所以在建立管理结构的过程中需要先临时地登记内存的使用情况，当内存管理模块初始化完成后，再根据登记的信息来更新这些管理结构的相关状态。这个临时过渡的内存管理器被称为 `bootmem allocator`。它使用位图来

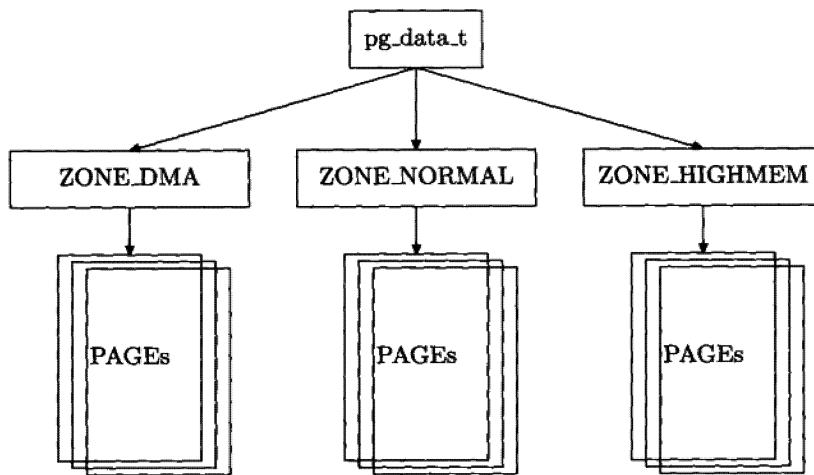


图 5.4 内存节点、区、页面的关系示意图

管理内存，位图的第 N 位表示第 N 个页面使用状态，0 代表空闲，1 代表被占用。位图占用的内存位于页表之后(见第83页图4.4)。因此进入 start_kernel 首先是设置好页位图，在内存模块初始化管理结构过程向 bootmem allocator 登记需要的内存，初始化完成后再根据 bootmem allocator 的页位图来更新这些内存管理结构的状态。bootmem allocator 的初始化是在 setup_memory()中进行的：

代码片段 5.8 节自 arch/x86/kernel/setup_32.c

```

1 [start_kernel() -> setup_arch() -> setup_memory()]
2
3 static unsigned long __init setup_memory(void)
4 {
5     /*
6      * partially used pages are not usable - thus
7      * we are rounding upwards:
8      */
9     min_low_pfn = PFN_UP(init_pg_tables_end);
10    find_max_pfn();
11    max_low_pfn = find_max_low_pfn();
12
13 #ifdef CONFIG_HIGHMEM
14     highstart_pfn = highend_pfn = max_pfn;
15     if (max_pfn > max_low_pfn) {
16         highstart_pfn = max_low_pfn;
17     }
18     printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
  
```

```

19     pages_to_mb(highend_pfn - highstart_pfn));
20     num_physpages = highend_pfn;
21     high_memory = (void *) __va(highstart_pfn * PAGE_SIZE - 1) + 1;
22 #else
23     num_physpages = max_low_pfn;
24     high_memory = (void *) __va(max_low_pfn * PAGE_SIZE - 1) + 1;
25 #endif
26 #ifdef CONFIG_FLATMEM
27     max_mapnr = num_physpages;
28 #endif
29     printk(KERN_NOTICE "%ldMB LOWMEM available.\n",
30           pages_to_mb(max_low_pfn));
31     setup_bootmem_allocator();
32     return max_low_pfn;
33 }

```

根据物理地址计算 PFN 时，需要确定向上取整还是向下取整，例如页大小为 4K 的情况下，对于物理地址 6K 来说，宏 PFN_UP 得到的结果是 2，PFN_DOWN 得到的结果是 1。第9行根据 init_pg_tables_end 计算出最小的空闲内存的 PFN，结果保存在变量 min_low_pfn 中。find_max_pfn()根据 e820_map 数组的物理内存信息计算出系统中最大的 PFN，find_max_low_pfn()计算出 Low Memory 区域中的最大 PFN。

在 setup_memory() 的最后，调用了 setup_bootmem_allocator() 设置 bootmem allocator:

代码片段 5.9 节自 arch/x86/kernel/setup_32.c

```

1 [start_kernel() -> setup_arch() ->
2   setup_memory() -> setup_bootmem_allocator()]
3
4 void __init setup_bootmem_allocator(void)
5 {
6     unsigned long bootmap_size;
7     /*
8      * Initialize the boot-time allocator (with low memory only):
9      */
10    bootmap_size = init_bootmem(min_low_pfn, max_low_pfn);
11
12    register_bootmem_low_pages(max_low_pfn);
13    /*
14     * Reserve the bootmem bitmap itself as well. We do this in two
15     * steps (first step was init_bootmem()) because this catches
16     * the (very unlikely) case of us accidentally initializing the
17     * bootmem allocator with an invalid RAM area.
18     */

```

```

19 reserve_bootmem(__pa_symbol(_text), (PFN_PHYS(min_low_pfn) +
20     bootmap_size + PAGE_SIZE-1) - __pa_symbol(_text));
21 /*
22  * reserve physical page 0 - it's a special BIOS page
23  * on many boxes, enabling clean reboots, SMP operation,
24  * laptop functions.
25  */
26 reserve_bootmem(0, PAGE_SIZE);
27 /* reserve EBDA region, it's a 4K region */
28 reserve_ebda_region();
29 .....
30
31 #ifdef CONFIG_BLK_DEV_INITRD
32 if (boot_params.hdr.type_of_loader &&
33     boot_params.hdr.ramdisk_image) {
34     unsigned long ramdisk_image = boot_params.hdr.ramdisk_image;
35     unsigned long ramdisk_size = boot_params.hdr.ramdisk_size;
36     unsigned long ramdisk_end = ramdisk_image + ramdisk_size;
37     unsigned long end_of_lowmem = max_low_pfn << PAGE_SHIFT;
38
39     if (ramdisk_end <= end_of_lowmem) {
40         reserve_bootmem(ramdisk_image, ramdisk_size);
41         initrd_start = ramdisk_image + PAGE_OFFSET;
42         initrd_end = initrd_start+ramdisk_size;
43     } else {
44         printk(KERN_ERR "initrd extends beyond end of memory "
45             "(0x%08lx > 0x%08lx)\n", disabling initrd\n",
46             ramdisk_end, end_of_lowmem);
47         initrd_start = 0;
48     }
49 }
50#endif
51 .....
52 }

```

第10行调用 `init_bootmem()` 把页位图中从 0 到 `max_low_pfn` 的页面全部置为占用状态(置 1)，随后 `register_bootmem_low_pages()` 根据 `e820_entry` 数组中的信息，把类型为 `E820_RAM` 地址区间内的位图状态改为空闲态(清 0)。这样就把 BIOS 区间的只读存储区排除了(见表5.1)。最后 `reserve_bootmem()` 把页位图中内核镜像、页表、页位图本身占用的内存状态设置为占用状态。如果 boot loader 加载了 RAM disk，还要设置 RAM disk 占用的内存状态。

之后，就会调用 `init_bootmem()` 进一步对 bootmem allocator 进行初始化设置。

代码片段 5.10 节自 mm/bootmem.c

```

1 [start_kernel() -> setup_arch() -> setup_memory() ->
2   setup_bootmem_allocator() -> init_bootmem()]
3
4 unsigned long __init init_bootmem(unsigned long start,
5                                   unsigned long pages)
6 {
7     max_low_pfn = pages;
8     min_low_pfn = start;
9     return init_bootmem_core(NODE_DATA(0), start, 0, pages);
10 }
11
12 static unsigned long __init
13     init_bootmem_core(pg_data_t *pgdat,
14                         unsigned long mapstart,
15                         unsigned long start,
16                         unsigned long end)
17 {
18     bootmem_data_t *bdata = pgdat->bdata;
19     unsigned long mapsize;
20
21     bdata->node_bootmem_map = phys_to_virt(PFN_PHYS(mapstart));
22     bdata->node_boot_start = PFN_PHYS(start);
23     bdata->node_low_pfn = end;
24     link_bootmem(bdata);
25
26     /*
27      * Initially all pages are reserved - setup_arch() has to
28      * register free RAM areas explicitly.
29      */
30     mapsize = get_mapsize(bdata);
31     memset(bdata->node_bootmem_map, 0xff, mapsize);
32
33     return mapsize;
34 }
```

宏 NODE_DATA 取得对应的内存节点结构，x86 上只有一个节点，被定义为 contig_page_data，相关定义如下：

代码片段 5.11 节自 mm/page_alloc.c

```

static bootmem_data_t contig_bootmem_data;
struct pglist_data contig_page_data = {
    .bdata = &contig_bootmem_data
```

```

};

#ifndef CONFIG_NEED_MULTIPLE_NODES

extern struct pglist_data contig_page_data;

#define NODE_DATA(nid)          (&contig_page_data)
#define NODE_MEM_MAP(nid)        mem_map
#define MAX_NODES_SHIFT         1

.....
typedef struct bootmem_data {
    unsigned long node_boot_start;
    unsigned long node_low_pfn;

    /* 页位图指针。 */
    void *node_bootmem_map;
    unsigned long last_offset;
    unsigned long last_pos;
    unsigned long last_success;

    struct list_head list;
} bootmem_data_t;

```

这个函数的 start 参数是 PFN_UP(init_pg_tables_end)，这就是页面位图的起始地址，现在把位图全部置位。接下来根据 e820 数组调用 register_bootmem_low_pages() 函数，最后调用 free_bootmem() 把低区内存对应的位图设置为 0。

代码片段 5.12 节自 arch/x86/kernel/e820_32.c

```

1 [start_kernel()->setup_arch()->setup_memory()->
2   setup_bootmem_allocator()->register_bootmem_low_pages]
3
4 void __init register_bootmem_low_pages(unsigned long max_low_pfn)
5 {
6     int i;
7
8     if (efi_enabled) {
9         efi_memmap_walk(free_available_memory, NULL);
10        return;
11    }
12    for (i = 0; i < e820.nr_map; i++) {
13        unsigned long curr_pfn, last_pfn, size;
14        /* Reserve usable low memory */
15        if (e820.map[i].type != E820_RAM)
16            continue;

```

```

17  /*
18   * We are rounding up the start address of usable memory:
19   */
20   curr_pfn = PFN_UP(e820.map[i].addr);
21   if (curr_pfn >= max_low_pfn)
22       continue;
23   /*
24   * ... and at the end of the usable range downwards:
25   */
26   last_pfn = PFN_DOWN(e820.map[i].addr + e820.map[i].size);
27
28   if (last_pfn > max_low_pfn)
29       last_pfn = max_low_pfn;
30   /*
31   * .. finally, did all the rounding and playing
32   * around just make the area go away?
33   */
34   if (last_pfn <= curr_pfn)
35       continue;
36   size = last_pfn - curr_pfn;
37   free_bootmem(PFN_PHYS(curr_pfn), PFN_PHYS(size));
38 }
39 }
```

5.3.2 页表初始化

启动阶段只映射了一部分内存，内核在开启分页机制后，能够直接使用的内存十分有限，现在要为全部的固定映射区建立页表。这个工作是在 `paging_init()` 中完成的。

代码片段 5.13 节自 `arch/x86/mm/init_32.c`

```

1 [start_kernel()>setup_arch()>paging_init()]
2
3 void __init paging_init(void)
4 {
5 #ifdef CONFIG_X86_PAE
6     set_nx();
7     if (nx_enabled)
8         printk("NX (Execute Disable) protection: active\n");
9 #endif
10    pagetable_init();
11    load_cr3(swapper_pg_dir);
```

```

12 #ifdef CONFIG_X86_PAE
13 /*
14  * We will bail out later - printk doesn't work right now so
15  * the user would just see a hanging kernel.
16 */
17 if (cpu_has_pae)
18     set_in_cr4(X86_CR4_PAE);
19#endif
20 __flush_tlb_all();
21 kmap_init();
22 }

```

其中 CONFIG_X86_PAE 处理 PAE，PAE 支持 36 位的地址总线，这里我们不讨论 PAE 的情况。页表初始化工作是在 pagetable_init 函数中完成的。初始化之后，load_cr3 把页目录物理地址 swapper_pg_dir 加载到 CR3 寄存器中，在开启分页机制前，swapper_pg_dir 已经加载到 CR3 寄存器中了，为什么这里要再次加载呢？这是由于之前页表的一部分已经在 CPU 的 TLB 缓存中，而 pagetable_init 函数初始化页表完成后，需要刷新 TLB 缓存。这是 Intel 的手册建议的，另外从处理器的角度来看，修改 CR3 后刷新 TLB 是必需的。现在来看看 pagetable_init 函数：

代码片段 5.14 节自 arch/x86/mm/init_32.c

```

1 [start_kernel() -> setup_arch() -> paging_init() -> pagetable_init()]
2
3 static void __init pagetable_init (void)
4 {
5     unsigned long vaddr, end;
6     pgd_t *pgd_base = swapper_pg_dir;
7     paravirt_pagetable_setup_start(pgd_base);
8     /* Enable PSE if available */
9     if (cpu_has_pse)
10         set_in_cr4(X86_CR4_PSE);
11     /* Enable PGE if available */
12     if (cpu_has_pge) {
13         set_in_cr4(X86_CR4_PGE);
14         __PAGE_KERNEL |= __PAGE_GLOBAL;
15         __PAGE_KERNEL_EXEC |= __PAGE_GLOBAL;
16     }
17     kernel_physical_mapping_init(pgd_base);
18     remap_numa_kva();
19
20     /*
21      * Fixed mappings, only the page table structure has to be

```

```

22     * created - mappings will be set by set_fixmap():
23     */
24     vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
25     end = (FIXADDR_TOP + PMD_SIZE - 1) & PMD_MASK;
26     page_table_range_init(vaddr, end, pgd_base);
27     permanent_kmaps_init(pgd_base);
28     paravirt_pagetable_setup_done(pgd_base);
29 }

```

这里 PSE 表示 4M 的分页方式，我们略去 PSE，该函数调用 kernel_physical_mapping_init() 初始化页表。之后分别处理高端内存区的固定映射及 Fixed mapping 映射(见第5.1.2节)。

代码片段 5.15 节自 arch/x86/mm/init_32.c

```

1 [start_kernel() -> setup_arch() -> paging_init() ->
2   kernel_physical_mapping_init()]
3
4 static void __init kernel_physical_mapping_init(pgd_t *pgd_base)
5 {
6     unsigned long pfn;
7     pgd_t *pgd;
8     pmd_t *pmd;
9     pte_t *pte;
10    int pgd_idx, pmd_idx, pte_ofs;
11
12    /* 取得虚拟地址 PAGE_OFFSET 对应的页目录索引。 */
13    pgd_idx = pgd_index(PAGE_OFFSET);
14    pgd = pgd_base + pgd_idx;
15    pfn = 0;
16
17    /* PTRS_PER_PGD 表示页目录表中有多少项，这里为 1024。 */
18    for (; pgd_idx < PTRS_PER_PGD; pgd++, pgd_idx++) {
19        pmd = one_md_table_init(pgd);
20        /* 保证只映射 max_low_pfn 个页面。 */
21        if (pfn >= max_low_pfn)
22            continue;
23        /* PTRS_PER_PMD 表示页中间目录中有多少项，这里为 1。 */
24        for (pmd_idx = 0; pmd_idx < PTRS_PER_PMD &&
25             pfn < max_low_pfn; pmd++, pmd_idx++) {
26            /* 从虚拟地址 PAGE_OFFSET 开始映射。 */
27            unsigned int address = pfn * PAGE_SIZE + PAGE_OFFSET;
28
29            if (cpu_has_pse) {

```

```

30     .....
31 } else {
32     pte = one_page_table_init(pmd);
33
34     /* 填写页表项，页表项所指的物理地址从 0 开始。*/
35     for (pte_ofs = 0;
36         pte_ofs < PTRS_PER_PTE && pfn < max_low_pfn;
37         pte++, pfn++, pte_ofs++, address += PAGE_SIZE) {
38         if (is_kernel_text(address))
39             set_pte(pte, pfn_pte(pfn, PAGE_KERNEL_EXEC));
40         else
41             set_pte(pte, pfn_pte(pfn, PAGE_KERNEL));
42     }
43 }
44 }
45 }
46 }

```

这个函数的参数就是页目录地址，也就是 `swapper_pg_dir`，然后找到虚拟地址 `PAGE_OFFSET` 在页目录中的对应位置，`PAGE_OFFSET` 的高 10 位是它的索引，也就是 `0x300`。在三级分页的情况下，第19行调用 `one_md_table_init` 分配一个页面，并且会设置页目录中的相关项目指向这个新分配的页面，但是 32 位 x86 使用两级分页就够了，所以 `one_md_table_init` 返回的是 `swapper_pg_dir` 本身，之后使用 `one_page_table_init` 分配一个页面，并且把这个页面的物理地址填入页目录项中。这里的第35行填写页表项目，由于分页机制已经开启，在第4.3节中，我们说过内核进入保护模式之前，在位于内核镜像之后的内存中，映射了足够内存空间用来做页表。这里 `one_page_table_init` 是向 `bootmem_allocator` 申请内存，`bootmem_allocator` 的分配方法很简单，就是根据页位图依次从低地址向高地址分配。现在从 PFN 0-`max_low_pfn` 的映射已经完成，内核可以在分页机制模式下访问所有低区内存了。

5.3.3 内存管理结构的初始化

内核能够访问所有的低区内存了，接下来就是要初始化各个管理结构，包括 `page`, `zone` 和 `node`。当这些结构都初始化完毕后，内核要根据 `bootmem_allocator` 的页位图设置这些管理结构的状态，最后回收页位图占用的内存。

代码片段 5.16 节自 `arch/x86/kernel/setup_32.c`

```

1 [start_kernel() -> setup_arch() -> zone_sizes_init()]
2
3 void __init zone_sizes_init(void)

```

```

4  {
5      unsigned long max_zone_pfns[MAX_NR_ZONES];
6      memset(max_zone_pfns, 0, sizeof(max_zone_pfns));
7
8      max_zone_pfns[ZONE_DMA] =
9          virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHIFT;
10
11     max_zone_pfns[ZONE_NORMAL] = max_low_pfn;
12 #ifdef CONFIG_HIGHMEM
13     max_zone_pfns[ZONE_HIGHMEM] = highend_pfn;
14     add_active_range(0, 0, highend_pfn);
15 #else
16     add_active_range(0, 0, max_low_pfn);
17 #endif
18
19     free_area_init_nodes(max_zone_pfns);
20 }

```

`add_active_range` 把一个有效的 PFN 范围添加到 `early_node_map` 数组中，数组中的每一个元素代表一段有效的连续 PFN 范围，被称为一个 active range。假设 PFN 0~15 已经在 `early_node_map` 数组中，使用 `add_active_range` 添加 PFN 16~18 的页面时，它们在 `early_node_map` 中将被合并。`early_node_map` 相关定义如下：

代码片段 5.17 节自 mm/page_alloc.c

```

1 /* mm/page_alloc.c 中定义的 early_node_map 数组。*/
2 static struct node_active_region __meminitdata
3     early_node_map[MAX_ACTIVE_REGIONS];
4
5 struct node_active_region {
6     unsigned long start_pfn;
7     unsigned long end_pfn;
8     int nid;
9 };

```

这个结构在接下来的 `free_area_init_nodes` 函数中会用到：

代码片段 5.18 节自 mm/page_alloc.c

```

1 [start_kernel() -> setup_arch() -> zone_sizes_init() ->
2     free_area_init_nodes()]
3
4 void __init free_area_init_nodes(unsigned long *max_zone_pfn)
5 {
6     unsigned long nid;

```

```
7 enum zone_type i;
8
9 /* Sort early_node_map as initialisation assumes it is sorted */
10 sort_node_map();
11
12 /* Record where the zone boundaries are */
13 memset(arch_zone_lowest_possible_pfn, 0,
14         sizeof(arch_zone_lowest_possible_pfn));
15 memset(arch_zone_highest_possible_pfn, 0,
16         sizeof(arch_zone_highest_possible_pfn));
17
18 /* From active_regions中找到最小的 PFN. */
19 arch_zone_lowest_possible_pfn[0] =
20     find_min_pfn_with_active_regions();
21
22 /* max_zone_pfn 数组是在前面 zone_sizes_init 中初始化的。*/
23 arch_zone_highest_possible_pfn[0] = max_zone_pfn[0];
24
25 /* 分别计算每一个管理区的最大和最小 PFN. */
26 for (i = 1; i < MAX_NR_ZONES; i++) {
27     if (i == ZONE_MOVABLE)
28         continue;
29     arch_zone_lowest_possible_pfn[i] =
30         arch_zone_highest_possible_pfn[i-1];
31
32     arch_zone_highest_possible_pfn[i] =
33         max(max_zone_pfn[i], arch_zone_lowest_possible_pfn[i]);
34 }
35
36 arch_zone_lowest_possible_pfn[ZONE_MOVABLE] = 0;
37 arch_zone_highest_possible_pfn[ZONE_MOVABLE] = 0;
38
39 /* Find the PFNs that ZONE_MOVABLE begins at in each node */
40 memset(zone_movable_pfn, 0, sizeof(zone_movable_pfn));
41 find_zone_movable_pfns_for_nodes(zone_movable_pfn);
42
43 /* Print out the zone ranges */
44 printk("Zone PFN ranges:\n");
45 for (i = 0; i < MAX_NR_ZONES; i++) {
46     if (i == ZONE_MOVABLE)
47         continue;
48     printk("  %-8s %8lu -> %8lu\n", zone_names[i],
```

```

49         arch_zone_lowest_possible_pfn[ i ],
50         arch_zone_highest_possible_pfn[ i ]);
51     }
52
53 /* Print out the PFNs ZONE_MOVABLE begins at in each node */
54 printk("Movable zone start PFN for each node\n");
55 for (i = 0; i < MAX_NUMNODES; i++) {
56     if (zone_movable_pfn[ i ])
57         printk("  Node %d: %lu\n", i, zone_movable_pfn[ i ]);
58 }
59
60 /* Print out the early_node_map[] */
61 printk("early_node_map[%d] active PFN ranges\n",
62       nr_nodemap_entries);
63 for (i = 0; i < nr_nodemap_entries; i++)
64     printk("  %3d: %8lu -> %8lu\n",
65           early_node_map[ i ].nid,
66           early_node_map[ i ].start_pfn,
67           early_node_map[ i ].end_pfn);
68
69 /* Initialise every node */
70 setup_nr_node_ids();
71 for_each_online_node(nid) {
72     pg_data_t *pgdat = NODE_DATA(nid);
73     free_area_init_node(nid, pgdat, NULL,
74                         find_min_pfn_for_node(nid), NULL);
75     /* Any memory on that node */
76     if (pgdat->node_present_pages)
77         node_set_state(nid, N_HIGH_MEMORY);
78     check_for_regular_memory(pgdat);
79 }
80 }

```

该函数计算每一个ZONE的PFN范围，最后打印出来，其信息如下：

代码片段 5.19 不同 zone 区间的 PFN 范围

Zone PFN ranges:		
DMA	0 ->	4096
Normal	4096 ->	229376
HighMem	229376 ->	521862
early_node_map[1] active PFN ranges		
0:	0 ->	521862

每个区的最大、最小 PFN 都确定下来后，函数最重要的一步就是在第71行，根据这些 PFN 信息，为每一个节点调用 free_area_init_node，x86 系统默认只有一个节点，就是 contig_page_data，free_area_init_node 为一个指定的内存节点初始化其 pglist_data 结构，为节点中的每一个区初始化其 zone 结构，计算该节点有多少个页面，并分配足够的 page 结构，并把这些 page 结构全部初始化为保留状态。以后我们将看到，内核最后会根据页面位图来设置这些 page 结构的状态。

代码片段 5.20 节自 mm/page_alloc.c

```

1 [start_kernel()>setup_arch()>zone_sizes_init()>
2   free_area_init_nodes()>free_area_init_node()]
3
4 void __meminit free_area_init_node(int nid,
5                                   struct pglist_data *pgdat,
6                                   unsigned long *zones_size,
7                                   unsigned long node_start_pfn,
8                                   unsigned long *zholes_size)
9 {
10    pgdat->node_id = nid;
11    pgdat->node_start_pfn = node_start_pfn;
12    /*
13     * 根据 PFN 计算该节点的总页面数，并保存
14     * 到 pgdat 结构的 node_present_pages 和 node_spanned_pages 成员中。
15     */
16    calculate_node_totalpages(pgdat, zones_size, zholes_size);
17
18    /*
19     * 根据总页面数，向 bootmem allocator 申请足够的 page 结构，
20     * 并起始地址保存到 pgdat 的 node_mem_map 成员，以及全局变量 mem_map 数组中。
21     * 数组中的第 N 项就对应第 N 个页面的 page 结构。
22     */
23    alloc_node_mem_map(pgdat);
24
25    /* 初始化该节点的 ZONE 结构。*/
26    free_area_init_core(pgdat, zones_size, zholes_size);
27 }

```

free_area_init_core()函数是对节点的一个指定的区进行初始化，该函数比较长，我们在这里省去一部分。

代码片段 5.21 节自 mm/page_alloc.c

```

1 [start_kernel()>setup_arch()>zone_sizes_init()>
2   free_area_init_nodes()>free_area_init_node()>free_area_init_core()]

```

```
3
4 static void __meminit
5   free_area_init_core(struct pglist_data *pgdat,
6                       unsigned long *zones_size,
7                       unsigned long *zholes_size)
8 {
9   enum zone_type j;
10  int nid = pgdat->node_id;
11  unsigned long zone_start_pfn = pgdat->node_start_pfn;
12  int ret;
13
14  pgdat_resize_init(pgdat);
15  pgdat->nr_zones = 0;
16
17  /*
18   * 当一个可阻塞的进程在该节点上申请内存，内存不够时，
19   * 需要把进程放入等待队列，并唤醒 kswapd 进程把某些内存腾
20   * 到磁盘的交换分区上，kswapd_wait 就是在该节点上的等待队列。
21   */
22  init_waitqueue_head(&pgdat->kswapd_wait);
23  pgdat->kswapd_max_order = 0;
24
25  for (j = 0; j < MAX_NR_ZONES; j++) {
26    struct zone *zone = pgdat->node_zones + j;
27    unsigned long size, realsize, memmap_pages;
28
29    .....
30    zone_pcp_init(zone);
31    INIT_LIST_HEAD(&zone->active_list);
32    INIT_LIST_HEAD(&zone->inactive_list);
33    zone->nr_scan_active = 0;
34    zone->nr_scan_inactive = 0;
35    zap_zone_vm_stats(zone);
36    zone->flags = 0;
37    if (!size)
38      continue;
39
40    set_pageblock_order(pageblock_default_order());
41    setup_usemap(pgdat, zone, size);
42    ret = init_currently_empty_zone(zone, zone_start_pfn,
43                                   size, MEMMAP_EARLY);
44    BUG_ON(ret);
```

```

45     zone_start_pfn += size;
46 }
47 }

```

上面的 for 循环中初始化相关的 zone 之后，在第42行调用 init_currently_empty_zone 对属于该 zone 的 page 结构进行初始化，page 的状态置为保留。这个工作主要是在 memmap_init 中完成的，memmap_init 是一个宏，被定义为 memmap_init_zone。

代码片段 5.22 节自 mm/page_alloc.c

```

1 [start_kernel()>setup_arch()>zone_sizes_init()>
2   free_area_init_nodes()>free_area_init_node()>
3     free_area_init_core()>init_currently_empty_zone()>
4       memmap_init_zone()]
5
6 void __meminit memmap_init_zone(unsigned long size,
7                                 int nid,
8                                 unsigned long zone,
9                                 unsigned long start_pfn,
10                                enum memmap_context context)
11 {
12     struct page *page;
13     unsigned long end_pfn = start_pfn + size;
14     unsigned long pfn;
15
16     for (pfn = start_pfn; pfn < end_pfn; pfn++) {
17         if (context == MEMMAP_EARLY) {
18             if (!early_pfn_valid(pfn))
19                 continue;
20             if (!early_pfn_in_nid(pfn, nid))
21                 continue;
22         }
23         /* pfn_to_page 用 pfn 作为 mem_map 的索引获取对应的 page 结构。 */
24         page = pfn_to_page(pfn);
25         set_page_links(page, zone, nid, pfn);
26         init_page_count(page);
27         reset_page_mapcount(page);
28         SetPageReserved(page);
29
30         if ((pfn & (pageblock_nr_pages-1)))
31             set_pageblock_migratetype(page, MIGRATE_MOVABLE);
32
33         INIT_LIST_HEAD(&page->lru);
34 #ifdef WANT_PAGE_VIRTUAL

```

```

35     /* The shift won't overflow because ZONE_NORMAL is below 4G. */
36     if (!is_highmem_idx(zone))
37         set_page_address(page, __va(pfn << PAGE_SHIFT));
38 #endif
39 }
40 }

```

这里把每一个 page 设置为保留状态，最后在 mem_init 中根据页面位图来设置对应的 page 状态，这个工作主要是在 free_all_bootmem_core() 函数中完成的。

代码片段 5.23 节自 mm/bootmem.c

```

1 [start_kernel() -> mem_init() -> free_all_bootmem() ->
2   free_all_bootmem_core()]
3
4 static unsigned long __init
5   free_all_bootmem_core(pg_data_t *pgdat)
6 {
7   struct page *page;
8   unsigned long pfn;
9   bootmem_data_t *bdata = pgdat->bdata;
10  unsigned long i, count, total = 0;
11  unsigned long idx;
12  unsigned long *map;
13  int gofast = 0;
14
15  BUG_ON(!bdata->node_bootmem_map);
16
17  count = 0;
18  /* first extant page of the node */
19  pfn = PFN_DOWN(bdata->node_boot_start);
20  idx = bdata->node_low_pfn - pfn;
21
22  /* 页面位图起始地址。*/
23  map = bdata->node_bootmem_map;
24  /* Check physaddr is O(LOG2(BITS_PER_LONG)) page aligned */
25  if (bdata->node_boot_start == 0 ||
26      ffs(bdata->node_boot_start) - PAGE_SHIFT >
27          ffs(BITS_PER_LONG))
28      gofast = 1;
29  for (i = 0; i < idx; ) {
30      unsigned long v = ~map[i / BITS_PER_LONG];
31
32      /* v 为全 1，则无须逐一扫描每一个 BIT，一次释放 32 个页面。*/

```

```
33     if (gofast && v == ~0UL) {
34         int order;
35
36         page = pfn_to_page(pfn);
37         count += BITS_PER_LONG;
38         order = ffs(BITS_PER_LONG) - 1;
39         __free_pages_bootmem(page, order);
40         i += BITS_PER_LONG;
41         page += BITS_PER_LONG;
42     } else if (v) {
43         /* 逐一扫描每一个BIT。 */
44         unsigned long m;
45
46         page = pfn_to_page(pfn);
47         for (m = 1; m && i < idx; m<<=1, page++, i++) {
48             if (v & m) {
49                 count++;
50                 __free_pages_bootmem(page, 0);
51             }
52         }
53     } else {
54         i += BITS_PER_LONG;
55     }
56     pfn += BITS_PER_LONG;
57 }
58 total += count;
59 /*
60  * Now free the allocator bitmap itself, it's not
61  * needed anymore:
62  */
63 page = virt_to_page(bdata->node_bootmem_map);
64 count = 0;
65 idx = (get_mapsize(bdata) + PAGE_SIZE-1) >> PAGE_SHIFT;
66 for (i = 0; i < idx; i++, page++) {
67     __free_pages_bootmem(page, 0);
68     count++;
69 }
70 total += count;
71 bdata->node_bootmem_map = NULL;
72
73     return total;
74 }
```

这个函数扫描页位图的每一个 BIT，如果为 0(注意第30行的取反运算)，则调用 `_free_pages_bootmem()` 设置页面的 page 结构的状态为空闲。`mem_init()` 最后打印出内存信息如下：

代码片段 5.24 虚拟地址空间

```
Memory: 2058164k/2087448k available (2015k kernel code, 28160k reserved,
915k data, 364k init, 1169944k highmem)
virtual kernel memory layout:
  fixmap : 0xffff4d000 - 0xffffffff000  ( 712 kB)
  pkmap  : 0xff800000 - 0xfffc00000  (4096 kB)
  vmalloc : 0xf8800000 - 0xff7fe000  ( 111 MB)
  lowmem  : 0xc0000000 - 0xf8000000  ( 896 MB)
  .init   : 0xc03e3000 - 0xc043e000  ( 364 kB)
  .data   : 0xc02f7e86 - 0xc03dce84  ( 915 kB)
  .text   : 0xc0100000 - 0xc02f7e86  (2015 kB)
```

5.4 内存的分配与回收

5.4.1 伙伴算法

内存管理模块初始化完毕后，就可以实施基本的物理页面分配和回收管理，程序通过 `alloc_pages/free_pages`⁶ 系列的函数来分配/释放内存。但是，随着不断的分配和释放，系统内存碎片必然会增加，从而导致即便系统中有足够的内存，但因为地址不连续而不能满足分配请求。不是可以使用页表把不连续的物理地址映射到连续的虚拟地址空间吗？如果采用虚拟地址空间映射来解决这个问题，会导致虚拟地址空间也出现碎片，而且对于某些 DMA 操作来说，由于外部设备没有 MMU 单元，它们只能使用物理地址连续的内存块。为了避免内存碎片的产生提出了伙伴算法，它把物理内存分为 11 个组，第 0、1、… 10 组分别包含 2^0 、 2^1 、… 2^{10} 个连续物理页面的内存。假设请求分配 4 个页面，根据该算法先到第 $2(2^2=4)$ 个组中去寻找空闲块，如果该组没有空闲块就到第 $3(2^3=8)$ 个组中去寻找，假设在第 3 个组中找到空闲块，就把其中的 4 个页面分配出去，剩余的 4 个页面放到第 2 个组中。如果第三个组还是没有空闲块，就到第 $4(2^4 = 16)$ 个组中寻找，如果在该组中找到空闲块，把其中的 4 个页面分配出去，剩余的 12 个页面被分成两部分，其中的 8 个页面放到第 3 个组，另外 4 个页面放到第 2 个组… 依此类推。同理，释放时会尽量向上合并空闲块。

⁶这些函数包括：`alloc_pages()`, `alloc_page()`, `_get_free_pages()`, `_get_free_page()`, `_get_zeroed_page()`, `_get_dma_pages()` 等。

1. 伙伴算法初始化和释放

这里先讨论释放过程，在 `_free_pages_bootmem()` 函数中，通过对每一个页面进行释放，从而完成对伙伴算法的初始化工作的。释放的主要工作就是递归向上合并伙伴块。在 5.2.2 节介绍的 `zone` 结构中，有一个 `free_area[MAX_ORDER]` 的数组，其中 `MAX_ORDER` 定义为 11(0~10 组)。数组中的每一个结构代表一个组，相关结构定义如下：

代码片段 5.25 `free_area` 结构的相关定义

```
#define MAX_ORDER 11

struct zone {
    .....
    struct free_area free_area[MAX_ORDER];
    .....
};

struct free_area {
    struct list_head free_list[MIGRATE_TYPES];
    unsigned long nr_free;
};

#define MIGRATE_UNMOVABLE 0
#define MIGRATE_RECLAMABLE 1
#define MIGRATE_MOVABLE 2
#define MIGRATE_RESERVE 3
/* can't allocate from here */
#define MIGRATE_ISOLATE 4
#define MIGRATE_TYPES 5
```

`free_area` 结构如图 5.5 所示，在 2.6.24 前的内核中 `free_area` 中只有一个链表，指向的是包含 2^n 个连续物理页面的块，但是从 2.6.24 开始，把每 1024 个页面组织成一个逻辑块⁷，每个块有一个类别属性，其中 `MIGRATE_MOVABLE` 是默认的类别，当某个类别的空闲伙伴块不足时，可以把其他链表上 `MIGRATE_MOVABLE` 块移过来，而 `MIGRATE_UNMOVABLE` 与它相反，是不可移动的；`MIGRATE_RESERVE` 是保留给内存不足时使用的内存块链表；`MIGRATE_RECLAMABLE` 是可交换回收的内存块链表；`MIGRATE_ISOLATE` 是不可分配的内存块链表。而 `free_area` 中链表的数量也增加到 5 个，假设某个包含 1024 个页面的块属性为 `MIGRATE_MOVABLE`，它被分配出去一部分，剩余的部分被拆散为 2^3 的连续页面，那么它只能链接到第 3 个 `free_area` 中的 `MIGRATE_MOVABLE` 链表中去。

⁷注意这里说 `free_area` 中的 2^n 个页面块和 1024 个页面一组的类别块，都是对内存的组织单位，是不同的管理需要而划分的，很容易混淆，以后我们把 1024 个页面一组的称为类别块。

zone 结构中的 pageblock_flags 专门记录每一个块对应的类别信息，在 build_all_zonelists 函数中，有一段代码决定是否开启这个分类的功能：

代码片段 5.26 节自 mm/page_alloc.c

```

1 [start_kernel() -> build_all_zonelists()]
2
3 void build_all_zonelists(void)
4 {
5     .....
6     vm_total_pages = nr_free_pagecache_pages();
7     if (vm_total_pages < (pageblock_nr_pages * MIGRATE_TYPES))
8         page_group_by_mobility_disabled = 1;
9     else
10        page_group_by_mobility_disabled = 0;
11     .....
12 }
```

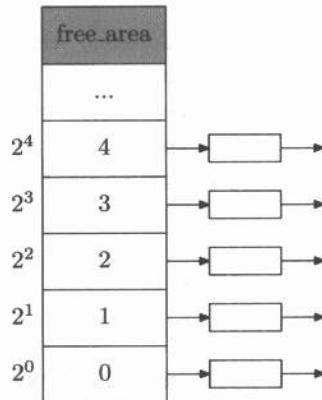


图 5.5 free_area 结构

如果开启分类功能，在 free_area_init_core() 中会调用 setup_usemap() 分配 pageblock_flags 的内存，并全部初始化为 0。由于 MIGRATE_TYPES 为 5，所以每个类别块(1024 个页面)需要 3 个 bit 来表示它的类别。

代码片段 5.27 节自 mm/page_alloc.c

```

1 static void __init setup_usemap(struct pglist_data *pgdat,
2                                 struct zone *zone,
3                                 unsigned long zonesize)
4 {
5     /* 根据 zone 中的 page 数量计算需要的内存字节数量。 */
6 }
```

```

6     unsigned long usemapsize = usemap_size(zonesize);
7     zone->pageblock_flags = NULL;
8     if (usemapsize) {
9         zone->pageblock_flags = alloc_bootmem_node(pgdat, usemapsize);
10        memset(zone->pageblock_flags, 0, usemapsize);
11    }
12 }

```

最后会在 free_area_init_core() 中把每一个类别块对应的 pageblock_flags 位设置为 MIGRATE_MOVABLE。另外还会通过 init_per_zone_pages_min() 把一部分设置为 MIGRATE_RESERVE 状态。

代码片段 5.28 节自 mm/page_alloc.c

```

[ start_kernel() -> setup_arch() -> zone_sizes_init() ->
  free_area_init_nodes() -> free_area_init_node() ->
  free_area_init_core() -> init_currently_empty_zone() ->
  memmap_init_zone()]

void __meminit memmap_init_zone(unsigned long size,
                                int nid,
                                unsigned long zone,
                                unsigned long start_pfn,
                                enum memmap_context context)
{
    .....
    for (pfn = start_pfn; pfn < end_pfn; pfn++) {
        .....
        if ((pfn & (pageblock_nr_pages-1)))
            set_pageblock_migratetype(page, MIGRATE_MOVABLE);
        .....
    }
}

```

在 5.3.3 节说到在建立 page 结构数 mem_map 数组时把每一个 page 结构置为保留态，最后 free_all_bootmem() 函数会扫描页面位图，释放相应的空闲页面，释放的同时会根据伙伴算法初始化 zone 结构的 free_area 数组，也就是合并伙伴，因此 mem_init 运行结束后，伙伴算法的相关初始化工作也就完成了。

代码片段 5.29 节自 mm/page_alloc.c

```

1 [ start_kernel() -> mem_init() -> free_all_bootmem() ->
2   free_all_bootmem_core() -> __free_pages_bootmem() ->
3   __free_pages() -> __free_pages_ok() -> free_one_page() ->

```

```
4     __free_one_page()]
5
6 static inline void __free_one_page(struct page *page,
7                                     struct zone *zone,
8                                     unsigned int order)
9 {
10    unsigned long page_idx;
11    int order_size = 1 << order;
12    int migratetype = get_pageblock_migratetype(page);
13
14    if (unlikely(PageCompound(page)))
15        destroy_compound_page(page, order);
16
17    /* 用 PFN 作为 mem_map 数组下标就可以索引到对应的 page 结构。 */
18    page_idx = page_to_pfn(page) & ((1 << MAX_ORDER) - 1);
19
20    VM_BUG_ON(page_idx & (order_size - 1));
21    VM_BUG_ON(bad_range(zone, page));
22
23    __mod_zone_page_state(zone, NR_FREE_PAGES, order_size);
24    while (order < MAX_ORDER-1) {
25        unsigned long combined_idx;
26        struct page *buddy;
27
28        buddy = __page_find_buddy(page, page_idx, order);
29        if (!page_is_buddy(page, buddy, order))
30            break;      /* Move the buddy up one level. */
31
32        list_del(&buddy->lru);
33        zone->free_area[order].nr_free--;
34        rmv_page_order(buddy);
35        combined_idx = __find_combined_index(page_idx, order);
36        page = page + (combined_idx - page_idx);
37        page_idx = combined_idx;
38        order++;
39    }
40
41    /* page 结构的 private=order. */
42    set_page_order(page, order);
43    list_add(&page->lru, &zone->free_area[order].free_list[migratetype]);
44    zone->free_area[order].nr_free++;
45 }
```

```

46
47 static inline struct page * __page_find_buddy(struct page *page,
48                                     unsigned long page_idx,
49                                     unsigned int order)
50 {
51     unsigned long buddy_idx = page_idx ^ (1 << order);
52     return page + (buddy_idx - page_idx);
53 }

```

这个函数的第一个参数是 `page` 结构指针，如果一个块中有多个 `page`，则参数为第一个 `page` 结构，第二个参数是 `page` 所在的 `zone`，假设要释放的 `page` 数量为 1，则第三个参数为 $0(2^0 = 1)$ ，依此类推。第24行的那个循环，主要是查看当前释放块伙伴是否空闲，如果空闲则合并它们。最后在第43行把它添加到 `zone` 对应的 `free_area` 数组中去。`page_idx` 是 `page` 结构在 `mem_map` 数组中的索引，其中 `__page_find_buddy` 设计得非常巧妙。这里我们举个例子来说明：如果一个块的大小是 4 个 `page`，则要向上合并的块为 8 个 `page`，合并后首地址 `page_idx` 开始的 `page` 的个数一定是 8 个 `page` 的整数倍。假设当前要释放的块大小是 4 个 `page`，`order=2`，`page_idx=11`，则它的“伙伴”的 `buddy_idx` 就是 15，而不能是 7，因为合并的上级为 8 个 `page`， $15-0$ 包含的页面个数是 8 的整数倍，如果把 7-4 这 4 个块与它合并，则会“拆散”7-0 这对“伙伴”。如果释放的 `page_idx=7`，则它的伙伴 `buddy_idx` 就是 3 而不能是 11。这样根据 `page_idx` 和 `order` 寻找它的伙伴的方法如下：

- (1) 如果 `page_idx+1` 不能被 $2^{order+1}$ 整除，则 `page_idx+2^{order}` 就是 `buddy_idx`，对应本例中的 $11+2^2$ 。
- (2) 如果 `page_idx+1` 能够被 $2^{order+1}$ 整除，则 `buddy_idx` 为 `page_idx-2^{order}`，对应本例中的 $7-2^2$ 。

下面我们来证明 `__page_find_buddy()` 函数中的 `page_idx` 异或($1 \ll order$)与这个方法等价。对于第一种情况：由于 `page_idx` 是要释放的块，所以可以肯定 `page_idx+1` 能够被 2^{order} 整除，如果不能够被 $2^{order+1}$ 整除，则 `page_idx+1` 的二进制中的第 `order` 位为 1，0 至 `order-1` 位全部为 0，`order+1` 开始以上的位无关紧要，进一步可以推出 `page_idx` 中第 `order` 位为 0，0 至 `order-1` 位全部为 1。所以 `page_idx+2^{order}` 就等于把 `page_idx` 中的第 `order` 位置 1。同理，对于第二种情况，`page_idx` 第 `order` 位为 1，`page_idx-2^{order}` 就等于把 `page_idx` 的第 `order` 位清 0。可以看到，把第 `order` 位与 1 异或，可以完成 0->1，1->0 的逻辑转换。

2. 伙伴算法的内存分配

分配相对复杂，内核收到 `alloc_pages()` 系列函数的分配请求时，首先需要确定是从哪一个节点上分配，然后再确定需要从节点的哪一个 `zone` 上分配，最后再根据伙伴算法，确定从 `zone` 的哪一个 `free_area` 数组成员上分配。在内存不足的情况下，还要回收内存，如

果内存还是不够，还要调度 kswapd 把必要的内存存储到交换分区中。内存分配模块总是试图从代价最小的节点上分配，而对 zone 的确定则根据 alloc_pages()系列函数的 gfp_mask 用以下规则确定：

- 如果 gfp_mask 参数设置了 __GFP_DMA 位，则只能从 ZONE_DMA 中分配。
- 如果 gfp_mask 参数设置了 __GFP_HIGHMEM 位，则能够从 ZONE_NORMAL, ZONE_NORMAL 和 ZONE_DMA 中分配。
- 如果 __GFP_DMA 和 __GFP_HIGHMEM 都没有设置，则能够从 ZONE_NORMAL 和 ZONE_DMA 中分配。

如果没有指定 __GFP_DMA 标志，则会尽量避免使用 ZONE_DMA 区的内存，只有当指定的区内存不足，而 ZONE_DMA 区又有充足的内存时，才会从 ZONE_DMA 中分配。

内存节点 pg_data_t 结构中的 node_zonelists[MAX_ZONELISTS] 数组就是用于上述分配策略的，内核在 start_kernel() 函数中调用 build_all_zonelists() 函数建立这个分配策略数组。其中 node_zonelists 是 zonelist 类型的结构体，其定义如下：

代码片段 5.30 节自 include/linux/mmzone.h

```
struct zonelist {
    struct zonelist_cache *zlcache_ptr;
    struct zone *zones[ MAX_ZONES_PER_ZONELIST + 1 ];
#ifndef CONFIG_NUMA
    struct zonelist_cache zlcache;
#endif
};
```

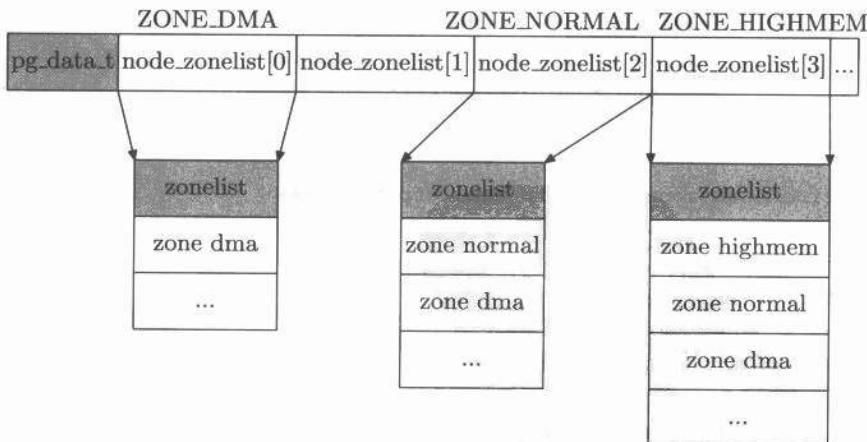


图 5.6 zonelists 分配策略

`build_all_zonelists()`函数初始化 `pg_data_t` 中的 `node_zonelists` 数组如图5.6所示，当请求分配时总是从CPU的本地内存节点开始，也就是代价最小的内存节点，x86系统上只有一个内存节点 `contig_page_data`。如果参数 `gfp_mask` 指定了 `_GFP_DMA` 标志，就试图从 `node_zonelists[0]` 的 `zonelist` 分配，它的 `zonelist` 只有一个 `zone` 指针指向 `ZONE_DMA`，所以对于 `_GFP_DMA`，只能从 `ZONE_DMA` 中分配；如果 `gfp_mask` 指定了 `_GFP_NORMAL` 标志，就试图从 `node_zonelists[2]` 的 `zonelist` 分配，这个 `zonelist` 有两个 `zone` 指针，第一个指向 `ZONE_NORMAL`，第二个指向 `ZONE_DMA`，所以对于 `_GFP_NORMAL`，先尝试从 `ZONE_NORMAL` 中分配，如果失败，再试图从 `ZONE_DMA` 中分配。对于多个节点的情况，代价越大的节点，它的 `zone` 指针在 `zonelist` 中越靠后。

现在我们来看看非NUMA系统的`alloc_pages`，NUMA和它差不多，只是多了一个`mempolicy`来定位先尝试从哪个节点分配。结合图5.6，很容易看懂下面这段代码，它主要根据`gfp_mask`来选择从哪个`node_zonelists`中分配，如果`gfp_mask`指定了`_GFP_DMA`标志，则选择`node_zonelists[0]`分配……

代码片段 5.31 节自 `include/linux/gfp.h`

```

1 static inline struct page *
2     alloc_pages_node(int nid,
3                         gfp_t gfp_mask,
4                         unsigned int order)
5 {
6     if (unlikely(order >= MAX_ORDER))
7         return NULL;
8     /* Unknown node is current node */
9     if (nid < 0)
10         nid = numa_node_id();
11
12     return __alloc_pages(gfp_mask, order,
13                          .NODE_DATA(nid)->node_zonelists +
14                          gfp_zone(gfp_mask));
15 }
16
17 static inline enum zone_type gfp_zone(gfp_t flags)
18 {
19     int base = 0;
20 #ifdef CONFIG_NUMA
21     if (flags & _GFP_THISNODE)
22         base = MAX_NR_ZONES;
23 #endif
24
25 #ifdef CONFIG_ZONE_DMA
26     if (flags & _GFP_DMA)

```

```

27     return base + ZONE_DMA;
28 #endif
29 #ifdef CONFIG_ZONE_DMA32
30     if (flags & __GFP_DMA32)
31         return base + ZONE_DMA32;
32 #endif
33     if ((flags & (__GFP_HIGHMEM | __GFP_MOVABLE)) ==
34         (__GFP_HIGHMEM | __GFP_MOVABLE))
35         return base + ZONE_MOVABLE;
36 #ifdef CONFIG_HIGHMEM
37     if (flags & __GFP_HIGHMEM)
38         return base + ZONE_HIGHMEM;
39 #endif
40     return base + ZONE_NORMAL;
41 }

```

当节点和 zone 都确定下来后，就要根据伙伴算法从 zone 的 free_area 实施分配了，MIGRATE_TYPE 被开启的情况下，还要进一步确定从 free_area 哪个 MIGRATE_TYPE 中分配，这是由 allocflags_to_migratetype() 完成的。

代码片段 5.32 节自 include/linux/gfp.h

```

1 static inline int allocflags_to_migratetype(gfp_t gfp_flags)
2 {
3     WARN_ON((gfp_flags & GFP_MOVABLE_MASK) == GFP_MOVABLE_MASK);
4
5     if (unlikely(page_group_by_mobility_disabled))
6         return MIGRATE_UNMOVABLE;
7     /* Group based on mobility */
8     return (((gfp_flags & __GFP_MOVABLE) != 0) << 1) |
9         ((gfp_flags & __GFP_RECLAMABLE) != 0);
10 }

```

现在一切准备就绪后，调用 __rmqueue() 从 free_area 中取出内存页面。

代码片段 5.33 节自 mm/page_alloc.c

```

1 static struct page * __rmqueue(struct zone *zone,
2                                 unsigned int order,
3                                 int migratetype)
4 {
5     struct page *page;
6     page = __rmqueue_smallest(zone, order, migratetype);
7     if (unlikely(!page))
8         page = __rmqueue_fallback(zone, order, migratetype);

```

```

9
10    return page;
11 }

```

它先调用 `_rmqueue_smallest()` 从指定的 MIGRATE_TYPE 链上分配内存，如果失败，再调用 `_rmqueue_fallback()` 尝试从其他的 MIGRATE_TYPE 链上分配。而尝试的依据是由一个二维数组 `fallbacks` 指定的。

代码片段 5.34 节自 mm/page_alloc.c

```

static int fallbacks[MIGRATE_TYPES][MIGRATE_TYPES-1] = {
    [MIGRATE_UNMOVABLE] = { MIGRATE_RECLAIMABLE,
                            MIGRATE_MOVABLE,
                            MIGRATE_RESERVE },
    [MIGRATE_RECLAIMABLE] = { MIGRATE_UNMOVABLE,
                            MIGRATE_MOVABLE,
                            MIGRATE_RESERVE },
    [MIGRATE_MOVABLE]     = { MIGRATE_RECLAIMABLE,
                            MIGRATE_UNMOVABLE,
                            MIGRATE_RESERVE },
    [MIGRATE_RESERVE]     = { MIGRATE_RESERVE,
                            MIGRATE_RESERVE,
                            MIGRATE_RESERVE },
};


```

举例来说：根据 `fallbacks` 数组，假设从 `MIGRATE_UNMOVABLE` 链表上分配失败，`_rmqueue_fallback()` 尝试的顺序依次为 `MIGRATE_RECLAIMABLE`, `MIGRATE_MOVABLE`, `MIGRATE_RESERVE` 链表。现在我们来看看 `_rmqueue_smallest()`:

代码片段 5.35 节自 mm/page_alloc.c

```

1 static struct page *_rmqueue_smallest(struct zone *zone,
2                                         unsigned int order,
3                                         int migratetype)
4 {
5     unsigned int current_order;
6     struct free_area * area;
7     struct page *page;
8
9     /* Find a page of the appropriate size in the preferred list */
10    for (current_order = order;

```

```
11     current_order < MAX_ORDER; ++current_order) {
12         area = &(zone->free_area[current_order]);
13         if (list_empty(&area->free_list[migratetype]))
14             continue;
15
16         page = list_entry(area->free_list[migratetype].next,
17                            struct page, lru);
18
19         list_del(&page->lru);
20         rmv_page_order(page);
21         area->nr_free--;
22         __mod_zone_page_state(zone, NR_FREE_PAGES, - (1UL << order));
23         expand(zone, page, order, current_order, area, migratetype);
24         return page;
25     }
26     return NULL;
27 }
```

这个函数根据 order 从最合适的 free_area 队列中分配，如果不成功就从更大的块中找，找到一个合适的块后把它摘下来，最后需要把大块剩余的那一部分解并放到对应的队列中去，这个工作是 expand() 函数完成的。

代码片段 5.36 节自 mm/page_alloc.c

```
1 static inline void expand(struct zone *zone,
2                           struct page *page,
3                           int low, int high,
4                           struct free_area *area,
5                           int migratetype)
6 {
7     unsigned long size = 1 << high;
8
9     while (high > low) {
10         area--;
11         high--;
12         size >>= 1;
13         VM_BUG_ON(bad_range(zone, &page[size]));
14         list_add(&page[size].lru, &area->free_list[migratetype]);
15         area->nr_free++;
16         set_page_order(&page[size], high);
17     }
18 }
```

假设需要分配 2^1 个页面，则 low=1，但在 2^2 页面块中找到空闲的，则 high=2，这个循环就把剩余的那个块添加到 2^1 对应的队列中。

5.4.2 SLUB 分配器

伙伴算法以页为单位管理内存，然而在大多数情况下，程序需要的内存并不是一个页面，而是几十、几百个字节的小块内存。于是需要在页的基础上再建立一种小块内存管理机制，以前这就是 SLAB 分配器的工作。但是由于 SLAB 太过复杂，它本身的管理结构需要占用过多的内存，并且效率也相对较低，为此出现了 SLUB 分配器。SLUB 提供了和 SLAB 一致的接口，不仅简化了 SLAB 分配器，也提供了更好的性能。目前来看内核中 SLAB 和 SLUB 将共存，但是 SLUB 已经成为内核默认的小块内存分配器，它将逐渐成为主流。

SLUB 是在页管理的基础上，把分配的内存分组管理，每个组分别包含 2^3 、 2^4 、 \dots 2^{11} 个字节，在 4KB 页大小的默认情况下，另外还有两个特别的组，其大小分别为 96 和 192 个字节，总共 11 个组。每一个 kmem_cache 结构代表一个组，每个组负责对应大小的内存分配和回收，每一次分配请求都是从大小最接近的那个组分配。SLUB 分配器向页面管理单元“批发”内存然后再“零售”出去。在 SLAB 中，程序需要通过 kmem_cache_create() 函数来创建专用的 SLAB 对象，这样即便两个程序要分配的内存对象大小一致，也需要创建两个独立的 kmem_cache 结构，过多的 kmem_cache 结构不仅造成管理结构本身内存浪费，另外在维护这些结构时也带来了不必要的消耗。因此 SLUB 只使用 kmalloc() 在这 11 个默认的 SLUB 对象的基础上进行内存分配就可以了，但是为了兼容 SLAB，SLUB 依然提供的 kmem_cache_create()，这个函数会尽量在这 11 个对象中寻找合适的 kmem_cache，然后增加 kmem_cache 结构中的 refcount 并返回，只有在不得已的情况下才会创建专门的 kmem_cache。SLUB 的相关结构定义如下：

代码片段 5.37 节自 include/linux/slub_def.h

```

1 struct kmem_cache {
2     /* Used for retrieving partial slabs etc */
3     unsigned long flags;
4     /* The size of an object including meta data */
5     int size;
6     /* The size of an object without meta data */
7     int objsize;
8     /* Free pointer offset. */
9     int offset;
10    int order;
11
12    struct kmem_cache_node local_node;
13    /* Allocation and freeing of slabs */
14    /* Number of objects in slab */

```

```
15 int objects;
16 /* Refcount for slab cache destroy */
17 int refcount;
18 void (*ctor)(struct kmem_cache *, void *);
19 /* Offset to metadata */
20 int inuse;
21 /* Alignment */
22 int align;
23 /* Name (only for display!) */
24 const char *name;
25 /* List of slab caches */
26 struct list_head list;
27 #ifdef CONFIG_SLUB_DEBUG
28 /* For sysfs */
29 struct kobject kobj;
30#endif
31
32 #ifdef CONFIG_NUMA
33 int defrag_ratio;
34 struct kmem_cache_node *node[MAX_NUMNODES];
35#endif
36 /* 多处理器情况下，每一个CPU有自己的SLAB。*/
37 #ifdef CONFIG_SMP
38 struct kmem_cache_cpu *cpu_slab[NR_CPUS];
39#else
40 struct kmem_cache_cpu cpu_slab;
41#endif
42 };
43
44 struct kmem_cache_cpu {
45 void **freelist;
46 struct page *page;
47 int node;
48 unsigned int offset;
49 unsigned int objsize;
50 };
51
52 struct kmem_cache_node {
53 /* Protect partial list and nr_partial */
54 spinlock_t list_lock;
55 unsigned long nr_partial;
56 atomic_long_t nr_slabs;
```

```

57     struct list_head partial;
58 #ifdef CONFIG_SLAB_DEBUG
59     struct list_head full;
60 #endif
61 };

```

SLUB 管理器需要建立如图5.7所示的管理结构⁸，其中右下角的 Page Frame 就是从页面管理单元那里“批发”的物理页面，每一次“批发” 2^{order} 个页面，其中 order 是在 kmem_cache 中指定的。之后，这些物理页面被按照对象大小组织成单向链表，大小是由 objsize 指定的。例如对于 16 字节的对象大小，那么在 Page Frame 中，每个 object 就是 16 个字节，链表的指针就存放在每个 object 中偏移为 offset 的地方，offset 默认为 0，也就是每个 object 的头部 4 个字节指向下一个 object，这样不会有问题是。但是由于 SLAB 程序可以在创建 slab 的时候指定一个“构造函数”，每一次“批发”后都调用这个“构造函数”对每一个小对象进行初始化，因此需要为单向链表的指针分配额外的存储空间。在这种情况下，为了兼容 SLAB，一个 object 的实际大小要大于分配给程序使用的大小。于是 kmem_cache 结构中的 size 就表示实际大小，而 objsize 表示分配出去可以使用的大小。在多 CPU 的系统中，为了避免过多地使用自旋锁带来的性能开销，每一个 CPU 有一个 kmem_cache_cpu 结构，“仓库”中的“货物”主要通过 kmem_cache_cpu 结构管理。一次“批发”出来的是连续的 2^{order} 个页面，SLUB 释放其中的一个或者多个 object 时，把它放到 kmem_cache_node 中，下次请求分配时，如果 kmem_cache_cpu 中的 freelist 为空，就到 kmem_cache_node 中分配，如果仍然为空，就进行“批发”。

内核在 start_kernel() 中对 11 个 SLUB 进行初始化，但是并不会“批发”物理页面，在处理第一个分配请求的时候，发现“仓库”中没有“货物”，才会“批发”。现在假设 SLUB 初始化刚刚完成，“仓库”为空，我们来看看分配请求的处理步骤：

- (1) 根据要求分配的字节数量，确定 kmem_cache 结构。
- (2) 根据 kmem_cache 和当前 CPU 确定 kmem_cache_cpu 结构，并查看该结构中的 freelist，如果不为 NULL，则可以直接从 freelist 上分配。现在 freelist 指针为 NULL。
- (3) 检查 kmem_cache_node 的 partial 链，看是否存在释放了的 object，如果有就从 partial 链上分配，现在假设这个链也为空。
- (4) “批发” 2^{order} 个页面，kmem_cache_cpu 结构中的 page 指向第一个页面对应的 page 结构，同时把页面初始化成大小为 size 的 object 链表。page 结构中的 freelist 指向第一个 object。page 结构中的 inuse 初始化为这个 Page Frame 所包含的 object 的总数。
- (5) 把第一个 object 分配出去，page 结构中的 freelist 被修改为 NULL，kmem_cache_cpu 的 freelist 指向 Page Frame 中的第二个 object。以后再分配时就从 kmem_cache_cpu 结构中的 freelist 上分配。

⁸这里说的 object 是指一片固定大小的内存，而 Page Frame 指一个或多个物理页面。

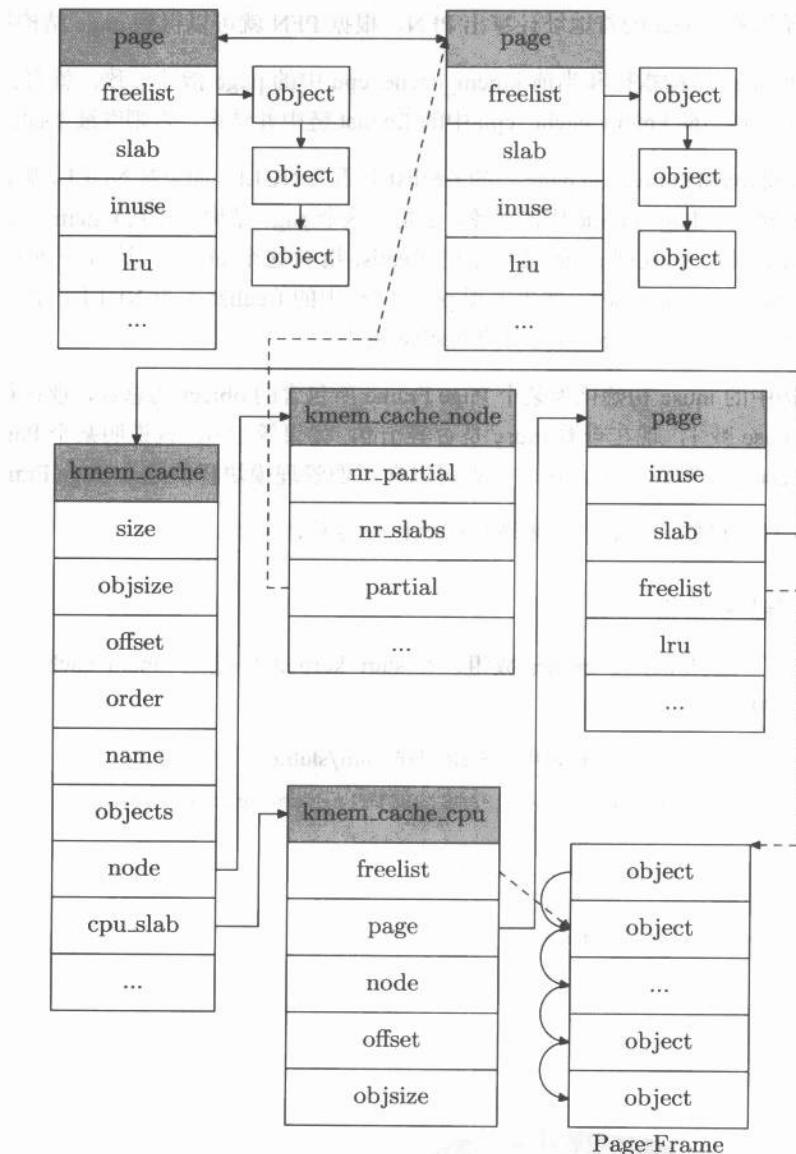


图 5.7 SLUB 管理结构

假设 Page Frame 中的全部 object 都被分配出去并且没有被释放，如果有新的分配请求到来，就会分配新的 Page Frame，同时 kmem_cache_cpu 中的 page 指针就会指向新的 page 结构，而 freelist 也会指向新的 Page Frame 中的 object，SLUB 并不记录每个 Page Frame 的相关信息。这些信息可以在释放函数 kfree()中根据 object 的首地址计算出来，下面是释放的步骤：

- (1) 根据要释放的 object 的首地址计算出 PFN，根据 PFN 就可以得到 page 结构指针。
 - (2) 如果这个 page 结构指针和当前 kmem_cache_cpu 中的 page 指针一致，就直接把释放的对象添加到当前 kmem_cache_cpu 中的 freelist 链中并结束。否则继续下面的步骤。
 - (3) 检查当前要释放的 object 的 page 中的 freelist 是否为 NULL，如果是 NULL，则说明这是第一次释放某个 Page Frame 中的对象，于是把这个 page 结构添加到 kmem_cache_node 的 partial 链上，并且把 page 结构中的 freelist 指向这个 object，这样下次释放一个 Page Frame 中的 object 时，如果发现 page 结构中的 freelist 不为 NULL，就可以把这个 object 直接链接到 page 结构中的 freelist 链上。
 - (4) page 结构中的 inuse 初始化为某个 Page Frame 所包含的 object 的总数，现在每释放一次就把 inuse 减 1，现在查看 inuse 是否等于 0，如果等于 0，就说明某个 Page Frame 中的 object 全部被释放了，这时候就可以向页面管理模块释放这个 Page Frame 了。

对 SLUB 有了整体认识后，再来看代码就比较容易理解了。

3. SLUB 分配器的初始化

内核定义了一个 `kmalloc_caches` 数组，在 `start_kernel()` 中调用 `kmem_cache_init()` 对这个数组进行初始化。

代码片段 5.38 节自 mm/slub.c

```
21                 GFP_KERNEL);  
22  
23     kmalloc_caches[ 0]. refcount = -1;  
24     caches++;  
25  
26     hotplug_memory_notifier(slab_memory_callback, 1);  
27 #endif  
28  
29     /* Able to allocate the per node structures */  
30     slab_state = PARTIAL;  
31     .  
32     /* Caches that are not of the two-to-the-power-of size */  
33     if (KMALLOC_MIN_SIZE <= 64) {  
34         create_kmalloc_cache(&kmalloc_caches[ 1],  
35                               "kmalloc-96", 96,  
36                               GFP_KERNEL);  
37         caches++;  
38     }  
39     if (KMALLOC_MIN_SIZE <= 128) {  
40         create_kmalloc_cache(&kmalloc_caches[ 2],  
41                               "kmalloc-192", 192,  
42                               GFP_KERNEL);  
43         caches++;  
44     }  
45  
46     for (i = KMALLOC_SHIFT_LOW; i < PAGE_SHIFT; i++) {  
47         create_kmalloc_cache(&kmalloc_caches[ i],  
48                               "kmalloc", 1 << i,  
49                               GFP_KERNEL);  
50         caches++;  
51     }  
52     .....  
53     printk(KERN_INFO "SLUB: Genslabs=%d, HWalign=%d, Order=%d-%d,  
54             MinObjects=%d, " " CPUs=%d, Nodes=%d\n", caches,  
55             cache_line_size(), slub_min_order, slub_max_order,  
56             slub_min_objects, nr_cpu_ids, nr_node_ids);  
57 }
```

这个函数调用 `create_kmalloc_cache()` 初始化 `kmalloc_caches` 数组的各个成员，第46行的 `KMALLOC_SHIFT_LOW` 默认为 3，这样 `kmalloc_caches` 数组的第 3 个成员维护的对象大小为 2^3 个字节，... 第 11 个成员维护的对象大小为 2^{11} 个字节，空出来的第 1, 2 个成员分别维护的对象大小为 96 和 128 个字节。在 `CONFIG_NUMA` 的情况下，需要为每一个

node 建立一个 `kmem_cache_node` 结构, `kmem_cache` 结构中的指针数组 `node` 成员就指向对应的 `kmem_cache_node` 结构, 但是对 SLUB 的初始化还未完成, `kmalloc_caches[0]` 就用于 `kmem_cache_node` 的专用缓存, 为此需要特殊处理 `kmalloc_caches[0]`。`PAGE_SHIFT` 被定义为 12, 这样除去第 0 个, 还有 11 个 SLUB 分配对象, `slab_caches` 指向这 11 个对象的头部。

4. SLUB 的内存分配

分配是通过 `kmalloc()` 函数进行的, 这个函数根据需要分配内存大小, 找到对应的 `kmalloc_caches`, 如果要求分配的大小超过半个页面, 就直接调用 `_get_free_pages()`。

代码片段 5.39 节自 `include/linux/slub_def.h`

```

1 static __always_inline void *kmalloc(size_t size, gfp_t flags)
2 {
3     /* 如果编译期能确定 size 的大小, __builtin_constant_p(size) 返回 1. */
4     if (__builtin_constant_p(size)) {
5         /* 如果大小超过半个页面直接调用 __get_free_pages(). */
6         if (size > PAGE_SIZE / 2)
7             return (void *)__get_free_pages(flags | __GFP_COMP,
8                                             get_order(size));
9         if (!(flags & 'SLUB_DMA')) {
10             /* 根据 size 在 kmem_cache 数组中找到最合适的 kmem_cache 结构。*/
11             struct kmem_cache *s = kmalloc_slab(size);
12             if (!s)
13                 return ZERO_SIZE_PTR;
14             return kmem_cache_alloc(s, flags);
15         }
16     }
17     return __kmalloc(size, flags);
18 }
19
20 void *__kmalloc(size_t size, gfp_t flags)
21 {
22     struct kmem_cache *s;
23     if (unlikely(size > PAGE_SIZE / 2))
24         return (void *)__get_free_pages(flags | __GFP_COMP,
25                                         get_order(size));
26     /* 根据 size 在 kmem_cache 数组中找到最合适的 kmem_cache 结构。*/
27     s = get_slab(size, flags);
28     if (unlikely(ZERO_OR_NULL_PTR(s)))
29         return s;
30     return slab_alloc(s, flags, -1, __builtin_return_address(0));

```

```
31 }
32
33 static void __always_inline *slab_alloc(struct kmem_cache *s,
34                                         gfp_t gfpflags,
35                                         int node,
36                                         void *addr)
37 {
38     void **object;
39     unsigned long flags;
40     struct kmem_cache_cpu *c;
41
42     local_irq_save(flags);
43     /* 从 kmem_cache 上找到当前 cpu 的 kmem_cache_cpu 结构。 */
44     c = get_cpu_slab(s, smp_processor_id());
45     /*
46      * 对照图5.7很容易看懂，如果 kmem_cache_cpu 的 freelist 为 NULL，
47      * 就调用 __slab_alloc() 先尝试从 partial 链表上分配，如果失败就向页面管理模块“批发”。
48      */
49     if (unlikely(!c->freelist || !node_match(c, node)))
50         object = __slab_alloc(s, gfpflags, node, addr, c);
51     /* 把 freelist 的头分配出去，freelist 指向链表的下一个。 */
52     else {
53         object = c->freelist;
54         c->freelist = object[c->offset];
55     }
56     local_irq_restore(flags);
57
58     if (unlikely((gfpflags & __GFP_ZERO) && object))
59         memset(object, 0, c->objszie);
60     return object;
61 }
62
63 static void *__slab_alloc(struct kmem_cache *s,
64                           gfp_t gfpflags,
65                           int node,
66                           void *addr,
67                           struct kmem_cache_cpu *c)
68 {
69     void **object;
70     struct page *new;
71
72     /* 如果 c->page 为 NULL，说明“仓库”空了，需要“批发”页面。 */
```

```
73     if (!c->page)
74         goto new_slab;
75     slab_lock(c->page);
76     if (unlikely(!node_match(c, node)))
77         goto another_slab;
78 load_freelist:
79     object = c->page->freelist;
80     /*
81      * c->page 不为 NULL, 但是 c->page->freelist 为 NULL, 表示曾经进行过“批发”,
82      * 但是现在“仓库”空了。这里需要注意两个不同结构中的 freelist.
83      */
84     if (unlikely(!object))
85         goto another_slab;
86     if (unlikely(SlabDebug(c->page)))
87         goto debug;
88     /* 把第一个 object 分配出去.*/
89     object = c->page->freelist;
90     /* kmem_cache_cpu 中的 freelist 指向下一个 object.*/
91     c->freelist = object[c->offset];
92     /*
93      * inuse 设置为 Page Frame 中的 objects 的数量, 每释放一次,
94      * inuse 会减 1, 当 inuse 为 0 的时候, 说明 Page Frame 中的 object
95      * 全部被分配出去后, 又被释放了, 这种情况下就可以释放 Page Frame 了。
96      */
97     c->page->inuse = s->objects;
98
99     c->page->freelist = NULL;
100    c->node = page_to_nid(c->page);
101    slab_unlock(c->page);
102    return object;
103 another_slab:
104     /* 主要是把 c->page 置为 NULL, 参见第73行.*/
105     deactivate_slab(s, c);
106 new_slab:
107     /* 先尝试从 partial 链表上分配.*/
108     new = get_partial(s, gfpflags, node);
109     if (new) {
110         c->page = new;
111         goto load_freelist;
112     }
113     if (gfpflags & __GFP_WAIT)
114         local_irq_enable();
```

```
115 /* 批发 2order 个页面，并初始化 object 链表等信息。*/
116 new = new_slab(s, gfpflags, node);
117
118 if (gfpflags & __GFP_WAIT)
119     local_irq_disable();
120
121 if (new) {
122     c = get_cpu_slab(s, smp_processor_id());
123     if (c->page)
124         flush_slab(s, c);
125     slab_lock(new);
126     SetSlabFrozen(new);
127     c->page = new;
128     goto load_freelist;
129 }
130
131 return NULL;
132 .....
133
134 static struct page *new_slab(struct kmem_cache *s,
135                             gfp_t flags,
136                             int node)
137 {
138     struct page *page;
139     struct kmem_cache_node *n;
140     void *start;
141     void *last;
142     void *p;
143
144     BUG_ON(flags & GFP_SLAB_BUG_MASK);
145     /* 分配页面。*/
146     page = allocate_slab(s,
147                          flags & (GFP_RECLAIM_MASK | GFP_CONSTRAINT_MASK),
148                          node);
149     if (!page)
150         goto out;
151     n = get_node(s, page_to_nid(page));
152     if (n)
153         atomic_long_inc(&n->nr_slabs);
154     page->slab = s;
155     page->flags |= 1 << PG_slab;
156     if (s->flags &
```

```
157     (SLAB_DEBUG_FREE | SLAB_RED_ZONE |
158      SLAB_POISON | SLAB_STORE_USER | SLAB_TRACE))
159     SetSlabDebug(page);
160     start = page_address(page);
161     if (unlikely(s->flags & SLAB_POISON))
162         memset(start, POISON_INUSE, PAGE_SIZE << s->order);
163
164     last = start;
165     /* 这个循环把分配来的 Page Frame 链接成 object 链，见图5.7。 */
166     for_each_object(p, s, start) {
167         setup_object(s, page, last);
168         set_freepointer(s, last, p);
169         last = p;
170     }
171     setup_object(s, page, last);
172     set_freepointer(s, last, NULL);
173     page->freelist = start;
174     page->inuse = 0;
175 out:
176     return page;
177 }
178
179 static struct page *allocate_slab(struct kmem_cache *s,
180                                     gfp_t flags,
181                                     int node)
182 {
183     struct page * page;
184     int pages = 1 << s->order;
185
186     if (s->order)
187         flags |= __GFP_COMP;
188     if (s->flags & SLAB_CACHE_DMA)
189         flags |= SLUB_DMA;
190     if (s->flags & SLAB_RECLAIM_ACCOUNT)
191         flags |= __GFP_RECLAIMABLE;
192     /* 终于看到前面说到的“批发”是如何进行的了。 */
193     if (node == -1)
194         page = alloc_pages(flags, s->order);
195     else
196         page = alloc_pages_node(node, flags, s->order);
197
198     if (!page)
```

```
199     return NULL;
200     mod_zone_page_state(page_zone(page),
201                           (s->flags & SLAB_RECLAIM_ACCOUNT) ?
202                               NR_SLAB_RECLAIMABLE : NR_SLAB_UNRECLAIMABLE,
203                           pages);
204     return page;
205 }
```

5. SLUB 的内存释放

释放的时候先把 object 添加到 partial 链表上，如果 Page Frame 中的 object 全部被释放了，就可以释放 Page Frame 了。释放的工作由 kfree()完成。

代码片段 5.40 节自 mm/slub.c

```
1 void kfree(const void *x)
2 {
3     struct page *page;
4     if (unlikely(ZERO_OR_NULL_PTR(x)))
5         return;
6     /* 根据地址确定对应的 page 结构指针。 */
7     page = virt_to_head_page(x);
8     if (unlikely(!PageSlab(page))) {
9         put_page(page);
10        return;
11    }
12    slab_free(page->slab,
13               page,
14               (void *)x,
15               __builtin_return_address(0));
16 }
17
18 static void __always_inline slab_free(struct kmem_cache *s,
19                                       struct page *page,
20                                       void *x,
21                                       void *addr)
22 {
23     void **object = (void *)x;
24     unsigned long flags;
25     struct kmem_cache_cpu *c;
26
27     local_irq_save(flags);
28     debug_check_no_locks_freed(object, s->objsize);
```

```
29     c = get_cpu_slab(s, smp_processor_id());
30     /*
31      * 如果当前 kmem_cache 结构中的 page 指针和要释放的 object 所对应的 page 指针一致,
32      * 就直接把这个 object 链接到 freelist 链表中。
33      */
34     if (likely(page == c->page && c->node >= 0)) {
35         object[c->offset] = c->freelist;
36         c->freelist = object;
37     } else
38         __slab_free(s, page, x, addr, c->offset);
39
40     local_irq_restore(flags);
41 }
42
43 static void __slab_free(struct kmem_cache *s,
44                         struct page *page,
45                         void *x,
46                         void *addr,
47                         unsigned int offset)
48 {
49     void *prior;
50     void **object = (void *)x;
51     slab_lock(page);
52     if (unlikely(SlabDebug(page)))
53         goto debug;
54     checks_ok:
55     prior = object[offset] = page->freelist;
56     page->freelist = object;
57     page->inuse--;
58
59     if (unlikely(SlabFrozen(page)))
60         goto out_unlock;
61     /*
62      * 如果 inuse 为 NULL, 说明 Page Frame 中的 object 全部被分配出去后,
63      * 又被释放了, 这种情况下就可以释放 Page Frame 了。见图5.7。
64      */
65     if (unlikely(!page->inuse))
66         goto slab_empty;
67     /*
68      * Objects left in the slab. If it was not on the partial
69      * list before then add it.
70      */
```

```
71  /*
72   * 如果 page 结构的 freelist 为 NULL, 说明这是第一次释放属于这个 Page Frame 的 object,
73   * 于是把这个 page 结构指针添加到 partial 链表中, 见图5.7。
74   */
75  if (unlikely(!prior))
76      add_partial_tail(get_node(s, page_to_nid(page)), page);
77
78 out_unlock:
79     slab_unlock(page);
80     return;
81 slab_empty:
82     if (prior)
83         /* Slab still on the partial list. */
84         remove_partial(s, page);
85     slab_unlock(page);
86     discard_slab(s, page);
87     return;
88 debug:
89     if (!free_debug_processing(s, page, x, addr))
90         goto out_unlock;
91     goto checks_ok;
92 }
```

第6章 中断与异常处理

中断是现代操作系统的一项重要技术，利用中断技术可以极大地提高系统吞吐量。深入理解中断处理的相关内容，是进一步学习设备驱动程序及异常处理(例如：缺页异常)的关键。而理解中断的关键在于：理解中断描述符表的作用，理解外部设备中断和CPU异常的区别，理解中断控制器的工作原理，理解中断控制器，外部设备的中断请求线，中断向量的关系。本章首先对这些概念进行介绍，其后将对中断的初始化、中断处理、软中断处理等内容进行深入介绍。

6.1 中断的分类

系统中有许多不同的功能都利用中断描述符表，调用哪一个中断处理例程是由中断向量决定的，广义上的中断是指利用中断描述符表进行控制转移，可以分为下面几种。

1. 外部可屏蔽中断

外部设备通过可编程中断控制器向CPU报告的中断。如图6.1所示，外部设备的中断请求线(IRQ)连接到一个中断控制器上，当一个外部设备需要发出中断时，会驱动对应的中断请求线进入有信号状态。中断控制器检测这个中断是否被屏蔽了，如果没有被屏蔽就驱动CPU的INTR中断请求线进入信号状态，这样CPU就能检测到这个中断了。如果该中断被屏蔽，中断控制器中的寄存器的某个位将记录这一请求，等到中断被开启时再驱动INTR。之后CPU通过中断应答从中断控制器的数据线上读取中断向量。如果多个设备在同一时刻通过不同的中断请求线发出中断请求，中断控制器也会将这些请求记录在不同的位中，如果这些中断都没有被屏蔽，中断控制器根据优先级，先向CPU报告优先级高的中断，再报告优先级低的中断。对于8259A来说，IRQn数字n越小优先级越大。

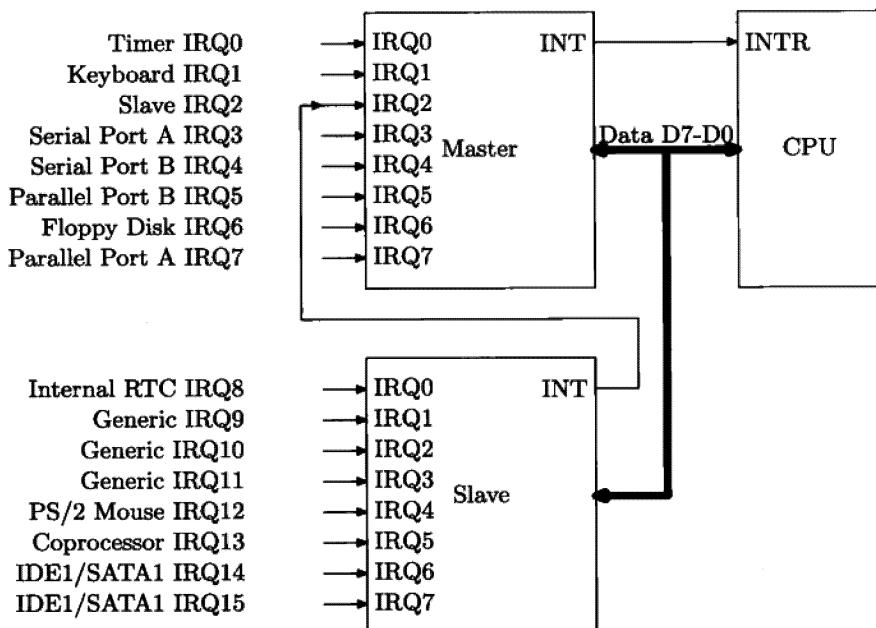


图 6.1 Intel 8259A 中断控制器

8259A 是早期在 PC 机上使用的可编程中断控制器，一片 8259A 可以管理 8 个中断请求线，对于 PC 来说，8 个中断请求线太少了，于是使用两片级联，其中从片的 INT 信号连接到主片的 IRQ2 上来，总共可以管理 15 个中断请求线。8259A 中 IRQ 越小优先级越高，图 6.1 中，假设 IRQ0 和 IRQ1 两根输入信号同时发出中断请求，8259A 通过 INT 信号线向 CPU 请求中断，CPU 进行中断应答，8259A 在应答周期把 IRQ0 的中断向量放到数据总线上，当 CPU 处理 IRQ0 的中断结束时，会向 8259A 发出 End Of Interrupt 通知，这时 8259A 才会向 CPU 报告 IRQ1 的中断请求。在现在的主板上已经看不到 8259A 的影子了(见第 1.4 节)，但典型的主板芯片组中，例如 Intel ICH7 中集成了 8259A 兼容功能。

注意这里说的外部可屏蔽中断，和中断控制器的屏蔽不是一回事。这里的可屏蔽是指是否能够通过 CPU 标志寄存器的关中断标志 IF 位来屏蔽中断，CPU 的指令流水操作的最后一步就执行中断检测，如果标志寄存器中的 IF 位为 1，就检查 INTR 上是否有信号，如果 IF 为 0，就不会去理会 INTR 线上的信号。从这里可以看出关闭外部中断有多种方式：

- 通过 cli 指令把标志寄存器中的 IF 位清零，这样就关闭了所有的外部中断。
- 通过中断控制器中的中断屏蔽寄存器，屏蔽某一特定的 IRQn，从而屏蔽该中断，但不影响其他中断。

(c) 大部分外部设备上，也设有控制寄存器，可以控制该设备是否发出中断，假设一个设备的中断请求线连接到中断控制器的 IRQn 上，当通过设备的中断允许寄存器来关闭设备中断时，IRQn 上始终不会进入信号状态。这是从源头上关闭了一个设备的中断。8259A 中有可以有 15 个中断源，但是随着外部设备的增多，15 个已经不够用，所以常常把多个设备的中断请求线连接到中断控制器的同一根 IRQ 线上，这就是共享中断。对于共享中断的多个设备，无论哪一个设备发出中断请求，CPU 通过中断应答从数据线上读出来的都是同一个中断向量，CPU 如何区分是哪个设备的中断呢？设备设有中断状态寄存器，中断处理程序可以通过读取各个设备的状态寄存器来区分是哪一个设备发出的中断。对于共享中断的多个设备，如果只想关闭某一个设备的中断，那么就可以利用设备的中断允许寄存器来进行。可以看出这些关中断的方式的作用范围逐步减小。另外，许多设备可以发出多种中断，例如网卡，分别在接收数据包完成时，发送数据包完成时，出错时都有可能发出中断，网卡驱动的中断处理程序通过读取中断状态寄存器来区分出中断原因，以便进行下一步处理。也可以通过中断允许寄存器来更为精确地控制是否开启接收中断、发送中断、或者错误中断，网卡驱动中的 NAPI 技术就是通过关闭接收中断，但开启发送和错误中断来进行性能优化的。

中断控制器如何驱动 INTR 进入信号状态，设备又如何驱动 IRQn 进入信号状态？根据配置有以下几种方式：

- (a) 电平触发又分为高电平触发和低电平触发，对于高电平触发，当该信号线(IRQn)被驱动到高电平时，触发中断。对于低电平触发与高电平触发相反，当控制器采样到低电平信号时，触发中断。设备驱动程序的中断处理例程会在处理结束后，通过设备的相关寄存器，向设备发送清中断命令，这时 IRQn 的信号被撤销。
- (b) 边缘触发正常情况下，信号线维持在一根稳定的状态，高电平或者低电平，通过驱动该信号线由高向低跳变或者由低向高跳变进入信号状态。对于这种情况，中断控制器会在采样到跳变时记录下这一中断请求，然后根据优先级和屏蔽情况向 CPU 报告。

2. 外部不可屏蔽中断

无论 CPU 的标志寄存器中的 IF 为何值，CPU 在指令流水的中断检查周期都会检查这一类型的中断。例如系统掉电，这个中断需要立刻响应，进入掉电保护。外部设备通过 CPU 的 NMI 信号线向 CPU 报告这一类别的中断。

狭义上的中断通常就是指上面介绍的外部中断，其中断源来自于 CPU 的外部。但是由于 CPU 内部的一些情况也要利用中断向量表进行目标转移，所以把它们称为内部中断。从 CPU 的角度看，外部中断是一个异步事件，它可能在任何时候发送，而内部中断是一个同步事件，它是执行某条指令时产生的。为了区分子外部中断，内部中断又被称为例外(Exception)。例外又被进一步细分为以下几种。

- 异常(Faults):** CPU 在指令执行时产生的，异常是可以修复的，例如缺页异常，当异常发生时，压入堆栈的是产生异常的那条指令，当 CPU 执行异常处理程序结束后，将重新执行那一条指令。
- 陷阱(Traps):** 在 CPU 执行自陷指令后，立刻通过中断描述符表执行预定的陷阱处理例程，陷阱处理例程执行结束后，将返回陷阱指令的下一条指令继续执行。单步异常或者 INT 3 等都属于陷阱。
- 终止(Aborts):** 严重的错误，发生这一类错误后，其返回地址是不可预知的，系统只能结束执行。

对于不同的中断，在中断初始化和中断处理过程中，其处理方式是不一样的，对于外部中断，除了初始化相关的中断描述符表项外，还要初始化中断控制器。因此区分不同类型的中断是必要的。我们知道，中断向量是 8 位的，那么它一共有 256 项(0~255)，那么系统是如何划分这个向量的呢？处理器使用了 0~31 号作为内部中断，而其余的则可以自由使用，一般通过软件编程配置 8259A 的 IRQ0 使用中断向量 32,IRQ1 使用 32...，直到 46。Linux 使用 Int 0x80 作为系统调用。表6.1是 Intel 保留的中断。

表 6.1 保护模式下 Intel 保留的中断

向量号	助记符	描述	类型	错误码	源
0	#DE	除法错误	Fault	无	DIV 和 IDIV 指令
1	#DB	调试异常	Fault/Trap	无	单步执行，内存断点
2	--	非屏蔽中断	Interrupt	无	非屏蔽中断源 NMI
3	#BP	断点异常	Trap	无	指令 INT 3
4	#OF	溢出	Trap	无	指令 INTO
5	#BR	越界	Fault	无	指令 BOUND
6	#UD	无效操作码	Fault	无	指令 UD2 或者无效指令
7	#NM	设备不可用(无协处理器)	Fault	无	浮点指令或者 WAIT/FWAIT 指令
8	#DF	双重错误	Abort	零	所有能产生异常，NMI 或者 INTR 的指令
9	--	协处理器段越界	Fault	无	浮点指令(386 后的 IA32 处理器不再产生这类异常)
10	#TS	无效的 TSS	Fault	有	任务切换或者访问 TSS
11	#NP	段不存在	Fault	有	加载段寄存器或者访问段
12	#SS	堆栈段错误	Fault	有	堆栈操作或者加载 SS
13	#GP	保护错误	Fault	有	内存或者其他保护检查
14	#PF	页错误	Fault	有	内存访问
15	--	Intel 保留			

续下页

保护模式下 Intel 保留的中断(续表)

向量号	助记符	描述	类型	错误码	源
16	#MF	x87FPU 浮点错误	Fault	无	x87 浮点错误, WAIT/FWAIT 指令
17	#AC	对齐校验	Fault	零	内存访问
18	#MC	Machine check	Abort	无	错误码和源依赖于具体模式(奔腾开始支持)
19	#XF	SIMD 浮点异常	Fault	无	SSE 和 SSE2 浮点指令(奔腾 3 开始支持)
20~31	--	Intel 保留			
32~255	--	用户定义			

保护模式下 Intel 保留的中断(完)

6.2 中断的初始化

现在我们知道内部中断和外部中断的区别了，对于内部中断的初始化，主要是设置中断向量表；而对于外部中断，除了中断向量表外，还要初始化中断控制器，以及中断控制器相关的管理结构。

6.2.1 异常初始化

在系统启动进入保护模式前，内核设置了中断向量表 `idt_table`(见第4.3节)，其中断处理函数都是 `ignore_int()`，在 `start_kernel()` 函数中需要重新设置 `idt_table` 了，这个工作由 `trap_init()` 和 `init_IRQ()` 完成，其中 `init_IRQ()` 除了设置 `idt_table` 外，还需要对中断控制器进行初始化，这里先来看看 `trap_init()`。

代码片段 6.1 节自 `arch/x86/kernel/traps_32.c`

```

1 void __init trap_init(void)
2 {
3     int i;
4
5     set_trap_gate(0, &divide_error);
6     set_intr_gate(1, &debug);
7     set_intr_gate(2, &nmi);
8     /* int3/4 can be called from all */
9     set_system_intr_gate(3, &int3);
10    set_system_gate(4, &overflow);
11    set_trap_gate(5, &bounds);
12    set_trap_gate(6, &invalid_op);

```

```

13 set_trap_gate(7, &device_not_available);
14 set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
15 set_trap_gate(9, &coprocessor_segment_overrun);
16 set_trap_gate(10, &invalid_TSS);
17 set_trap_gate(11, &segment_not_present);
18 set_trap_gate(12, &stack_segment);
19 set_trap_gate(13, &general_protection);
20 set_intr_gate(14, &page_fault);
21 set_trap_gate(15, &spurious_interrupt_bug);
22 set_trap_gate(16, &coprocessor_error);
23 set_trap_gate(17, &alignment_check);
24 #ifdef CONFIG_X86_MCE
25     set_trap_gate(18, &machine_check);
26 #endif
27     set_trap_gate(19, &simd_coprocessor_error);
28
29 .....
30 set_system_gate(SYSCALL_VECTOR, &system_call);
31
32 /* Reserve all the builtin and the syscall vector. */
33 for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
34     set_bit(i, used_vectors);
35 set_bit(SYSCALL_VECTOR, used_vectors);
36 /*
37     * Should be a barrier for any external CPU state.
38 */
39 cpu_init();
40 trap_init_hook();
41 }

```

trap_init()主要调用 set_xxx_gate(), 其中第一个参数是中断向量，第二个参数是中断处理函数，set_xxx_gate()就是按照中断门的格式填写中断向量表的。

代码片段 6.2 节自 include/asm-x86/desc_32.h

```

1 /* present, system, DPL-0, LDT */
2 #define DESCTYPE_LDT 0x82
3
4 /* present, system, DPL-0, 32-bit TSS */
5 #define DESCRIPTYPE_TSS 0x89
6
7 /* present, system, DPL-0, task gate */
8 #define DESCRIPTYPE_TASK 0x85
9

```

```
10 /* present, system, DPL-0, interrupt gate */
11 #define DESCTYPE_INT 0x8e
12
13 /* present, system, DPL-0, trap gate */
14 #define DESCTYPE_TRAP 0x8f
15
16 /* DPL-3 */
17 #define DESCTYPE_DPL3 0x60
18
19 /* !system */
20 #define DESCTYPE_S 0x10
21
22 void set_intr_gate(unsigned int n, void *addr)
23 {
24     _set_gate(n, DESCTYPE_INT, addr, __KERNEL_CS);
25 }
26
27 static inline void set_system_intr_gate(unsigned int n, void *addr)
28 {
29     _set_gate(n, DESCTYPE_INT | DESCTYPE_DPL3, addr, __KERNEL_CS);
30 }
31
32 static void __init set_trap_gate(unsigned int n, void *addr)
33 {
34     _set_gate(n, DESCTYPE_TRAP, addr, __KERNEL_CS);
35 }
36
37 static void __init set_system_gate(unsigned int n, void *addr)
38 {
39     _set_gate(n, DESCTYPE_TRAP | DESCTYPE_DPL3, addr, __KERNEL_CS);
40 }
41
42 static void __init set_task_gate(unsigned int n, unsigned int gdt_entry)
43 {
44     _set_gate(n, DESCTYPE_TASK, (void *)0, (gdt_entry<<3));
45 }
```

`set_xxx_gate()`分别使用相应的 TYPE 来调用 `_set_gate()`，这里以 `set_intr_gate()`为例，它设置中断门，关于中断门的格式见第1.2节。在 `set_intr_gate()`中，其 TYPE 为 `DESCTYPE_INT`，被定义为 `0x8E`，即 `1000 1110`，根据图1.9及表1.2可以看到，最高位存在位为 1，表示在物理内存中，接下来 DPL 为 0，S 位为 0 表示这个描述符是一个系统段描述符，其类型为 `1110` 表示这是中断门。`set_intr_gate()`设置的段选择子为 `__KERNEL_CS`，表示中断处理程序的

地址在内核的代码段中。在下面的 `pack_gate()` 中，`a` 表示 64 位的中断描述符的低 32 位，低 32 位由 16 位的段选择子和中断处理程序入口地址的低 16 位组成，`b` 表示中断描述符的高 32 位，由门类型、门的标志，还有中断处理程序的入口地址的高 16 位组成，`pack_gate()` 的作用就是要凑出一个中断描述符项，最后调用 `write_idt_entry()` 把它写到中断向量表中。

代码片段 6.3 节自 `include/asm-x86/desc_32.h`

```

1 static inline void _set_gate(int gate,
2                             unsigned int type,
3                             void *addr,
4                             unsigned short seg)
5 {
6     __u32 a, b;
7     pack_gate(&a, &b, (unsigned long)addr, seg, type, 0);
8     write_idt_entry(idt_table, gate, a, b);
9 }
```

`_set_gate` 首先调用 `pack_gate` 根据门的类型、入口地址等信息，“组装”一个 64 位的门描述符，保存在 `a,b` 中，然后调用 `write_idt_entry` 把 `a,b` 写入到中断描述符表中。其中 `pack_gate` 定义如下：

代码片段 6.4 节自 `include/asm-x86/desc_32.h`

```

1 static inline void pack_gate(__u32 *a, __u32 *b,
2                             unsigned long base,
3                             unsigned short seg,
4                             unsigned char type,
5                             unsigned char flags)
6 {
7     *a = (seg << 16) | (base & 0xffff);
8     *b = (base & 0xffff0000) | ((type & 0xff) << 8) | (flags & 0xff);
9 }
```

其中 `write_idt_entry` 被定义为 `write_dt_entry`，相关定义如下：

代码片段 6.5 节自 `include/asm-x86/desc_32.h`

```

1 #define write_idt_entry(dt, entry, a, b) write_dt_entry(dt, entry, a, b)
2
3 static inline void write_dt_entry(struct desc_struct *dt,
4                                  int entry,
5                                  u32 entry_low,
6                                  u32 entry_high)
7 {
8     dt[entry].a = entry_low;
```

```

9     dt[entry].b = entry_high;
10 }

```

6.2.2 中断的初始化

接下来就是在 start_kernel()中调用 init_IRQ()对外部中断进行初始化，相对于外部中断来说，init_IRQ()除了设置中断向量表，还需要初始化中断控制器以及相关的管理结构。这里先介绍中断控制器的管理结构。

不同的硬件系统可能拥有不同的中断控制器，因此内核定义了一个 irq_chip 结构来描述中断控制器，该结构定义如下：

代码片段 6.6 节自 include/linux/irq.h

```

1 struct irq_chip {
2     const char *name;
3     unsigned int (*startup)(unsigned int irq);
4     void     (*shutdown)(unsigned int irq);
5     void     (*enable)(unsigned int irq);
6     void     (*disable)(unsigned int irq);
7
8     void     (*ack)(unsigned int irq);
9     void     (*mask)(unsigned int irq);
10    void    (*mask_ack)(unsigned int irq);
11    void    (*unmask)(unsigned int irq);
12    void    (*eoi)(unsigned int irq);
13
14    void    (*end)(unsigned int irq);
15    void    (*set_affinity)(unsigned int irq, cpumask_t dest);
16    int     (*retrigger)(unsigned int irq);
17    int     (*set_type)(unsigned int irq, unsigned int flow_type);
18    int     (*set_wake)(unsigned int irq, unsigned int on);
19
20    /* Currently used only by UML, might disappear one day.*/
21 #ifdef CONFIG_IRQ_RELEASE_METHOD
22     void    (*release)(unsigned int irq, void *dev_id);
23 #endif
24
25     /*
26      * For compatibility, ->typename is copied into ->name.
27      * Will disappear.
28      */
29     const char *typename;
30 };

```

这个结构中就是一些函数指针，无论中断控制器有什么区别，对于内核来说，硬件差异都被屏蔽了。内核无须关心中断控制器之间的硬件差异，中断控制器的驱动模块只需要提供硬件相关的操作函数，然后定义一个 `irq_chip` 对象就可以了，内核调用 `startup` 对中断控制器进行初始化，调用 `enable` 和 `disable` 对中断控制器进行开启和关闭操作，等等。其中 8259A 的中断控制器对象如下：

代码片段 6.7 节自 `arch/x86/kernel/i8259_32.c`

```

1 static struct irq_chip i8259A_chip = {
2     .name      = "XT-PIC",
3     .mask      = disable_8259A_irq,
4     .disable   = disable_8259A_irq,
5     .unmask    = enable_8259A_irq,
6     .mask_ack = mask_and_ack_8259A,
7 };

```

这几个函数分别用于操作 8259A 中断控制器，感兴趣的读者可以参照 8295A 的 datasheet，自行分析这几个函数。

由于同一个外部中断可能被多个外部设备共享，而每个不同的外部设备都可以动态地注册和撤销自己的中断处理例程，所以外部设备的中断不可能像异常处理例程那么简单。内核定义了一个 `irq_desc` 结构，每一个中断需要一个这样的结构来描述。

代码片段 6.8 节自 `include/linux/irq.h`

```

1 struct irq_desc {
2     irq_flow_handler_t handle_irq;
3     struct irq_chip *chip;
4     struct msi_desc *msi_desc;
5     void *handler_data;
6     void *chip_data;
7     /* IRQ action list */
8     struct irqaction *action;
9     /* IRQ status */
10    unsigned int status;
11
12    /* nested irq disables */
13    unsigned int depth;
14    /* nested wake enables */
15    unsigned int wake_depth;
16    /* For detecting broken IRQs */
17    unsigned int irq_count;
18    unsigned int irqsUnhandled;
19    /* Aging timer for unhandled count */
20    unsigned long lastUnhandled;

```

```

21     spinlock_t      lock;
22     .....
23 } __cacheline_internodealigned_in_smp;

```

这个结构中最重要的成员就是 `handle_irq` 和 `action`, `handle_irq` 是一个函数指针, 一个 `irq_desc` 结构中有一个 `action` 链表, 成员 `action` 是一个 `irqaction` 结构体, 定义如下:

代码片段 6.9 节自 `include/linux/interrupt.h`

```

1 struct irqaction {
2     irq_handler_t handler;
3     unsigned long flags;
4     cpumask_t mask;
5     const char *name;
6     void *dev_id;
7     struct irqaction *next;
8     int irq;
9     struct proc_dir_entry *dir;
10 };

```

x86 一共有 256 个中断, 除了 32 个 Intel 保留的内部中断, 还剩下 224 个中断, 内核中定义了一个 `irq_desc` 数组共 224 项, 当发生 n 号外部中断时, 中断处理函数会利用 n 索引到 `irq_desc` 数组的第 n 个 `irq_desc` 成员, 然后调用 `irq_desc` 结构中的 `handle_irq()` 函数。而 `handle_irq()` 又会调用 `action` 中的每一个 `handler` 函数。其简化后的等价代码如下:

代码片段 6.10 `handler_irq` 示例代码

```

1 struct irq_desc *desc = &irq_desc[n];
2 struct irqaction *action = desc->action;
3 do {
4     ret = action->handler(irq, action->dev_id);
5     if (ret == IRQ_HANDLED)
6         status |= action->flags;
7     retval |= ret;
8     action = action->next;
9 } while (action);

```

如果多个设备共享一个中断号, 那么它的 `irq_desc` 中就有多个 `irq_action` 结构, 设备驱动程序调用 `request_irq()` 注册中断时, 函数 `request_irq()` 的工作就是构造一个 `irq_action` 结构, 并把它挂接到对应的 `action` 链表中。从上面的代码可以看出当中断发生时, 它们的中断处理函数都会被调用一次, 那么就有一个问题, 假设 Dev1 和 Dev2 共享中断 n, Dev2 请求中断, 可是 Dev1 和 Dev2 的中断处理函数都得到了执行, 这会有问题吗? 这会降低性能吗? 每一个外部设备都有中断状态寄存器, 而它们的中断处理程序就是读取设备的中断状态寄存器, 然后根据中断状态进行下一步处理, 现在 Dev1 没有请求中断, 所以 Dev1 的中

断处理程序在入口处读取 Dev1 的中断状态寄存器，判断出该设备没有发出过中断请求，所以立即退出了。如果恰巧在 Dev2 发出中断请求后，Dev1 也要请求中断了，那就顺便把 Dev1 的中断处理了。所以不会产生什么问题，对性能的影响也是微乎其微的。

现在相关的管理结构都理清楚了，可以来分析外部中断的初始化了。其初始化工作是在 init_IRQ 中完成的：

代码片段 6.11 节自 arch/x86/kernel/i8259_32.c

```
1 /* init_IRQ 是 native_init_IRQ 的别名。*/
2 void init_IRQ(void) __attribute__((weak, alias("native_init_IRQ")));
3
4 void __init native_init_IRQ(void)
5 {
6     int i;
7     /* 初始化中断控制器及相关管理结构。*/
8     pre_intr_init_hook();
9
10    /* 设置对应的中断向量表。*/
11    for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
12        /*
13         * vector = 32+i, 这是由于中断描述符表中的前 32 项是 Intel 保留给 CPU 内部中断,
14         * trap_init() 已经对它进行了初始化。
15         */
16        int vector = FIRST_EXTERNAL_VECTOR + i;
17        if (i >= NR_IRQS)
18            break;
19
20        /* SYSCALL_VECTOR was reserved in trap_init. */
21        /*
22         * 在 trap_init() 中会为设置好的中断向量设置一个标记, 这里检查这个标记,
23         * 如果设置过, 就不用再设置对应的中断描述符表项了。
24         * 中断处理函数地址保存在 interrupt 数组中。
25         */
26        if (!test_bit(vector, used_vectors))
27            set_intr_gate(vector, interrupt[i]);
28    }
29
30    intr_init_hook();
31    if (boot_cpu_data.hard_math && !cpu_has_fpu)
32        setup_irq(FPU_IRQ, &fpu_irq);
33    irq_ctx_init(smp_processor_id());
34 }
```

init_IRQ 的工作分为两部分，先来看第一部分：

代码片段 6.12 节自 arch/x86/mach-default/setup.c

```

1 void __init pre_intr_init_hook(void)
2 {
3     init_ISA_irqs();
4 }
```

init_ISA_irqs 函数主要是对 8259A 进行初始化设置，定义如下：

代码片段 6.13 节自 arch/x86/mach-default/setup.c

```

1 void __init init_ISA_irqs (void)
2 {
3     int i;
4
5 #ifdef CONFIG_X86_LOCAL_APIC
6     init_bsp_APIC();
7 #endif
8     /*
9      * 对 8259A 芯片进行初始化，其中包括设置起始中断向量为 32，默认情况下
10     * 8259A 的 irq0 对应中断向量 0，但是 0~31 是 Intel 为内部中断保留的，所以
11     * 要把 8259A 的 irq0 映射为 32。请感兴趣的读者参考 8259A 可编程中断控制器
12     * 的 datasheet 自行分析 init_8259A()。
13     */
14     init_8259A(0);
15
16     for (i = 0; i < NR_IRQS; i++) {
17         /*
18          * irq_desc 中的 action 为 NULL，初始状态为 disabled，等到驱动程序调用
19          * request_irq() 函数时，会把相应的 irq_action 结构挂接到 action 链表中。
20          */
21         irq_desc[i].status = IRQ_DISABLED;
22         irq_desc[i].action = NULL;
23         irq_desc[i].depth = 1;
24         /*
25          * 8259A 有 15 个中断源，所以把这 15 个 irq_desc 中的 chip 设置为前面提到的
26          * i8259A_chip，中断处理函数为 handle_level_irq()，就是这个函数负责
27          * 循环调用 action 链表上面的函数。
28          */
29         if (i < 16) {
30             set_irq_chip_and_handler_name(i, &i8259A_chip,
31                                         handle_level_irq,
32                                         "XT");
```

```

33     } else {
34         /* 其他的没有被使用，都设置为默认的 no_irq_chip. */
35         irq_desc[i].chip = &no_irq_chip;
36     }
37 }
38 }
```

现在回到 init_IRQ()函数中，看看它对中断向量表的初始化过程，set_intr_gate(vector, interrupt[i])，用来把 interrupt[i]中保存的函数指针设置到中断向量表中。interrupt 被定义在 arch/x86/kernel/entry_32.S 中：

代码片段 6.14 节自 arch/x86/kernel/entry_32.S

```

1 /* 数据段开始。*/
2 .data
3 ENTRY(interrupt)
4 .text
5 /* 代码段开始。*/
6
7 ENTRY(irq_entries_start)
8     RING0_INT_FRAME
9 vector=0
10 /* 重复 NR_IRQS 次。*/
11 .rept NR_IRQS
12     ALIGN
13     .if vector
14     CFI_ADJUST_CFA_OFFSET -4
15     .endif
16     1: pushl $~(vector)
17     CFI_ADJUST_CFA_OFFSET 4
18     jmp common_interrupt
19     .previous
20     .long 1b
21     .text
22     vector=vector+1
23     .endr
24 END(irq_entries_start)
25
26     .previous
27 END(interrupt)
28     .previous
```

上面这段代码中，使用了大量的宏，为了方便理解，我们来看看预处理后的代码片断：

代码片段 6.15 interrupt 数组

```

1 # 在数据段定义一个 interrupt, 按 4 字节的边界对齐, 不足的用 0x90 填充。
2 .data
3 .globl interrupt; .align 4,0x90; interrupt:
4
5 # 下面是代码段。
6 .text
7
8 .globl irq_entries_start; .align 4,0x90; irq_entries_start:
9   ignore simple; ignore; ignore esp, 3*4; ignore eip, -3*4
10 vector=0
11 # 重复 224 次
12 .rept 224
13 # 按 4 字节的边界对齐, 不足的用 0x90 填充。
14   .align 4,0x90
15   .if vector
16     ignore -4
17   .endif
18   1: pushl $~(vector)
19   ignore 4
20   jmp common_interrupt
21
22 # 进入数据段。
23 .previous
24 # 在数据段中放置上面标号为 1 的地址。
25   .long 1b
26 .text
27   vector=vector+1
28 .endr
29 .size irq_entries_start, .-irq_entries_start
30
31 .previous
32 .size interrupt, .-interrupt
33 .previous

```

上面这段代码告诉汇编器, 在数据段定义一个 interrupt 作为起始, 然后进入代码段, 放置 pushl \$(vector)¹ 和 jmp common_interrupt 两条指令, 指令要求在 4 的边界上对齐, 不足的用 0x90(nop)补齐。然后使用.previous 切换到数据段中, 并在数据段中放置标号为 1 的地址(.long 1b)。.previous 是的作用是在最近的段之间切换。当汇编器处理上面这片代码时, 最开始, 汇编器遇到.data 进入数据段, 然后遇到.text 进入代码段, 然后又遇到.previous 又

¹注意这里的中断号 vector 是从 0 开始的。

进入数据段。.rept 使这个过程一直重复 224 次。最终汇编出来的代码如下：

代码片段 6.16 interrupt 入口

```

1 <irq_entries_start>:
2 # 数据段中的 interrupt[ 0] 指向这里。
3 push $0xffffffff
4 jmp c0104f28 <common_interrupt>
5 nop
6 # 数据段中的 interrupt[ 1] 指向这里。
7 push $0xfffffff
8 jmp c0104f28 <common_interrupt>
9 nop
10 # 数据段中的 interrupt[ 2] 指向这里。
11 push $0xffffffd
12 jmp c0104f28 <common_interrupt>
13 nop
14 .....

```

在 init_IRQ() 函数中，调用 set_intr_gate(vector, interrupt[i]) 来设置中断描述符表，这里数据段中的 interrupt，是不能直接被 C 编译器使用的，因为 i8259_32.c 的代码中.globl interrupt 只能保证在链接期，目标文件 i8259_32.o 中有一个全局标号 interrupt。C 编译器在编译期还不能确定 interrupt 的数据类型，也不知道 interrupt 数组成员的大小²，所以在 hw_irq_32.h 文件中定义了一个指针数组 extern void (*interrupt[NR_IRQS])(void); 它的目的是让 C 编译器知道 interrupt[i] 和 interrupt[i+1] 之间的地址差别是 4 个字节。

从上面的代码可以看出，对于第 0~n 的中断，进入上面的代码后，先把 n 取反后压入堆栈，然后跳转到 common_interrupt 处继续执行。

6.2.3 中断请求服务队列的初始化

对于外部中断来说，设备驱动程序可以调用 request_irq() 把一个中断处理程序挂接到中断请求队列中来，request_irq() 代码如下：

代码片段 6.17 节自 kernel/irq/manage.c

```

1 /* handler 就是设备驱动程序的中断处理函数。*/
2 int request_irq(unsigned int irq,
3                 irq_handler_t handler,
4                 unsigned long irqflags,
5                 const char *devname,
6                 void *dev_id)
7 {

```

² 请读者仔细体会这句话，如果不能理解，则需要补充程序编译、链接方面的知识。

```
8 struct irqaction *action;
9 int retval;
10
11 #ifdef CONFIG_LOCKDEP
12 /*
13  * Lockdep wants atomic interrupt handlers:
14  */
15 irqflags |= IRQF_DISABLED;
16 #endif
17
18 /*
19  * Sanity-check: shared interrupts must pass in a real dev-ID,
20  * otherwise we'll have trouble later trying to figure out
21  * which interrupt is which (messes up the interrupt freeing
22  * logic etc).
23 */
24 if ((irqflags & IRQF_SHARED) && !dev_id)
25     return -EINVAL;
26 if (irq >= NR_IRQS)
27     return -EINVAL;
28 if (irq_desc[irq].status & IRQ_NOREQUEST)
29     return -EINVAL;
30 if (!handler)
31     return -EINVAL;
32
33 /* 为新的 action 结构分配内存。*/
34 action = kmalloc(sizeof(struct irqaction), GFP_ATOMIC);
35 if (!action)
36     return -ENOMEM;
37
38 /* 初始化 action 结构。*/
39 action->handler = handler;
40 action->flags = irqflags;
41 cpus_clear(action->mask);
42 action->name = devname;
43 action->next = NULL;
44 action->dev_id = dev_id;
45 .....
46
47 /* 把这个 action 挂接到对应的 irq_desc 队列中。*/
48 retval = setup_irq(irq, action);
49 if (retval)
```

```
50     kfree(action);
51
52     return retval;
53 }
```

setup_irq 函数，把一个 irqaction 挂接到对应的 irq_desc 队列中，定义如下：

代码片段 6.18 节自 kernel/irq/manage.c

```
1 int setup_irq(unsigned int irq, struct irqaction *new)
2 {
3     struct irq_desc *desc = irq_desc + irq;
4     struct irqaction *old, **p;
5     const char *old_name = NULL;
6     unsigned long flags;
7     int shared = 0;
8
9     if (irq >= NR_IRQS)
10    return -EINVAL;
11     if (desc->chip == &no_irq_chip)
12    return -ENOSYS;
13     .....
14     spin_lock_irqsave(&desc->lock, flags);
15     p = &desc->action;
16     /*
17      * 如果这个 irq 请求队列中已经存在某个设备的中断处理函数，那么需要检查
18      * 旧的处理函数和新的处理函数是否都允许中断共享，只要某一个不允许共享，
19      * 则不能把新的中断处理函数添加到该请求队列中。
20     */
21     old = *p;
22     if (old) {
23         if (!((old->flags & new->flags) & IRQF_SHARED) ||
24             ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK)) {
25             old_name = old->name;
26             goto mismatch;
27         }
28         .....
29         /* add new interrupt at end of irq queue */
30         /* 找到队列尾部。*/
31         do {
32             p = &old->next;
33             old = *p;
34         } while (old);
35         shared = 1;
```

```
36 }
37
38 /* 新的中断处理函数被添加到队列尾部。*/
39 *p = new;
40
41 /* Exclude IRQ from balancing */
42 if (new->flags & IRQF_NOBALANCING)
43     desc->status |= IRQ_NO_BALANCING;
44 /*
45  * 如果没有共享，则说明这是第一个添加到中断请求队列中的中断处理函数，
46  * 这里需要做一些必要的初始化工作。
47 */
48 if (!shared) {
49     irq_chip_set_defaults(desc->chip);
50     .....
51 }
52 /* Reset broken irq detection when installing new handler */
53 desc->irq_count = 0;
54 desc->irqs_unhandled = 0;
55 spin_unlock_irqrestore(&desc->lock, flags);
56
57 new->irq = irq;
58 register_irq_proc(irq);
59 new->dir = NULL;
60 register_handler_proc(irq, new);
61
62 return 0;
63 mismatch:
64 #ifdef CONFIG_DEBUG_SHIRQ
65     if (!(new->flags & IRQF_PROBE_SHARED)) {
66         printk(KERN_ERR "IRQ handler type mismatch for IRQ %d\n", irq);
67         if (old_name)
68             printk(KERN_ERR "current handler: %s\n", old_name);
69         dump_stack();
70     }
71 #endif
72     spin_unlock_irqrestore(&desc->lock, flags);
73     return -EBUSY;
74 }
```

6.3 中断与异常处理

6.3.1 特权转换与堆栈变化

我们知道，在任意时刻 CPU 的当前执行环境可能是内核态，也可能是用户态，而中断处理一定是在内核态进行的。如果 CPU 的当前执行环境是在内核态，而被中断，我们称之为无特权转换；如果 CPU 的当前执行环境是用户态，被中断，则被称之为特权转换。虽然我们总是说中断发生时，CPU 会在堆栈中保存中断现场，等到中断返回时再从堆栈中恢复现场，但是由于不同的执行环境使用独立的堆栈，所以，特权转换的中断和无特权转换的中断的现场有所不同。

其中无特权转换中断和特权转换中断之间的区别可以从图6.2和图6.3中看出。假设当前 CPU 的执行环境为用户态，中断发生时，CPU 会从任务状态段中取出 SS0:ESP0 加载到 SS 和 ESP 寄存器中(见第1.1.2节，图1.7)，这样 CPU 就切换到内核态堆栈，同时会把用户态使用的堆栈 SS:ESP 保存在新的堆栈中。当中断处理结束后，使用 iret 进行中断返回时，CPU 会从当前的堆栈中恢复 CS:EIP 以及 EFLAG，如果返回后的执行环境和当前的执行环境不一致，CPU 还会根据保存下来的 SS:ESP 恢复用户态堆栈。

某些中断有错误号，而某些中断没有错误号，具体情况见表6.1。对于有错误号的情况，CPU 会压入一个错误代码到堆栈中，当使用 iret 指令进行中断返回时，ErrorCode 不会被自动清除，所以有错误号的中断处理程序，在处理结束后，为了保证堆栈平衡，需要清除堆栈中的 ErrorCode。

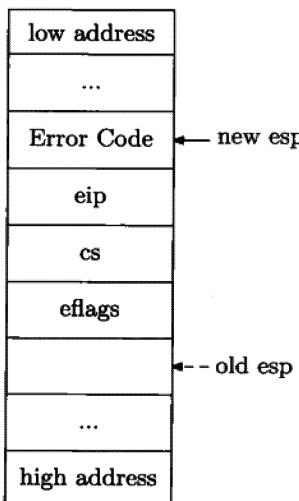


图 6.2 无特权转换中断的堆栈变化

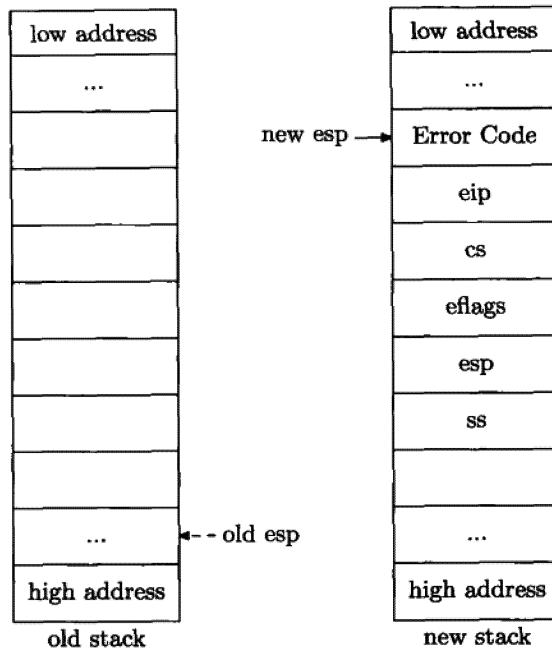


图 6.3 特权转换中断的堆栈变化

6.3.2 中断处理

外部中断 *n* 的入口代码首先在堆栈中压入 *n*, 然后跳转到 common_interrupt 继续执行(见第6.2.2节)。common_interrupt 首先进行现场保护, 然后调用 do_IRQ(), do_IRQ()会根据 irq_desc 结构中的 action 结构链表, 逐个调用中断服务函数。common_interrupt 位于 arch/x86/kernel/entry_32.S 文件中, CFI_XXX 宏是给调试器使用的, 这里为了方便讨论, 把它略去了。

代码片段 6.19 节自 arch/x86/kernel/entry_32.S

```

1 # 此时已经切换到内核态的堆栈。
2 push    $0xffffffff
3 jmp     c0104f28 <common_interrupt>
4 nop
5 push    $0xfffffff
6 jmp     c0104f28 <common_interrupt>
7 nop
8 .....
9 common_interrupt:
10    SAVE_ALL
11    TRACE IRQS_OFF

```

6.3 中断与异常处理

```

12    movl %esp,%eax
13    call do_IRQ
14    jmp ret_from_intr
15 ENDPROC(common_interrupt)
16
17 # SAVE_ALL 的作用就是现场保护。
18 #define SAVE_ALL \
19     cld; \
20     pushl %fs; \
21     pushl %es; \
22     pushl %ds; \
23     pushl %eax; \
24     pushl %ebp; \
25     pushl %edi; \
26     pushl %esi; \
27     pushl %edx; \
28     pushl %ecx; \
29     pushl %ebx; \
30     movl $(__USER_DS), %edx; \
31     movl %edx, %ds; \
32     movl %edx, %es; \
33     movl $(__KERNEL_PERCPU), %edx; \
34     movl %edx, %fs

```

SAVE_ALL 保存现场后，就调用 do_IRQ(), do_IRQ 有一个 pt_regs 的指针参数，先来看看 pt_regs 的定义：

代码片段 6.20 节自 include/asm-x86/ptrace.h

```

1 struct pt_regs {
2     +0x00 long ebx;
3     +0x04 long ecx;
4     +0x08 long edx;
5     +0x0C long esi;
6     +0x10 long edi;
7     +0x14 long ebp;
8     +0x18 long eax;
9     +0x1C int   xds;
10    +0x20 int   xes;
11    +0x24 int   xfs;
12    /* int   xgs; */
13    +0x28 long orig_eax;
14    +0x2C long eip;
15    +0x30 int   xcs;

```

```

16 +0x34 long eflags;
17 +0x38 long esp;
18 +0x3C int  xss;
19 };

```

可以看到 `pt_regs` 结构恰好就是中断发生时，压入堆栈的内容，第13行开始前面的是中断处理程序压入的，后面的是 `eip`, `cs` 等是 CPU 进入中断处理函数之前，自动压入的。其中 `orig_eax` 就是中断号取反后的结果。`do_IRQ()` 是一个 `fastcall` 函数，它的第一个参数使用 `eax` 传递，所以前面调用 `do_IRQ()` 之前，执行了 `movl %esp,%eax`, `esp` 就是堆栈中构造出来的 `pt_regs` 结果指针，`eax` 是传递给 `do_IRQ()` 的参数。

代码片段 6.21 节自 `arch/x86/kernel/irq_32.c`

```

1 fastcall unsigned int do_IRQ(struct pt_regs *regs)
2 {
3     struct pt_regs *old_regs;
4     /* high bit used in ret_from_code */
5     /*
6      * 压入堆栈的 irq 取反得到起始 irq，注意是从 0 开始的，0~31 的中断向量
7      * 是 CPU 内部保留的，从 31 开始，这个向量的作用是找到合适的中断入口程序，
8      * 不要把这个 irq 和中断向量号搞混淆了。
9     */
10    int irq = ~regs->orig_eax;
11    /* 找到 irq_desc 结构。 */
12    struct irq_desc *desc = irq_desc + irq;
13    .....
14    old_regs = set_irq_regs(regs);
15    irq_enter();
16    .....
17    desc->handle_irq(irq, desc);
18
19    irq_exit();
20    set_irq_regs(old_regs);
21    return 1;
22 }

```

在 `init_ISA_irqs()` 中调用 `set_irq_chip_and_handler_name()` 函数把 `desc->handle_irq` 设置为 `handle_level_irq()`(见第6.2.2节)，`handle_level_irq` 负责调用所有该 `irq` 的所有中断服务例程，中断处理结束后，调用 `irq_exit()` 处理延迟的软件中断(见第6.4节)。

代码片段 6.22 节自 `kernel/irq/chip.c`

```

1 void fastcall handle_level_irq(unsigned int irq, struct irq_desc *desc)
2 {

```

```
3     unsigned int cpu = smp_processor_id();
4     struct irqaction *action;
5     irqreturn_t action_ret;
6
7     spin_lock(&desc->lock);
8     /* 向中断控制器发送中断应答命令，并屏蔽该中断。 */
9     mask_ack_irq(desc, irq);
10
11    /* 如果另外一个CPU在处理同一个中断，则退出。 */
12    if (unlikely(desc->status & IRQ_INPROGRESS))
13        goto out_unlock;
14    desc->status &= ~(IRQ_REPLAY | IRQ_WAITING);
15    kstat_cpu(cpu).irqs[irq]++;
16
17    action = desc->action;
18    if (unlikely(!action || (desc->status & IRQ_DISABLED)))
19        goto out_unlock;
20
21    desc->status |= IRQ_INPROGRESS;
22    spin_unlock(&desc->lock);
23
24    action_ret = handle_IRQ_event(irq, action);
25    if (!noirqdebug)
26        note_interrupt(irq, desc, action_ret);
27
28    spin_lock(&desc->lock);
29    desc->status &= ~IRQ_INPROGRESS;
30    if (!(desc->status & IRQ_DISABLED) && desc->chip->unmask)
31        desc->chip->unmask(irq);
32    out_unlock:
33    spin_unlock(&desc->lock);
34 }
```

第9行，向中断控制器发出中断应答，并通知中断控制器屏蔽这条中断请求线，对于某些多处理器来说，第9行可能仅仅是在当前CPU上屏蔽该中断，这取决于不同的中断控制器的具体实现，举例来说假设现在CPU0在处理IRQ5的中断，执行第9行后，此时IRQ5的设备再次发出中断请求，中断控制器并不会通知CPU0(当前CPU的IRQ5被屏蔽)，但是从下面的分析可以看出，内核认为中断控制器此时有可能向另外一个CPU报告该中断。当然，如果此时有更高优先级的中断，中断控制器会立即向该CPU报告。

对于同一个中断来说，假设CPU正在执行某个中断服务程序，在处理结束前，设备再一次发送中断请求，比如网卡在这期间又收到一个网络数据包，虽然已经通知中断控制器

屏蔽了这个中断，但是另外一个 CPU 可能会接手处理这个中断，两个 CPU 同时运行同一个中断处理程序，这是不允许的，假设第一个 CPU 正在对某个资源进行处理，而另外一个 CPU 会竞争该资源。举例来说 CPU1 正执行对某个链表的删除操作，而 CPU2 又来使用该链表中的成员。第12行先判断这个中断的 IRQ_INPROGRESS 是否被设置了，如果没有则进入中断处理并设置 IRQ_INPROGRESS 标志，如果设置了 IRQ_INPROGRESS 就直接退出。对于第二种情况，基于下面的原因，新的中断并不会丢失。

- (1) 设备驱动程序的中断处理例程会根据设备上的中断状态寄存器的指示，一次处理完就绪的中断。举例来说，假设某块网卡接收到一个网络数据包，并中断 CPU1，CPU1 关闭中断并进入网卡驱动程序的中断服务例程。在 CPU1 完成中断处理之前，网卡设备又接收到一个网络数据包，并再次发出中断请求，CPU2 接手处理，它检测到 IRQ_INPROGRESS 标志，于是直接退出了。CPU1 正在执行网卡驱动程序的中断服务例程能够根据网卡中断状态控制器的指示，发现第二个数据包已经接收完毕，正在等待处理。它一次处理完成了所有中断，于是中断服务结束，中断返回。
- (2) 假设 CPU1 已经从网卡的中断服务例程返回到 handle_level_irq()，但是还没来得急撤销 IRQ_INPROGRESS 标志，也就是说第28行还没来得及执行，CPU2 执行到第12行，检测到于是直接退出(CPU2 先于 CPU1 获取到 desc->lock 自旋锁)。这样 CPU1 在开中断后，就会立即接收到这个中断处理，于是第二次接手处理这个中断。这样，同一个中断被严格地串行化了。

 注意，上面的讨论是针对同一个 irq 的情况，对于不同的 irq，根据中断优先级，是可能嵌套执行的，读者不要混淆这两种情况。

之后，内核会调用 handle_IRQ_event()，这个函数将根据 irq 号，循环调用设备驱动程序注册的中断服务处理函数。

代码片段 6.23 节自 kernel/irq/handle.c

```

1 irqreturn_t handle_IRQ_event(unsigned int irq, struct irqaction *action)
2 {
3     irqreturn_t ret, retval = IRQ_NONE;
4     unsigned int status = 0;
5
6     handle_dynamic_tick(action);
7
8     if (!(action->flags & IRQF_DISABLED))
9         local_irq_enable_in_hardirq();
10    /* 循环调用中断服务例程。 */
11    do {
12        ret = action->handler(irq, action->dev_id);
13        if (ret == IRQ_HANDLED)

```

```
14     status |= action->flags;
15     retval |= ret;
16     action = action->next;
17 } while (action);
18
19 /* 外部中断事件可以被看做是一个随机事件，这里是为随机数算法取得更好的随机效果。*/
20 if (status & IRQF_SAMPLE_RANDOM)
21     add_interrupt_randomness(irq);
22 local_irq_disable();
23
24 return retval;
25 }
```

6.3.3 异常处理

相对于中断处理来说，异常处理要简单一些，异常处理不存在动态的多个服务例程，也不需要和外部中断控制器打交道。与中断处理一样，异常处理也会在堆栈上保存一个 pt_regs 结构，现在以 divide_error 为例讨论异常处理。在 trap_init() 中，已经把中断向量 0 的中断入口设置为 divide_error，divide_error 是在 arch/x86/kernel/entry_32.S 文件中定义的：

代码片段 6.24 节自 arch/x86/kernel/entry_32.S

```
1 ENTRY(divide_error)
2     /* 对于某些异常，CPU 会自动压入一个异常代码，为了统一处理，
3      * 对于没有异常号的那些异常，则软件手工压入 0。
4     pushl $0    # no error code
5     pushl $do_divide_error
6     jmp error_code
7 END(divide_error)
8
9 error_code:
10    /* 在堆栈保存现场。*/
11    /* the function address is in %fs's slot on the stack */
12    pushl %es
13    pushl %ds
14    pushl %eax
15    pushl %ebp
16    pushl %edi
17    pushl %esi
18    pushl %edx
19    pushl %ecx
20    pushl %ebx
```

```

21    cld
22    pushl %fs
23    movl $(_KERNEL_PERCPU), %ecx
24    movl %ecx, %fs
25    popl %ecx
26    /*
27     * PT_xx 定义为某个保存在堆栈中某个寄存器的偏移，例如上面 ebx 是最后保存的，
28     * 所以相对于 esp，PT_EBX 就是 0。
29     */
30    # get the function address
31    /* 取出前面压入的 do_divide_error 的地址到 edi 寄存器。*/
32    movl PT_FS(%esp), %edi
33    # get the error code
34    movl PT_ORIG_EAX(%esp), %edx
35    # no syscall to restart
36    movl $-1, PT_ORIG_EAX(%esp)
37    mov %ecx, PT_FS(%esp)
38    movl $(_USER_DS), %ecx
39    movl %ecx, %ds
40    movl %ecx, %es
41
42    # pt_regs pointer
43    movl %esp, %eax
44    /* 调用 do_divide_error. */
45    call *%edi
46    jmp ret_from_exception

```

`do_divide_error` 定义在 `arch/x86/kernel/traps_32.c` 文件中，`do_divide_error` 的调用属性也是 `fastcall`，它有两个参数，第一个参数通过 `eax` 传递，第二个参数通过 `edx` 传递，上面已经把参数准备好了。这个函数是宏展开后得到的结果：

代码片段 6.25 节自 `arch/x86/kernel/traps_32.c`

```

1 #define DO_VM86_ERROR_INFO(trapnr, signr, str, name, sicode, siaddr) \
2 fastcall void do_##name(struct pt_regs * regs, long error_code) \
3 { \
4     siginfo_t info; \
5     info.si_signo = signr; \
6     info.si_errno = 0; \
7     info.si_code = sicode; \
8     info.si_addr = (void __user *)siaddr; \
9     trace_hardirqs_fixup(); \
10    if (notify_die(DIE_TRAP, str, regs, error_code, \
11                  trapnr, signr) == NOTIFY_STOP) \

```

```
12     return; \
13     do_trap(trapnr, signr, str, 1, regs, error_code, &info); \
14 }
15
16 DO_VM86_ERROR_INFO(0, SIGFPE,
17                     "divide error",
18                     divide_error,
19                     FPE_INTDIV,
20                     regs->eip)
```

`do_divide_error()`准备一个 `siginfo_t` 结构后，调用 `do_trap()` 函数，许多异常都要进入 `do_trap()`。`do_trap()` 需要根据异常现场的 `cs` 寄存器判断异常是在内核态还是在用户态产生的，并做如下处理：

1. 如果是运行在内核态发生的异常，则尝试利用内核异常处理链表修复该异常，如果失败，则打印 `oops`。
2. 如果是运行在用户态发生的异常，则需要向该进程发送 `SIGFPE` 信号(见第7章)。

`do_trap()` 的具体代码如下：

代码片段 6.26 节自 `arch/x86/kernel/traps_32.c`

```
1 static void __kprobes do_trap(int trapnr,
2                                 int signr,
3                                 char *str,
4                                 int vm86,
5                                 struct pt_regs *regs,
6                                 long error_code,
7                                 siginfo_t *info)
8 {
9     /* 当前进程。*/
10    struct task_struct *tsk = current;
11    .....
12    /* 判断异常发生时的运行模式。*/
13    if (!user_mode(regs))
14        goto kernel_trap;
15
16    /* 用户模式，向用户进程发送信号(见第7章。)*/
17    trap_signal: {
18        tsk->thread.error_code = error_code;
19        tsk->thread.trap_no = trapnr;
20
21        if (info)
```

```

22     force_sig_info(signr, info, tsk);
23     else
24         force_sig(signr, tsk);
25     return;
26 }
27 /* 内核模式，尝试利用内核异常处理链修复异常。*/
28 kernel_trap: {
29     if (!fixup_exception(regs)) {
30         tsk->thread.error_code = error_code;
31         tsk->thread trap_no = trapnr;
32         die(str, regs, error_code);
33     }
34     return;
35 }
36 .....
37 }

```

6.4 软件中断与延迟函数

中断是一个异步事件，因此中断服务例程通常是在任意进程环境中执行的，也就是说中断处理例程的开发者不能假设当前的执行环境是某一个进程。另外，由于驱动程序可以注册中断服务例程，因此内核不能保证中断服务例程总是在开中断的情况下执行，即便是开中断执行，低优先级的中断仍然是被屏蔽的。为了提高系统的相应性能，我们希望中断服务例程尽快处理完紧急的工作，然后立即退出。假设一块网卡接收到了数据包，网卡驱动程序把这个数据包从网卡接收队列复制到一个接收队列中，便立即退出。而进行协议分析，协议栈处理的工作则被延迟。这些可延迟的工作由 softirq 和 tasklet 来完成。

6.4.1 softirq

softirq 被称为软中断，它也有自己的优先级，每次中断处理结束时会检查是否有 softirq 等待处理。每一个 softirq 用一个 softirq_action 结构来表示，内核定义了 softirq_action 数组，数组下标越小，其优先级越高。softirq 结构如下：

代码片段 6.27 节自 include/linux/interrupt.h

```

struct softirq_action
{
    /* 延迟函数。*/
    void (*action)(struct softirq_action *);
    /* 参数。*/
}

```

```

void *data;
};

/* 32 个软中断。*/
static struct softirq_action softirq_vec[32] __cacheline_aligned_in_smp;

```

内核使用 open_softirq() 注册一个延迟函数。

代码片段 6.28 节自 kernel/softirq.c

```

1 void open_softirq(int nr, void (*action)(struct softirq_action*),
2                   void *data)
3 {
4     softirq_vec[nr].data = data;
5     softirq_vec[nr].action = action;
6 }

```

值得注意的是，open_softirq() 操作 softirq_vec 结构时，没有任何保护措施，比如自旋锁等。这是因为内核保证 open_softirq() 初始化后就不再改变，内核启动时调用 open_softirq() 注册的 softirq 如表 6.2 所示。

表 6.2 内核 softirq

softirq	优先级	处理函数	描述
HI_SOFTIRQ	0	tasklet_hi_action	处理高优先级 tasklet
TIMER_SOFTIRQ	1	run_timer_softirq	处理定时器的 tasklet
NET_TX_SOFTIRQ	2	net_tx_action	发送网络数据包
NET_RX_SOFTIRQ	4	net_rx_action	网络数据包接收
BLOCK_SOFTIRQ	5	blk_done_softirq	块设备读写
TASKLET_SOFTIRQ	6	tasklet_action	处理低优先级 tasklet
SCHED_SOFTIRQ	7	run_rebalance_domains	多 CPU 任务队列平衡
HRTIMER_SOFTIRQ	8	run_hrtimer_softirq	高精度时钟

在中断服务例程中通过调用 raise_softirq() 请求软件中断，来完成某些非紧急任务。每一个 CPU 都有一个 irq_cpustat_t 类型的结构，该结构的成员 __softirq_pending 用来指示是否有软中断等待处理，raise_softirq() 函数通过把 __softirq_pending 的第 n 位置 1 触发 softirq_vec 数组中的第 n 个软中断。每当中断处理结束时，都会根据当前 cpu 的 __softirq_pending 的状态来判断是否有响应的软中断等待处理。

代码片段 6.29 节自 kernel/softirq.c

```

1 [do_IRQ() -> irq_exit()]
2
3 void irq_exit(void)

```

```

4 {
5     account_system_vtime(current);
6     trace_hardirq_exit();
7     sub_preempt_count(IRQ_EXIT_OFFSET);
8
9     if (!in_interrupt() && local_softirq_pending())
10      invoke_softirq();
11
12     .....
13     preempt_enable_no_resched();
14 }

```

如果一个低优先级的中断进入 do_IRQ(), 但是在执行到 irq_exit()之前, 一个高优先级的中断打断了当前执行流程, 当高优先级中断处理结束时, 必然要返回到打断的低优先级中断处继续执行, 此时返回的是中断环境, 第9行先保证不在中断环境中处理 softirq。local_softirq_pending()通过检查当前 cpu 的 __softirq_pending 来判断是否有延迟的 softirq, 如果有, 则调用 invoke_softirq():

代码片段 6.30 节自 kernel/softirq.c

```

1 [do_IRQ() -> irq_exit() -> do_softirq()]
2
3 asmlinkage void do_softirq(void)
4 {
5     __u32 pending;
6     unsigned long flags;
7
8     if (in_interrupt())
9         return;
10    local_irq_save(flags);
11    pending = local_softirq_pending();
12    if (pending)
13        __do_softirq();
14
15    local_irq_restore(flags);
16 }

```

__do_softirq 定义如下:

代码片段 6.31 节自 kernel/softirq.c

```

1 [do_IRQ() -> irq_exit() -> do_softirq() -> __do_softirq()]
2 asmlinkage void __do_softirq(void)
3 {
4     .....

```

```
5     pending = local_softirq_pending();
6 restart:
7     /* Reset the pending bitmask before enabling irqs */
8     set_softirq_pending(0);
9     local_irq_enable();
10    h = softirq_vec;
11
12    do {
13        if (pending & 1) {
14            h->action(h);
15            rcu_bh_qsctr_inc(cpu);
16        }
17        h++;
18        pending >>= 1;
19    } while (pending);
20
21    local_irq_disable();
22    pending = local_softirq_pending();
23    if (pending && --max_restart)
24        goto restart;
25
26    if (pending)
27        wakeup_softirqd();
28    .....
29 }
```

第5行起先把 `_softirq_pending` 赋值给 `pending` 变量，接着把 `_softirq_pending` 清零。在 `do-while` 循环中，如果 `pending` 的第 n 位为 1，则调用一次 `softirq_vec[n]->action`，但是每一次调用该函数所消耗的时间却不是固定的，以 `net_rx_action()` 为例，接收队列中的包越多，则 `net_rx_action()` 消耗的时间越多。当网络流量过大时，接收中断和软中断都会同时增加，那么就有可能在处理 `softirq` 时被中断，并且中断服务例程会再次请求 `softirq`，于是在第22行再次判断是否有软中断等待处理。如果有软中断等待处理，就跳转到 `restart` 处继续处理。软中断和中断服务例程一样，也是在任意进程环境中执行的，所以不能在软中断环境中进行进程调度，在中断和软中断持续过多的情况下，如果不加控制地跳转到 `restart` 处理软中断，那么系统中的其他进程会因长期得不到调度而“饥饿”，从而导致系统的交互响应性下降。如果处理一定次数就直接返回，虽然可以保证交互响应性，但是会带来新的问题。假设处理了 N 次后，还有软中断等待处理，但是直接返回了，假设之后的一段时间间隙中没有中断，那么即便 CPU 空闲着，队列中的软件中断还是得不到及时处理。为了解决这个问题，新的内核为每一个 CPU 创建了一个 `ksoftirqd` 内核线程，专门为该 CPU 处理软件中断，如果重复了 `max_restart` 次，还有软件中断等待处理，则调用 `wakeup_softirqd()` 唤

醒 ksoftirqd 内核线程，然后退出。ksoftirqd 线程和系统中其他进程公平竞争处理器，从而实现在保证交互响应性的前提下处理软件中断。现在来看看 ksoftirqd()。

代码片段 6.32 节自 kernel/softirq.c

```
1 static int ksoftirqd(void * __bind_cpu)
2 {
3     /* 把 ksoftirqd 内核线程的状态设置为 TASK_INTERRUPTIBLE. */
4     set_current_state(TASK_INTERRUPTIBLE);
5
6     while (!kthread_should_stop()) {
7         preempt_disable();
8         /* 如果没有软件中断需要处理，则调用 schedule() 主动让出 CPU. */
9         if (!local_softirq_pending()) {
10             preempt_enable_no_resched();
11             schedule();
12             preempt_disable();
13         }
14         /* 当需要时，ksoftirqd 被唤醒，并继续执行到这里，现在把状态设置为 TASK_RUNNING. */
15         __set_current_state(TASK_RUNNING);
16
17         while (local_softirq_pending()) {
18             /* CPU 热插拔支持。*/
19             if (cpu_is_offline((long) __bind_cpu))
20                 goto wait_to_die;
21             /* 处理软件中断。*/
22             do_softirq();
23             preempt_enable_no_resched();
24             cond_resched();
25             preempt_disable();
26         }
27         preempt_enable();
28         /* 处理结束，把状态设置为 TASK_INTERRUPTIBLE. */
29         set_current_state(TASK_INTERRUPTIBLE);
30     }
31     __set_current_state(TASK_RUNNING);
32     return 0;
33
34 wait_to_die:
35     .....
36     return 0;
37 }
```

从上面的分析中可以看到，软中断的运行过程中没有使用自旋锁保护，因此多个 CPU 可能在同一时刻执行同一个软中断函数，这就要求软中断函数必须是可重入的，它要么全部使用 percpu 变量，或者自己使用自旋锁对关键资源进行保护。为了保证内核的健壮性，内核做了最坏的打算，它“不相信”其他模块驱动等能满足这个要求，所以软中断不能在运行中动态创建与删除。从表6.2可以看到，内核使用的 softirq 都是启动时创建的，内核保证这几个函数能够满足以上要求，如果其他的模块驱动要使用延迟函数，则需要使用要求更低的 tasklet。

6.4.2 tasklet

tasklet 是建立在 softirq 的基础上的，表6.2中的 tasklet_hi_action()和 tasklet_action()就是专门处理 tasklet 的，前者的优先级高于后者。每一个 CPU 都有独立的 tasklet 队列，所以不需要使用自旋锁保护 tasklet，从而降低了对延迟函数的要求。内核通过 tasklet 向驱动程序模块提供动态延迟函数接口。结构 tasklet_struct 用来描述一个 tasklet 对象，其定义如下：

代码片段 6.33 节自 include/linux/interrupt.h

```
1 struct tasklet_struct
2 {
3     struct tasklet_struct *next;
4     unsigned long state;
5     atomic_t count;
6     void (*func)(unsigned long);
7     unsigned long data;
8 };
```

每一个 CPU 都有一个 tasklet_hi_vec 和 tasklet_vec 变量，分别是高优先级和普通优先级 tasklet 的队列头。tasklet_init()用于初始化一个 tasklet_struct 结构，之后可以调用 tasklet_hi_schedule()或 tasklet_schedule()来请求延迟调用一个 tasklet，前者把一个指定的 tasklet 添加到高优先级的 tasklet 队列，后者添加到普通优先级 tasklet 队列。

代码片段 6.34 节自 kernel/softirq.c

```
1 void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
2                   unsigned long data)
3 {
4     t->next = NULL;
5     t->state = 0;
6     atomic_set(&t->count, 0);
7     t->func = func;
8     t->data = data;
9 }
```

`tasklet_hi_schedule` 函数将调用 `_tasklet_hi_schedule` 完成实际的工作，定义如下：

代码片段 6.35 节自 `kernel/softirq.c`

```

1 [tasklet_hi_schedule() -> __tasklet_hi_schedule()]
2
3 void fastcall __tasklet_hi_schedule(struct tasklet_struct *t)
4 {
5     unsigned long flags;
6
7     local_irq_save(flags);
8     t->next = __get_cpu_var(tasklet_hi_vec).list;
9     __get_cpu_var(tasklet_hi_vec).list = t;
10    raise_softirq_irqoff(HI_SOFTIRQ);
11    local_irq_restore(flags);
12 }
```

`tasklet_hi_schedule()` 函数把一个 `tasklet` 添加到 CPU 的 `HI_SOFTIRQ` 队列中，然后再调用 `raise_softirq_irqoff()` 请求软中断。当调度到软件中断时，就会执行 `tasklet_hi_action()` 函数。

代码片段 6.36 节自 `kernel/softirq.c`

```

1 static void tasklet_hi_action(struct softirq_action *a)
2 {
3     struct tasklet_struct *list;
4
5     local_irq_disable();
6     list = __get_cpu_var(tasklet_hi_vec).list;
7     __get_cpu_var(tasklet_hi_vec).list = NULL;
8     local_irq_enable();
9
10    while (list) {
11        struct tasklet_struct *t = list;
12        list = list->next;
13        /* 保证某一时刻，最多只有一个 CPU 执行同一个 tasklet 函数。*/
14        if (tasklet_trylock(t)) {
15            if (!atomic_read(&t->count)) {
16                if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
17                    BUG();
18                t->func(t->data);
19                tasklet_unlock(t);
20                continue;
21            }
22            tasklet_unlock(t);
```

```
23     }
24
25     local_irq_disable();
26     t->next = __get_cpu_var(tasklet_hi_vec).list;
27     __get_cpu_var(tasklet_hi_vec).list = t;
28     __raise_softirq_irqoff(HI_SOFTIRQ);
29     local_irq_enable();
30 }
31 }
```

6.5 中断与异常返回

在前面已经看到，中断处理结束后，会通过 `jmp ret_from_intr` 返回，而异常处理结束则通过 `jmp ret_from_exception` 返回。中断返回前，还需要完成以下工作：

(1) 处理延迟的调度请求假设有 Ph 和 Pl 两个进程，进程 Ph 的优先级大于进程 Pl，现在进程 Ph 通过系统调用(比如 `sys_read`)进入内核态，它进行磁盘 IO 读操作，但是磁盘缓存队列中没有现成的数据，于是磁盘驱动程序向磁盘设备发出读请求，此时进程 Ph 主动放弃 CPU，进入等待队列。然后进程 Pl 被调度运行，片刻后，磁盘完成读操作并向 CPU 发出中断，磁盘中断服务例程通知调度器把进程 Ph 重新加入就绪队列，调度器此时发现当前就绪队列中的进程 Ph 的优先级大于当前进程 Pl，于是请求调度 Ph，但是延迟到中断处理结束。此时进程 Pl 没有主动放弃 CPU，而是被进程 Ph 抢占了 CPU 资源。进程 Ph 的调度时机分为以下两种情况：

- 中断发生时进程 Pl 运行在用户态环境，此时中断处理结束后，在返回进程 Pl 用户态前，内核先检查到这个延迟的调度请求，于是通知调度器重新调度，调度器根据调度算法选择进程 Ph 运行。
- 中断发生时进程 Pl 运行在内核态，此时如果内核编译时配置了 `CONFIG_PREEMPT`，则直接通知调度器重新调度；如果没有配置 `CONFIG_PREEMPT`，则不允许在内核态发生抢占，于是恢复中断现场，返回到进程 Pl 的内核态继续执行。当进程 Pl 在内核态由于请求某种资源得不到及时满足而主动放弃 CPU，或者进程 Pl 要返回用户态前时，这个延迟的调度请求才会被处理，于是进程 Ph 才会获取到 CPU 资源。可以看出 `CONFIG_PREEMPT` 把抢占的时机从用户态扩充到了内核态。

为什么调度器发现更高优先级的进程进入就绪队列时，不立即调度高优先级的进程运行，而要延迟到中断处理结束呢？这个要求是由多方面的原因造成的，这里举例说明：为了降低对中断服务例程的要求，试想设备驱动注册的中断服务例程各式各样，内核不能假定它们的行为，如果在中断处理例程中允许调度，那么就可能某个设备驱动程序的中断处理函数，不小心把中断关闭后，就切换到另外一个进程，此后CPU接收不到任何可屏蔽中断，没有了中断，操作系统还能正常运行吗？另外在中断服务例程中，有更加紧急的指令需要执行，比如设备DMA的内存队列满了，需要分配新的内存等。

(2) 延迟的信号处理(见第7章)假设p1为shell，p2是该shell的一个前台进程，现在p2通过系统调用(比如sys_read)进入内核态读取数据，但是目标设备没有准备好，于是p2进入等待队列，一段时间后，用户无法忍受一个长时间毫无反应的进程占用自己的shell，于是按下ctrl+c键，此时通过键盘中断，shell进程p1捕捉到这个消息，根据控制台的默认设置，p1需要向p2发送SIGINT信号，同时为了让p2有机会处理该信号，进程p2会进入就绪队列，等待某个合适的时机，轮到p2执行时，p2会执行自己的默认进程退出函数³。于是p2结束了，同理，如果p2是在后台运行，可以通过发送SIGKILL信号来迫使它执行退出函数。

(3) Virtual-8086 处理当前CPU是在虚拟8086模式下执行的，我们不讨论这种情况。

在明白上述背景知识后，再来看中断返回的代码就容易得多了，ret_from_intr和ret_from_exception都是entry_32.S文件中定义的，下面的分析假设内核配置了CONFIG_PREEMPT编译选项：

代码片段 6.37 节自 arch/x86/kernel/entry_32.S

```

1 ret_from_exception:
2     # 关中断，对于x86平台宏展开后就是cli指令。
3     preempt_stop(CLBR_ANY)
4     # 对于中断返回，中断已经是关闭的。
5 ret_from_intr:
6     # 把当前进程的thread_info结构指针存入ebp寄存器。
7     GET_THREAD_INFO(%ebp)
8     # 根据中断现场的cs和eip寄存器判断中断前夕运行的状态。
9 check_userspace:
10    # mix EFLAGS and CS
11    movl PT_EFLAGS(%esp), %eax
12    movb PT_CS(%esp), %al
13    andl $(VM_MASK | SEGMENT_RPL_MASK), %eax
14    cmpl $USER_RPL, %eax
15    # not returning to v8086 or userspace

```

³假设进程p2没有为SIGINT设置自定义的信号处理函数。

```
16    # 返回内核态。
17    jb resume_kernel
18    .....
19    # 返回内核态。
20 #ifdef CONFIG_PREEMPT
21 ENTRY(resume_kernel)
22    # 关闭中断。
23    DISABLE_INTERRUPTS(CLBR_ANY)
24    # 如果当前进程的 thread_info 结构的 preempt_count 为 0，则允许内核态抢占。
25    # non-zero preempt_count ?
26    cmpl $0, TI_preempt_count(%ebp)
27
28    # 如果不允许内核态抢占则中断返回。
29    jnz restore_nocheck
30
31 need_resched:
32    # 如果当前进程的 thread_info 结构的 flag 成员设置了 TIF_NEED_RESCHED,
33    # 则说明有延迟的调度请求需要处理。
34    # need_resched set ?
35    movl TI_flags(%ebp), %ecx
36    testb $TIF_NEED_RESCHED, %cl
37
38    # 没有延迟的调度请求。
39    jz restore_all
40
41    # interrupts off (exception path) ?
42    testl $IF_MASK, PT_EFLAGS(%esp)
43    jz restore_all
44
45    # 处理延迟的调度请求。
46    call preempt_schedule_irq
47    jmp need_resched
48 END(resume_kernel)
49#endif
```

如果执行到第42行，则说明在返回内核态之前有一个延迟的调度请求等待处理，但是中断和异常都可能导致内核执行到这里，如果是在内核态的中断处理中发生了异常，而异常处理程序很可能修复了这个异常，根据前面的讨论，中断处理更为紧急，因此这种情况下，需要暂时抛开这个延迟的调度请求，返回中断处理函数中继续执行。问题的关键就是如何准确地判断出这种情况呢？第42行的原理就是：判断“中断”前的现场中的 eflags，如果 IF=0，说明该“中断”前中断是关闭的，就说明是中断处理函数中触发的一个异常进

入这里，那么直接跳转到 `restore_all` 处返回到中断处理程序中继续执行，等中断处理结束，执行中断返回时，再处理延迟的调度请求。

 获取自旋锁也会关闭中断，假设某段非中断处理内核代码获取了自旋锁后的某个时刻发生了异常，也会导致第42行暂时不处理延迟调度请求，但是没有关系，能够关闭中断的代码一定是运行在内核态的，从内核态返回到用户态时，这个延迟调度请求无论如何都会被处理。

如果顺利调用 `preempt_schedule_irq()`，则当前进程被放入就绪队列并让出 CPU，某个时候再次调度这个进程时，它从第47行再次跳回 `need_resched` 处，再次检查是否有延迟的调度请求，重复这个过程，直到顺利地从 `restore_all` 返回。

现在来看看返回到用户态的情况：

代码片段 6.38 节自 `arch/x86/kernel/entry_32.S`

```

1 # 返回用户态，处理延迟的调度请求，以及延迟的信号。
2 ENTRY(resume_userspace)
3     LOCKDEP_SYS_EXIT
4     DISABLE_INTERRUPTS(CLBR_ANY)
5
6     # 检测当前进程是否有延迟的请求等待处理。
7     movl TI_flags(%ebp), %ecx
8     andl $_TIF_WORK_MASK, %ecx
9     # 处理延迟请求。
10    jne work_pending
11    # 恢复现场，中断返回。
12    jmp restore_all
13 END(ret_from_exception)
14
15 # 处理延迟的调度请求和延迟的信号。
16 work_pending:
17     testb $_TIF_NEED_RESCHED, %cl
18     # 如果没有延迟的调度请求，则跳转。
19     jz work_notifysig
20 work_resched:
21     call schedule
22     LOCKDEP_SYS_EXIT
23     DISABLE_INTERRUPTS(CLBR_ANY)
24
25     TRACE_IRQS_OFF
26     movl TI_flags(%ebp), %ecx
27     andl $_TIF_WORK_MASK, %ecx
28     jz restore_all

```

```
29     testb $_TIF_NEED_RESCHED, %cl
30     jnz work_resched
31
32 work_notifysig:
33 # deal with pending signals and notify-resume requests
34 .....
35     movl %esp, %eax
36     xorl %edx, %edx
37 # 处理延迟的信号，这里略去了 VM86 的代码。
38     call do_notify_resume
39     jmp resume_userspace
40 END(work_pending)
41
42 # 信号处理完后返回到这里。
43 ENTRY(resume_userspace)
44     LOCKDEP_SYS_EXIT
45     DISABLE_INTERRUPTS(CLBR_ANY)
46     movl TI_flags(%ebp), %ecx
47     andl $_TIF_WORK_MASK, %ecx
48 # 再次循环处理延迟的信号，最后从 restore_all 返回。
49     jne work_pending
50     jmp restore_all
51 END(ret_from_exception)
```

当延迟的调度请求和信号都处理完毕后，再次调度到这个进程时(如果有请求调度)，会从 `restore_all` 处返回，`restore_all` 的作用就是恢复 `SAVE_ALL` 保存的中断现场，并执行中断返回。

6.6 中断优先级回顾

通过前面的分析，我们已经知道，CPU 首先处理优先级最高的中断，当全部中断都处理完毕后，就处理延迟的软件中断，然后处理延迟的调度请求，最后在返回用户态前会检查当前进程是否有延迟的信号等待处理，最后返回用户态。可以看到在中断处理过程中，有一个隐含的优先级。但是 Linux 内核中没有抽象出一个直观统一的优先级概念，受 Windows 内核⁴的启发，我们来对这个处理过程进行更加直观的总结，我们把统一的优先级称为 IRQL，每一个级别有一个对应的 IRQL，优先级越高，对应的 IRQL 越高。

对于内核来说，假设当前有两个设备同时发出中断请求，中断控制器向 CPU 报告优先级高的中断，低优先级的中断被延迟，内核把当前运行级别提升到高优先级的 IRQL。在高

⁴见《深入理解 Windows 操作系统》。

优先级中断处理例程中，一个更高优先级的进程的IO请求被满足，或者内核向某个进程发送信号，于是该进程进入就绪队列，由于在中断和软中断的IRQL运行级别都不能进行任务切换。因此内核此时会请求延迟调度，当前中断处理结束后，CPU发送EIO命令通知中断控制器中断处理结束，此时，中断控制器立刻向CPU报告延迟的低优先级中断，CPU把当前运行级别降低到低优先级中断，当中断处理全部结束后，IRQL降到softirq的级别，softirq处理结束后⁵，IRQL进一步降低，此时内核处理延迟的进程调度，调度器会根据调度算法将CPU分配给每一个进程，当该进程返回用户态时，内核处理该进程的信号，最后返回用户态继续执行。

6.7 关于高级可编程中断控制器

由于8259A只适用于单处理器，为了满足多处理器的需求，后来设计了高级可编程中断控制器⁶。大多数x86平台都包含了APIC，和PIC一样，Intel的南桥ICH系列中也包含了APIC的兼容功能，如图6.4所示。每一个CPU内部有一个本地APIC，本地APIC可以产生时钟中断，可以向其他的处理器发送处理器间中断等。系统中可以有一个或多个I/O APIC，每个I/O APIC支持24个中断输入信号，I/O APIC和Local APIC之间通过总线连接。

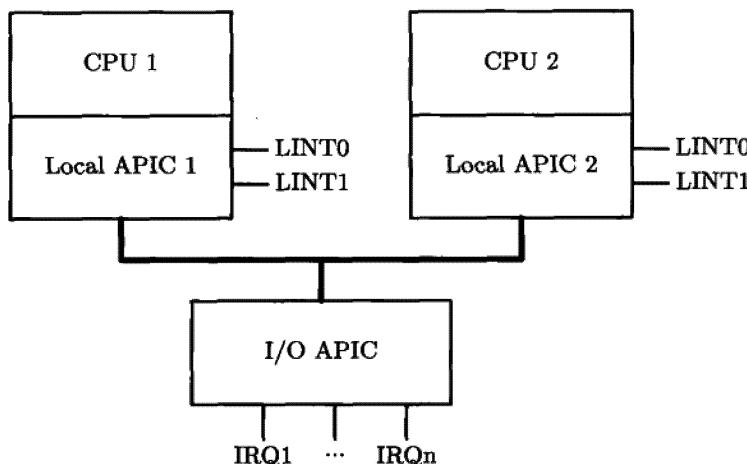


图 6.4 高级可编程中断控制器

外部设备可以通过I/O APIC的IRQn请求中断，而Local APIC中的LINT0和LINT1的作用是可软件配置的，通常在没有I/O APIC的情况下，LINT0被当做INTR用于连接8259A的中断请求信号，而LINT1被用做NMI。APIC中的每个LINTn和IRQn分别有一个

⁵softirq不一定被全部处理，也可能是请求调度了ksoftirqd进程。

⁶即 Advanced Programmable Interrupt Controller(APIC)。

64位配置寄存器，被称为 Interrupt Redirection Table。这些寄存器被映射到内存地址空间(见第5.1节)。配置寄存器的最低八位就是该中断输入信号对应的中断向量，另外通过配置寄存器还可以控制中断触发方式，中断需要向哪一个CPU或者全部CPU报告。关于详细的配置信息请参考《Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide》和《Intel 82093AA I/O Advanced Programmable Interrupt Controller》。

启动阶段调用 init_ISA_irqs()对8259A进行了初始化(见第6.2节)，但是如果随后系统检测到APIC的支持，就会根据APIC的配置情况，重新设置相关中断向量表，并关闭8259A。

6.7.1 APIC 初始化

系统启动阶段需要根据配置，重新设置中断向量等相关信息。这些配置是硬件相关的，为此Intel制定了MultiProcessor Specification。它规定了主板设计者必须把硬件相关的信息按照统一的数据格式集成到BIOS中，这样操作系统就可以通过BIOS获取到相关信息。如果内核配置了CONFIG_X86_SMP, CONFIG_X86_LOCAL_APIC以及CONFIG_X86_IO_APIC，那么启动时可以通过命令行参数apic=debug看到APIC的详细信息。APIC的初始化过程如下：

代码片段 6.39 节自 arch/x86/kernel/mpparse_32.c

```
1 [setup_arch() -> setup_memory() -> setup_bootmem_allocator() ->
2   find_smp_config() -> find_smp_config()]
3
4 void __init find_smp_config (void)
5 {
6     unsigned int address;
7
8     if (smp_scan_config(0x0, 0x400) ||
9         smp_scan_config(639*0x400, 0x400) ||
10        smp_scan_config(0xF0000, 0x10000))
11     return;
12     address = get_bios_ebda();
13     if (address)
14         smp_scan_config(address, 0x400);
15 }
```

BIOS会提供一个Intel中规定的MP table，它的头4个字节为_MP_，smp_scan_config()就是根据在BIOS数据区寻找这个表，如果找到，则打印found SMP MP-table at xxx。之后内核在setup_arch()中调用acpi_boot_table_init()，acpi_boot_init()和get_smp_config()，这几个函数根据MultiProcessor Specification的规定对MP table进行分析，同时会调用acpi_table_print_madt_entry()打印出分析出来的APIC的相关信息。关于MP table的细节请

参考 MultiProcessor Specification，这里以一个常见的配置输出为例，介绍 APIC 的典型配置。

代码片段 6.40 APIC 初始化信息

```

1 .....
2 ACPI: Local APIC address 0xfee00000
3
4 local apic 的 id 为 0 并已启用。
5 ACPI: LAPIC (acpi_id[0x00] lpic_id[0x00] enabled)
6 CPU0 的 Local APIC 版本为 2.0.
7 Processor #0 6:14 APIC version 20
8 local apic 的 id 为 1 并已启用。
9 ACPI: LAPIC (acpi_id[0x01] lpic_id[0x01] enabled)
10 CPU1 的 Local APIC 版本为 2.0.
11 Processor #1 6:14 APIC version 20
12 CPU0 和 CPU1 的 Local APIC 中的 LINT1 用于 NMI 信号连接，并且上升沿有效。
13 ACPI: LAPIC_NMI (acpi_id[0x00] high edge lint[0x1])
14 ACPI: LAPIC_NMI (acpi_id[0x01] high edge lint[0x1])
15 I/O APIC 的 id, 地址, 版本号。
16 ACPI: IOAPIC (id[0x01] address[0xfec00000] gsi_base[0])
17 IOAPIC[0]: apic_id 1, version 32, address 0xfec00000, GSI 0-23
18 ACPI: INT_SRC_OVR (bus 0 bus_irq 0 global_irq 2 dfl dfl)
19 ACPI: INT_SRC_OVR (bus 0 bus_irq 9 global_irq 9 high level)
20 ACPI: IRQ0 used by override.
21 ACPI: IRQ2 used by override.
22 ACPI: IRQ9 used by override.
23
24 系统中有一个 I/O APIC，其模式为 Flat。
25 Enabling APIC mode: Flat. Using 1 I/O APICs
26 ACPI: HPET id: 0x8086a201 base: 0xfed00000
27 Using ACPI (MADT) for SMP configuration information
28 .....

```

根据上面的信息，Local APIC 和 I/O APIC 配置寄存器的物理地址分别是 0xfee00000 和 0xfec00000，但是内核需要访问的是虚拟地址，因此内核需要为 APIC 建立页表映射。

代码片段 6.41 节自 arch/x86/kernel/apic_32.c

```

1 [trap_init() -> init_apic_mappings()]
2
3 void __init init_apic_mappings(void)
4 {
5     unsigned long apic_phys;
6

```

```

7   if (!smp_found_config && detect_init_APIC()) {
8       apic_phys = (unsigned long) alloc_bootmem_pages(PAGE_SIZE);
9       apic_phys = __pa(apic_phys);
10  } else
11      apic_phys = mp_lapic_addr;
12
13  set_fixmap_nocache(FIX_APIC_BASE, apic_phys);
14  printk(KERN_DEBUG "mapped APIC to %08lx (%08lx)\n",
15         APIC_BASE, apic_phys);
16
17  if (boot_cpu_physical_apicid == -1U)
18      boot_cpu_physical_apicid = GET_APIC_ID(apic_read(APIC_ID));
19
20 #ifdef CONFIG_X86_IO_APIC
21  {
22      unsigned long ioapic_phys, idx = FIX_IO_APIC_BASE_0;
23      int i;
24      for (i = 0; i < nr_ioapics; i++) {
25          .....
26          set_fixmap_nocache(idx, ioapic_phys);
27          printk(KERN_DEBUG "mapped IOAPIC to %08lx (%08lx)\n",
28                 __fix_to_virt(idx), ioapic_phys);
29          idx++;
30      }
31  }
32 #endif
33 }
```

接下来会调用 `apic_intr_init()` 为 Local APIC 设置中断向量。

代码片段 6.42 节自 `arch/x86/kernel/apic_32.c`

```

1 [start_kernel() -> init_IRQ() -> intr_init_hook() -> apic_intr_init()]
2
3 #define LOCAL_TIMER_VECTOR 0xef
4 #define SPURIOUS_APIC_VECTOR 0xff
5 #define ERROR_APIC_VECTOR 0xfe
6 #define THERMAL_APIC_VECTOR 0xf0
7
8 void __init apic_intr_init(void)
9 {
10 #ifdef CONFIG_SMP
11     smp_intr_init();
12 #endif
```

```

13  /* self generated IPI for local APIC timer */
14  set_intr_gate(LOCAL_TIMER_VECTOR, apic_timer_interrupt);
15
16  /* IPI vectors for APIC spurious and error interrupts */
17  set_intr_gate(SPURIOUS_APIC_VECTOR, spurious_interrupt);
18  set_intr_gate(ERROR_APIC_VECTOR, error_interrupt);
19
20  /* thermal monitor LVT interrupt */
21 #ifdef CONFIG_X86_MCE_P4THERMAL
22  set_intr_gate(THERMAL_APIC_VECTOR, thermal_interrupt);
23#endif
24 }

```

这里为 Local APIC 设置了一些新的中断处理函数，例如：在中断向量 0xef 设置了中断入口函数 `apic_timer_interrupt()`，接下来对 Local APIC 中的配置寄存器进行配置后，这样当 Local APIC 产生中断时，就会调用对应的中断处理函数。

代码片段 6.43 节自 `arch/x86/kernel/apic_32.c`

```

1 int __init APIC_init_uniprocessor (void)
2 {
3     .....
4     verify_local_APIC();
5     connect_bsp_APIC();
6     .....
7     /* 设置 Local APIC. */
8     setup_local_APIC();
9
10    /* 设置 I/O APIC. */
11 #ifdef CONFIG_X86_IO_APIC
12     if (smp_found_config)
13         if (!skip_ioapic_setup && nr_ioapics)
14             setup_IO_APIC();
15#endif
16    /* 配置 Local APIC 的时钟中断。*/
17    setup_boot_clock();
18
19    return 0;
20 }

```

由于在 `init_ISA_irqs()` 中已经对 8259A 进行了初始化，现在 `setup_IO_APIC()` 要设置新的中断并关闭 8259A。

代码片段 6.44 节自 `arch/x86/kernel/io_apic_32.c`

```

1 static void __init setup_IO_APIC_irqs(void)
2 {
3     .....
4     if (IO_APIC_IRQ(irq)) {
5         vector = assign_irq_vector(irq);
6         entry.vector = vector;
7         ioapic_register_intr(irq, vector, IOAPIC_AUTO);
8
9         /* 关闭 8259A 的 15 个中断源。*/
10        if (!apic && (irq < 16))
11            disable_8259A_irq(irq);
12    }
13    .....
14 }

```

ioapic_register_intr 定义如下：

代码片段 6.45 节自 arch/x86/kernel/io_apic_32.c

```

1 static void ioapic_register_intr(int irq,
2                                 int vector,
3                                 unsigned long trigger)
4 {
5     if ((trigger == IOAPIC_AUTO &&
6          IO_APIC_irq_trigger(irq)) ||
7         trigger == IOAPIC_LEVEL) {
8         irq_desc[irq].status |= IRQ_LEVEL;
9         set_irq_chip_and_handler_name(irq,
10                                     &ioapic_chip,
11                                     handle_fasteoi_irq,
12                                     "fasteoi");
13     } else {
14         irq_desc[irq].status &= ~IRQ_LEVEL;
15         set_irq_chip_and_handler_name(irq,
16                                     &ioapic_chip,
17                                     handle_edge_irq,
18                                     "edge");
19     }
20     set_intr_gate(vector, interrupt[irq]);
21 }

```

这里 set_irq_chip_and_handler_name() 函数和 interrupt 数组已经在 init_IRQ() 中介绍过了(见第6.2.2节)，重新设置了 irq_desc 数组，需要注意的是，APIC 的中断向量是软件可配置的，虽然关闭了 8259A，但是它并没有使用 8259A 的中断向量，例如 8259A 中时钟中断

使用的向量是 0x20，而 Local APIC 的时钟使用的是 0xef，在重新设置的过程中，并没有覆盖 8259A 的中断向量表项。最后，系统使用的就是 APIC 的中断了。例如：

代码片段 6.46 节自/proc/interrupts

CPU0	CPU1	
0: 554459	0	IO-APIC-edge timer
1: 15716	0	IO-APIC-edge i8042
8: 7	0	IO-APIC-edge rtc
9: 23545	0	IO-APIC-fasteoi acpi
12: 675	0	IO-APIC-edge i8042
14: 13270	0	IO-APIC-edge libata
15: 35820	0	IO-APIC-edge libata
16: 10433	0	IO-APIC-fasteoi uhci_hcd: usb3
17: 2	0	IO-APIC-fasteoi uhci_hcd: usb1
18: 0	0	IO-APIC-fasteoi uhci_hcd: usb2
19: 1	0	IO-APIC-fasteoi uhci_hcd: usb4
20: 2964	0	IO-APIC-fasteoi eth1
21: 646	0	IO-APIC-fasteoi HDA Intel
NMI: 0	0	Non-maskable interrupts
LOC: 27442	275378	Local timer interrupts
RES: 230202	241475	Rescheduling interrupts
CAL: 78	171	function call interrupts
TLB: 93	370	TLB shootdowns
TRM: 0	0	Thermal event interrupts
SPU: 0	0	Spurious interrupts
ERR: 0		
MIS: 0		

第7章 信号机制

信号机制是类 UNIX 系统中的一种重要的进程间通信手段之一。我们经常使用信号来向一个进程发送一个简短的消息。例如：假设我们启动一个进程通过 socket 读取远程主机发送过来的网络数据包，此时由于网络因素当前主机还没有收到相应的数据，当前进程被设置为可中断等待状态(TASK_INTERRUPTIBLE)，此时我们已经失去耐心，想提前结束这个进程，于是可以通过 kill 命令向这个进程发送 KILL 信号，内核会唤醒该进程，执行它的信号处理函数，KILL 信号的默认处理是退出该进程。当然并不是一定要进程处于 TASK_INTERRUPTIBLE 状态时，才能够处理信号。

另外应用程序可以通过 signal() 等函数来为一个信号设置默认处理函数。例如当用户按下 CTRL+C 时，shell 将会发送 SIGINT 信号，SIGINT 的默认处理函数是执行进程的退出代码，但是下面的例子把 SIGINT 的响应函数设置为 int_handler。

代码片段 7.1 signal 示例代码

```
1 #include <signal.h>
2 #include <stdio.h>
3
4 void int_handler()
5 {
6     printf("SIGINT signal handler.\n");
7 }
8
9 int main()
10 {
11     signal(SIGINT, int_handler);
12     printf("int_handler set for SIGINT\n");
13
14     while (1) {
15         printf("go to sleep.\n");
16         sleep(60);
17     }
18
19     printf("Exit.\n");
```

```
20     return 0;
21 }
```

当执行上面这段代码时，按下 CTRL+C 后，会执行 `int_handler()` 这个函数，进程并不会退出，首先当执行到 `sleep()` 时，进程进入可中断等待状态，之后如果按下 CTRL+C，进程会被唤醒执行 `SIGINT` 的处理函数 `int_handler()`，为了结束这个进程，只能使用 `kill` 命令来发送 `SIGKILL` 信号。

信号的分发和处理是在内核态进行的，但是从上面的这个例子中可以看出，信号的处理函数可能是在用户态，在这种情况下，内核需要内核态构建一个临时的用户态环境，然后调用用户态的信号处理函数。

7.1 信号机制的管理结构

信号只是一个数字，数字 0~31 表示不同的信号，如表 7.1 所示。`task_struct` 结构中的 `blocked` 有点类似于中断屏蔽寄存器，如表 7.1 所示。

表 7.1 信号列表

编号	信号名	默认动作	说明
1	<code>SIGHUP</code>	进程终止	终端断开连接
2	<code>SIGINT</code>	进程终止	用户在键盘上按下 CTRL+C
3	<code>SIGQUIT</code>	进程意外结束(Dump)	用户在键盘上按下 CTRL+\
4	<code>SIGKILL</code>	进程意外结束(Dump)	遇到非法指令
5	<code>SIGTRAP</code>	进程意外结束(Dump)	遇到断点，用于调试
6	<code>SIGABRT</code>	进程意外结束(Dump)	
6	<code>SIGIOT</code>	进程意外结束(Dump)	
7	<code>SIGBUS</code>	进程意外结束(Dump)	总线错误
8	<code>SIGFPE</code>	进程意外结束(Dump)	浮点异常
9	<code>SIGKILL</code>	进程终止	其他进程发送 <code>SIGKILL</code> 将导致目标进程终止
10	<code>SIGUSR1</code>	进程终止	应用程序可自定义使用
11	<code>SIGSEGV</code>	进程意外结束(Dump)	非法的内存访问
12	<code>SIGUSR2</code>	进程终止	应用程序可自定义使用
13	<code>SIGPIPE</code>	进程终止	管道读取端已经关闭，写入端进程会收到该信号
14	<code>SIGALRM</code>	进程终止	定时器到期
15	<code>SIGTERM</code>	进程终止	发送该信号使目标进程终止
16	<code>SIGSTKFLT</code>	进程终止	堆栈错误
17	<code>SIGCHLD</code>	忽略	子进程退出时会向父进程发送该信号

续下页

信号列表(续表)

编号	信号名	默认动作	说明
18	SIGCONT	忽略	进程继续执行
19	SIGSTOP	进程暂停	发送该信号会使目标进程进入 TASK_STOPPED 状态
20	SIGTSTP	进程暂停	在终端上按下 CTRL+Z
21	SIGTTIN	进程暂停	后台进程从控制终端读取数据
22	SIGTTOU	进程暂停	后台进程向控制终端写入数据
23	SIGURG	忽略	socket 收到设置紧急指针标志的网络数据包
24	SIGXCPU	进程意外结束(Dump)	进程使用 CPU 已经超过限制
25	SIGXFSZ	进程意外结束(Dump)	进程使用文件已经超过限制
26	SIGVTALRM	进程终止	进程虚拟定时器到期
27	SIGPROF	进程终止	进程 Profile 定时器到期
28	SIGMNCH	忽略	进程终端窗口大小改变
29	SIGIO	进程终止	用于异步 IO
29	SIGPOLL	进程终止	用于异步 IO
30	SIGPWR	进程终止	电源失效
31	SIGUNUSED	进程终止	保留未使用

信号列表(完)

注意在表7.1中的默认动作是指，在没有任何程序为相应的信号设置信号处理函数的情况下，内核接收到该信号的默认处理方式，但是在实际中，有可能不是这样处理的。另外，在这里，进程终止一般是指进程通过 `do_exit()` 退出，而进程意外结束(Dump)则表示进程遇到了一个异常。默认情况下，内核会根据进程当的内存信息，在进程的当前目录中生成一个 Core Dump 文件，以后用户可以通过这个文件来分析进程异常的原因。这个工作主要通过 `do_coredump()` 来完成。

由于早期只有 31 个信号，内核仅仅使用一个 32 位的变量 `signal` 来表示进程接收到的信号，因此如果要向一个进程发送一个信号，就把 `singal` 的第 n 位设置为 1，这非常类似于中断状态寄存器的功能(见第6章)，同时，还有一个 `blocked` 的变量，用来做信号屏蔽，这类似于中断屏蔽寄存器。这样做好处是可以“很快”判断出一个进程收到了哪些信号，如果采用数组或链表，则需要扫描整个队列，但是这也带来了新的问题，如果向一个进程发送了 SIGINT 信号，在这个信号处理之前，再次发送 SIGINT，当这个进程开始处理信号时，它只知道接收到了 SIGINT，而无法判断出有几个 SIGINT 需要处理。此后加入了信号队列，把收到的信号保存在这个队列中，就可以解决这个问题了。但是为了兼容的目的，仍然保留了旧的信号处理方式，因此 1~32 还是按原有的方式进行处理，而 33~64 则使用新的机制，为了区别对待编号为 33~64 的信号又被称为实时信号。需要注意的是：这里的“实时”和实时操作系统中的“实时”没有任何联系，实时信号在处理速度上并不会比普

通信号快，它们之间的区别就是，普通信号会对多次的同一信号进行“合并”处理，而实时信号则一一处理。因此我们在这里仅仅对普通信号进行讨论。

信号机制的相关管理结构位于 `task_struct` 结构中，其主要结构如图7.1所示。

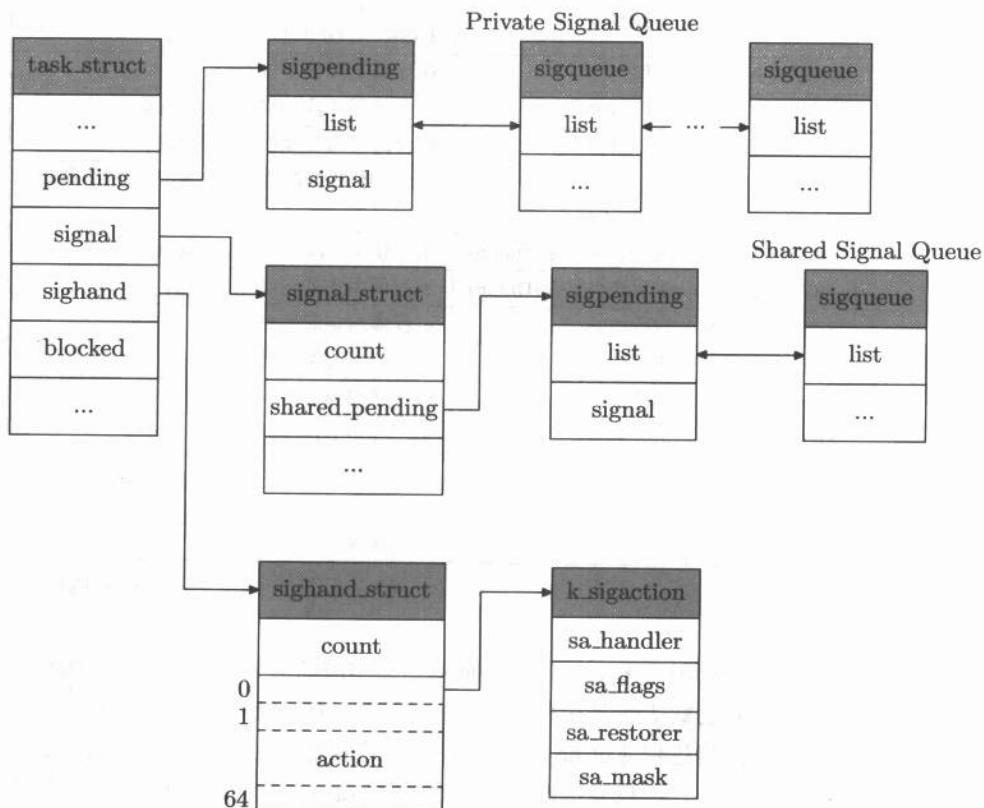


图 7.1 信号管理相关结构示意图

实时信号引入了信号队列，为了处理上的方便，普通信号也使用信号队列，仅仅从数字上来区分实时信号还是普通信号。由于在 Linux 中，进程对象和线程对象都是 `task_struct`，因此需要区别对待线程的信号和进程的信号。在图7.1中，`Private Signal Queue` 是线程信号队列(在 Linux 中被称为轻权进程)，而 `Shared Signal Queue` 是进程的信号队列(在 Linux 中被称为进程组)。对于进程信号，则由进程组中的每一个线程(轻权进程)共享。在图7.1中，`pending` 和 `shared_peding` 分别为 `Private Sigaln Queue` 和 `Shared Signal Queue`。其类型都是 `sigpending`，定义如下：

代码片段 7.2 节自 `include/linux/signal.h`

```

1 struct sigpending {
2     struct list_head list;
  
```

```

3     sigset_t signal;
4 };

```

list 用于连接信号队列，而 signal 是一个位图，每一位表示一个对应的信号，用于指示信号队列中有哪些信号等待处理，其类型为 sigset_t，定义如下：

代码片段 7.3 节自 include/asm/signal.h

```

1 typedef struct {
2     unsigned long sig[ 2];
3 } sigset_t;

```

sigset_t 是一个数组，总共 64 位，对应 64 个信号位图(32 个普通信号，32 个实时信号)。将来在信号发送的分析中，我们会看到，对于普通信号，只需要把 sigset_t 中对应的位置 1 就可以了，而对于实时信号，还需要把信号的相关信息添加到 list 的信号队列中，信号队列类型为 sigqueue，定义如下：

代码片段 7.4 节自 include/linux/signal.h

```

1 struct sigqueue {
2     struct list_head list;
3     int flags;
4     siginfo_t info;
5     struct user_struct *user;
6 };

```

sigqueue 中的 list 是队列链表指针，info 为这个信号的相关信息，主要成员定义如下：

代码片段 7.5 节自 include/asm-generic/siginfo.h

```

1 typedef struct siginfo {
2     /* 信号 */
3     int si_signo;
4     int si_errno;
5     int si_code;
6     .....
7 }

```

图7.1中的 sighthand 保存信号处理函数指针，它的作用类似于中断向量表(见第6章)。类型为 sighthand_struct，定义如下：

代码片段 7.6 节自 include/linux/sched.h

```

1 struct sighthand_struct {
2     atomic_t count;
3     struct k_sigaction action[ _NSIG ];
4     spinlock_t siglock;

```

```

5     wait_queue_head_t signalfd_wqh;
6 };

```

这里 _NSIG 为 64, 数组 action, 对应 64 个信号处理函数的相关信息, 类型为 k_sigaction, 在 x86 平台中, k_sigaction 是 sigaction 的一个包装, sigaction 定义如下:

代码片段 7.7 节自 include/asm-x86/signal.h

```

1 struct sigaction {
2     __sighandler_t sa_handler;
3     unsigned long sa_flags;
4     __sigrestore_t sa_restorer;
5
6     /* mask last for extensibility */
7     sigset_t sa_mask;
8 };

```

sa_handler 就是信号处理函数指针。另外在 task_struct 结构中还有一个 blocked 可以用来做信号屏蔽。

明白了上面的主要数据结构的作用之后, 很容易想到信号的处理主要有以下几方面。

- 设置信号回调函数: 内核把函数的相关信息保存到对应的 sigaction 结构中。
- 信号的发送: 通过相关系统调用把一个指定的信号发送到目标进程, 如果该信号没有被屏蔽, 就把信号的相关信息添加到信号队列中, 如果有必要就唤醒目标进程。
- 信号响应: 进程被唤醒后, 根据信号队列中的信息, 调用信号回调函数。

现在看来, 信号机制是不是不再复杂了呢? 接下来我们将继续对以上几方面进行详细讨论。

7.2 信号发送

应用程序发送信号时, 主要通过 kill 进行的。没错, 不要被"kill"迷惑, 它并不是发送 SIGKILL 信号专用函数。这个函数通过系统调用 sys_kill()进入内核, 它接收两个参数:

- (1) 第一个参数为目标进程 id, kill()可以向进程(进程组), 线程(轻权进程)发送信号, 因此 pid 有以下几种情况。

- pid > 0: 目标进程(可能是轻权进程)由 pid 指定。
- pid = 0: 信号被发送到当前进程组中的每一个进程。
- pid = -1: 信号被发送到任何一个进程, init 进程(PID=1)和以及当前进程无权发送信号的进程除外。

- pid < -1: 信号被发送到目标进程组，其组 id 由参数中 pid 的绝对值指定。

(2) 第二个参数为需要发送的信号。

由于 sys_kill() 处理的情况比较多，我们从 tkill() 入手，这个函数把信号发送到由参数 pid 指定的线程(轻权进程)中。tkill() 的内核入口是 sys_tkill()，定义如下：

代码片段 7.8 节自 kernel/signal.c

```
1 asmlinkage long
2 sys_tkill(int pid, int sig)
3 {
4     /* This is only valid for single tasks */
5     if (pid <= 0)
6         return -EINVAL;
7
8     return do_tkill(0, pid, sig);
9 }
```

sys_tkill() 函数仅仅是对参数 pid 进行检查，之后调用 do_tkill()，代码如下：

代码片段 7.9 节自 kernel/signal.c

```
1 static int do_tkill(int tgid, int pid, int sig)
2 {
3     int error;
4     struct siginfo info;
5     struct task_struct *p;
6
7     /* 根据参数初始化一个 siginfo 结构。*/
8     error = -ESRCH;
9     info.si_signo = sig;
10    info.si_errno = 0;
11    info.si_code = SI_TKILL;
12    info.si_pid = task_tgid_vnr(current);
13    info.si_uid = current->uid;
14
15    read_lock(&tasklist_lock);
16    /* 获取由 pid 指定的线程的 task_struct 结构。*/
17    p = find_task_by_vpid(pid);
18
19    if (p && (tgid <= 0 || task_tgid_vnr(p) == tgid)) {
20
21        /* 权限检查。*/
22        error = check_kill_permission(sig, &info, p);
23    }
24
25    if (error)
26        goto out;
27
28    /* 将信息写入到 sigqueueinfo 结构中。*/
29    info.si_value.sival_int = sig;
30    info.si_info.si_signo = sig;
31    info.si_info.si_errno = 0;
32    info.si_info.si_code = SI_TKILL;
33    info.si_info.si_pid = task_tgid_vnr(current);
34    info.si_info.si_uid = current->uid;
35
36    /* 将信息写入到 task_struct 结构中。*/
37    p->sиг = &info;
38    p->sigval = sig;
39
40    /* 通知线程。*/
41    wake_up_process(p);
42
43    /* 释放锁。*/
44    read_unlock(&tasklist_lock);
45
46    /* 返回结果。*/
47    return error;
48
49    /* 错误处理。*/
50    out:
51        /* 释放锁。*/
52        read_unlock(&tasklist_lock);
53        /* 返回错误。*/
54        return error;
55 }
```

```

23     /*
24      * The null signal is a permissions and
25      * process existence probe.
26      * No signal is actually delivered.
27     */
28     if (!error && sig && p->sighand) {
29         spin_lock_irq(&p->sighand->siglock);
30         handle_stop_signal(sig, p);
31         error = specific_send_sig_info(sig, &info, p);
32         spin_unlock_irq(&p->sighand->siglock);
33     }
34 }
35 read_unlock(&tasklist_lock);
36
37 return error;
38 }

```

在 do_tkill() 函数中，如果顺利地通过了权限检查，就调用 handle_stop_signal() 对某些特殊信号进行处理，例如当收到 SIGSTOP 时，需要把信号队列中的 SIGCONT 全部删除。在此之后，再调用 specific_send_sig_info() 把信号加入到信号队列。

代码片段 7.10 节自 kernel/signal.c

```

1 static int specific_send_sig_info(int sig,
2                                 struct siginfo *info,
3                                 struct task_struct *t)
4 {
5     int ret = 0;
6
7     BUG_ON(!irqs_disabled());
8     assert_spin_locked(&t->sighand->siglock);
9
10    /* 信号被忽略。*/
11    /* Short-circuit ignored signals. */
12    if (sig_ignored(t, sig))
13        goto out;
14
15    /* Support queueing exactly one non-rt signal, so that we
16       can get more detailed information about the cause of
17       the signal. */
18    if (LEGACY_QUEUE(&t->pending, sig))
19        goto out;
20
21    ret = send_signal(sig, info, t, &t->pending);

```

```
22     if (!ret && !sigismember(&t->blocked, sig))
23         signal_wake_up(t, sig == SIGKILL);
24     out:
25     return ret;
26 }
```

首先调用 `sig_ignored()` 检查信号是否被忽略，然后检查发送的信号是不是普通信号，如果是普通信号，就需要根据信号位图来检查当前信号队列中是否存在该信号，如果已经存在，对于普通信号不需要做任何处理。然后调用 `send_signal()` 来完成实际的发送工作，`send_signal()` 是信号发送的重点，除 `sys_tkill()` 之外的函数，最终都是通过 `send_signal()` 来完成信号发送工作的。这里我们注意到向 `send_signal()` 传递的参数是 `t->pending`，也就是连接 `Private Signal Queue` 的那条链(见图7.1)。最后，如果发送成功就调用 `signal_wake_up()` 来唤醒目标进程，这样就可以保证该进程进入就绪状态，从而有机会被调度执行信号处理函数。

现在我们来看看 `send_signal()` 函数，这个函数的主要工作就是分配并初始化一个 `sigqueue` 结构，然后把它添加到信号队列中。

代码片段 7.11 节自 `kernel/signal.c`

```
1 static int send_signal(int sig,
2                         struct siginfo *info,
3                         struct task_struct *t,
4                         struct sigpending *signals)
5 {
6     struct sigqueue * q = NULL;
7     int ret = 0;
8
9     /*
10      * Deliver the signal to listening signalfds.
11      * This must be called with the sighand lock held.
12      */
13     signalfd_notify(t, sig);
14
15     /*
16      * fast-pathed signals for kernel-internal things
17      * like SIGSTOP or SIGKILL.
18      */
19     if (info == SEND_SIG_FORCED)
20         goto out_set;
21
22     /* Real-time signals must be queued if sent by sigqueue, or
23      * some other real-time mechanism. It is implementation
24      * defined whether kill() does so. We attempt to do so, on
25      * the principle of least surprise, but since kill is not
```

```
26     allowed to fail with EAGAIN when low on memory we just
27     make sure at least one signal gets delivered and don't
28     pass on the info struct. */
29
30     /* 分配 sigqueue 结构。*/
31     q = __sigqueue_alloc(t, GFP_ATOMIC,
32                           (sig < SIGRTMIN &
33                            (is_si_special(info) ||
34                             info->si_code >= 0)));
35
36     if (q) {
37         /* 如果成功分配到 sigqueue 结构，就把它添加到队列中，并对其进行初始化。*/
38         list_add_tail(&q->list, &signals->list);
39         switch ((unsigned long) info) {
40             case (unsigned long) SEND_SIG_NOINFO:
41                 q->info.si_signo = sig;
42                 q->info.si_errno = 0;
43                 q->info.si_code = SI_USER;
44                 q->info.si_pid = task_pid_vnr(current);
45                 q->info.si_uid = current->uid;
46                 break;
47             case (unsigned long) SEND_SIG_PRIV:
48                 q->info.si_signo = sig;
49                 q->info.si_errno = 0;
50                 q->info.si_code = SI_KERNEL;
51                 q->info.si_pid = 0;
52                 q->info.si_uid = 0;
53                 break;
54             default:
55                 /* 拷贝 sigqueue 结构。*/
56                 copy_siginfo(&q->info, info);
57                 break;
58         }
59     } else if (!is_si_special(info)) {
60         if (sig >= SIGRTMIN && info->si_code != SI_USER)
61             /*
62             * Queue overflow, abort. We may abort if
63             * the signal was rt and sent by user using
64             * something other than kill().
65             */
66         return -EAGAIN;
67     }
```

```
68
69 out_set:
70 /* 设置信号位图。*/
71 sigaddset(&signals->signal, sig);
72 return ret;
73 }
```

从前面的讨论中，我们看到当信号被添加到信号队列之后，会调用 `signal_wake_up()` 唤醒这个进程，`signal_wake_up()` 定义如下：

代码片段 7.12 节自 `kernel/signal.c`

```
1 void signal_wake_up(struct task_struct *t, int resume)
2 {
3     unsigned int mask;
4
5     /* 为进程设置 TIF_SIGPENDING 标志。*/
6     set_tsk_thread_flag(t, TIF_SIGPENDING);
7
8     /*
9      * For SIGKILL, we want to wake it up in the stopped/traced
10     * case. We don't check t->state here because there is a
11     * race with it executing another processor and just now
12     * entering stopped state. By using wake_up_state, we
13     * ensure the process will wake up and handle its death
14     * signal.
15     */
16
17     mask = TASK_INTERRUPTIBLE;
18     if (resume)
19         mask |= TASK_STOPPED | TASK_TRACED;
20     if (!wake_up_state(t, mask))
21         kick_process(t);
22 }
```

`signal_wake_up()` 首先为进程设置 `TIF_SIGPENDING` 标志，来说明该进程有延迟的信号等待处理。然后再调用 `wake_up_state()` 唤醒目标进程，如果目标进程已经在其他的 CPU 上运行，`wake_up_state()` 将返回 0，此时调用 `kick_process()` 向该 CPU 发送一个处理器间中断。通过第 6 章的讨论我们知道，当中断返回前夕，会为当前进程处理延迟的信号。

此后当该进程被调度时，在进程返回用户态空间前，会调用 `do_notify_resume()` 来处理该进程的信号（见第 6 章）。

7.3 信号处理

当进程被调度时，会调用 `do_notify_resume()` 来处理信号队列中的信号。信号处理主要就是调用 `sighand_struct` 结构中对应的信号处理函数。`do_notify_resume()` 函数定义如下：

代码片段 7.13 节自 `arch/x86/kernel/signal_32.c`

```
1 static void fastcall do_signal(struct pt_regs *regs)
2 {
3     siginfo_t info;
4     int signr;
5     struct k_sigaction ka;
6     sigset_t *oldset;
7     /*
8      * We want the common case to go fast, which
9      * is why we may in certain cases get here from
10     * kernel mode. Just return without doing anything
11     * if so. vm86 regs switched out by assembly code
12     * before reaching here, so testing against kernel
13     * CS suffices.
14     */
15
16     if (!user_mode(regs))
17         return;
18
19     if (test_thread_flag(TIF_RESTORE_SIGMASK))
20         oldset = &current->saved_sigmask;
21     else
22         oldset = &current->blocked;
23
24     signr = get_signal_to_deliver(&info, &ka, regs, NULL);
25     if (signr > 0) {
26         /* Re-enable any watchpoints before delivering the
27          * signal to user space. The processor register will
28          * have been cleared if the watchpoint triggered
29          * inside the kernel.
30         */
31         if (unlikely(current->thread.debugreg[7]))
32             set_debugreg(current->thread.debugreg[7], 7);
33
34         /* Whee! Actually deliver the signal. */
35         if (handle_signal(signr, &info, &ka, oldset, regs) == 0) {
36             /* a signal was successfully delivered; the saved
```

```
37     * sigmask will have been stored in the signal frame,
38     * and will be restored by sigreturn, so we can simply
39     * clear the TIF_RESTORE_SIGMASK flag */
40 if (test_thread_flag(TIF_RESTORE_SIGMASK))
41     clear_thread_flag(TIF_RESTORE_SIGMASK);
42 }
43
44 return;
45 }
46 .....
47 }
```

执行 do_signal() 函数时，进程一定是处于内核空间，通常进程只有通过中断或者系统调用才能进入内核空间，regs 保存着系统调用或者中断时的现场(见第6章)。在第16行中，调用 user_mode() 根据 regs 中的 CS 寄存器来判断，中断现场环境是不是用户态环境，如果不是，就不对信号进行任何处理，直接从 do_signal() 函数返回。这是为什么呢？如果 user_mode() 发现 regs 的现场是内核态，那就意味着这不是一次系统调用的返回，也不是一次普通的中断返回，而是一次嵌套中断返回(或者在系统调用过程中发生了中断)。此时大概的执行路径是应该这样的：假设进程现在运行在用户态，此时发生一次中断，进程进入内核态(此时 user_mode(regs) 返回 1，意味着中断现场是用户态)，此后在中断返回前，发生了一个更高优先级的中断，于是 CPU 开始执行高优先级的处理函数(此时 user_mode(regs) 返回 0，意味着中断现场是在内核态)。当高优先级中断处理结束后，在它返回时，是不应该处理信号的，因为信号的优先级比中断优先级低(见第6章)。在这种情况下，对信号的处理将会延迟到低优先级中断处理结束之后。相对于 Windows 内核来说，尽管 Linux 中没有一组显式的操作函数来实现这一系列的优先级管理方案，但是 Linux 内核和 Windows 内核都使用了同样的机制，优先级关系为：高优先级中断-> 低优先级中断-> 软中断(类似 Windows 内核中的 DPC)-> 信号(类似 Windows 内核中的 APC)-> 进程运行。

如果 user_mode() 返回 1，在第24行，调用 get_signal_to_deliver()，这个函数从当前进程的信号队列(包括 Private Signal Queue 和 Shared Signal Queue)取出等待处理的信号(调用 dequeue_signal() 函数)，然后根据信号定位到对应的 sighand_struct 结构，如果信号的处理函数 sa_handler 为 SIG_IGN，就忽略该信号，继续取下一个信号。如果信号的处理函数 sa_handler 为 SIG_DFL，意味着按照信号默认的处理方式对待就可以了(例如直接调用 do_coredump() 等)。

如果 get_signal_to_deliver() 函数返回值大于 0，说明这个信号的处理函数是在用户态空间(程序通过 signal() 等函数设置的自定义信号处理函数)。在这种情况下，进程当前处于内核态，而信号处理函数却位于用户态，为此必须在进程的用户态构造一个临时的堆栈环境(因为进程的信号处理函数在进行函数调用以及使用局部变量时需要使用堆栈)，然后进入用户态执行信号处理函数，最后再返回内核态继续执行。在这个过程中，有以下

问题需要解决：

- 临时的用户态堆栈在哪里呢？这个问题很好解决，因为可以直接使用进程现有的用户态堆栈，这里要保证的是：使用结束后这个堆栈必须和使用前是一模一样的。
- 临时堆栈解决之后，需要确定的是通过什么方法来保证返回到用户态后，进程执行的是信号的处理函数，在第6章，我们知道在进入内核态时，内核态堆栈中保存了一个中断现场，也就是一个 `pt_regs` 结构，中断返回地址就保存在 `pt_regs` 中的 `eip` 中，因此这里我们只要把当前进程的 `pt_regs` 结构中的 `eip` 设置为 `sa_handler`，然后返回用户态后就开始从 `sa_handler` 处开始执行了。修改方法如下：

```
regs->eip = (unsigned long) ka->sa.sa_handler;
```

- 当信号的用户态处理函数执行结束时，需要再次进入内核态，还原用户态堆栈，并且修改 `pt_regs` 中的 `eip`，保证将来能够按照正常的方式返回用户态。我们知道进程要主动进入内核态只有通过系统调用，触发异常等方法，为此内核专门提供了一个系统调用 `sys_sigreturn()`，但是如何调用 `sys_sigreturn()` 呢？强制安排信号处理函数最后必须调用一个 `sigreturn()` 不是一个好方法，因为不了解内核的程序员会对这个限制感到不解，为此程序员常常忘记在它们的信号处理函数的末尾调用 `sigreturn()`，如果真是这样，`gcc` 也检测不出这个错误。为此内核在临时堆栈中插入下面的指令：

```
popl %eax
movl __NR_sigreturn, %eax
int $0x80
```

当用户态信号处理函数运行结束时，会从堆栈的顶部取出返回地址，因此内核在构建临时堆栈时，把上面这段代码的入口地址保存在堆栈顶部，这样当信号处理完成后，就会顺利地通过系统调用 `sigreturn()` 进入内核。

- 当通过构造的 `sigreturn()` 返回到内核态之后，内核需要顺利地返回用户态执行原来的代码(信号处理前应该返回的用户空间状态)，但是此时进入内核空间的 `pt_regs` 上下文是通过 `sigreturn()` 构成的，而最初的内核堆栈中的 `pt_regs` 上下文在第一次返回用户态空间执行信号处理函数时，已经被“销毁”(内核态堆栈的 `pt_regs` 上下文，在中断返回后，就不复存在了)。而现在必须通过最初的 `pt_regs` 上下文返回用户态，为此，在构建临时堆栈环境时，内核会把当时的 `pt_regs` 上下文备份到临时堆栈中(位于用户态堆栈。)，当通过系统调用 `sigreturn()` 再次进入内核时，内核从用户态空间还原出原始的 `pt_regs`，最后正常返回。

通过上面的讨论，我们知道在这个迂回的处理过程中，关键在于用户态的临时建立的堆栈环境，这是一个 `sigframe` 结构：

代码片段 7.14 节自 arch/x86/kernel/sigframe_32.h

```

1 struct sigframe
2 {
3     /* 保存返回地址。 */
4     char __user *pretcode;
5     /* 信号。 */
6     int sig;
7     /* 保存一组寄存器上下文。 */
8     struct sigcontext sc;
9     /* 保存浮点寄存器上下文。 */
10    struct _fpstate fpstate;
11    unsigned long extramask[_NSIG_WORDS-1];
12
13    /* 保存进入 sigreturn() 系统调用的代码。 */
14    char retcode[8];
15 }

```

其中 sigcontext 和 _fpstate 的作用类似于 pt_regs 用于保存相关寄存器上下文，原始的 pt_regs 的相关信息就备份在这里。而整个 sigframe 结构是通过 get_sigframe() 函数在进程的用户态空间分配的，get_sigframe() 定义如下：

代码片段 7.15 节自 arch/x86/kernel/signal.c

```

1 static inline void __user *
2     get_sigframe(struct k_sigaction *ka,
3                 struct pt_regs * regs,
4                 size_t frame_size)
5 {
6     unsigned long esp;
7
8     /* Default to using normal stack */
9     esp = regs->esp;
10
11    .....
12
13    esp -= frame_size;
14
15    /*
16     * Align the stack pointer according to the i386 ABI,
17     * i.e. so that on function entry ((sp + 4) & 15) == 0.
18     */
19     esp = ((esp + 4) & -16ul) - 4;
20
21     return (void __user *) esp;
22 }

```

`get_sigframe` 通过用户态空间堆栈的 `esp+sizeof(struct sigframe)` 在用户态堆栈的顶端分配一片临时存储空间，将来使用完成后，再通过 `esp+sizeof(struct sigframe)` 还原。

通过上面的讨论，我们再回到 `do_signal()` 中来，如果有用户态的信号处理函数，`do_signal()` 会调用 `handle_signal()`，`handle_signal()` 将调用 `setup_frame()` 或者 `setup_rt_frame()` 来完成实际的工作，这里我们以 `setup_frame()` 为例进行进一步的讨论。

代码片段 7.16 节自 `arch/x86/kernel/signal_32.c`

```

1 static int setup_frame(int sig,
2                         struct k_sigaction *ka,
3                         sigset_t *set,
4                         struct pt_regs * regs)
5 {
6     void __user *restorer;
7     struct sigframe __user *frame;
8     int err = 0;
9     int usig;
10
11    /* 在用户态堆栈分配一个 sigframe 结构。 */
12    frame = get_sigframe(ka, regs, sizeof(*frame));
13
14    /* 检查是否可写。 */
15    if (!access_okVERIFY_WRITE, frame, sizeof(*frame)))
16        goto give_sigsegv;
17
18    usig = current_thread_info()->exec_domain &&
19          current_thread_info()->exec_domain->signal_invmap &&
20          sig < 32 ?
21              current_thread_info()->exec_domain->signal_invmap[sig] :
22              sig;
23
24    /* 把相关信息从内核态备份到用户态堆栈的 sigframe 结构中。 */
25    err = __put_user(usig, &frame->sig);
26    if (err)
27        goto give_sigsegv;
28
29    err = setup_sigcontext(&frame->sc,
30                           &frame->fpstate,
31                           regs,
32                           set->sig[0]);
33    if (err)
34        goto give_sigsegv;
35    if (_NSIG_WORDS > 1) {
36        err = __copy_to_user(&frame->extramask,
```

```
36             &set->sig[1],  
37             sizeof(frame->extramask));  
38     if (err)  
39         goto give_sigsegv;  
40 }  
41 /*  
42 * 在这里 restorer 指向用户态堆栈 frame->retcode,  
43 * retcode 就是保存手工构造的 sigreturn() 系统代码。  
44 */  
45 if (current->binfo->hasvdso)  
46     restorer = (void *)VDSO_SYM(&__kernel_sigreturn);  
47 else  
48     restorer = (void *)&frame->retcode;  
49  
50 if (ka->sa.sa_flags & SA_RESTORER)  
51     restorer = ka->sa.sa_restorer;  
52 /*  
53 * 把保存 sigreturn() 代码的首地址写入到用户态堆栈顶端，  
54 * 也就是 frame->pretcode 中，当用户态信号处理函数结束时，  
55 * 就会把这个地址作为返回地址。  
56 */  
57 /* Set up to return from userspace. */  
58 err |= __put_user(restorer, &frame->pretcode);  
59 /*  
60 * This is popl %eax ; movl $,%eax ; int $0x80  
61 *  
62 * WE DO NOT USE IT ANY MORE! It's only left here  
63 * for historical reasons and because gdb uses it  
64 * as a signature to notice signal handler stack frames.  
65 */  
66 /*  
67 * 在用户态堆栈 (frame->retcode) 中构造以下指令：  
68 * popl \%eax  
69 * movl __NR_sigreturn, \%eax  
70 * int \$0x80  
71 */  
72 err |= __put_user(0xb858, (short __user *)(frame->retcode+0));  
73 err |= __put_user(__NR_sigreturn, (int __user *)(frame->retcode+2));  
74 err |= __put_user(0x80cd, (short __user *)(frame->retcode+6));  
75  
76 if (err)  
77     goto give_sigsegv;
```

```

78
79     /* Set up registers for signal handler */
80     /* 把 pt_regs 结构中的 eip 设置为进程的用户态信号处理函数地址。*/
81     regs->esp = (unsigned long) frame;
82     regs->eip = (unsigned long) ka->sa.sa_handler;
83     regs->eax = (unsigned long) sig;
84     regs->edx = (unsigned long) 0;
85     regs->ecx = (unsigned long) 0;
86
87     regs->xds = __USER_DS;
88     regs->xes = __USER_DS;
89     regs->xss = __USER_DS;
90     regs->xcs = __USER_CS;
91
92     /*
93      * Clear TF when entering the signal handler, but
94      * notify any tracer that was single-stepping it.
95      * The tracer may want to single-step inside the
96      * handler too.
97
98      */
99     regs->eflags &= ~TF_MASK;
100    if (test_thread_flag(TIF_SINGLESTEP))
101        ptrace_notify(SIGTRAP);
102
103 #if DEBUG_SIG
104     printk("SIG deliver (%s:%d): sp=%p pc=%p ra=%p\n",
105           current->comm, current->pid, frame,
106           regs->eip, frame->preicode);
107 #endif
108
109     return 0;
110     give_sigsegv();
111     force_sigsegv(sig, current);
112     return -EFAULT;
113 }

```

当 `setup_frame()` 返回后，一切准备就绪，因此可以从内核态返回了，这样就顺利地过渡到用户态的信号处理函数。当这个函数处理结束后，会通过 `sys_sigreturn()` 再次进入内核态，现在我们来看看 `sys_sigreturn()` 是如何处理的。

代码片段 7.17 节自 `arch/x86/kernel/signal_32.c`

```

1 asmlinkage int sys_sigreturn(unsigned long __unused)
2 {
3     struct pt_regs *regs = (struct pt_regs *) &__unused;

```

```
4 struct sigframe __user *frame = (struct sigframe __user *) (regs->esp-8);
5 sigset_t set;
6 int eax;
7
8 if (!access_ok(VERIFY_READ, frame, sizeof(*frame)))
9     goto badframe;
10 if (__get_user(set.sig[0], &frame->sc.oldmask) ||
11     (_NSIG_WORDS > 1 &&
12      __copy_from_user(&set.sig[1],
13                      &frame->extramask,
14                      sizeof(frame->extramask))))
15     goto badframe;
16
17 sigdelsetmask(&set, ~_BLOCKABLE);
18 spin_lock_irq(&current->sighand->siglock);
19 current->blocked = set;
20 recalc_sigpending();
21 spin_unlock_irq(&current->sighand->siglock);
22
23 if (restore_sigcontext(regs, &frame->sc, &eax))
24     goto badframe;
25 return eax;
26
27 badframe:
28 if (showUnhandledSignals && printk_ratelimit())
29     printk("%s[%d] bad frame in sigreturn frame: %p eip: %lx"
30           " esp: %lx oeax: %lx\n",
31           task_pid_nr(current) > 1 ? KERN_INFO : KERN_EMERG,
32           current->comm, task_pid_nr(current), frame, regs->eip,
33           regs->esp, regs->orig_eax);
34
35 force_sig(SIGSEGV, current);
36 return 0;
37 }
```

这个函数主要调用 `restore_sigcontext()` 根据用户态堆栈上的 `sigframe` 备份还原 `pt_regs`。`restore_sigcontext()` 很简单，当它返回之后，通过 `sys_sigreturn()` 在内核态堆栈上建立的 `pt_regs` 已经变为调用信号处理函数之前的 `pt_regs()`。这样，`sys_sigreturn()` 返回用户态时，就顺利地过渡到信号处理之前的状态了。

最后我们再来看看用户态信号处理函数是如何设置的，应用程序可以通过 `sys_signal()` 等系统调用为一个指定的信号设置用户态处理函数。`sys_signal()` 定义如下：

代码片段 7.18 节自 kernel/signal.c

```

1 asmlinkage unsigned long
2 sys_signal(int sig, __sighandler_t handler)
3 {
4     struct k_sigaction new_sa, old_sa;
5     int ret;
6
7     new_sa.sa.sa_handler = handler;
8     new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;
9     sigemptyset(&new_sa.sa.sa_mask);
10    ret = do_sigaction(sig, &new_sa, &old_sa);
11
12    return ret ? ret : (unsigned long)old_sa.sa.sa_handler;
13 }

```

信号由 sys_signal()的第一个参数指定，信号处理函数的地址由第二个参数指定。sys_signal()根据这两个参数在堆栈设置一个 k_sigaction 结构，然后调用 do_sigaction()。

代码片段 7.19 节自 kernel/signal.c

```

1 int do_sigaction(int sig,
2                  struct k_sigaction *act,
3                  struct k_sigaction *oact)
4 {
5     struct k_sigaction *k;
6     sigset_t mask;
7
8     if (!valid_signal(sig) || sig < 1 || (act && sig_kernel_only(sig)))
9         return -EINVAL;
10    k = &current->sighand->action[sig-1];
11    spin_lock_irq(&current->sighand->siglock);
12    /*
13     * 把原来的 k_sigaction 结构保存到 oact 结构中。
14     * 注意这里是整个数据结构进行复制。
15     */
16    if (oact)
17        *oact = *k;
18
19    if (act) {
20        sigdelsetmask(&act->sa.sa_mask,
21                      sigmask(SIGKILL) | sigmask(SIGSTOP));
22        /* 把新的 k_sigaction 结构复制到进程的 sighand->action 中。 */
23        *k = *act;
24        .....

```

```
25     }
26     spin_unlock_irq(&current->sighand->siglock);
27     return 0;
28 }
```

第8章 系统调用

在保护模式的操作系统中，通常应用程序运行在最低权限级别，而内核运行在高权限级别。低级别的程序不能轻易访问到高级别程序的代码、数据等资源。而另一方面，操作系统内核又必须向应用程序提供一组统一的接口，为应用程序提供服务。例如当应用程序调用 `read()` 从磁盘读取数据时，文件系统驱动程序，磁盘驱动程序，以及磁盘等相关资源都在内核空间，应用程序是无权直接访问的，它必须通过这个接口把请求传递到内核，由内核来进行具体的访问操作。为此内核提供了一个特殊的中断，被称为系统调用，其中断向量为 `0x80`，应用程序可以通过指向 `int $0x80` 进入系统调用，最后内核将通过 `iret` 返回。奔腾 2 及其后续 CPU 为系统调用提供了专门的汇编指令 `sysenter` 和 `sysexit`，但是内核仍然保持了对 `int $0x80` 的兼容，本章我们将以 `int $0x80` 为例对系统调用进行讨论。

8.1 Libc 和系统调用

通常在 C 程序中调用 `read()` 进行文件读取，调用 `exit()` 结束当前进程，这些函数最终都是通过系统调用，分别进入内核中的 `sys_read()` 和 `sys_exit()`。不难想象，C 运行时库提供的 `read()`，`exit()` 等 API 仅仅是对 `int $0x80` 的一个包装。但是 C 运行时库提供的 API 中，很多都需要通过系统调用进入内核（也有一部分是在库中实现的），而 `int $0x80` 是一个特殊的中断，在第6章中，我们看到这个中断的入口函数是 `system_call`。

内核启动阶段，通过 `trap_init()` 把 `0x80` 的中断处理函数设置为 `system_call`：

代码片段 8.1 节自 `arch/x86/kernel/traps_32.c`

```
1 #define SYSCALL_VECTOR      0x80
2
3 void __init trap_init(void)
4 {
5     .....
6     set_system_gate(SYSCALL_VECTOR, &system_call);
7     .....
8 }
```

目前内核总共提供了 300 多个系统调用，它们都是通过 int \$0x80 进入内核，其入口都是 system_call，然后由 system_call 调用其他的 sys_xxx 函数。那么 system_call 如何确定要调用哪一个系统调用服务例程呢？首先内核会为每一个系统调用服务例程分配一个唯一的编号，我们把这个编号称为系统调用号。C 库的实现代码在执行 int \$0x80 之前，必须把相应的系统调用服务例程对应的编号保存到 EAX 寄存器中，在 int \$0x80 之后，进入内核态执行 system_call，system_call 定义如下：

代码片段 8.2 节自 arch/x86/kernel/entry_32.S

```

1 ENTRY(system_call)
2     .....
3     # 如果系统调用号越界，就跳转到 syscall_badsys。
4     cmpl $(nr_syscalls), %eax
5     jae syscall_badsys
6     syscall_call:
7     call *sys_call_table(%eax, 4)
8     # store the return value
9     movl %eax, PT_EAX(%esp)
10    syscall_exit:
11     .....

```

从第7行可以看出，系统调用的函数地址保存在 sys_call_table 这个数组中，eax 作为数组索引¹，而 sys_call_table 是在 syscall_table_32.S 中定义的：

代码片段 8.3 节自 arch/x86/kernel/syscall_table_32.S

```

1 ENTRY(sys_call_table)
2     /* 0 - old "setup()" system call, used for restarting */
3     .long sys_restart_syscall
4     .long sys_exit
5     .long sys_fork
6     .long sys_read
7     .long sys_write
8     .long sys_open    /* 5 */
9     .long sys_close
10    .long sys_waitpid
11    .long sys_creat
12    .long sys_link
13    .long sys_unlink /* 10 */
14    .long sys_execve
15    .long sys_chdir
16    .....

```

¹这里相当于 *(sys_call_table+(eax*4))。

从这里可以看出每增加一个系统调用时，都必须把系统调用服务例程的地址填到这个地址表中。

现在 `system_call` 已经知道如何根据 `eax` 的值来确定需要调用哪一个函数，但是不同的函数有不同的参数，这些参数都是用户态向内核传递的，现在还需要确定内核进入 `system_call` 后，如何找到这些参数。和系统调用一样，这些参数也是通过寄存器传递的，参数依次保存在 `EBX,ECX,EDX,ESI,EDI,EBP` 中，现在我们来看看用户态程序是如何传递参数的。为了简单起见，这里以一个最简单的系统调用 `exit` 为例，`exit` 接收一个参数，类型为 `int`。

代码片段 8.4 系统调用示例代码

```
1 #include <stdlib.h>
2
3 int main()
4 {
5     exit(0);
6     return 0;
7 }
```

使用下面的命令编译这个程序：

```
$ gcc systest.c -static -o systest
```

然后使用 `gdb` 来进行反汇编分析：

```
$ gdb systest
```

首先我们看看 `main()` 函数是如何调用库函数 `exit(0)` 的：

```
1 (gdb) disassemble main
2
3 Dump of assembler code for function main:
4 0x080481f0 <main+0>: lea    0x4(%esp),%ecx
5 0x080481f4 <main+4>: and   $0xffffffff,%esp
6 0x080481f7 <main+7>: pushl -0x4(%ecx)
7 0x080481fa <main+10>: push   %ebp
8 0x080481fb <main+11>: mov    %esp,%ebp
9 0x080481fd <main+13>: push   %ecx
10 0x080481fe <main+14>: sub    $0x4,%esp
11 0x08048201 <main+17>: movl   $0x0,(%esp)
12 0x08048208 <main+24>: call   0x8048960 <exit>
```

从 `main()` 函数的反汇编代码可以看出，从第10行开始，首先把 `exit()` 的参数 0 放置到堆栈的顶部，然后调用 C 库函数 `exit()`。当进入 `exit()` 的代码时，堆栈布局如图8.1所示。

现在再来看看 `exit()` 的反汇编代码：

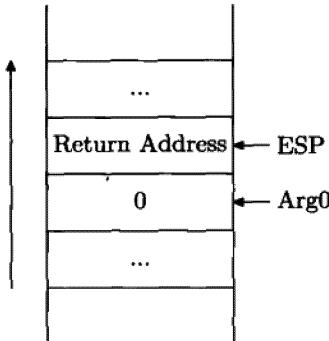


图 8.1 函数调用堆栈示意图

```

1 (gdb) disassemble _exit
2 Dump of assembler code for function _exit:
3 0x0804dfbc <_exit+0>: mov    0x4(%esp),%ebx
4 0x0804dfc0 <_exit+4>: mov    $0xfc,%eax
5 0x0804dfc5 <_exit+9>: int    $0x80
6 0x0804dfc7 <_exit+11>: mov    $0x1,%eax
7 0x0804dfcc <_exit+16>: int    $0x80
8 0x0804dfce <_exit+18>: hlt

```

从 `exit()` 的反汇编代码中可以看到，`0x4(%esp)` 是 `main()` 函数传递进来的参数(见图8.1)，这样 `ebx` 寄存器的值就被设置为 0。之后把 `0xfc` 保存到 `eax` 寄存器中，最后再通过 `int $0x80` 进行系统调用。通过 `sys_call_table` 可以发现，`eax` 寄存器中的系统调用号 `0xfc` 对应的内核函数为 `sys_exit_group()`。当 `sys_exit_group()` 系统调用返回后，中断现场被恢复，此时 `ebx` 仍然寄存器仍然为 0，这时再把 `eax` 寄存器设置为 1，然后再次通过 `int $0x80` 进行系统调用。通过 `sys_call_table` 可以发现，系统调用号 1 对应的内核函数为 `sys_exit()`。

从上面的讨论中我们看到，当只有一个参数时，参数保存在 `EBX` 寄存器中。当有两个参数时，第一个参数保存在 `EBX` 寄存器中，第二个参数保存在 `ECX` 寄存器中，其他情况依此类推，顺序为 `EBX,ECX,EDX,ESI,EDI,EBP`。因此系统调用的参数最多不能超过 6 个，由于受到寄存器位宽限制，每一个参数不能超过 32 位。

参数传递的问题解决了，那么通过什么方式通知内核态的代码，传递了几个参数呢？其实不需要什么机制来通知内核传递了几个参数。因为 C 运行时库的系统调用接口和内核紧密相关，双方的代码都严格遵守预先的“约定”，也就是说，内核的某个系统调用需要几个参数，C 运行时库的相关接口就传递几个，比如 `sys_exit()` 需要一个参数，因此 C 库中的 `exit()` 就传递一个。下面我们来看看内核是如何使用这些参数的。

在第6章中我们知道，进入中断处理时，会调用 `SAVE_ALL` 这个宏把中断现场的相关寄存器保存到内核态堆栈中：

代码片段 8.5 节自 arch/x86/kernel/entry_32.S

```

1 ENTRY(system_call)
2     # save orig_eax
3     pushl %eax
4     SAVE_ALL
5
6     .....
7     cmpl $(nr_syscalls), %eax
8     jae syscall_badsys
9     syscall_call:
10    call *sys_call_table(%eax, 4)
11
12    # store the return value
13    movl %eax, PT_EAX(%esp)
14 syscall_exit:
15    .....

```

在 system_call 中，调用 SAVE_ALL 之后，就会通过 sys_call_table 调用相应的系统函数，这时内核态堆栈布局如图8.2所示。

从图8.2中可以看出，当通过 sys_call_table 调用具体的 sys_xxx 函数时，通过 SAVE_ALL 在内核态堆栈中保存的中断现场，正好符合 C 函数调用的约定(见第2章)，例如 sys_exit() 函数：

代码片段 8.6 节自 kernel/exit.c

```

1 asmlinkage long sys_exit(int error_code)
2 {
3     do_exit((error_code&0xff)<<8);
4 }

```

编译后的 sys_exit() 会从 4(%esp) 这个地址取第一个参数 error_code，这正是保存在堆栈中的 EBX 寄存器的值。最后当 sys_exit() 返回时，把返回值保存到 eax 寄存器中，当返回到 system_call 时，会把 eax 保存到堆栈中。从这里可以看出，通过 SAVE_ALL 保存的中断现场 pt_regs 结构并没有被破坏，仅仅是改变了中断现场的 eax 寄存器的值，最后返回到用户态时，就可以通过 eax 寄存器来获取系统调用的返回值(见第6章)。

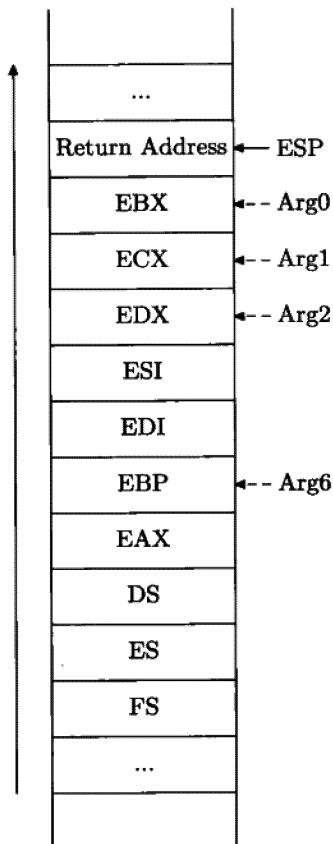


图 8.2 系统调用堆栈示意图

第9章 时钟机制

在基于优先级的分时系统中，时钟机制具有重要意义，它包括以下几方面：

- 一个外部时钟，依靠主板上的电池，在系统断电的情况下，也能维持时钟的准确性。当内核启动时就是通过它来获取当前系统时间的。在 x86 体系上，这个时钟又被称为 Real Time Clock(RTC)。RTC 是主板上的一个 CMOS 芯片，可以通过 0x70 和 0x71 端口操作 RTC。
- 一个时钟中断源，能够周期性地向 CPU 发出中断（比如 10 毫秒。），内核在中断处理例程中检查当前进程时间片是否到期，从而为调度器提供分时依据。另外内核根据 RTC 获取到初始时间后，依靠时钟中断来维护内核的时间。早期的 x86 系统，利用 Programmable Interval Timer(PIT) 来产生周期性的时钟中断，时钟中断通过 8259A 的 IRQ0 向 CPU 报告，早期的 PIT 是 Intel 的 8254 芯片，现在集成在 ICH 中。可以通过 0x40~0x43 端口访问 PIT。

由于 RTC 的计时是以秒为单位的，无法精确到毫秒、微妙甚至纳秒级别。为了获得更精确的时间，内核在启动时从 RTC 中读取到初始时间，然后在每一次时钟中断时，利用初始时间加上时钟中断周期，例如十毫秒。这样内核可以把时钟精确到十毫秒的级别。但是由于内核在运行过程中，需要频繁地开/关中断，如果时钟中断产生时，中断是关闭的，等到中断开启时，流逝的时间已经超过了十毫秒。为了获取更为精确的时间，后来出现了 Time Stamp Counter (TSC)，TSC 是 CPU 中的一个寄存器，该寄存器随着 CPU 的每一个时钟¹周期加一，假设当前 CPU 的主频为 1GHZ，该寄存器每纳秒加一。内核以 RTC 为基础，每次时钟中断通过汇编指令 rdtscl 读取 TSC，这样内核在时钟中断中，通过 TSC 就能够计算出两次时钟中断间流逝的精确时间。于是能够把时间精确到纳秒级别，并且毫不受开/关中断的影响。

随着硬件的发展，后来出现了 APIC Timer，High Precision Event Timer(HPET)，以及 ACPI Power Management Timer。在多处理器计算机中，如果使用 PIT 作为时钟中断源，无法为每一个 CPU 提供一个准确的时钟中断周期。所以就在每一个 CPU 中提供一个 Local APIC Timer，让其独立为本地 CPU 提供中断周期。HPET 内含 8 个计数寄

¹这里的时钟是指数字电路的驱动时钟，以后不再强调，请读者根据上下文区分不同的时钟。

存器，每一个计数器可以由独立的外部时钟驱动，这些计数器随着每一个时钟周期加一，每一个计数器有 32 个比较寄存器，当计数寄存器和比较寄存器相等时，就会发出中断请求。关于 HPET 的详细信息请参阅《High Precision Event Timers (HPET) Specification》。

- 软件定时器，它建立在时钟中断基础上，能够在时钟到期时，调用指定的时钟函数。

从上面可以看出，外部时钟设备的主要作用分别是提供精确的计时功能和定期产生中断的功能。内核把前者抽象为 `clocksource` 对象，后者抽象为 `clock_event_device` 对象。

9.1 clocksource 对象

9.1.1 clocksource 概述

虽然通过 RTC 可以计时，但是由于 RTC 时钟精度相对较低，频繁地通过 IO 读取 RTC 时间也不划算。通过前面的讨论可以看到可计时的设备很多，例如 TSC，HPET 等。因此内核在启动时通过 RTC 获取一个起始时间，然后就可以利用 TSC，HPET 等机制维护自己的时间。然而随着可以用来计时的外部设备的增多，为了统一不同时钟的接口，内核把每一个可用来计时的时钟抽象为 `clocksource` 结构。

代码片段 9.1 节自 `include/linux/clocksource.h`

```
1 struct clocksource {
2     char *name;
3     struct list_head list;
4     int rating;
5     /* 读取时钟函数指针。 */
6     cycle_t (*read)(void);
7     cycle_t mask;
8     u32 mult;
9     u32 shift;
10    unsigned long flags;
11    cycle_t (*vread)(void);
12    void (*resume)(void);
13    .....
14    cycle_t cycle_last __cacheline_aligned_in_smp;
15    u64 xtime_nsec;
16    s64 error;
17
18 #ifdef CONFIG_CLOCKSOURCE_WATCHDOG
19     /* Watchdog related data, used by the framework */
20     struct list_head wd_list;
```

```

21     cycle_t wd_last;
22 #endif
23 };

```

clocksource 结构中, name 是时钟的名字, 所有的时钟通过 list 连接, read 就是读取时间的操作函数。可以通过 sys 文件系统中的 clocksource 看到系统中的时钟²:

代码片段 9.2 clock source 列表

```

cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm jiffies

cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc

```

从上面可以看出, 当前系统中有 4 个可用时钟, 分别是 tsc, hpet, acpi_pm 和 jiffies, current_clocksource 是 tsc, 可用通过 echo "hpet" > current_clocksource 文件来改变内核当前的时钟源。其中 jiffies 就是利用时钟中断周期维护的时钟, 在没有其他时钟源的情况下使用。另外还可以在启动时向内核传递参数 clocksource = xxx 来指定系统使用的时钟源。当内核选定一个当前时钟源时, 会打印出如下消息(以 TSC 为例):

代码片段 9.3 clock source 初始化消息

```
Time: tsc clocksource has been installed.
```

9.1.2 clocksource 初始化

时钟源的初始化工作包括从 RTC 中读取初始时间, 创建 clocksource 结构, 由于内核中有多个时钟源, 因此还要选定一个当前使用的时钟源, 并把全局变量 clock 指向当前使用的 clocksource 结构。

代码片段 9.4 节自 kernel/time/timekeeping.c

```

1 [start_kernel() -> timekeeping_init()]
2
3 void __init timekeeping_init(void)
4 {
5     unsigned long flags;
6     /* 从 RTC 中读出当前系统时间。 */
7     unsigned long sec = read_persistent_clock();
8     write_seqlock_irqsave(&xtime_lock, flags);
9     ntp_clear();
10    /* 选定一个时钟。 */

```

²对于不同的机器, 输出可能不一致。

```

11   clock = clocksource_get_next();
12   clocksource_calculate_interval(clock, NTP_INTERVAL_LENGTH);
13   clock->cycle_last = clocksource_read(clock);
14
15   /* 设置全局时间变量 xtime 和 wall_to_monotonic. 见第9.4.1节。 */
16   xtime.tv_sec = sec;
17   xtime.tv_nsec = 0;
18   set_normalized_timespec(&wall_to_monotonic,
19     -xtime.tv_sec, -xtime.tv_nsec);
20   total_sleep_time = 0;
21
22   write_sequnlock_irqrestore(&xtime_lock, flags);
23 }

```

clock 是 clocksource 类型的全局变量，由于在调用 timekeeping_init 时，系统中的时钟源的初始化工作还没有完成，所以第11行获取到的时钟是 jiffies。第13行其实通过调用 clock->read()，从时钟源读出当前时间放到 clock->cycle_last 中，在下一次时钟中断时利用 clock->read()减去 clock->cycle_last，就得到了这段间隔中流逝的时间。我们来看看时钟 jiffies。

代码片段 9.5 节自 kernel/time/jiffies.c

```

1 static cycle_t jiffies_read(void)
2 {
3     return (cycle_t) jiffies;
4 }
5
6 struct clocksource clocksource_jiffies = {
7     .name      = "jiffies",
8     .rating    = 1, /* lowest valid rating*/
9     .read      = jiffies_read,
10    .mask     = 0xffffffff, /*32bits*/
11    .mult     = NSEC_PER_JIFFY << JIFFIES_SHIFT,
12    .shift    = JIFFIES_SHIFT,
13 };

```

其中 jiffies 是一个全局变量，每一次时钟中断会加一，时钟源 jiffies³的 rating 为 1，rating 越小，选用的优先级越低，也就是说，只有系统中没有其他任何专门的硬件时钟源时，才使用 jiffies 作为时钟源。之后会初始化其他的时钟源。

代码片段 9.6 节自 arch/x86/kernel/time_32.c

```
1 [start_kernel() -> time_init()]
```

³请读者根据上下文区分这两个 jiffies，一个是时钟源的名字，另外一个是全局变量。

```

2
3 extern void (*late_time_init)(void);
4
5 #define choose_time_init() hpet_time_init
6 void __init time_init(void)
7 {
8     /* 初始化 TSC 时钟源。*/
9     tsc_init();
10    late_time_init = choose_time_init();
11 }

```

由于 TSC 在 CPU 的每个时钟周期加一，为了完成计时功能，内核必须知道 CPU 的时钟周期。`tsc_init()`会计算并打印出 CPU 当前主频。

代码片段 9.7 CPU 的时钟频率

```
Detected 1596.500 MHz processor.
```

`late_time_init` 是一个函数指针，这是因为 `time_init()` 在 `mem_init()` 前执行，而 APIC 使用内存地址空间(见第5.1.2节)，因此对 APIC 时钟的初始化工作必须等 APIC 的 fixmap 地址的页表映射建立之后才能进行。同样 HPET 时钟也一样，这里的 `choose_time_init` 被定义为 `hpet_time_init()`。当内存初始化完毕时，会调用 `late_time_init()`。

代码片段 9.8 节自 init/main.c

```

1 [start_kernel()]
2
3 asmlinkage void __init start_kernel(void)
4 {
5     .....
6     mem_init();
7     kmem_cache_init();
8     setup_per_cpu_pageset();
9     numa_policy_init();
10    if (late_time_init)
11        late_time_init();
12
13    calibrate_delay();
14    pidmap_init();
15    .....
16 }
17
18 void __init hpet_time_init(void)
19 {

```

```

20     if (!hpet_enable())
21         setup_pit_timer();
22     time_init_hook();
23 }

```

如果系统支持 HPET 时钟，则 hpet_enable() 会初始化该时钟，并调用 clocksource_register() 函数向内核注册 clocksource_hpet 时钟源。然后会初始化时钟中断。

代码片段 9.9 节自 arch/x86/mach-default/setup.c

```

1 [start_kernel() -> hpet_time_init() -> time_init_hook()]
2
3 static struct irqaction irq0 = {
4     .handler = timer_interrupt,
5     .flags = IRQF_DISABLED | IRQF_NOBALANCING | IRQF_IRQPOLL,
6     .mask = CPU_MASK_NONE,
7     .name = "timer"
8 };
9
10 void __init time_init_hook(void)
11 {
12     irq0.mask = cpumask_of_cpu(0);
13     setup_irq(0, &irq0);
14 }

```

上面的代码时钟中断处理函数设置为 timer_interrupt，如果支持 APIC，以后内核会设置中断向量 0xef 的处理函数为 apic_timer_interrupt()（见第 6.7 节）。

前面说过时钟源 jiffies 的 rating 为 1，rating 的数值越小其优先级最低，现在来看看其他几个时钟源的相关定义。

代码片段 9.10 节自 arch/x86/kernel/tsc_32.c

```

1 static struct clocksource clocksource_tsc = {
2     .name      = "tsc",
3     .rating    = 300,
4     .read      = read_tsc,
5     .mask      = CLOCKSOURCE_MASK(64),
6     .mult      = 0, /* to be set */
7     .shift     = 22,
8     .flags     = CLOCK_SOURCE_IS_CONTINUOUS |
9                  CLOCK_SOURCE_MUST_VERIFY,
10 };
11
12 static struct clocksource clocksource_hpet = {

```

```

13     .name      = "hpet",
14     .rating    = 250,
15     .read      = read_hpet,
16     .mask      = HPET_MASK,
17     .shift     = HPET_SHIFT,
18     .flags     = CLOCK_SOURCE_IS_CONTINUOUS,
19     .resume    = hpet_restart_counter,
20 #ifdef CONFIG_X86_64
21     .vread     = vread_hpet,
22 #endif
23 };
24
25 static struct clocksource clocksource_acpi_pm = {
26     .name      = "acpi_pm",
27     .rating    = 200,
28     .read      = acpi_pm_read,
29     .mask      = (cycle_t)ACPI_PM_MASK,
30     .mult      = 0, /*to be calculated*/
31     .shift     = 22,
32     .flags     = CLOCK_SOURCE_IS_CONTINUOUS,
33
34 };

```

从这里可以看出，默认情况下，内核优先使用 TSC 作为当前时钟源。

9.2 tickless 机制

9.2.1 tickless 由来

在分时系统中，时钟中断的重要性不言而喻，如果一个进程在一段时间间隔内，没有进行 IO 操作，也没有主动让出 CPU，比如 sleep, wait 等操作，那么时钟中断会迫使该进程时间片到期时让出 CPU。另外软件定时器也是建立在时钟中断的基础上的，典型的时钟中断周期为 10ms，这也是调度算法和软件定时器的极限。为了提高软件时钟精度，就需要减小时钟中断周期，例如把时钟中断周期改为 1ms。然而过短的时钟中断周期又难免带来性能的浪费。当 CPU 无事可做时⁴，我们希望相关的空闲设备进入节能状态，这是电源管理的基本要求。假设现在 CPU 运行在 idle 状态，于是进入节能状态，那么 CPU 何时结束节能状态呢？我们假设系统中所有进程都在非就绪队列的前提下考虑以下几种情况。

(1) 对于因 IO 操作类进程。必须等到设备发出中断⁵，CPU 结束节能状态，运行中断处理

⁴可以使用 vmstat 命令来查看 CPU 在 idle 状态的时间比。

⁵这里，我们把因 IO 超时结束等待状态的情况归为第二类。

例程，如果 IO 操作满足进程要求，则唤醒相关进程，并调度运行。

- (2) 对于时间等待类进程，等到定时器到期，CPU 需要结束节能状态，调度相关进程运行。

假设一个进程通过 sleep 操作请求睡眠 1 毫秒，为了保证软件时钟的精度，需要减小时钟中断周期，如果一个进程通过 sleep 操作请求睡眠 10 分钟，并且当前没有可运行的进程，于是我们希望 CPU 进入节能状态。但是每毫秒会产生一个时钟中断，频繁地打断了 CPU 的节能状态，这些时钟中断是多余的。为此提出了 tickless 机制，当使用 tickless 后，CPU 在 idle 时，不会产生不必要的时钟中断，从而保证 CPU 尽量处于节能状态。使用 tickless 机制后，中断周期不再是固定的，而是动态改变的，因此又被称为 Dynamic ticks。其原理如下：

- (1) 正常的进程运行情况下，外部时钟中断源按一定的中断周期发出时钟中断，为分时调度提供依据。
- (2) 没有进程可调度时，CPU 运行 idle 进程进入节能状态，并且把外部中断源的中断时间设置为最早到期的软件定时器。假设最早到期的时钟是 5 分钟，则这 5 分钟期间，不会产生时钟中断。当然，这并不意味着这期间 CPU 一定会处于节能状态，还有其他的外部中断可能会打断这一状态，比如磁盘 IO 操作结束中断，或者网卡接收中断等，从而使某个进程等待结束进入就绪队列。

据称，在 Intel 移动芯片平台中，使用 tickless 技术后，可以节能 1.2 瓦，读者可以运行 Intel 的 powertop 来查看系统的电能使用情况。我们知道，系统的时间是在每一次时钟中断例程中更新的，当系统进入 tickless 状态很长时间后，一个磁盘中断唤醒一个进程，之后该进程通过系统调用获取系统时间，此时将会得到错误的时间，为了避免这种错误的发生，在 tickless 模式下，每一次进入中断处理时都会更新系统时间。

代码片段 9.11 节自 kernel/softirq.c

```
1 [ do_IRQ() -> irq_enter()]
2 void irq_enter(void)
3 {
4     __irq_enter();
5     /* 更新系统时间。*/
6 #ifdef CONFIG_NO_HZ
7     if (idle_cpu(smp_processor_id()))
8         tick_nohz_update_jiffies();
9 #endif
10 }
```

9.2.2 clock event device 对象概述

采用 tickless 机制后，时钟中断周期不再是固定的，因此需要在运行中频繁地操作时钟中断源设备，来配置下一次发出中断的时机，然而随着外部时钟中断源的增多，为了屏蔽各种设备的操作差异，内核使用 `clock_event_device` 结构来表示一个设备，其定义如下：

代码片段 9.12 节自 `include/linux/clockchips.h`

```

1 struct clock_event_device {
2     const char *name;
3     unsigned int features;
4     unsigned long max_delta_ns;
5     unsigned long min_delta_ns;
6     unsigned long mult;
7     int shift;
8     int rating;
9     int irq;
10    cpumask_t cpumask;
11    int (*set_next_event)(unsigned long evt, struct clock_event_device *);
12    void (*set_mode)(enum clock_event_mode mode, struct clock_event_device *);
13    void (*event_handler)(struct clock_event_device *);
14    void (*broadcast)(cpumask_t mask);
15    struct list_head list;
16    enum clock_event_mode mode;
17    ktime_t next_event;
18 };

```

这个结构中，`name` 表示该设备的名称，`features` 表示设备的功能，其中 `CLOCK_EVT_FEAT_AT_PERIODIC` 表示可周期性的中断，`CLOCK_EVT_FEAT_ONESHOT` 表示单次中断，Dynamic tick 就依赖于这种模式。每一次中断后需要设置下一次中断的时机。`max_delta_ns` 和 `min_delta_ns` 分别表示该设备的最大和最小中断周期，单位为纳秒。计时器由一个计数寄存器和一个比较寄存器构成，计数器在每一个主频周期加 1，比较寄存器用来设置定时器，当这两个寄存器的值相等时，就会触发时钟中断。当主频为 0.5GHz 时，计数器每 2 纳秒加 1。`mult` 和 `shift` 是用来把纳秒转换成周期数的，假设现在要求设置一个 10 纳秒的定时器，转换成时钟周期的就是 $(10 * \text{mult}) >> \text{shift}$ 。`rating` 是优先级。`irq` 是中断请求号，`cpumask` 表示这个中断设备所属的 CPU 掩码，`set_next_event` 函数的作用就是设置定时器下一次发出时钟中断的时机，`set_mode` 是改变设备工作模式的函数指针，`event_handler` 是中断处理函数。`mode` 表示当前设备的工作模式，可以是 `periodic` 和 `oneshot` 模式。64 位的 `next_event` 表示下一次该设备发出中断的时间，单位为纳秒。

我们可以通过 `/proc/timer_list` 文件看到系统中的时钟事件设备以及软件定时器⁶。

⁶对于不同的机器，输出可能不一致。

代码片段 9.13 节自/proc/timer_list

```
1 $ cat /proc/timer_list
2
3 Timer List Version: v0.3
4 HRTIMER_MAX_CLOCK_BASES: 2
5
6 /* 系统当前时间，单位纳秒。*/
7 now at 50790034132537 nsecs
8 .....
9 /* clock event device 对象。*/
10 Tick Device: mode: 1
11 Clock Event Device: hpet
12 max_delta_ns: 2147483647
13 min_delta_ns: 3352
14 mult: 61496110
15 shift: 32
16 mode: 3
17 next_event: 50790036000000 nsecs
18 set_next_event: hpet_legacy_next_event
19 set_mode: hpet_legacy_set_mode
20 event_handler: tick_handle_oneshot_broadcast
21 tick_broadcast_mask: 00000003
22 tick_broadcast_oneshot_mask: 00000001
23
24 Tick Device: mode: 1
25 Clock Event Device: lapic
26 max_delta_ns: 1008931166
27 min_delta_ns: 1804
28 mult: 35709862
29 shift: 32
30 mode: 1
31 next_event: 50790036000000 nsecs
32 set_next_event: lapic_next_event
33 set_mode: lapic_timer_setup
34 event_handler: hrtimer_interrupt
35
36 Tick Device: mode: 1
37 Clock Event Device: lapic
38 max_delta_ns: 1008931166
39 min_delta_ns: 1804
40 mult: 35709862
41 shift: 32
```

```

42 mode:          3
43 next_event:    50790037000000 nsecs
44 set_next_event: lpic_next_event
45 set_mode:      lpic_timer_setup
46 event_handler: hrtimer_interrupt

```

上面有 3 个 clock event device 对象，其中 lpic 是两个 CPU 的本地时钟，当 CPU 进入一定的节能状态时，lpic 时钟也会停止，此时需要另外的时钟接手处理原本属于 lpic 的时钟事件。这个过程被称为 broadcast。因此有一个 tick_broadcast_device 指向当前使用的 broadcast clock event device 对象。以 HPET 为例，当 lpic 停止时，HPET 接手处理所有的时钟事件，它检查 mask 中的全部 CPU 的时钟队列，为它们处理到期的时钟。

系统中每一个可发出时钟中断的设备都由一个 clock event device 对象来表示。各个 CPU 又如何选用这些 clock event device 对象呢？为了维护 CPU 和 clock event device 对象的对应关系，内核定义了一个 tick_device 结构。

代码片段 9.14 节自 include/linux/tick.h

```

1 DECLARE_PER_CPU(struct tick_device, tick_cpu_device);
2
3 struct tick_device {
4     struct clock_event_device *evtdev;
5     enum tick_device_mode mode;
6 };

```

每一个 CPU 都有一个 tick_device 结构，其 evtdev 指向该 CPU 当前使用的 clock event device 对象。

9.2.3 clock event device 对象的初始化

初始化工作主要就是探测并初始化时钟中断源设备，并建立起 clock_event_device 设备。可以提供中断的设备包括 PIT，HPET，Local APIC 等。

1. HPET clock event device 对象初始化

代码片段 9.15 节自 arch/x86/kernel/time_32.c

```

1 #define choose_time_init() hpet_time_init
2
3 [start_kernel() -> choose_time_init()]
4 void __init hpet_time_init(void)
5 {
6     if (!hpet_enable())

```

```

7     setup_pit_timer();
8     time_init_hook();
9 }

```

如果没有探测到 HPET，则初始化 PIT，这里我们不讨论 PIT 的情况，仅仅列出 PIT 要注册的 `clock_event_device` 结构。

代码片段 9.16 节自 `arch/x86/kernel/i8253.c`

```

1 struct clock_event_device pit_clockevent = {
2     .name      = "pit",
3     .features  = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT,
4     .set_mode   = init_pit_timer,
5     .set_next_event = pit_next_event,
6     .shift     = 32,
7     .irq       = 0,
8 };

```

代码片段 9.17 节自 `arch/x86/kernel/hpet.c`

```

1 [start_kernel() -> choose_time_init() -> hpet_enable()]
2
3 int __init hpet_enable(void)
4 {
5     unsigned long id;
6     /* 探测硬件系统是否支持 HPET。 */
7     if (!is_hpet_capable())
8         return 0;
9     /* 为 HPET 设置 fixmap 内存映射。 */
10    hpet_set_mapping();
11    hpet_period = hpet_readl(HPET_PERIOD);
12    if (hpet_period < HPET_MIN_PERIOD ||
13        hpet_period > HPET_MAX_PERIOD)
14        goto out_nohtpet;
15
16    id = hpet_readl(HPET_ID);
17    /* 为 HPET 注册 clocksource 对象。 */
18    if (hpet_clocksource_register())
19        goto out_nohtpet;
20    /* 为 HPET 注册 clock event device 对象。 */
21    if (id & HPET_ID_LEGSSUP) {
22        hpet_legacy_clockevent_register();
23        return 1;
24    }

```

```

25     return 0;
26
27 out_nohtpet:
28     hpet_clear_mapping();
29     boot_hpet_disable = 1;
30     return 0;
31 }

```

可以看到 HPET 既可以用于计时，也可以用于提供时钟中断。其中 `hpet_legacy_clockevent_register()` 会计算当前时钟主频，并对 HPET 的 `clock_event_device` 结构进行初始化，最后注册该设备：

代码片段 9.18 节自 `arch/x86/kernel/hpet.c`

```

1 static struct clock_event_device hpet_clockevent = {
2     .name      = "hpet",
3     .features  = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT,
4     .set_mode   = hpet_legacy_set_mode,
5     .set_next_event = hpet_legacy_next_event,
6     .shift      = 32,
7     .irq        = 0,
8     .rating     = 50,
9 };
10
11 [start_kernel() -> choose_time_init() -> hpet_enable() ->
12   hpet_legacy_clockevent_register()]
13
14 static void hpet_legacy_clockevent_register(void)
15 {
16     uint64_t hpet_freq;
17
18     /* Start HPET legacy interrupts */
19     hpet_enable_legacy_int();
20     /* 根据频率计算相关参数。*/
21     hpet_freq = 100000000000000ULL;
22     do_div(hpet_freq, hpet_period);
23     hpet_clockevent.mult = div_sc((unsigned long)hpet_freq,
24                                   NSEC_PER_SEC, 32);
25     /* Calculate the min / max delta */
26     hpet_clockevent.max_delta_ns = clockevent_delta2ns(0x7FFFFFFF,
27                                                       &hpet_clockevent);
28     hpet_clockevent.min_delta_ns = clockevent_delta2ns(0x30,
29                                                       &hpet_clockevent);
30     /* 设置其 cpumask 为本地 CPU. */

```

```

31     hpet_clockevent.cpumask = cpumask_of_cpu(smp_processor_id());
32     /* 注册 hpet clock event device 对象。*/
33     clockevents_register_device(&hpet_clockevent);
34     /* 全局变量 global_clock_event 指向当前使用的 clock event device 对象。*/
35     global_clock_event = &hpet_clockevent;
36     printk(KERN_DEBUG "hpet clockevent registered\n");
37 }

```

`global_clock_event` 指向当前使用的全局的 `clock event device` 对象，当 `lapic` 也进入节能状态时，`clock event device` 需要接手 `lapic` 的时钟事件。我们注意到注册的 `hpet_clockevent` 设备并没有设置 `event_handler` 函数指针，这个函数指针将在以后初始化。如果硬件不支持 HPET 或者启动时通过内核参数设置 `hpet=disable`，则 `global_clock_event` 就指向了 `pit` 的 `clock event device` 结构。

代码片段 9.19 节自 `kernel/time/clockevents.c`

```

1 [start_kernel() -> choose_time_init() -> hpet_enable() ->
2   hpet_legacy_clockevent_register() -> clockevents_register_device()]
3
4 void clockevents_register_device(struct clock_event_device *dev)
5 {
6     BUG_ON(dev->mode != CLOCK_EVT_MODE_UNUSED);
7     spin_lock(&clockevents_lock);
8     list_add(&dev->list, &clockevent_devices);
9     clockevents_do_notify(CLOCK_EVT_NOTIFY_ADD, dev);
10    clockevents_notify_released();
11    spin_unlock(&clockevents_lock);
12 }

```

首先把注册的 `clock event device` 对象添加到全局的 `clockevent_devices` 链表中，然后请求 `CLOCK_EVT_NOTIFY_ADD` 的 `notify` 事件。内核在 `start_kernel()` 函数中初始化了一个全局的 `notify` 对象。

代码片段 9.20 节自 `kernel/time/tick-common.c`

```

1 static struct notifier_block tick_notifier = {
2     .notifier_call = tick_notify,
3 };
4
5 [start_kernel() -> tick_init()]
6 void __init tick_init(void)
7 {
8     clockevents_register_notifier(&tick_notifier);
9 }

```

所以 clockevents_do_notify 的 notify 请求最终由 tick_notify()函数处理。

代码片段 9.21 节自 kernel/time/tick-common.c

```

1 static int tick_notify(struct notifier_block *nb,
2                         unsigned long reason,
3                         void *dev)
4 {
5     switch (reason) {
6     case CLOCK_EVT_NOTIFY_ADD:
7         return tick_check_new_device(dev);
8     case CLOCK_EVT_NOTIFY_BROADCAST_ON:
9     case CLOCK_EVT_NOTIFY_BROADCAST_OFF:
10    case CLOCK_EVT_NOTIFY_BROADCAST_FORCE:
11        tick_broadcast_on_off(reason, dev);
12        break;
13    case CLOCK_EVT_NOTIFY_BROADCAST_ENTER:
14    case CLOCK_EVT_NOTIFY_BROADCAST_EXIT:
15        tick_broadcast_oneshot_control(reason);
16        break;
17    case CLOCK_EVT_NOTIFY_CPU_DEAD:
18        tick_shutdown_broadcast_oneshot(dev);
19        tick_shutdown_broadcast(dev);
20        tick_shutdown(dev);
21        break;
22    case CLOCK_EVT_NOTIFY_SUSPEND:
23        tick_suspend();
24        tick_suspend_broadcast();
25        break;
26    case CLOCK_EVT_NOTIFY_RESUME:
27        tick_resume();
28        break;
29    default:
30        break;
31    }
32
33    return NOTIFY_OK;
34 }
```

这里 CLOCK_EVT_NOTIFY_ADD 事件将被 tick_check_new_device()函数处理，在内核中不同的 CPU 可以使用不同的 clock event device，每一个 CPU 都有一个 tick_device 的结构来表示当前 CPU 使用的 clock event device 对象。tick_check_new_device()的作用是通知当前 CPU 现在有一个新的 clock event device 对象可以使用了，CPU 现在需要做出决策，

需要使用新的 clock event device 对象，还是维持不变。

代码片段 9.22 节自 kernel/time/tick-common.c

```
1 struct tick_device {
2     struct clock_event_device *evtdev;
3     enum tick_device_mode mode;
4 };
5
6 static int tick_check_new_device(struct clock_event_device *newdev)
7 {
8     struct clock_event_device *curdev;
9     struct tick_device *td;
10    int cpu, ret = NOTIFY_OK;
11    unsigned long flags;
12    cpumask_t cpumask;
13
14    spin_lock_irqsave(&tick_device_lock, flags);
15    cpu = smp_processor_id();
16    if (!cpu_isset(cpu, newdev->cpumask))
17        goto out_bc;
18    td = &per_cpu(tick_cpu_device, cpu);
19    curdev = td->evtdev;
20    cpumask = cpumask_of_cpu(cpu);
21
22    /* cpu local device ? */
23    if (!cpus_equal(newdev->cpumask, cpumask)) {
24        /*
25         * 如果该设备不是 CPU 的本地设备，则先判断新注册的设备是否能够向该 CPU 发 IRQ。
26         * 如果新注册的设备不能向该 CPU 发出中断请求，则维持不变。
27         */
28        if (!irq_can_set_affinity(newdev->irq))
29            goto out_bc;
30        /*
31         * 如果新设备不是 CPU 的本地设备，且当前 CPU 使用的设备是本地设备，
32         * 则还是使用原设备。
33         */
34        if (curdev && cpus_equal(curdev->cpumask, cpumask))
35            goto out_bc;
36    }
37    if (curdev) {
38        /*
39         * 如果 CPU 当前使用的设备支持 ONESHOT 工作模式，而新注册的设备不支持，
40         * 则使用源设备。
41     }
```

```

41     */
42     if ((curdev->features & CLOCK_EVT_FEAT_ONESHOT) &&
43         !(newdev->features & CLOCK_EVT_FEAT_ONESHOT))
44         goto out_bc;
45     /*
46      * 如果新注册的设备和CPU当前使用的设备特性一致，则根据rating来判断设备的
47      * 优先级。新设备的rating小于当前设备的rating，则维持当前使用的设备不变。
48      */
49     if (curdev->rating >= newdev->rating)
50         goto out_bc;
51     }
52     /* 经过上面的一系列判断后，运行到这里说明CPU需要使用新注册的设备。*/
53     if (tick_is_broadcast_device(curdev)) {
54         /* 关闭目前使用的设备。*/
55         clockevents_set_mode(curdev, CLOCK_EVT_MODE_SHUTDOWN);
56         curdev = NULL;
57     }
58     clockevents_exchange_device(curdev, newdev);
59     tick_setup_device(td, newdev, cpu, cpumask);
60     if (newdev->features & CLOCK_EVT_FEAT_ONESHOT)
61         tick_oneshot_notify();
62     spin_unlock_irqrestore(&tick_device_lock, flags);
63     return NOTIFY_STOP;
64
65 out_bc:
66     if (tick_check_broadcast_device(newdev))
67         ret = NOTIFY_STOP;
68     spin_unlock_irqrestore(&tick_device_lock, flags);
69     return ret;
70 }

```

如果有 broadcast 功能的 clock event device 对象注册，并且没有被某个 CPU 用做自己的 clock event device 对象，那么最后的 tick_check_broadcast_device() 试图把它作为 broadcast 设备，并设置全局变量 tick_broadcast_device。

代码片段 9.23 节自 kernel/time/tick-broadcast.c

```

1 int tick_check_broadcast_device(struct clock_event_device *dev)
2 {
3     if ((tick_broadcast_device.evtdev &&
4         tick_broadcast_device.evtdev->rating >= dev->rating) ||
5         (dev->features & CLOCK_EVT_FEAT_C3STOP))
6     return 0;
7

```

```
8     clockevents_exchange_device(NULL, dev);
9     tick_broadcast_device.evtdev = dev;
10    if (!cpus_empty(tick_broadcast_mask))
11        tick_broadcast_start_periodic(dev);
12    return 1;
13 }
```

如果 `tick_check_new_device()` 函数发现需要使用新注册的设备作为该 CPU 的当前设备，会调用 `tick_setup_device()` 函数，它将根据不同的情况设置 `clock_event_device` 结构的 `handler` 指针。

代码片段 9.24 节自 `kernel/time/tick-common.c`

```
1 static void tick_setup_device(struct tick_device *td,
2                               struct clock_event_device *newdev,
3                               int cpu, cpumask_t cpumask)
4 {
5     ktime_t next_event;
6     void (*handler)(struct clock_event_device *) = NULL;
7     /*
8      * First device setup ?
9      */
10    /*
11     * 如果当前 CPU 还没有 clock event device, 就默认新设备为周期性的设备,
12     * 并计算该设备的中断周期, 其中 NSEC_PER_SEC 表示一秒中的纳秒数, HZ 是编
13     * 译时配置的每秒的中断次数, 所以 tick_period 就是中断周期, 单位为纳秒。*/
14    if (!td->evtdev) {
15        if (tick_do_timer_cpu == -1) {
16            tick_do_timer_cpu = cpu;
17            tick_next_period = ktime_get();
18            tick_period = ktime_set(0, NSEC_PER_SEC / HZ);
19        }
20        /*
21         * Startup in periodic mode first.
22         */
23        td->mode = TICKDEV_MODE_PERIODIC;
24    } else {
25        /* 如果当前 CPU 已经有一个 clock event device. */
26        handler = td->evtdev->event_handler;
27        next_event = td->evtdev->next_event;
28    }
29    /* 当前 CPU 使用新的 clock event device. */
30    td->evtdev = newdev;
31    /*
```

```

32     * When the device is not per cpu, pin the interrupt to the
33     * current cpu:
34     */
35     if (!cpus_equal(newdev->cpumask, cpumask))
36         irq_set_affinity(newdev->irq, cpumask);
37     if (tick_device_uses_broadcast(newdev, cpu))
38         return;
39     if (td->mode == TICKDEV_MODE_PERIODIC)
40         tick_setup_periodic(newdev, 0);
41     else
42         tick_setup_oneshot(newdev, handler, next_event);
43 }

```

首先，如果该设备支持 broadcast 功能，则调用 tick_device_uses_broadcast，最后，如果新注册的 clock event device 工作模式是周期性的，就调用 tick_setup_periodic()设置 event_handler 函数指针，并且会配置设备的中断周期。如果是 ONESHOT 模式，则调用 tick_setup_oneshot()设置 event_handler 和 next_event，并设置设备下一次发出中断的时机。这里再次强调，对于周期模式，只需要对设备进行一次初始化，而对于 ONESHOT 模式，每一次中断后，都需要设置设备下一次中断的时机。

代码片段 9.25 节自 kernel/time/tick-common.c

```

1 void tick_setup_periodic(struct clock_event_device *dev,
2                           int broadcast)
3 {
4     tick_set_periodic_handler(dev, broadcast);
5
6     /* Broadcast setup ? */
7     if (!tick_device_is_functional(dev))
8         return;
9     if (dev->features & CLOCK_EVT_FEAT_PERIODIC) {
10         clockevents_set_mode(dev, CLOCK_EVT_MODE_PERIODIC);
11     } else {
12         unsigned long seq;
13         ktime_t next;
14         do {
15             seq = read_seqbegin(&xtime_lock);
16             next = tick_next_period;
17         } while (read_seqretry(&xtime_lock, seq));
18         clockevents_set_mode(dev, CLOCK_EVT_MODE_ONESHOT);
19         for (;;) {
20             if (!clockevents_program_event(dev, next, ktime_get()))
21                 return;

```

```

22     next = ktime_add(next, tick_period);
23 }
24 }
25 }
26
27 void tick_set_periodic_handler(struct clock_event_device *dev,
28                                 int broadcast)
29 {
30     if (!broadcast)
31         dev->event_handler = tick_handle_periodic;
32     else
33         dev->event_handler = tick_handle_periodic_broadcast;
34 }
```

至此 clock event device 的 event_handler 函数指针已经设置完毕，起初 hpet 的 event_handler 被设置为 tick_handle_periodic，以后我们将看到 clock event device 的 event_handler 可能在运行中改变。

2. Local APIC clock event device 对象初始化

首先，我们来看看 Local APIC 的 clock event device 对象的定义。

代码片段 9.26 节自 arch/x86/kernel/apic_32.c

```

1 static struct clock_event_device lapic_clockevent = {
2     .name      = "lapic",
3     .features  = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT |
4                  CLOCK_EVT_FEAT_C3STOP | CLOCK_EVT_FEAT_DUMMY,
5
6     .shift     = 32,
7     .set_mode  = lapic_timer_setup,
8     .set_next_event = lapic_next_event,
9     .broadcast = lapic_timer_broadcast,
10    .rating   = 100,
11    .irq      = -1,
12 };
13
14 static DEFINE_PER_CPU(struct clock_event_device, lapic_events);
```

Local APIC 的 clock event device 对象的注册是在 setup_APIC_timer()函数中完成的，对于启动的 CPU 该函数由 setup_boot_APIC_clock()函数调用，而对于非启动 CPU 则由 setup_secondary_APIC_clock()函数调用。

代码片段 9.27 节自 arch/x86/kernel/apic_32.c

```

1 [start_kernel() -> kernel_init() -> smp_prepare_cpus() ->
2   native_smp_prepare_cpus() -> smp_boot_cpus() -> setup_boot_clock() ->
3   setup_boot_APIC_clock()]
4
5 void __init setup_boot_APIC_clock(void)
6 {
7     /* 对于 Local APIC, 每一个 CPU 都有一个 clock event device 对象。*/
8     struct clock_event_device *levt = &__get_cpu_var(lapic_events);
9     const long pm_100ms = PMTMR_TICKS_PER_SEC/10;
10    const long pm_thresh = pm_100ms/100;
11    .....
12    setup_APIC_timer();
13 }
14
15 static void __devinit setup_APIC_timer(void)
16 {
17     struct clock_event_device *levt = &__get_cpu_var(lapic_events);
18
19     memcpy(levt, &lapic_clockevent, sizeof(*levt));
20     levt->cpumask = cpumask_of_cpu(smp_processor_id());
21     clockevents_register_device(levt);
22 }
```

同样，lapic 的 clock event device 对象的 event_handler 函数指针也会在运行时动态改变。现在我们再一次通过 timer_list 文件看看系统中的 clock event device 对象⁷。

代码片段 9.28 节自 /proc/timer_list

```

1 $ cat /proc/timer_list
2
3 Timer List Version: v0.3
4 HRTIMER_MAX_CLOCK_BASES: 2
5 now at 3044531539404 nsecs
6 .....
7 Tick Device: mode:      1
8 Clock Event Device: hpet
9 .....
10 next_event:      3044532000000 nsecs
11 set_next_event: hpet_legacy_next_event
12 set_mode:        hpet_legacy_set_mode
13 event_handler:   tick_handle_oneshot_broadcast
```

⁷对于不同的机器，输出可能不一致。

```
14 .....
15
16 Tick Device: mode:      1
17 Clock Event Device: lapic
18 .....
19 next_event:      3044532000000 nsecs
20 set_next_event: lapic_next_event
21 set_mode:        lapic_timer_setup
22 event_handler:  hrtimer_interrupt
23
24 Tick Device: mode:      1
25 Clock Event Device: lapic
26 .....
27 next_event:      3044533000000 nsecs
28 set_next_event: lapic_next_event
29 set_mode:        lapic_timer_setup
30 event_handler:  hrtimer_interrupt
```

通过对 timer_list 文件的分析，我们可以看到 hpet 和 lapic 的 event_handler 分别是 tick_handle_oneshot_broadcast 和 hrtimer_interrupt 函数，其中 tick_handle_oneshot_broadcast() 将在 lapic 进入节能状态时，接手全部 CPU 的时钟，hrtimer_interrupt() 是高精度时钟的中断函数(见第9章第9.3节)。

9.3 High-Resolution Timers

通常软件定时器是建立在周期性的时钟中断的基础上的，在 Dynamic Ticks 出现之前，时钟中断周期通常是 10 毫秒。可以通过减小时钟中断周期的方法来提高软件时钟精度，但是过高的时钟中断频率将浪费 CPU 的运算周期。于是出现了 High-Resolution Timers(hrtimers)。通过可编程的硬件定时器，把它的到期时间设置为软件定时器队列中最早到期的时间，当时钟到期后，再把剩余的软件定时器队列的最早到期时间编程到硬件定时器中，这样可以获得极高的软件定时器精度，又不至于影响系统性能。而通过 HPET 硬件的支持，内核可以同时设置大量的硬件定时器。

9.3.1 High-Resolution Timers 管理结构

hrtimer 结构代表一个高精度定时器对象，其定义如下：

代码片段 9.29 节自 `include/linux/hrtimer.h`

```
1 struct hrtimer {
2     struct rb_node    node;
```

```

3      ktime_t      expires;
4      enum hrtimer_restart    (*function)(struct hrtimer *);
5      struct hrtimer_clock_base *base;
6      unsigned long      state;
7 #ifdef CONFIG_HIGH_RES_TIMERS
8      enum hrtimer_cb_mode    cb_mode;
9      struct list_head      cb_entry;
10 #endif
11 #ifdef CONFIG_TIMER_STATS
12     void      *start_site;
13     char      start_comm[16];
14     int      start_pid;
15 #endif
16 };

```

由于软件定时器需要进行频繁的添加、查找、删除等操作，为了加快速度，采用红黑树管理，`rb_node`就是用于红黑树管理的，`expires`是定时器到期的时间，`function`是定时器函数，`state`是定时器的状态。`start_comm`和`start_pid`分别代表创建该定时器的进程名和进程ID。`cb_mode`指定该时钟回调函数的执行环境，有以下几种情况：

- (1) HRTIMER_CB_SOFTIRQ 时钟回调函数需要在时钟软中断环境中执行。
- (2) HRTIMER_CB_IRQSAFE 时钟回调函数需要在时钟中断环境中执行。
- (3) HRTIMER_CB_IRQSAFE_NO_RESTART 时钟不需要重新设置，也就是说该时钟是一次性的，某些时钟可以固定一个周期，每当时钟到期后都触发一次，这就需要重设。
- (4) HRTIMER_CB_IRQSAFE_NO_SOFTIRQ 当采用 High-Resolution Timers 后，需要根据软件设置的定时器来设置时钟设备的中断时机，那么原来需要周期性完成的工作，例如更新 jiffies，更新系统时间等工作，该在什么时候来完成呢？为了解决这个问题，在每个 CPU 的 High-Resolution Timers 队列中，总是有一个 tick emultation 时钟，当这个时钟到期时，就会执行上面说的这些工作。HRTIMER_CB_IRQSAFE_NO_SOFTIRQ 就是为这种特殊模式设置的。它的时钟回调函数必须在时钟中断环境下执行。

由于用户可能会修改系统时间，因此定时器被划分为两类，其中 `CLOCK_REALTIME` 是系统的实时定时器，也就是我们日常使用的绝对时间，对系统时间的修改将影响 `CLOCK_REALIME` 定时器。`CLOCK_MONOTONIC` 是内核启动以来的相对时间。因此又抽象出一个 `hrtimer_clock_base` 结构，其定义如下：

代码片段 9.30 节自 `include/linux/hrtimer.h`

```

1 struct hrtimer_clock_base {
2     struct hrtimer_cpu_base *cpu_base;

```

```

3   clockid_t index;
4   struct rb_root active;
5   struct rb_node *first;
6   ktime_t resolution;
7   ktime_t (*get_time)(void);
8   ktime_t (*get_softirq_time)(void);
9   ktime_t softirq_time;
10 #ifdef CONFIG_HIGH_RES_TIMERS
11   ktime_t offset;
12   int (*reprogram)(struct hrtimer *t,
13                     struct hrtimer_clock_base *b,
14                     ktime_t n);
15 #endif
16 };

```

`CLOCK_REALTIME` 时钟和 `CLOCK_MONOTONIC` 时钟各有一个 `hrtimer_clock_base` 结构，`hrtimer` 组成一个红黑树，其中 `first` 指向 `hrtimer` 的树根，`resolution` 是这类时钟的精度，`get_time` 是读取时间的函数，`reprogram` 函数用于操作某个外部硬件时钟设备，通过它可以设置硬件时钟下一次发出中断的时机。

在多处理器系统中，每一个 CPU 都有自己的时钟队列，因此每个 CPU 都要管理两个 `hrtimer_clock_base` 结构，分别是 `CLOCK_REALTIME` 时钟和 `CLOCK_MONOTONIC` 时钟，所以为每个 CPU 定义了 `hrtimer_cpu_base` 结构，而上面 `hrtimer_clock_base` 结构中的 `cpu_base` 就指向对应的 `hrtimer_cpu_base` 结构，`hrtimer_cpu_base` 定义如下：

代码片段 9.31 节自 `include/linux/hrtimer.h`

```

1 #define HRTIMER_MAX_CLOCK_BASES 2
2
3 struct hrtimer_cpu_base {
4     spinlock_t lock;
5     struct lock_class_key lock_key;
6     struct hrtimer_clock_base clock_base[HRTIMER_MAX_CLOCK_BASES];
7 #ifdef CONFIG_HIGH_RES_TIMERS
8     ktime_t expires_next;
9     int hres_active;
10    struct list_head cb_pending;
11    unsigned long nr_events;
12 #endif
13 };

```

`percpu` 变量 `hrtimer_bases` 就是 CPU 的 `hrtimer_cpu_base` 结构，其定义如下：

代码片段 9.32 节自 `kernel/hrtimer.c`

```

1 DEFINE_PER_CPU(struct hrtimer_cpu_base, hrtimer_bases) =
2 {
3     .clock_base =
4     {
5         {
6             .index = CLOCK_REALTIME,
7             .get_time = &ktime_get_real,
8             .resolution = KTIME_LOW_RES,
9         },
10        {
11            .index = CLOCK_MONOTONIC,
12            .get_time = &ktime_get,
13            .resolution = KTIME_LOW_RES,
14        },
15    }
16 };

```

上面分别初始定义了每个 CPU 的 CLOCK_REALTIME 时钟和 CLOCK_MONOTONIC 时钟，我们可以通过 timer_list 文件看到 CPU 的时钟。

代码片段 9.33 节自 /proc/timer_list

```

1 $ cat /proc/timer_list
2 Timer List Version: v0.3
3 HRTIMER_MAX_CLOCK_BASES: 2
4 now at 9230479046504 nsecs
5
6 cpu: 0
7 clock 0:
8     .index:      0
9     .resolution: 1 nsecs
10    .get_time:   ktime_get_real
11    .offset:     1213185148720372967 nsecs
12 active timers:
13 clock 1:
14     .index:      1
15     .resolution: 1 nsecs
16     .get_time:   ktime_get
17     .offset:     0 nsecs
18 active timers:
19 #0: <f245debc>, tick_sched_timer, S:01, tick_nohz_restart_sched_tick,
20     swapper/0
21 # expires at 9230480000000 nsecs [in 953496 nsecs]
22 #1: <f245debc>, it_real_fn, S:01, do_setitimer, syslogd/5659

```

```
22 # expires at 9235599619034 nsecs [in 5120572530 nsecs]
23 #2: <f245debc>, hrtimer_wakeup, S:01, do_nanosleep, plcc/6259
24 # expires at 9265226256945 nsecs [in 34747210441 nsecs]
25 #3: <f245debc>, hrtimer_wakeup, S:01, do_nanosleep, cron/5728
26 # expires at 9272327206698 nsecs [in 41848160194 nsecs]
27 #4: <f245debc>, hrtimer_wakeup, S:01, do_nanosleep, atd/5710
28 # expires at 10825842104141 nsecs [in 1595363057637 nsecs]
29 .expires_next : 9230480000000 nsecs
30 .hres_active : 1
31 .nr_events : 286607
32 .nohz_mode : 2
33 .idle_tick : 9230472000000 nsecs
34 .tick_stopped : 0
35 .idle_jiffies : 2232616
36 .idle_calls : 881579
37 .idle_sleeps : 559775
38 .idle_entrytime : 9230476026093 nsecs
39 .idle_sleeptime : 8670043951709 nsecs
40 .last_jiffies : 2232619
41 .next_jiffies : 2232625
42 .idle_expires : 9230500000000 nsecs
43 jiffies: 2232619
44
45 cpu: 1
46 clock 0:
47 .index: 0
48 .resolution: 1 nsecs
49 .get_time: ktime_get_real
50 .offset: 1213185148720372967 nsecs
51 active timers:
52 clock 1:
53 .index: 1
54 .resolution: 1 nsecs
55 .get_time: ktime_get
56 .offset: 0 nsecs
57 active timers:
58 #0: <f245debc>, tick_sched_timer, S:01, tick_noht_restart_sched_tick,
      swapper/0
59 # expires at 9230481000000 nsecs [in 1953496 nsecs]
60 #1: <f245debc>, hrtimer_wakeup, S:01, futex_wait, firefox/11155
61 # expires at 9230488291972 nsecs [in 9245468 nsecs]
62 .expires_next : 9230481000000 nsecs
```

```

63 .hres_active      : 1
64 .nr_events        : 300066
65 .nohz_mode        : 2
66 .idle_tick        : 9230473000000 nsecs
67 .tick_stopped     : 0
68 .idle_jiffies    : 2232617
69 .idle_calls       : 565926
70 .idle_sleeps     : 294841
71 .idle_entrytime   : 9230475369182 nsecs
72 .idle_sleeptime   : 8748658547288 nsecs
73 .last_jiffies    : 2232618
74 .next_jiffies    : 2232620
75 .idle_expires    : 9230480000000 nsecs
76 jiffies: 2232619
77
78 .....

```

上面每个 CPU 都有两个 clock, 分别对应 CLOCK_REALTIME 和 CLOCK_MONOTONIC, 另外还可以看到各个 CPU 的时钟队列, 以及队列中最早到期的时间等信息。

9.3.2 High-Resolution Timers 初始化

由于启动时, 时钟队列是空的, 因此 High-Resolution Timers 的初始化比较简单, 初始化工作是由 `hrtimers_init()` 来完成的:

代码片段 9.34 节自 `kernel/hrtimer.c`

```

1 [start_kernel() -> hrtimers_init()]
2
3 static struct notifier_block __cpuinitdata hrtimers_nb = {
4     .notifier_call = hrtimer_cpu_notify,
5 };
6
7 void __init hrtimers_init(void)
8 {
9     hrtimer_cpu_notify(&hrtimers_nb,
10                     (unsigned long)CPU_UP_PREPARE,
11                     (void *)(long)smp_processor_id());
12     register_cpu_notifier(&hrtimers_nb);
13 }
14 #ifdef CONFIG_HIGH_RES_TIMERS
15     open_softirq(HRTIMER_SOFTIRQ, run_hrtimer_softirq, NULL);
16 #endif

```

17 }

hrtimers_init()注册了一个 notifier 对象，其处理函数为 hrtimer_cpu_notify()，起初这个 notifier 对象还没有注册，因此直接以 CPU_UP_PREPARE 为参数调用了 hrtimer_cpu_notify()函数，hrtimer_cpu_notify()将调用 init_hrtimers_cpu()，这个函数比较简单，因此就不进一步分析了。

代码片段 9.35 节自 kernel/hrtimer.c

```
1 static int __cpuinit hrtimer_cpu_notify(struct notifier_block *self,
2                                         unsigned long action,
3                                         void *hcpu)
4 {
5     unsigned int cpu = (long)hcpu;
6
7     switch (action) {
8     case CPU_UP_PREPARE:
9     case CPU_UP_PREPARE_FROZEN:
10        init_hrtimers_cpu(cpu);
11        break;
12 #ifdef CONFIG_HOTPLUG_CPU
13     case CPU_DEAD:
14     case CPU_DEAD_FROZEN:
15        clockevents_notify(CLOCK_EVT_NOTIFY_CPU_DEAD, &cpu);
16        migrate_hrtimers(cpu);
17        break;
18 #endif
19     default:
20        break;
21    }
22
23    return NOTIFY_OK;
24 }
25
26 static void __cpuinit init_hrtimers_cpu(int cpu)
27 {
28     struct hrtimer_cpu_base *cpu_base = &per_cpu(hrtimer_bases, cpu);
29     int i;
30
31     spin_lock_init(&cpu_base->lock);
32     lockdep_set_class(&cpu_base->lock, &cpu_base->lock_key);
33     for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++)
34         cpu_base->clock_base[i].cpu_base = cpu_base;
```

```

35     hrtimer_init_hres(cpu_base);
36 }

```

内核启动之初，并没有启动 High-Resolution Timers，每当执行时钟软中断时，会检查能否需要进入 High-Resolution Timers 模式，如果能则开始进一步的初始化工作。

代码片段 9.36 节自 kernel/hrtimer.c

```

1 [ run_timer_softirq() -> hrtimer_run_queues()]
2
3 void hrtimer_run_queues(void)
4 {
5     struct hrtimer_cpu_base *cpu_base = &__get_cpu_var(hrtimer_bases);
6     int i;
7
8     /* 如果 High-Resolution Timers 已经启用则返回。*/
9     if (hrtimer_hres_active())
10        return;
11
12    if (tick_check_oneshot_change(!hrtimer_is_hres_enabled()))
13        if (hrtimer_switch_to_hres())
14            return;
15    hrtimer_get_softirq_time(cpu_base);
16    /* 执行 High-Resolution Timers 的软中断函数。*/
17    for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++)
18        run_hrtimer_queue(cpu_base, i);
19 }
20
21 static int hrtimer_switch_to_hres(void)
22 {
23     int cpu = smp_processor_id();
24     struct hrtimer_cpu_base *base = &per_cpu(hrtimer_bases, cpu);
25     unsigned long flags;
26
27     if (base->hres_active)
28         return 1;
29     local_irq_save(flags);
30     /* 初始化。*/
31     if (tick_init_highres()) {
32         local_irq_restore(flags);
33         printk(KERN_WARNING "Could not switch to high resolution mode on CPU %d
34             \n", cpu);
35     }

```

```
36 base->hres_active = 1;
37 base->clock_base[CLOCK_REALTIME].resolution = KTIME_HIGH_RES;
38 base->clock_base[CLOCK_MONOTONIC].resolution = KTIME_HIGH_RES;
39
40 /* 设置一个 tick emultation 时钟. */
41 tick_setup_sched_timer();
42
43 /* "Retrigger" the interrupt to get things going */
44 retrigger_next_event(NULL);
45 local_irq_restore(flags);
46 printk(KERN_DEBUG "Switched to high resolution mode on CPU %d\n",
47        smp_processor_id());
48 return 1;
49 }
```

先来看看 tick_init_highres():

代码片段 9.37 节自 kernel/time/tick-oneshot.c

```
1 [run_timer_softirq() -> hrtimer_run_queues() -> hrtimer_switch_to_hres() ->
   tick_init_highres()]
2
3 int tick_init_highres(void)
4 {
5     return tick_switch_to_oneshot(hrtimer_interrupt);
6 }
7
8 int tick_switch_to_oneshot(void (*handler)(struct clock_event_device *))
9 {
10    struct tick_device *td = &__get_cpu_var(tick_cpu_device);
11    struct clock_event_device *dev = td->evtdev;
12    .....
13    td->mode = TICKDEV_MODE_ONESHOT;
14    /*
15     * 把 CPU 本地 clock event device 的 event_handler
16     * 设置为 hrtimer_interrupt()。
17     */
18    dev->event_handler = handler;
19    clockevents_set_mode(dev, CLOCK_EVT_MODE_ONESHOT);
20    tick_broadcast_switch_to_oneshot();
21    return 0;
22 }
23
24 void tick_broadcast_switch_to_oneshot(void)
```

```

25 {
26     struct clock_event_device *bc;
27     unsigned long flags;
28
29     spin_lock_irqsave(&tick_broadcast_lock, flags);
30     /*
31      * 把全局 broadcast device 的 event_handler
32      * 设置为 tick_handle_oneshot_broadcast.
33      */
34     tick_broadcast_device.mode = TICKDEV_MODE_ONESHOT;
35     bc = tick_broadcast_device.evtdev;
36     if (bc)
37         tick_broadcast_setup_oneshot(bc);
38     spin_unlock_irqrestore(&tick_broadcast_lock, flags);
39 }
40
41 void tick_broadcast_setup_oneshot(struct clock_event_device *bc)
42 {
43     bc->event_handler = tick_handle_oneshot_broadcast;
44     clockevents_set_mode(bc, CLOCK_EVT_MODE_ONESHOT);
45     bc->next_event.tv64 = KTIME_MAX;
46 }

```

由于 HPET 和 PIT 都可以作为 broadcast tick device，当没有 HPET 时，才会使用 PIT。全局变量 global_clock_event 和 tick_broadcast_device⁸就是 HPET 或者 PIT，至此，终于明白为什么通过/proc/timer_list 文件看到的 event_handler 分别是 tick_handle_oneshot_broadcast()和 hrtimer_interrupt()了。

现在，我们再来看看 tick_setup_sched_timer()，这个函数负责设置一个 tick emultation 设备，切换到 High-Resolution Timers 后，时钟中断设备主要不是为了周期性的 tick 而发出中断，所以，每一个 CPU 有一个虚拟的 tick 定时器，它的结构如下：

代码片段 9.38 节自 include/linux/tick.h

```

1 struct tick_sched {
2     struct hrtimer    sched_timer;
3     unsigned long    check_clocks;
4     enum tick_nohz_mode nohz_mode;
5     ktime_t        idle_tick;
6     int            tick_stopped;
7     unsigned long   idle_jiffies;
8     unsigned long   idle_calls;

```

⁸见第242页。

```
9     unsigned long    idle_sleeps;
10    ktime_t      idle_entrytime;
11    ktime_t      idle_sleeptime;
12    ktime_t      sleep_length;
13    unsigned long   last_jiffies;
14    unsigned long   next_jiffies;
15    ktime_t      idle_expires;
16 };
```

tick_setup_sched_timer()的主要作用就是设置结构中的 hrtimer:

代码片段 9.39 节自 kernel/time/tick-sched.c

```
1 void tick_setup_sched_timer(void)
2 {
3     struct tick_sched *ts = &__get_cpu_var(tick_cpu_sched);
4     ktime_t now = ktime_get();
5     u64 offset;
6     /*
7     * Emulate tick processing via per-CPU hrtimers:
8     */
9     /* 初始化 hrtimer.*/
10    hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
11    ts->sched_timer.function = tick_sched_timer;
12    ts->sched_timer.cb_mode = HRTIMER_CB_IRQSAFE_NO_SOFTIRQ;
13
14    /* Get the next period (per cpu) */
15    ts->sched_timer.expires = tick_init_jiffy_update();
16    offset = ktime_to_ns(tick_period) >> 1;
17    do_div(offset, num_possible_cpus());
18    offset *= smp_processor_id();
19    ts->sched_timer.expires = ktime_add_ns(ts->sched_timer.expires, offset);
20
21    /* 添加 hrtimer.*/
22    for (;;) {
23        hrtimer_forward(&ts->sched_timer, now, tick_period);
24        hrtimer_start(&ts->sched_timer,
25                      ts->sched_timer.expires,
26                      HRTIMER_MODE_ABS);
27
28        /* Check, if the timer was already in the past */
29        if (hrtimer_active(&ts->sched_timer))
30            break;
31    }
32    now = ktime_get();
```

```

32     )
33
34 #ifdef CONFIG_NO_HZ
35     if (tick_nohz_enabled)
36         ts->nohz_mode = NOHZ_MODE_HIGHRES;
37 #endif
38 }

```

这个函数把 tick emulation 设备的回调函数设置为 tick_sched_timer(), 我们将在第9.4对该函数进行深入分析。

9.3.3 High-Resolution Timers 操作

现在 High-Resolution Timer 已经初始化完毕，接下来我们来看看 hrtimer 的建立，回调函数的执行，hrtimer 的删除等工作。

1. hrtimer 的初始化

初始化是由 hrtimer_init()完成的：

代码片段 9.40 节自 kernel/hrtimer.c

```

1 void hrtimer_init(struct hrtimer *timer,
2                     clockid_t clock_id,
3                     enum hrtimer_mode mode)
4 {
5     struct hrtimer_cpu_base *cpu_base;
6
7     memset(timer, 0, sizeof(struct hrtimer));
8
9     /* 获取当前 CPU 的 hrtimer_cpu_base 结构指针。 */
10    cpu_base = &__raw_get_cpu_var(hrtimer_bases);
11
12    if (clock_id == CLOCK_REALTIME && mode != HRTIMER_MODE_ABS)
13        clock_id = CLOCK_MONOTONIC;
14    /* 每个 CPU 有两个时钟队列，根据模式判断这个时钟该初始化所使用的队列。 */
15    timer->base = &cpu_base->clock_base[clock_id];
16    hrtimer_init_timer_hres(timer);
17
18    /* 初始化统计信息。 */
19 #ifdef CONFIG_TIMER_STATS
20     timer->start_site = NULL;
21     timer->start_pid = -1;

```

```
22     memset(timer->start_comm, 0, TASK_COMM_LEN);
23 #endif
24 }
25
26 static inline void hrtimer_init_timer_hres(struct hrtimer *timer)
27 {
28     INIT_LIST_HEAD(&timer->cb_entry);
29 }
```

2. hrtimer 的添加

为了能够进行快速添加、删除、查找等操作，hrtimer 采用红黑树管理。添加操作包括红黑树的操作，另外时钟中断设备上面可以设置的硬件定时器毕竟有限，如果当前添加的时钟的到期时间早于当前硬件时钟设备设置的发出中断的时间，那么就需要重新设置硬件时钟设备发出中断的时间。

代码片段 9.41 节自 kernel/hrtimer.c

```
1 hrtimer_start(struct hrtimer *timer,
2                 ktime_t tim,
3                 const enum hrtimer_mode mode)
4 {
5     struct hrtimer_clock_base *base, *new_base;
6     unsigned long flags;
7     int ret;
8
9     base = lock_hrtimer_base(timer, &flags);
10
11    /* 如果这个 hrtimer 已经在队列中，则先将其移除。*/
12    /* Remove an active timer from the queue: */
13    ret = remove_hrtimer(timer, base);
14    /*
15     * 检查是否需要转换 clock base，也就是判断添加到 CLOCK_REALTIME 队列
16     * 还是 CLOCK_MONOTONIC 队列。如果这个 hrtimer 已经在队列中（有可能在另
17     * 外的 CPU 的队列中），并且有没完成的任务而不能被删除，而现在再次要
18     * 添加时钟队列和原来的不一致。
19     */
20    /* Switch the timer base, if necessary: */
21    new_base = switch_hrtimer_base(timer, base);
22    /*
23     * 真实时间的时钟，那么需要在过期时间的基础上加上当前时间，
24     * 相对时钟则不必进行该项操作。
25     */
```

```

26     if (mode == HRTIMER_MODE_REL) {
27         tim = ktime_add(tim, new_base->get_time());
28
29 #ifdef CONFIG_TIME_LOW_RES
30         tim = ktime_add(tim, base->resolution);
31 #endif
32         if (tim.tv64 < 0)
33             tim.tv64 = KTIME_MAX;
34     }
35     timer->expires = tim;
36
37     /* 设置/proc/timer_stats 文件的统计信息。*/
38     timer_stats_hrtimer_set_start_info(timer);
39
40     /*
41      * 也许这个 hrtimer 以前被添加到另外的 CPU 时钟队列中，且不能被删除。
42      * 只有 hrtimer 所在的 CPU 和执行当前代码的 CPU 为同一个 CPU 时，才允许必
43      * 要时重新设置时钟中断设备的下一次中断时间。
44      */
45
46     /*
47      * Only allow reprogramming if the new base is on this CPU.
48      * (it might still be on another CPU if the timer was pending)
49      */
50     enqueue_hrtimer(timer, new_base,
51                      new_base->cpu_base == &__get_cpu_var(hrtimer_bases));
52     unlock_hrtimer_base(timer, &flags);
53 }

```

`enqueue_hrtimer()`把 `hrtimer` 对象添加到队列中，如果过期时间是最早的，就要重新设置时钟中断设备了⁹。

代码片段 9.42 节自 `kernel/hrtimer.c`

```

1 static void enqueue_hrtimer(struct hrtimer *timer,
2                             struct hrtimer_clock_base *base,
3                             int reprogram)
4 {
5     struct rb_node **link = &base->active.rb_node;
6     struct rb_node *parent = NULL;
7     struct hrtimer *entry;
8     int leftmost = 1;

```

⁹这里根据过期时间设置红黑树，关于红黑树操作，请读者查阅数据结构的相关书籍。

```
9  /*
10   * Find the right place in the rbtree:
11   */
12  while (*link) {
13      parent = *link;
14      entry = rb_entry(parent, struct hrtimer, node);
15      if (timer->expires.tv64 < entry->expires.tv64) {
16          link = &(*link)->rb_left;
17      } else {
18          link = &(*link)->rb_right;
19          leftmost = 0;
20      }
21  }
22  /*
23   * Insert the timer to the rbtree and check whether it
24   * replaces the first pending timer
25   */
26  if (leftmost) {
27      /*
28       * Reprogram the clock event device. When the timer is already
29       * expired hrtimer_enqueue_reprogram has either called the
30       * callback or added it to the pending list and raised the
31       * softirq.
32       *
33       * This is a NOP for !HIGHRES
34       */
35      if (reprogram && hrtimer_enqueue_reprogram(timer, base))
36          return;
37
38      base->first = &timer->node;
39  }
40
41  rb_link_node(&timer->node, parent, link);
42  rb_insert_color(&timer->node, &base->active);
43  /*
44   * HRTIMER_STATE_ENQUEUED is or'ed to the current state to preserve the
45   * state of a possibly running callback.
46   */
47  timer->state |= HRTIMER_STATE_ENQUEUED;
48 }
```

对时钟中断设备的到期时间的重新设置由 `hrtimer_enqueue_reprogram()` 完成，但是从申请添加时钟执行到这里，也需要一个短暂的时间段，也许在这段时间中，这个 `hrtimer` 就过期了，`hrtimer_enqueue_reprogram()` 需要检测这种情况，否则把时钟的到期时间设置为“过去”是很危险的。

代码片段 9.43 节自 `kernel/hrtimer.c`

```

1 static inline int hrtimer_enqueue_reprogram(struct hrtimer *timer,
2                                         struct hrtimer_clock_base *base)
3 {
4     if (base->cpu_base->hres_active && hrtimer_reprogram(timer, base)) {
5         /* Timer is expired, act upon the callback mode */
6         switch(timer->cb_mode) {
7             case HRTIMER_CB_IRQSAFE_NO_RESTART:
8                 /*
9                  * We can call the callback from here. No restart
10                 * happens, so no danger of recursion
11                 */
12                 BUG_ON(timer->function(timer) != HRTIMER_NORESTART);
13                 return 1;
14             case HRTIMER_CB_IRQSAFE_NO_SOFTIRQ:
15                 /*
16                  * This is solely for the sched tick emulation with
17                  * dynamic tick support to ensure that we do not
18                  * restart the tick right on the edge and end up with
19                  * the tick timer in the softirq ! The calling site
20                  * takes care of this.
21                 */
22                 return 1;
23             case HRTIMER_CB_IRQSAFE:
24             case HRTIMER_CB_SOFTIRQ:
25                 /*
26                  * Move everything else into the softirq pending list !
27                 */
28                 list_add_tail(&timer->cb_entry,
29                               &base->cpu_base->cb_pending);
30                 timer->state = HRTIMER_STATE_PENDING;
31                 raise_softirq(HRTIMER_SOFTIRQ);
32                 return 1;
33             default:
34                 BUG();
35         }
36     }

```

```
37     return 0;
38 }
```

这段代码注释得很详细，如果 hrtimer 已经过期，则 hrtimer_reprogram()返回-ETIME，因此要根据定时器回调函数的执行环境，选择现在执行其回调函数，还是请求 hrtimer 的软件中断来执行其回调函数。现在再来看看 hrtimer_reprogram()：

代码片段 9.44 节自 kernel/hrtimer.c

```
1 static int hrtimer_reprogram(struct hrtimer *timer,
2                               struct hrtimer_clock_base *base)
3 {
4     ktime_t *expires_next = &__get_cpu_var(hrtimer_bases).expires_next;
5     ktime_t expires = ktime_sub(timer->expires, base->offset);
6     int res;
7
8     /* 如果定时器回调函数已经在另外一个 CPU 上执行。*/
9     /*
10      * When the callback is running, we do not reprogram the clock event
11      * device. The timer callback is either running on a different CPU or
12      * the callback is executed in the hrtimer_interrupt context. The
13      * reprogramming is handled either by the softirq, which called the
14      * callback or at the end of the hrtimer_interrupt.
15      */
16    if (hrtimer_callback_running(timer))
17        return 0;
18
19    if (expires.tv64 >= expires_next->tv64)
20        return 0;
21    /*
22     * Clockevents returns -ETIME, when the event was in the past.
23     */
24    res = tick_program_event(expires, 0);
25    if (!IS_ERR_VALUE(res))
26        *expires_next = expires;
27    return res;
28 }
29
30 int tick_program_event(ktime_t expires, int force)
31 {
32     struct clock_event_device *dev = __get_cpu_var(tick_cpu_device).evtdev;
33     ktime_t now = ktime_get();
34     while (1) {
35         int ret = clockevents_program_event(dev, expires, now);
```

```
36
37     if (!ret || !force)
38         return ret;
39     now = ktime_get();
40     expires = ktime_add(now, ktime_set(0, dev->min_delta_ns));
41 }
42 }
43
44 int clockevents_program_event(struct clock_event_device *dev,
45                               ktime_t expires,
46                               ktime_t now)
47 {
48     unsigned long long clc;
49     int64_t delta;
50
51     if (unlikely(expires.tv64 < 0)) {
52         WARN_ON_ONCE(1);
53         return -ETIME;
54     }
55     delta = ktime_to_ns(ktime_sub(expires, now));
56     if (delta <= 0)
57         return -ETIME;
58     dev->next_event = expires;
59     if (dev->mode == CLOCK_EVT_MODE_SHUTDOWN)
60         return 0;
61
62     if (delta > dev->max_delta_ns)
63         delta = dev->max_delta_ns;
64     if (delta < dev->min_delta_ns)
65         delta = dev->min_delta_ns;
66     clc = delta * dev->mult;
67     clc >>= dev->shift;
68
69     /* 调用 clock event device 对象的 set_next_event()。 */
70     return dev->set_next_event((unsigned long) clc, dev);
71 }
```

3. hrtimer 的删除

删除工作由 `_remove_hrtimer()` 来完成，同样如果删除的时钟到期时间是最早的，则需要重新设置时钟中断设备的到期时间。

代码片段 9.45 节自 kernel/hrtimer.c

```

1 [remove_hrtimer() -> __remove_hrtimer()]
2
3 static void __remove_hrtimer(struct hrtimer *timer,
4                               struct hrtimer_clock_base *base,
5                               unsigned long newstate,
6                               int reprogram)
7 {
8     /* High res. callback list. NOP for !HIGHRES */
9     if (hrtimer_cb_pending(timer))
10        hrtimer_remove_cb_pending(timer);
11    else {
12        /*
13         * Remove the timer from the rbtree and replace the
14         * first entry pointer if necessary.
15         */
16        if (base->first == &timer->node) {
17            base->first = rb_next(&timer->node);
18            /* Reprogram the clock event device. if enabled */
19            if (reprogram && hrtimer_hres_active())
20                hrtimer_force_reprogram(base->cpu_base);
21        }
22        rb_erase(&timer->node, &base->active);
23    }
24    timer->state = newstate;
25 }
```

4. hrtimer 的系统调用举例

hrtimer 是 POSIX 的高精度时钟的内核实现，其上层 C 接口包括 nanosleep(), clock_settime() 等函数，正是因为有了 hrtimer，Linux 才能按照 POSIX 的规定为应用程序提供高精度定时器，这里以 nanosleep() 为例，来介绍 hrtimer 是如何和应用程序一起工作的。nanosleep 通过系统调用执行的内核函数为 sys_nanosleep()。

代码片段 9.46 节自 kernel/hrtimer.c

```

1 asmlinkage long sys_nanosleep(struct timespec __user *rqtp,
2                                struct timespec __user *rmt)
3 {
4     struct timespec tu, rmt;
5     int ret;
6 }
```

```
7  if (copy_from_user(&tu, rqtp, sizeof(tu)))
8      return -EFAULT;
9  if (!timespec_valid(&tu))
10     return -EINVAL;
11  ret = hrtimer_nanosleep(&tu, rmtp ? &rmt : NULL,
12                         HRTIMER_MODE_REL,
13                         CLOCK_MONOTONIC);
14  if (ret && rmtp) {
15      if (copy_to_user(rmtp, &rmt, sizeof(*rmtp)))
16          return -EFAULT;
17  }
18
19  return ret;
20 }
21
22 long hrtimer_nanosleep(struct timespec *rqtp,
23                         struct timespec *rmtp,
24                         const enum hrtimer_mode mode,
25                         const clockid_t clockid)
26 {
27     struct restart_block *restart;
28     struct hrtimer_sleeper t;
29     ktime_t rem;
30
31     /* 初始化一个 hrtimer. */
32     hrtimer_init(&t.timer, clockid, mode);
33     t.timer.expires = timespec_to_ktime(*rqtp);
34     if (do_nanosleep(&t, mode))
35         return 0;
36     .....
37 }
38
39 static int __sched do_nanosleep(struct hrtimer_sleeper *t,
40                                 enum hrtimer_mode mode)
41 {
42     /* 把 hrtimer 的回调函数设置为 hrtimer_wakeup(). */
43     hrtimer_init_sleeper(t, current);
44     do {
45         /* 当前进程设置为等待状态。*/
46         set_current_state(TASK_INTERRUPTIBLE);
47         /* 把 hrtimer 添加到队列中。*/
48         hrtimer_start(&t->timer, t->timer.expires, mode);
```

```
49
50     /* 当前进程主动让出 CPU. */
51     if (likely(t->task))
52         schedule();
53
54     /* 删除该 hrtimer. */
55     hrtimer_cancel(&t->timer);
56     mode = HRTIMER_MODE_ABS;
57
58 } while (t->task && !signal_pending(current));
59
60 return t->task == NULL;
61 }
62
63 void hrtimer_init_sleeper(struct hrtimer_sleeper *sl,
64                           struct task_struct *task)
65 {
66     sl->timer.function = hrtimer_wakeup;
67     sl->task = task;
68 #ifdef CONFIG_HIGH_RES_TIMERS
69     sl->timer.cb_mode = HRTIMER_CB_IRQSAFE_NO_RESTART;
70 #endif
71 }
```

必要时，调用 `nanosleep()` 的进程会主动让出 CPU，这时调度器会调度其他进程，当定时器到期时，会执行回调函数 `hrtimer_wakeup()`，该函数负责唤醒睡眠的进程。

代码片段 9.47 节自 `kernel/hrtimer.c`

```
1 static enum hrtimer_restart hrtimer_wakeup(struct hrtimer *timer)
2 {
3     struct hrtimer_sleeper *t = container_of(timer,
4                                               struct hrtimer_sleeper,
5                                               timer);
6     struct task_struct *task = t->task;
7     t->task = NULL;
8     if (task)
9         wake_up_process(task);
10
11    return HRTIMER_NORESTART;
12 }
```

9.4 时钟中断处理

前面介绍了各种 clock event device 对象，内核中能发出时钟中断的设备也很多，包括 PIT, HPET, Local APIC 等。读者一定感到迷惑，clock event device 对象是怎么和时钟中断对应起来的？一般来说，PIT 和 HPET 都使用 0x20 号中断向量，其中断处理程序为 timer_interrupt()，而 Local APIC 则使用中断向量 0xef，其处理入口为 apic_timer_interrupt()（见第6章）。我们先来看看 timer_interrupt()。

代码片段 9.48 节自 arch/x86/kernel/time_32.c

```

1 irqreturn_t timer_interrupt(int irq, void *dev_id)
2 {
3     .....
4     do_timer_interrupt_hook();
5     .....
6     return IRQ_HANDLED;
7 }
8
9 struct clock_event_device *global_clock_event;
10
11 static inline void do_timer_interrupt_hook(void)
12 {
13     global_clock_event->event_handler(global_clock_event);
14 }
```

在 clock event device 对象初始化过程中，我们看到 global_event_device 指针初始指向 hpet_clockevent 对象¹⁰，而它的 event_handler 是可以动态改变的，我们留意到启动之初其 event_handler 为 tick_handle_periodic() 函数。现在来看看 Local APIC 的时钟中断。

代码片段 9.49 节自 arch/x86/kernel/entry_32.S

```

1 BUILD_INTERRUPT(apic_timer_interrupt, LOCAL_TIMER_VECTOR)
2
3 #define BUILD_INTERRUPT(name, nr) \
4 ENTRY(name)           \
5     pushl $(nr);       \
6     SAVE_ALL;          \
7     movl %esp,%eax;    \
8     call smp_##name;   \
9     jmp ret_from_intr; \
10 ENDPROC(name)
```

由此可见，apic_timer_interrupt 保存中断现场后，会调用 smp_apic_timer_interrupt()。

¹⁰如果没有 HPET，则初始化为 pit_clockevent 对象。

代码片段 9.50 节自 arch/x86/kernel/apic_32.c

```

1 void fastcall smp_apic_timer_interrupt(struct pt_regs *regs)
2 {
3     struct pt_regs *old_regs = set_irq_regs(regs);
4
5     ack_APIC_irq();
6     irq_enter();
7     local_apic_timer_interrupt();
8     irq_exit();
9
10    set_irq_regs(old_regs);
11 }
12
13
14 static void local_apic_timer_interrupt(void)
15 {
16     int cpu = smp_processor_id();
17     struct clock_event_device *evt = &per_cpu(lapic_events, cpu);
18
19     if (!evt->event_handler) {
20         printk(KERN_WARNING "Spurious LAPIC timer interrupt on cpu %d\n", cpu);
21
22         /* Switch it off */
23         lapic_timer_setup(CLOCK_EVT_MODE_SHUTDOWN, evt);
24         return;
25     }
26     per_cpu(irq_stat, cpu).apic_timer_irqs++;
27     evt->event_handler(evt);
28 }

```

同样 lapic 的 event_handler 也可以动态改变。到这里，我们已经知道 clock event device 对象和具体的时钟中断设备及 CPU 的关系了。简单地总结，每一个 CPU 的 Local APIC 的时钟抽象为本地 CPU 的 lapic clock event device 对象，而 PIT，HPET 这一类设备的 clock event device 对象对应于 global_clock_event 对象。当 Local APIC 的时钟发出中断时，由本地 CPU 的 lapic clock event device 对象的 event_handler 处理。当 PIT，HPET 发出中断时，由全局的 clock event device 处理。

虽然 clock event device 对象的 event_handler 可以动态改变，但是时钟中断函数要完成的工作却是不变的。主要包括更新系统时间，增加 jiffies，统计当前进程的运行时间，统计 kernel profile 信息，请求时钟软中断等。

现在先来看看 tick_handle_periodic():

代码片段 9.51 节自 kernel/time/tick-common.c

```

1 void tick_handle_periodic(struct clock_event_device *dev)
2 {
3     int cpu = smp_processor_id();
4     ktime_t next;
5
6     tick_periodic(cpu);
7     if (dev->mode != CLOCK_EVT_MODE_ONESHOT)
8         return;
9
10    next = ktime_add(dev->next_event, tick_period);
11    for (;;) {
12        if (!clockevents_program_event(dev, next, ktime_get()))
13            return;
14        tick_periodic(cpu);
15        next = ktime_add(next, tick_period);
16    }
17 }

```

在 ONESHOT 模式下，每一次时钟中断处理时都需要设置下一次中断的时机，第10行就设置下一次中断时机。但是对于周期性的时钟来说，一般只需要在初始化时把中断周期告诉中断设备就可以了。时钟中断的工作是在 tick_periodic()中完成的：

代码片段 9.52 节自 kernel/time/tick-common.c

```

1 static void tick_periodic(int cpu)
2 {
3     if (tick_do_timer_cpu == cpu) {
4         write_seqlock(&xtime_lock);
5         /* Keep track of the next tick event */
6         tick_next_period = ktime_add(tick_next_period, tick_period);
7         do_timer(1);
8         write_sequnlock(&xtime_lock);
9     }
10
11     update_process_times(user_mode(get_irq_regs()));
12     profile_tick(CPU_PROFILING);
13 }

```

do_timer 函数是更新系统时间(见第9.4.1节)，update_process_times 函数是统计进程的时间信息，同时还会请求时钟软中断，profile_tick()用于性能分析。这些工作是时钟中断必须完成的，因此我们暂时把这几个函数先放一放。再来看看 hrtimer_interrupt()：

代码片段 9.53 节自 kernel/hrtimer.c

```
1 void hrtimer_interrupt(struct clock_event_device *dev)
2 {
3     struct hrtimer_cpu_base *cpu_base =
4         &__get_cpu_var(hrtimer_bases);
5
6     struct hrtimer_clock_base *base;
7     ktime_t expires_next, now;
8     int i, raise = 0;
9     BUG_ON(!cpu_base->hres_active);
10    cpu_base->nr_events++;
11    dev->next_event.tv64 = KTIME_MAX;
12    retry:
13    now = ktime_get();
14    expires_next.tv64 = KTIME_MAX;
15    base = cpu_base->clock_base;
16    /* 分别处理 CLOCK_REALTIME 和 CLOCK_MONOTONIC。 */
17    for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++) {
18        ktime_t basenow;
19        struct rb_node *node;
20        spin_lock(&cpu_base->lock);
21        basenow = ktime_add(now, base->offset);
22        while ((node = base->first)) {
23            struct hrtimer *timer;
24            timer = rb_entry(node, struct hrtimer, node);
25            /*
26             * 由于是根据红黑树按序处理，所以当遇到第一个还没过期的定时器时，
27             * 就说明剩下的 hrtimer 全部都没有过期。
28             */
29            if (basenow.tv64 < timer->expires.tv64) {
30                ktime_t expires;
31                expires = ktime_sub(timer->expires, base->offset);
32                /* expires_next 用来跟踪时钟中断设备下一次发出中断的时机。 */
33                if (expires.tv64 < expires_next.tv64)
34                    expires_next = expires;
35                break;
36            }
37
38            /* 定时器回调函数需要在软中断中处理移除该 hrtimer，并请求软件中断。 */
39            /* Move softirq callbacks to the pending list */
40            if (timer->cb_mode == HRTIMER_CB_SOFTIRQ) {
41                __remove_hrtimer(timer, base, HRTIMER_STATE_PENDING, 0);
```

```

42     list_add_tail(&timer->cb_entry, &base->cpu_base->cb_pending);
43     raise = 1;
44     continue;
45 }
46 __remove_hrtimer(timer, base, HRTIMER_STATE_CALLBACK, 0);
47 timer_stats_account_hrtimer(timer);
48 /*
49  * Note: We clear the CALLBACK bit after
50  * enqueue_hrtimer to avoid reprogramming of
51  * the event hardware. This happens at the end
52  * of this function anyway.
53 */
54 /* 定时器的回调函数需要在中断环境中执行，则直接允许其回调函数。*/
55 if (timer->function(timer) != HRTIMER_NORESTART) {
56     BUG_ON(timer->state != HRTIMER_STATE_CALLBACK);
57     /* 必要时再次排队该 hrtimer. */
58     enqueue_hrtimer(timer, base, 0);
59 }
60 timer->state &= ~HRTIMER_STATE_CALLBACK;
61 }
62 spin_unlock(&cpu_base->lock);
63 base++;
64 }
65 cpu_base->expires_next = expires_next;
66 /* Reprogramming necessary ? */
67 if (expires_next.tv64 != KTIME_MAX) {
68     if (tick_program_event(expires_next, 0))
69         goto retry;
70 }
71 /* Raise softirq ? */
72 if (raise)
73     raise_softirq(HRTIMER_SOFTIRQ);
74 }

```

软件设置的定时器的回调函数千差万别，读者可以参见第265页代码片段9.47的例子来理解这段代码。这里我们要说的是由 tick_setup_sched_timer()¹¹函数设置的 tick emulation 时钟的回调函数 tick_sched_timer()，这个函数负责完成时钟中断的常规工作。

代码片段 9.54 节自 kernel/time/tick-sched.c

```

1 static enum hrtimer_restart tick_sched_timer(struct hrtimer *timer)
2 {

```

¹¹见第256页代码片段9.39。

```
3 struct tick_sched *ts = container_of(timer, struct tick_sched,
4     sched_timer);
5 struct hrtimer_cpu_base *base = timer->base->cpu_base;
6 struct pt_regs *regs = get_irq_regs();
7 ktime_t now = ktime_get();
8 int cpu = smp_processor_id();
9
10 #ifdef CONFIG_NO_HZ
11 /*
12  * Check if the do_timer duty was dropped. We don't care about
13  * concurrency: This happens only when the cpu in charge went
14  * into a long sleep. If two cpus happen to assign themselves to
15  * this duty, then the jiffies update is still serialized by
16  * xtime_lock.
17 */
18 if (unlikely(tick_do_timer_cpu == -1))
19     tick_do_timer_cpu = cpu;
20
21 /* Check, if the jiffies need an update */
22 if (tick_do_timer_cpu == cpu)
23     tick_do_update_jiffies64(now);
24 /*
25  * Do not call, when we are not in irq context and have
26  * no valid regs pointer
27 */
28 if (regs) {
29 /*
30  * When we are idle and the tick is stopped, we have to touch
31  * the watchdog as we might not schedule for a really long
32  * time. This happens on complete idle SMP systems while
33  * waiting on the login prompt. We also increment the "start of
34  * idle" jiffy stamp so the idle accounting adjustment we do
35  * when we go busy again does not account too much ticks.
36 */
37 if (ts->tick_stopped) {
38     touch_softlockup_watchdog();
39     ts->idle_jiffies++;
40 }
41 /*
42  * update_process_times() might take tasklist_lock, hence
43  * drop the base lock. sched-tick hrtimers are per-CPU and
```

```

44     * never accessible by userspace APIs, so this is safe to do.
45     */
46     spin_unlock(&base->lock);
47     update_process_times(user_mode(regs));
48     profile_tick(CPU_PROFILING);
49     spin_lock(&base->lock);
50 }
51
52 /* Do not restart, when we are in the idle loop */
53 if (ts->tick_stopped)
54     return HRTIMER_NORESTART;
55 hrtimer_forward(timer, now, tick_period);
56
57 return HRTIMER_RESTART;
58 }

```

第22行调用 `tick_do_update_jiffies64()` 来更新系统时间(见第9.4.1节)。每一次进入中断处理时，在 `do_IRQ()` 中调用 `set_irq_regs()` 设置了 `percpu` 变量 `_irq_regs`，从 `do_IRQ()` 返回时又调用 `set_irq_regs()` 还原了该变量，所以第28行不为 0，就说明当前处于中断上下文的运行环境中，所以就调用 `update_process_timers()` 和 `profile_tick()` 来统计进程时间和性能分析的信息。

最后，再来看看 `tick_handle_oneshot_broadcast()` 函数：

代码片段 9.55 节自 `kernel/time/tick-broadcast.c`

```

1 static void tick_handle_oneshot_broadcast(struct clock_event_device *dev)
2 {
3     struct tick_device *td;
4     cpumask_t mask;
5     ktime_t now, next_event;
6     int cpu;
7
8     spin_lock(&tick_broadcast_lock);
9 again:
10    dev->next_event.tv64 = KTIME_MAX;
11    next_event.tv64 = KTIME_MAX;
12    mask = CPU_MASK_NONE;
13    now = ktime_get();
14
15    /* 查看哪些 CPU 上有过时的时钟。*/
16    /* Find all expired events */
17    for (cpu = first_cpu(tick_broadcast_oneshot_mask);
18         cpu != NR_CPUS;

```

```
19         cpu = next_cpu(cpu, tick_broadcast_oneshot_mask)) {
20     td = &per_cpu(tick_cpu_device, cpu);
21     if (td->evtdev->next_event.tv64 <= now.tv64)
22         cpu_set(cpu, mask);
23     else if (td->evtdev->next_event.tv64 < next_event.tv64)
24         next_event.tv64 = td->evtdev->next_event.tv64;
25 }
26 /*
27 * Wakeup the cpus which have an expired event.
28 */
29 tick_do_broadcast(mask);
30 if (next_event.tv64 != KTIME_MAX) {
31     /*
32      * Rerarm the broadcast device. If event expired,
33      * repeat the above
34      */
35     if (tick_broadcast_set_event(next_event, 0))
36         goto again;
37 }
38 spin_unlock(&tick_broadcast_lock);
39 }
40
41 int tick_do_broadcast(cpumask_t mask)
42 {
43     int ret = 0, cpu = smp_processor_id();
44     struct tick_device *td;
45     /*
46      * Check, if the current cpu is in the mask
47      */
48     if (cpu_isset(cpu, mask)) {
49         cpu_clear(cpu, mask);
50         td = &per_cpu(tick_cpu_device, cpu);
51         td->evtdev->event_handler(td->evtdev);
52         ret = 1;
53     }
54
55     if (!cpus_empty(mask)) {
56         /*
57          * It might be necessary to actually check whether the devices
58          * have different broadcast functions. For now, just use the
59          * one of the first device. This works as long as we have this
60          * misfeature only on x86 (lapic)
```

```

61     */
62     cpu = first_cpu(mask);
63     td = &per_cpu(tick_cpu_device, cpu);
64     td->evtdev->broadcast(mask);
65     ret = 1;
66 }
67 return ret;
68 }

```

在 tickless 模式中，如果 CPU 进入深度节能状态，其 Local APIC 的时钟也停止了，所以全局的 broadcast 时钟中断需要接手 CPU 的本地时钟，如果某个 CPU 上有过期的时钟，第51行调用 CPU 的 clock event device 的 event_handler()函数，也就是 lapic 的 hrtimer_interrupt()。到此为止，我们看到，tick_handle_periodic()，hrtimer_interrupt()以及 tick_handle_oneshot_broadcast()这三个函数，殊途同归，最终在必要时都会完成时钟中断的基本工作。

9.4.1 时钟维护

内核中时钟维护包括以下几点。

1. 时钟中断次数 jiffies

jiffies 每一次时钟中断时，jiffies 会加 1，jiffies 乘以时钟中断周期，就得到一个相对时间。jiffies 是一个全局变量，对于多 CPU 的情况，只有启动 CPU 的时钟中断会使 jiffies 加 1。宏 HZ 定义了一秒内发出的时钟中断次数，例如 HZ 为 100，则时钟中断周期为 10ms。对于 Dynamic ticks 机制情况，由于系统启动之初，总是首先被初始化为非 Dynamic ticks 机制，这时总是有一个时钟中断周期，当启用 Dynamic ticks 机制后，在其时钟中断中会计算逝去的时间，并根据之前的时钟周期调整 jiffies。jiffies 和 jiffies_64 是同一个变量，其定义如下：

代码片段 9.56 节自 kernel/timer.c

```

1 u64 jiffies_64 __cacheline_aligned_in_smp = INITIAL_JIFFIES;
2
3 #define INITIAL_JIFFIES ((unsigned long)(unsigned int) (-300*HZ))

```

jiffies_64 是 64 位，在 32 位系统中不能直接处理，jiffies_64 被初始化为 -300*HZ，典型的 HZ 为 100，则中断周期为 10ms，则 5 分钟后 jiffies_64 会从 32 位中溢出，很快让它溢出，如果处理有错，能很快发现。另外在 vmlinux.lds 文件中定义了 jiffies_64 的别名：

代码片段 9.57 节自 arch/x86/kernel/vmlinux.lds

```

1 jiffies = jiffies_64;

```

注意链接脚本中等号和 C 语言中的等号所代表的意义是不同的，这里是告诉链接器，符号 jiffies 和 jiffies_64 的内存地址一致，就相当于变量的别名了。可以通过 vmlinux 的符号表来检验这一点：

代码片段 9.58 jiffies 与 jiffies_64 为同一个变量

```
1 $ nm vmlinux | grep jiffies
2
3 c0856c00 D jiffies
4 c0856c00 D jiffies_64
```

2. 当前系统时间 xtime

全局变量 xtime 记录着当前系统的时间，其初始值是通过 RTC 时钟获取的。start_kernel()函数会调用 timekeeping_init()来初始化 xtime，xtime 定义如下：

代码片段 9.59 节自 include/linux/time.h

```
1 struct timespec {
2     /* seconds */
3     time_t tv_sec;
4     /* nanoseconds */
5     long tv_nsec;
6 };
7
8 struct timespec xtime __attribute__ ((aligned (16)));
```

3. 系统相对时间 wall_to_monotonic

由于系统时间是可以修改的，为了记录系统从启动到“现在”的时间，于是定义了 wall_to_monotonic，其他也是 timespec 类型的全局变量。它也是在 timekeeping_init()中初始化的：

代码片段 9.60 节自 kernel/time/timekeeping.c

```
1 void __init timekeeping_init(void)
2 {
3     unsigned long flags;
4     unsigned long sec = read_persistent_clock();
5     .....
6     xtime.tv_sec = sec;
7     xtime.tv_nsec = 0;
8     set_normalized_timespec(&wall_to_monotonic,
9                           -xtime.tv_sec,
```

```

10                     -xtime.tv_nsec);
11     total_sleep_time = 0;
12     .....
13 }

```

第8行用 xtime 取反来初始化 wall_to_monotonic，将来可以利用 xtime + wall_to_monotonic 来计算系统启动的相对时间，启动之初，其和为 0。在系统时钟中断中，只需要更新 xtime，任何时候都可以通过它们的和来计算出相对时间，而时钟中断时不需要对 wall_to_monotonic 进行操作，从而达到了优化的目的。当用户调整时间日期时会修改 xtime，但是为了保证相对时间的准确性，这时也需要修改 wall_to_monotonic。

代码片段 9.61 节自 kernel/time/timekeeping.c

```

1 [sys_settimeofday() -> do_sys_settimeofday() -> do_settimeofday()]
2
3 int do_settimeofday(struct timespec *tv)
4 {
5     unsigned long flags;
6     time_t wtm_sec, sec = tv->tv_sec;
7     long wtm_nsec, nsec = tv->tv_nsec;
8     .....
9     nsec -= __get_nsec_offset();
10    wtm_sec = wall_to_monotonic.tv_sec + (xtime.tv_sec - sec);
11    wtm_nsec = wall_to_monotonic.tv_nsec + (xtime.tv_nsec - nsec);
12
13    set_normalized_timespec(&xtime, sec, nsec);
14    /* 相应的更正 wall_to_monotonic.*/
15    set_normalized_timespec(&wall_to_monotonic, wtm_sec, wtm_nsec);
16    .....
17    /* signal hrtimers about time change */
18    clock_was_set();
19
20    return 0;
21 }

```

时钟中断中会调用 do_timer() 来更新 jiffies 和 xtime:

代码片段 9.62 节自 kernel/timer.c

```

1 void do_timer(unsigned long ticks)
2 {
3     jiffies_64 += ticks;
4     update_times(ticks);
5 }

```

```
6
7 static inline void update_times(unsigned long ticks)
8 {
9     update_wall_time();
10    calc_load(ticks);
11 }
```

do_timer()的参数 ticks，表示 tick 的次数，首先根据 ticks 增加 jiffies_64，然后再更新 xtime，这个工作由 update_wall_time()来完成，其中 clock 指向当前使用的 clocksource 对象(见第9.1节)。

代码片段 9.63 节自 kernel/time/timekeeping.c

```
1 void update_wall_time(void)
2 {
3     cycle_t offset;
4
5     /* Make sure we're fully resumed: */
6     if (unlikely(timekeeping_suspended))
7         return;
8
9 #ifdef CONFIG_GENERIC_TIME
10    /* 从clocksource中读取当前时间。*/
11    offset = (clocksource_read(clock) - clock->cycle_last) &
12          clock->mask;
13 #else
14    offset = clock->cycle_interval;
15 #endif
16    clock->xtime_nsec += (s64)xtime.tv_nsec << clock->shift;
17    while (offset >= clock->cycle_interval) {
18        /* accumulate one interval */
19        clock->xtime_nsec += clock->xtime_interval;
20        clock->cycle_last += clock->cycle_interval;
21        offset -= clock->cycle_interval;
22
23        /* 增加xtime的秒数。*/
24        if (clock->xtime_nsec >= (u64)NSEC_PER_SEC << clock->shift) {
25            clock->xtime_nsec -= (u64)NSEC_PER_SEC << clock->shift;
26            xtime.tv_sec++;
27            second_overflow();
28        }
29        /* accumulate error between NTP and clock interval */
30        clock->error += current_tick_length();
31        clock->error -= clock->xtime_interval <<
```

```

32             (TICK_LENGTH_SHIFT - clock->shift);
33     }
34
35     /* correct the clock when NTP error is too big */
36     clocksource_adjust(offset);
37     /* store full nanoseconds into xtime */
38     xtime.tv_nsec = (s64)clock->xtime_nsec >> clock->shift;
39     clock->xtime_nsec -= (s64)xtime.tv_nsec << clock->shift;
40     update_xtime_cache(cyc2ns(clock, offset));
41     /* check to see if there is a new clocksource to use */
42     change_clocksource();
43     /* 当前定义为空。*/
44     update_vsyscall(&xtime, clock);
45 }

```

这个函数比较简单，注释得也比较清楚，这里有几点需要注意。第36行是用于网络时间协议，也就是 Network Time Protocol。在第9.1节，我们说过，可以利用/sys 目录下的 current_clocksource 文件来修改内核当前使用的 clocksource 对象，第42行处理对 clocksource 进行修改的情况。

Periodic ticks 模式和 Dynamic ticks 模式调用 do_timer()是有差别的，由于 Periodic ticks 模式下，中断周期是固定的，所以每次 jiffies 只需要增加 1，而 Dynamic ticks 则需要计算 jiffies 的增加数量，Periodic ticks 处理模式如下：

代码片段 9.64 节自 kernel/time/tick-common.c

```

1 [tick_handle_periodic() -> tick_periodic()]
2 static void tick_periodic(int cpu)
3 {
4     .....
5     do_timer(1);
6     .....
7 }

```

而 Dynamic ticks 的处理模式如下：

代码片段 9.65 节自 kernel/time/tick-sched.c

```

1 [hrtimer_interrupt() -> tick_sched_timer() -> tick_do_update_jiffies64()]
2
3 static void tick_do_update_jiffies64(ktime_t now)
4 {
5     unsigned long ticks = 0;
6     ktime_t delta;
7

```

```

8  /* Reevaluate with xtime_lock held */
9  write_seqlock(&xtime_lock);
10 delta = ktime_sub(now, last_jiffies_update);
11 if (delta.tv64 >= tick_period.tv64) {
12     delta = ktime_sub(delta, tick_period);
13     last_jiffies_update = ktime_add(last_jiffies_update, tick_period);
14     /* Slow path for long timeouts */
15     if (unlikely(delta.tv64 >= tick_period.tv64)) {
16         s64 incr = ktime_to_ns(tick_period);
17         ticks = ktime_divns(delta, incr);
18         last_jiffies_update = ktime_add_ns(last_jiffies_update, incr*ticks);
19     }
20     do_timer(++ticks);
21 }
22 write_sequnlock(&xtime_lock);
23 }

```

`last_jiffies_update` 记录着上一次更新 jiffies 的时间，如果当前时间和 `last_jiffies_update` 的差值 `delta` 大于初始的中断周期¹²，就需要计算更新 jiffies。而 jiffies 需要增加的次数就等于用时间差 `delta` 除以中断周期。但是由于内核中除法运算相对较慢，而且 Dynamic ticks 模式是为了高精度时钟而设置的，因此大多数情况下，jiffies 需要增加的次数不会超过 1。所以先用 `delta` 减去中断周期，如果差值还大于中断周期，就做除法运算，最终更新 ticks 并调用 `do_timer()` 函数。

9.4.2 进程时间信息统计

`update_process_times()` 函数会更新当前进程的运行时间，检查当前进程的时间片是否到期，请求时钟软中断等。

代码片段 9.66 节自 `kernel/timer.c`

```

1 void update_process_times(int user_tick)
2 {
3     struct task_struct *p = current;
4     int cpu = smp_processor_id();
5
6     /* Note: this timer irq context must be accounted for as well. */
7     /* 统计当前进程的运行时间。*/
8     account_process_tick(p, user_tick);
9     /* 请求时间软中断。*/
10    run_local_timers();

```

¹²前面说过，系统启动时，总是默认先进入 Periodic ticks 模式，因此总是有一个默认的中断周期。

```
11  if (rcu_pending(cpu))
12    rCU_check_callbacks(cpu, user_tick);
13  /* 减少当前进程的时间片，如果当前进程时间片已经用完，则请求延迟调度。*/
14  scheduler_tick();
15  run_posix_cpu_timers(p);
16 }
17
18 void scheduler_tick(void)
19 {
20  int cpu = smp_processor_id();
21  struct rq *rq = cpu_rq(cpu);
22  struct task_struct *curr = rq->curr;
23  u64 next_tick = rq->tick_timestamp + TICK_NSEC;
24
25  spin_lock(&rq->lock);
26  __update_rq_clock(rq);
27  if (unlikely(rq->clock < next_tick))
28    rq->clock = next_tick;
29  rq->tick_timestamp = rq->clock;
30  update_cpu_load(rq);
31  /* 根据不同的调度算法，来计算时间片。*/
32  if (curr != rq->idle) /* FIXME: needed? */
33    curr->sched_class->task_tick(rq, curr);
34  spin_unlock(&rq->lock);
35
36 #ifdef CONFIG_SMP
37  rq->idle_at_tick = idle_cpu(cpu);
38  trigger_load_balance(rq, cpu);
39#endif
40 }
```

最后时间中断还要调用 profile_tick() 处理性能分析的情况，我们不打算讨论性能分析的统计，这里说一说它的基本原理。当启用 profile 功能时，每一次时钟中断时，会根据中断上下文的 EIP 计算出对应的函数名，中断时 EIP 位于某个函数的次数越多，则说明这段代码中该函数越耗 CPU 资源。

9.5 软件定时器

这里讨论的定时器和 High-Resolution Timer 不同，这里的定时器主要面对内核模块中常用的 add_timer(), mod_timer() 等接口，它是在 jiffies 的基础上实现的，其回调函数在时钟软中断中执行，精度相对较低。

9.5.1 基本管理结构

每一个软件定时器由一个 timer_list 结构表示，其定义如下：

代码片段 9.67 节自 include/linux/timer.h

```
1 struct timer_list {
2     struct list_head entry;
3     /* 过期时间。*/
4     unsigned long expires;
5     /* 定时器回调函数。*/
6     void (*function)(unsigned long);
7     /* 回调函数的参数。*/
8     unsigned long data;
9     struct tvec_t_base_s *base;
10 #ifdef CONFIG_TIMER_STATS
11     void *start_site;
12     char start_comm[16];
13     int start_pid;
14 #endif
15 };
```

系统中有大量的软件定时器，时钟软中断处理函数需要根据当前的 jiffies，从软件定时器队列中找出过期的定时器，再调用它的回调函数，定时器回调函数可以再次调用 mod_timer() 设置下一次的到期时间。所以定时器的组织方式非常关键。这个队列由一个 percpu 变量 tvec_bases 来维护，相关定义如下：

代码片段 9.68 节自 kernel/timer.c

```
1 #define TVN_BITS (CONFIG_BASE_SMALL ? 4 : 6)
2 #define TVR_BITS (CONFIG_BASE_SMALL ? 6 : 8)
3 #define TVN_SIZE (1 << TVN_BITS)
4 #define TVR_SIZE (1 << TVR_BITS)
5 #define TVN_MASK (TVN_SIZE - 1)
6 #define TVR_MASK (TVR_SIZE - 1)
7
8 typedef struct tvec_s {
```

```

9   struct list_head vec[TVN_SIZE];
10 } tvec_t;
11
12 typedef struct tvec_root_s {
13   struct list_head vec[TVR_SIZE];
14 } tvec_root_t;
15
16 struct tvec_t_base_s {
17   spinlock_t lock;
18   struct timer_list *running_timer;
19   /* 整个队列中的软件时钟最早到期的 jiffies. */
20   unsigned long timer_jiffies;
21   /* list 数组。*/
22   tvec_root_t tv1;
23   tvec_t tv2;
24   tvec_t tv3;
25   tvec_t tv4;
26   tvec_t tv5;
27 } __cacheline_aligned;

```

在 `tvec_t_base_s` 结构中, `timer_jiffies` 表示整个定时器队列中最早到期的 jiffies, `tv1, tv2, ..., tv5`, 都是 `list_head` 结构, 我们默认 `CONFIG_BASE_SMALL` 为 0, 则数组 `tv1` 中有 2^8 个 `list_head` 成员, `tv2, ..., tv5` 分别有 2^6 个 `list_head` 成员。以 `timer_jiffies` 为起始, 到期时间在 $0 \sim 2^8 - 1$ 个 jiffies 之内的定时器连接到 `tv1` 的数组成员中, 到期时间在 $2^8 \sim 2^{14} - 1$ 个 jiffies 之内的定时器连接在 `tv2` 的数组中, ..., 依此类推, 直到 `tv5`, 总共正好是 2^{32} 个队列。这样做是为了加快搜索过期的定时器的速度, 其原理将在 9.5.3 节中继续讨论。

9.5.2 初始化

初始化工作主要是为 `percpu` 变量 `tvc_bases` 调用 `INIT_LIST_HEAD` 来设置结构中的 `tvn` 的链表。

代码片段 9.69 节自 `kernel/timer.c`

```

1 [start_kernel() -> init_timers()]
2
3 static struct notifier_block __cpuinitdata timers_nb = {
4   .notifier_call  = timer_cpu_notify,
5 };
6
7 void __init init_timers(void)
8 {

```

```
9  /* notifier 对象注册前，手动调用 timer_cpu_notify。*/
10 int err = timer_cpu_notify(&timers_nb,
11                           (unsigned long)CPU_UP_PREPARE,
12                           (void *)(long)smp_processor_id());
13 /* 建立/proc/timer_stats 文件，用于统计。*/
14 init_timer_stats();
15
16 BUG_ON(err == NOTIFY_BAD);
17 /* 注册一个 notifier 对象，其回调函数为 timer_cpu_notify()。*/
18 register_cpu_notifier(&timers_nb);
19 /* 注册时钟软中断函数。*/
20 open_softirq(TIMER_SOFTIRQ, run_timer_softirq, NULL);
21 }
```

timer_cpu_notify()主要是为cpu的hotplug而设置的，我们不考虑这种情况，主要来看看它的初始化。

代码片段 9.70 节自 kernel/timer.c

```
1 tvec_base_t boot_tvec_bases;
2
3 static int __cpuinit timer_cpu_notify(struct notifier_block *self,
4                                       unsigned long action,
5                                       void *hcpu)
6 {
7     long cpu = (long)hcpu;
8     switch(action) {
9         case CPU_UP_PREPARE:
10        case CPU_UP_PREPARE_FROZEN:
11            if (init_timers_cpu(cpu) < 0)
12                return NOTIFY_BAD;
13            break;
14 #ifdef CONFIG_HOTPLUG_CPU
15        case CPU_DEAD:
16        case CPU_DEAD_FROZEN:
17            migrate_timers(cpu);
18            break;
19 #endif
20        default:
21            break;
22    }
23    return NOTIFY_OK;
24 }
```

```
26 static int __cpuinit init_timers_cpu(int cpu)
27 {
28     int j;
29     tvec_base_t *base;
30     static char __cpuinitdata tvec_base_done[NR_CPUS];
31
32     if (!tvec_base_done[cpu]) {
33         static char boot_done;
34         if (boot_done) {
35             base = kmalloc_node(sizeof(*base),
36                                 GFP_KERNEL | __GFP_ZERO,
37                                 cpu_to_node(cpu));
38
39             if (!base)
40                 return -ENOMEM;
41             /* Make sure that tvec_base is 2 byte aligned */
42             if (tbbase_get_deferrable(base)) {
43                 WARN_ON(1);
44                 kfree(base);
45                 return -ENOMEM;
46             }
47             per_cpu(tvec_bases, cpu) = base;
48         } else {
49             boot_done = 1;
50             base = &boot_tvec_bases;
51         }
52         tvec_base_done[cpu] = 1;
53     } else {
54         base = per_cpu(tvec_bases, cpu);
55     }
56
57     spin_lock_init(&base->lock);
58     lockdep_set_class(&base->lock, base_lock_keys + cpu);
59
60     for (j = 0; j < TVN_SIZE; j++) {
61         INIT_LIST_HEAD(base->tv5.vec + j);
62         INIT_LIST_HEAD(base->tv4.vec + j);
63         INIT_LIST_HEAD(base->tv3.vec + j);
64         INIT_LIST_HEAD(base->tv2.vec + j);
65     }
66     for (j = 0; j < TVR_SIZE; j++)
67         INIT_LIST_HEAD(base->tvl.vec + j);
```

```
68     base->timer_jiffies = jiffies;
69
70     return 0;
71 }
```

在第1行定义了一个 `tvec_base_t` 结构 `boot_tvec_bases`, 对于多 CPU 的系统, 每一个 CPU 都有一个这样的结构, 但是在编译时, 无法知道内核要运行的平台上有多少 CPU, 因此当第一个 CPU 初始化时就使用该结构, 而以后的 CPU 需要 `kmalloc_node` 来分配内存。由于 `init_timers()` 函数是在内存初始化前调用的, 但是 `init_timers()` 是在内存初始化完成之前调用的, 因此第一个 CPU 此时不能进行内存分配。最后, 每个 CPU 会初始化自己的 `tvec_base_t` 结构的 `tvn` 链表, 已经 lock, 其 `timer_jiffies` 初始化为当前 `jiffies`。

9.5.3 注册与过期处理

软件定时器注册首先要初始化一个 `timer_list` 对象, 然后调用 `add_timer()` 注册这个 `timer_list` 对象, 当定时器到期时, 会在时钟软中断环境中调用定时器回调函数, 回调函数一般会再次调用 `mod_timer()` 设置下一次到期的时间。

代码片段 9.71 节自 `kernel/timer.c`

```
1 /* timer_list 对象的回调函数由程序设置。*/
2 void fastcall init_timer(struct timer_list *timer)
3 {
4     timer->entry.next = NULL;
5     timer->base = __raw_get_cpu_var(tvec_bases);
6 #ifdef CONFIG_TIMER_STATS
7     timer->start_site = NULL;
8     timer->start_pid = -1;
9     memset(timer->start_comm, 0, TASK_COMM_LEN);
10 #endif
11 }
12
13 static inline void add_timer(struct timer_list *timer)
14 {
15     BUG_ON(timer_pending(timer));
16     __mod_timer(timer, timer->expires);
17 }
18
19 int mod_timer(struct timer_list *timer, unsigned long expires)
20 {
21     BUG_ON(!timer->function);
```

```
23     timer_stats_timer_set_start_info(timer);
24
25     if (timer->expires == expires && timer_pending(timer))
26         return 1;
27
28     return __mod_timer(timer, expires);
29 }
30
31 int __mod_timer(struct timer_list *timer, unsigned long expires)
32 {
33     tvec_base_t *base, *new_base;
34     unsigned long flags;
35     int ret = 0;
36
37     timer_stats_timer_set_start_info(timer);
38     BUG_ON(!timer->function);
39
40     base = lock_timer_base(timer, &flags);
41     if (timer_pending(timer)) {
42         detach_timer(timer, 0);
43         ret = 1;
44     }
45
46     new_base = __get_cpu_var(tvec_bases);
47     if (base != new_base) {
48         /*
49          * We are trying to schedule the timer on the local CPU.
50          * However we can't change timer's base while it is running,
51          * otherwise del_timer_sync() can't detect that the timer's
52          * handler yet has not finished. This also guarantees that
53          * the timer is serialized wrt itself.
54         */
55         if (likely(base->running_timer != timer)) {
56             /* See the comment in lock_timer_base() */
57             timer_set_base(timer, NULL);
58             spin_unlock(&base->lock);
59             base = new_base;
60             spin_lock(&base->lock);
61             timer_set_base(timer, base);
62         }
63     }
64     timer->expires = expires;
```

```

65     internal_add_timer(base, timer);
66     spin_unlock_irqrestore(&base->lock, flags);
67
68     return ret;
69 }

```

如果注册的 timer 之前注册在另外一个 CPU 的 tvec_bases 上，现在需要把它移到到本地 CPU 的 tvec_bases 上，但是如果其回调函数正在另外一个 CPU 上运行，那么就不能移动。这个函数注释比较清楚，我们来看看 internal_add_timer():

代码片段 9.72 节自 kernel/timer.c

```

1 static void internal_add_timer(tvec_base_t *base,
2                               struct timer_list *timer)
3 {
4     unsigned long expires = timer->expires;
5     /* 计算添加的 timer 的过期时间和队列中最早到期的 timer 的时间差。 */
6     unsigned long idx = expires - base->timer_jiffies;
7     struct list_head *vec;
8
9     /* 0 <= idx < TVR_SIZE13 则添加到 tv1 的对应数组中。 */
10    if (idx < TVR_SIZE) {
11        int i = expires & TVR_MASK;
12        vec = base->tv1.vec + i;
13    } else if (idx < 1 << (TVR_BITS + TVN_BITS)) {
14        int i = (expires >> TVR_BITS) & TVN_MASK;
15        vec = base->tv2.vec + i;
16    } else if (idx < 1 << (TVR_BITS + 2 * TVN_BITS)) {
17        int i = (expires >> (TVR_BITS + TVN_BITS)) & TVN_MASK;
18        vec = base->tv3.vec + i;
19    } else if (idx < 1 << (TVR_BITS + 3 * TVN_BITS)) {
20        int i = (expires >> (TVR_BITS + 2 * TVN_BITS)) & TVN_MASK;
21        vec = base->tv4.vec + i;
22    } else if ((signed long) idx < 0) {
23        vec = base->tv1.vec + (base->timer_jiffies & TVR_MASK);
24    } else {
25        int i;
26        /* If the timeout is larger than 0xffffffff on 64-bit
27         * architectures then we use the maximum timeout:
28         */
29        if (idx > 0xffffffffUL) {
30            idx = 0xffffffffUL;

```

¹³TVR_SIZE 默认为 2⁸

```

31     expires = idx + base->timer_jiffies;
32 }
33 i = (expires >> (TVR_BITS + 3 * TVN_BITS)) & TVN_MASK;
34 vec = base->tv5.vec + i;
35 }
36 /*
37  * Timers are FIFO:
38 */
39 list_add_tail(&timer->entry, vec);
40 }

```

软件定时器的回调函数由时钟软中断函数 run_timer_softirq() 处理。

代码片段 9.73 节自 kernel/timer.c

```

1 static void run_timer_softirq(struct softirq_action *h)
2 {
3     tvec_base_t *base = __get_cpu_var(tvec_bases);
4
5     hrtimer_run_queues();
6
7     /* timer_jiffies 是最早到期的时间。*/
8     if (time_after_eq(jiffies, base->timer_jiffies))
9         __run_timers(base);
10 }
11 static inline void __run_timers(tvec_base_t *base)
12 {
13     struct timer_list *timer;
14
15     spin_lock_irq(&base->lock);
16     while (time_after_eq(jiffies, base->timer_jiffies)) {
17         struct list_head work_list;
18         struct list_head *head = &work_list;
19         int index = base->timer_jiffies & TVR_MASK;
20
21         if (!index && (!cascade(base, &base->tv2, INDEX(0))) &&
22             (!cascade(base, &base->tv3, INDEX(1))) &&
23             !cascade(base, &base->tv4, INDEX(2)))
24             cascade(base, &base->tv5, INDEX(3));
25
26         ++base->timer_jiffies;
27
28         list_replace_init(base->tvl.vec + index, &work_list);

```

```
30     while (!list_empty(head)) {
31         void (*fn)(unsigned long);
32         unsigned long data;
33
34         timer = list_first_entry(head, struct timer_list, entry);
35         fn = timer->function;
36         data = timer->data;
37
38         timer_stats_account_timer(timer);
39
40         set_running_timer(base, timer);
41         detach_timer(timer, 1);
42         spin_unlock_irq(&base->lock);
43
44         int preempt_count = preempt_count();
45         fn(data);
46         if (preempt_count != preempt_count()) {
47             .....
48             BUG();
49         }
50     }
51     spin_lock_irq(&base->lock);
52 }
53 }
54 set_running_timer(base, NULL);
55 spin_unlock_irq(&base->lock);
56 }
```

软件时钟设计得比较巧妙，为了更加容易理解，我们举个通用的例子来说明它的工作原理。timer_jiffies 在初始化时，被设置为 jiffies，当下一次时钟中断发生时，jiffies 增加 1，此时在时钟软中断中，jiffies > timer_jiffies，说明可能有时钟过期了，于是调用 __run_timers() 函数，在第27行中，每次执行 timer_jiffies++，直到 timer_jiffies 比 jiffies 大 1 时，while 循环结束。在下一个时钟中断发生时，jiffies 再次增加 1，又赶上 timer_jiffies，于是又会进入该循环一次。因此多数情况下，进入这个循环都只需要处理一条链表上的时钟就可以了。但是也有例外的情况，例如在执行到第15行获取到自旋锁 base_lock 之前，如果产生了一个时钟中断，那么 jiffies 又会增加，这样就有多个时钟链表需要处理¹⁴。但是无论如何，时钟软中断返回时，timer_jiffies 都比当前的 jiffies 大 1。

¹⁴另外，在 tickless 机制中，当前调度到 idle 进程，并且系统最早到期的定时器超过一个 jiffies 时，就会计算最早到期时间，并使用这个时间来设置时钟中断时间及 timer_jiffies，此时 timer_jiffies 和 jiffies 的差不为 1。但是在这种情况下，当执行时钟软中断时，仍然只有一条链表需要处理。

现在假设当前队列为空，当前 jiffies % (256) = 250，于是 timer_jiffies % (256) = 251，现在要注册 2 个定时器对象，其中一个在 3 个 jiffies 之后过期，而另外一个，在 300 个 jiffies 过期之后，根据 internal_add_timer() 函数片断：

代码片段 9.74 节自 kernel/timer.c

```

1 static void internal_add_timer(tvec_base_t *base,
2                                struct timer_list *timer)
3 {
4     unsigned long expires = timer->expires;
5     unsigned long idx = expires - base->timer_jiffies;
6     struct list_head *vec;
7
8     if (idx < TVR_SIZE) {
9         int i = expires & TVR_MASK;
10        vec = base->tv1.vec + i;
11    } else if (idx < 1 << (TVR_BITS + TVN_BITS)) {
12        int i = (expires >> TVR_BITS) & TVN_MASK;
13        vec = base->tv2.vec + i;
14    }
15    .....
16    list_add_tail(&timer->entry, vec);
17 }
```

对于第一个定时器对象，其 expires - base->timer_jiffies = 2，小于 256，因此要添加到 tv1 中，且 expires & 256 = 253，因此第一个定时器要添加到 tv1.vec[253] 中¹⁵。而第二个定时器则要添加到 tv2 中，且(expires >> TVR_BITS) & TVN_MASK = 44，所以第二个定时器要添加到 tv2.vec[44] 中。

现在我们来看看定时器的处理，由于 timer_jiffies 比 jiffies 大 1，因此每次时钟执行时钟软件中断时，time_after_eq(jiffies, timer_jiffies) 条件都会满足，但是进入 __run_timers() 仅仅是执行 timer_jiffies++，就退出了。等到第三次，jiffies%256=253 时，处理过程分析如下：

代码片段 9.75 节自 kernel/timer.c

```

1 static inline void __run_timers(tvec_base_t *base)
2 {
3     .....
4     /* 此时 index=253. */
5     int index = base->timer_jiffies & TVR_MASK;
6     if (!index && (!cascade(base, &base->tv2, INDEX(0))) &&
7         (!cascade(base, &base->tv3, INDEX(1))) &&
8         !cascade(base, &base->tv4, INDEX(2)))
```

¹⁵可以看到，虽然 expires 比 timer_jiffies 大 2，但是该定时器却不是添加到 tv1.vec[2] 中。

```

9     cascade(base, &base->tv5, INDEX(3));
10
11    ++base->timer_jiffies;
12    /*
13     * 把 tv1. vec[ 253] 的链表头复制到 work_list 链表中,
14     * 同时初始化 tv1. vec[ 253] 为一个空链表。
15     */
16    list_replace_init(base->tv1. vec + index, &work_list);
17
18    /* 循环调用 work_list 链表上的定时器的回调函数。*/
19    struct list_head *head = &work_list;
20    while (!list_empty(head)) {
21        void (*fn)(unsigned long);
22        unsigned long data;
23
24        timer = list_first_entry(head, struct timer_list, entry);
25        fn = timer->function;
26        data = timer->data;
27
28        fn(data);
29    }
30 }

```

之后，随着时间的推移，当 `timer_jiffies & TVR_MASK = 0` 时，会调用 `cascade(base, &base->tv2, INDEX(0))` 函数，它的作用是把 `tv2` 中的某条定时器链表移动到 `tv1` 中：

代码片段 9.76 节自 kernel/timer.c

```

1 /* 此时是取 timer_jiffies 的第 8-13 位，而忽略其他的位。*/
2 #define INDEX(N) \
3     ((base->timer_jiffies >> (TVR_BITS + (N) * TVN_BITS)) & TVN_MASK)
4
5 static int cascade(tvec_base_t *base, tvec_t *tv, int index)
6 {
7     /* cascade all the timers from tv up one level */
8     struct timer_list *timer, *tmp;
9     struct list_head tv_list;
10    /*
11     * 把 tv2. vec[ index] 的链表头复制到 tv_list 链表中,
12     * 同时初始化 tv1. vec[ 253] 为一个空链表。
13     */
14    list_replace_init(tv->vec + index, &tv_list);
15    /*
16     * 对该链表上的每一个定时器对象，重新调用 internal_add_timer(),

```

```
17     * 这样就把某些定时器就移动到 tv1 链表上。
18     */
19     list_for_each_entry_safe(timer, tmp, &tv_list, entry) {
20         BUG_ON(tbase_get_base(timer->base) != base);
21         internal_add_timer(base, timer);
22     }
23
24     return index;
25 }
```

细心的读者一定会想，在第19行为什么不是对链表头那个定时器调用 `internal_add_timer()` 函数呢？这样不就把整条链表上的定时器添加到了 `tv1` 中吗？这就说明这种情况下，该链表上的定时器可能不是移动到 `tv1` 中的同一条链表中。现在我们假设 `timer_jiffies=X`，其中一个 `timer` 的 `expires=X+257`，另外一个 `timer` 的 `expires=X=258`，由于 `expires` 的第 8~13 位相等，因此添加在 `tv2` 的同一条链表上。随着时间的流逝，当需要移动 `tv2` 上的这条链表时，由于这两个定时器的 `expires` 的 0~7 位不同，因此应该移动到 `tv1` 中不同的链表上。

第10章 进程管理

进程是一个可执行文件的执行体，除了包括指令之外，它还包含了大量的资源，它拥有独立的虚拟地址空间，Linux 中使用 `task_struct` 结构来描述一个进程。在第1章，我们看到虚拟地址空间把不同进程隔离开，极大地提高了系统的健壮性。但是在某些情况下，它也带来了新的问题。假设现在有一个 GUI 进程，当用户按下某个菜单项时，需要执行大量的运算，在运算结束前，用户按下其他菜单项(比如取消)，是不会得到响应的¹。于是，我们希望把大量运算的代码放到子进程中执行，只要保证 GUI 进程的优先级大于子进程，那么就能保证用户的及时响应²。但是此时这个子进程需要复制共享父进程的所有资源，在进程切换过程时，需要切换地址空间。

为了减小系统开销，提出了线程的概念，在 Linux 中又被称为轻权进程，它们共享同一个地址空间，可以根据情况共享进程的某些或者全部资源。引入线程的概念后，线程称为 CPU 调度的基本单位，而进程则是线程对象和各种资源对象的容器。但是 Linux 采用了一种“偷懒”的方法，Linux 没有专门的线程对象，当需要建立一个线程时，实际上内核创建的是一个进程对象，也就是 `task_struct`，只不过这个进程对象和父进程共享了大量的资源，所以被称为轻权进程³(Lightweight Processes)。Linux 建立进程和线程的接口也一致，比如 `fork()`，而通过不同的参数来指定要建立的是进程还是线程。调用 `fork()` 函数将返回两次，一次是在父进程中，另外一次是在子进程中，一定会让大多数人感到迷惑。其实 `fork()` 就是把当前的进程对象 `task_struct` 复制一份，这样进程队列中就多了一个进程对象，由于两个进程相同，所以调度器调度到父进程时，返回一次，调度到子进程时，返回一次。

¹其大致过程是这样的：假设这个需要进行大量运算的函数为 `fn()`，当 CPU 执行这个函数时，用户通过鼠标按下另外的菜单，此时发出鼠标中断，CPU 中断处理结束时，返回到 `fn()` 中继续执行(假设当前进程优先级最高)。而菜单项的响应则要等待 `fn()` 返回。

²其大致过程是这样的：假设这个需要大量运算的函数为 `fn()`，现在把 `fn()` 安排在一个子进程中，CPU 执行子进程的 `fn()` 函数，用户通过鼠标按下另外的菜单，此时发出鼠标中断，CPU 中断处理结束时，由于 GUI 进程的优先级高于这个子进程，因此中断返回时会调度 GUI 进程，从而保证了及时响应性。

³以后我们将使用线程这个术语。

10.1 进程描述符

Linux 内核中，每一个进程由一个 `task_struct` 结构描述，该结构包含了进程的大量信息，又被称为进程控制块(PCB)。该结构比较大，这里列出主要结构如下：

代码片段 10.1 节自 `include/linux/sched.h`

```
1 struct task_struct {
2     /* 进程状态。*/
3     volatile long state;
4     /* 进程退出状态。*/
5     int exit_state;
6     int exit_code;
7     int exit_signal;
8
9     /* 进程 ID。*/
10    pid_t pid;
11    pid_t tgid;
12
13    /* 进程的可执行文件名，不包含路径，最大长度为 15 字节。*/
14    char comm[TASK_COMM_LEN];
15
16    /* 进程的亲缘关系。*/
17    /* real parent process (when being debugged) */
18    struct task_struct *real_parent;
19    /* parent process */
20    struct task_struct *parent;
21
22    /* list of my children */
23    struct list_head children;
24    /* linkage in my parent's children list */
25    struct list_head sibling;
26    /* threadgroup leader */
27    struct task_struct *group_leader;
28
29    /* 内核态堆栈。*/
30    void *stack;
31
32    /* 进程的内存布局信息。*/
33    struct mm_struct *mm, *active_mm;
34
35    /* filesystem information */
36    struct fs_struct *fs;
```

```
37  /* 打开的文件描述符。*/
38  /* open file information */
39  struct files_struct *files;
40
41  /* 时间统计信息。*/
42  unsigned int rt_priority;
43  cputime_t utime, stime, utimescaled, stimescaled;
44  cputime_t gtime;
45  cputime_t prev_utime, prev_stime;
46  unsigned long nvcsw, nivcsw;
47  /* monotonic time */
48  struct timespec start_time;
49  /* boot based time */
50  struct timespec real_start_time;
51
52  /* 用户及认证信息。*/
53  uid_t uid, euid, suid, fsuid;
54  gid_t gid, egid, sgid, fsgid;
55  struct group_info *group_info;
56  kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
57  unsigned keep_capabilities:1;
58  struct user_struct *user;
59
60  /* 进程的信号处理相关信息。*/
61  struct signal_struct *signal;
62  struct sighand_struct *sighand;
63
64  sigset_t blocked, real_blocked;
65  /* To be restored with TIF_RESTORE_SIGMASK */
66  sigset_t saved_sigmask;
67  struct sigpending pending;
68 }
```

10.1.1 进程状态

进程状态 state 是进程调度的依据。内核定义了几个宏表示不同的状态，主要有以下几个。

- **TASK_RUNNING**: 可运行状态，处于可运行态的进程要么正在某个 CPU 上运行，要么位于就绪队列中，等待调度运行。
- **TASK_INTERRUPTIBLE**: 可中断等待状态，进程由于所需资源得不到满足，从而进

入等待队列，但是该状态能够被信号中断。当一个正在运行的进程因进行磁盘 IO 操作而进入可中断等待状态时，在 IO 操作完成之前，用户可以向该进程发送 SIGKILL，从而使该进程提前结束等待状态，进入可运行态，以便响应 SIG_KILL，执行进程退出代码，从而结束该进程。

- **TASK_UNINTERRUPTIBLE**: 不可中断等待状态，进程由于所需资源得不到满足，而进入等待队列，但是不能被信号中断。当一个正在运行的进程因进行磁盘 IO 操作而进入可中断等待状态时，在 IO 操作完成之前，该进程不会因 SIG_KILL 等信号而结束等待状态，只有当磁盘操作完成后，把该进程唤醒进入就绪队列。由于该进程在等待队列中是不能被信号打断的，所以称为不可中断的等待状态。
- **TASK_STOPPED**: 暂停状态，处于该状态的进程同样是不可调度的，它和等待状态的区别在于，处于等待状态的进程一般是由于资源得不到满足而放弃 CPU，而暂停状态是由于其他进程引起的，比如一个进程没有执行任何 IO 操作或者 sleep 等操作，那么可以通过发送 SIGSTOP 等信号把一个进程设置为暂停状态，之后可以通过 SIGCONT 信号取消暂停状态。
- **TASK_TRACED**: 进程被调试器设置为暂停状态，例如 ptrace() 系统调用。
- **TASK_DEAD**: 进程已经退出，等待调度器删除其 task_struct 结构。当进程调用退出时(例如调用 exit 或者从 main 函数返回)，可能需要向父进程发送信号，当父进程进行信号处理时，需要获取子进程的信息，因此这时不能删除子进程的 task_struct，另外每一个进程都有一个内核态堆栈，当前进程调用 exit() 时，在切换到另外一个进程前，总是要使用内核态堆栈，因此当进程调用 exit() 时，完成必要的处理之后，就把 state 设置为 TASK_DEAD，并切换到其他进程。当顺利地切换到其他进程后，由于该进程状态设置为 TASK_DEAD，因此这个进程不会被调度，之后当调度器检查到状态为 TASK_DEAD 的进程时，就会删除这个进程的 task_struct 结构，这样这个进程彻底“消失”了。

`exit_code` 是退出码，而 `exit_state` 退出状态，它们都是为在进程退出设置的。在 Linux 内核中，父进程建立一个子进程后，可以通过 `waitpid()` 等系列函数等待子进程结束。当子进程结束时，向父进程发送 SIGCHLD 信号。所以设置了 `exit_state`，它有如下两种状态。

- **EXIT_ZOMBIE**: 父进程等待子进程结束时发送的 SIGCHLD 信号，此时子进程已退出，并且 SIGCHLD 信号已经发送，但是父进程还没有被调度运行，该进程退出时 `exit_state` 被设置为 EXIT_ZOMBIE 状态。
- **EXIT_DEAD**: 父进程对子进程的退出信号“没兴趣”，进程退出时，没有任何进程通过 `waitpid()` 调用等待该进程的的 SIGCHLD 信号，该进程退出时 `exit_state` 被设置为 EXIT_DEAD 状态。

10.1.2 进程标识

每一个进程都有一个唯一的标识，被称为 PID，系统启动时“手工”建立的 swapper 进程的 pid 为 0，以后建立新进程时，使用前一个进程的 pid+1，作为新进程的 pid。pid 是 32 位的数字，但是为了兼容 16 位的代码，默认最大的 pid 为 32768，用户可以通过 /proc/sys/kernel/pid_max 文件来设置最大的 pid。随着时间的推移，pid 很快会达到最大值，此时需要通过 pidmap 位图来判断哪些 pid 是可用的。

在某些多线程的操作系统中（例如 Windows），一个进程拥有一个或多个线程，进程有进程 ID，这些线程又有不同的线程 ID。POSIX 标准规定对于同一个进程的多个线程，有一个进程 ID，多个线程 ID，这样便可以根据 PID 向整个进程发送信号。但是在 Linux 中，每一个进程都对应一个 task_struct 结构，当建立线程时，其实也是建立一个 task_struct 结构，所以 task_struct 具有双重身份，既可以作为进程对象，也可以作为线程对象。这样即使是同一个进程的不同线程，它们的 pid 都不同，为此又定义了 tgid。其实 tgid 才是真正意义上的进程 ID。当创建一个进程时，该进程的 pid 和 tgid 一致，此后该进程创建一个轻权进程时，新进程有一个新的 task_struct，新的 pid，但是其 tgid 和原来的进程一致。这两个进程被称为一个线程组（Thread Group.），而第一个进程就是 thread group leader。所以我们认为 tgid 是真正意义上的 PID，而 pid 只是 TID（Thread ID.）。

内核在运行过程中，根据一个 ID 获取进程的 task_struct 结构是一个很频繁的操作，然而系统中的进程数非常多，尤其是在服务器上，为了提高查找效率，内核使用了 hash 表，其中 pid_hash 为 hash 表的首地址，在 start_kernel() 函数中，会调用 pidhash_init() 为 hash 表分配内存。

代码片段 10.2 节自 kernel/pid.c

```
1  /*
2   * hash 函数，nr 为 pid，ns 为进程 id 的 namespace。进程 id 的 namespace 是
3   * 为虚拟化设置的，例如 OpenVZ，它允许同时在一台电脑上虚拟运行多个
4   * 操作系统，这样在不同的 namespace 中的不同进程，可以拥有相同的 pid。
5   */
6  #define pid_hashfn(nr, ns) hash_long((unsigned long)nr + \
7      (unsigned long)ns, pidhash_shift)
8  /* hash 表的首地址。*/
9  static struct hlist_head *pid_hash;
10
11 void __init pidhash_init(void)
12 {
13     int i, pidhash_size;
14     unsigned long megabytes = nr_kernel_pages >> (20 - PAGE_SHIFT);
15
16     pidhash_shift = max(4, fls(megabytes * 4));
17     pidhash_shift = min(12, pidhash_shift);
```

```

18     pidhash_size = 1 << pidhash_shift;
19
20     printk("PID hash table entries: %d (order: %d, %Zd bytes)\n",
21             pidhash_size, pidhash_shift,
22             pidhash_size * sizeof(struct hlist_head));
23
24     /* 为 pid hash 表分配内存。*/
25     pid_hash = alloc_bootmem(pidhash_size * sizeof(*(pid_hash)));
26     if (!pid_hash)
27         panic("Could not alloc pidhash!\n");
28
29     /* 初始化链表。*/
30     for (i = 0; i < pidhash_size; i++)
31         INIT_HLIST_HEAD(&pid_hash[i]);
32 }

```

对于两个不同的 pid，调用 pid_hashfn()得到的 hash 表的索引可能一样，这被称为 hash 冲突。对于冲突的 pid 的进程，使用一个双向链表连接，组织方式如图10.1所示。

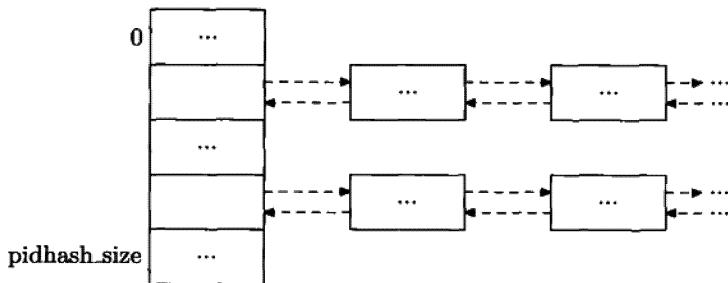


图 10.1 pid hash table

10.1.3 进程的亲缘关系

进程的亲缘关系有父子关系和兄弟关系，所有的进程组织成一棵树，init 进程是所有进程的祖先。task_struct 结构中 children 是子进程链表， sibling 是兄弟进程链表， real_parent 和 parent 都指向父进程。在 Linux 内核中，当父进程建立一个子进程后，可以通过 waitpid()等系列的函数等待子进程结束，这样父进程进入等待状态，当子进程结束时根据 real_parent 向父进程发送 SIGCHLD 消息。但是为什么需要 real_parent 和 parent 两个父进程指针呢？当一个进程处于被调试（跟踪）状态时，需要向调试器发送信号，而调试器可以动态的附加或者释放一个进程，因此当一个进程被调试时，其 parent 临时地指向调试器进程，而 real_parent 指向该进程的真正的父进程。当调试器退出时，其 parent 又和 real_parent 一样

指向父进程。

当父进程退出时，该如何设置这些子进程 `parent` 指针呢？如果父进程位于一个 `Thread Group` 中，则该 `Thread Group` 中的另外一个进程“收养”这些“孤儿”进程。否则 `init` 进程会“收养”所有的“孤儿”进程。

10.1.4 进程的内核态堆栈

进程的内核态堆栈默认为 2 个页面 8KB，由于在运行中需要频繁地获取当前进程的 `task_struct` 指针，因此内核定义了一个 `current`，在任何时候，`current` 都指向当前进程的 `task_struct`。早期的内核把 `task_struct` 放在堆栈的顶部，这样在内核态，`current` 这个宏就是 `ESP` 的最低 13 位清零，来得到当前进程的 `task_struct` 指针。但是随着 `task_struct` 结构的不断增大，把这个结构嵌入 8KB 的堆栈顶部已经不合适，于是分离出一个很小的 `thread_info` 结构嵌入到堆栈中。如图 10.2 所示，`thread_info` 结构中的 `task` 指针指向进程的 `task_struct` 结构，而进程的 `task_struct` 结构的 `stack` 成员则指向内核态堆栈。注意堆栈是由高地址向低地址方向增长的，而 `thread_info` 位于低地址，也就是堆栈的顶端。通过把 `ESP` 低 13 位清零的方式获取的是当前进程的 `thread_info` 结构，而该结构的 `task` 则指向当前进程的 `task_struct`。但是现在内核定义了 `percpu` 变量 `current_task`，每次调度一个新进程时，就会设置 `current_task` 变量指向新进程的 `task_struct` 结构。

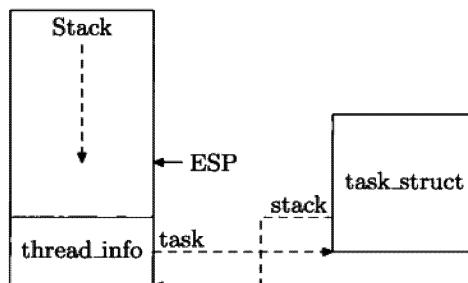


图 10.2 进程内核态堆栈

因此 `current` 宏定义如下：

代码片段 10.3 节自 `include/asm-x86/current_32.h`

```

1 DECLARE_PER_CPU(struct task_struct *, current_task);
2 static __always_inline struct task_struct *get_current(void)
3 {
4     return x86_read_percpu(current_task);
5 }
6
7 #define current get_current()

```

10.1.5 进程的虚拟内存布局

每一个用户态进程拥有独立的 3GB 用户态虚拟地址空间，共享 1GB 的内核态空间，而这 3GB 的地址不可能都映射了物理地址，而 task_struct 的 mm 成员就是用来描述这 3GB 的虚拟地址信息的。对于内核态进程，由于没有 3GB 的用户态虚拟地址空间，所以其 mm 结构为 NULL。可是为什么要在 task_struct 中设置 mm 和 active_mm 两个 mm_struct 成员呢？这是由于内核线程没有用户态地址空间，所以它的 mm 设置为 NULL，但是由于页目录的地址是保存在 mm 结构中的，从其他进程切换到这个内核态进程时，调度器可能需要切换页表，为此增加了一个 active_mm，对于 mm 为 NULL 的内核态线程，就借用其他进程的 mm_struct，也就是说把它的 active_mm 指向其他进程的 mm 结构。当进行进程切换时，统一使用 active_mm 就可以了。但是其他进程不是有自己独立的页表吗？由于内核态线程只使用内核地址空间，因此这不会有问题。mm_struct 定义如下：

代码片段 10.4 节自 include/linux/mm_types.h

```

1 struct mm_struct {
2     /* list of VMAs */
3     struct vm_area_struct * mmap;
4     struct rb_root mm_rb;
5     struct vm_area_struct * mmap_cache;
6     .....
7     /* base of mmap area */
8     unsigned long mmap_base;
9     /* size of task vm space */
10    unsigned long task_size;
11    .....
12    /* 进程页目录。 */
13    pgd_t * pgd;
14    /* How many users with user space? */
15    atomic_t mm_users;
16    /* How many references to "struct mm_struct" (users count as 1) */
17    atomic_t mm_count;
18    /* number of VMAs */
19    int map_count;
20    .....
21    /* 进程的执行文件镜像。 */
22    unsigned long total_vm, locked_vm, shared_vm, exec_vm;
23    unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
24    unsigned long start_code, end_code, start_data, end_data;
25    unsigned long start_brk, brk, start_stack;
26    unsigned long arg_start, arg_end, env_start, env_end;
27 };

```

用户态进程都有一个可执行文件，`start_code`, `end_code` 分别指向该进程代码段起始地址和结束地址，`start_data`, `end_data` 指向进程数据段的起始地址和结束地址，`start_stack` 指向进程的堆栈起始地址，`start_brk` 指向进程的堆（heap）的起始地址，`arg_start`, `arg_end` 指向进程的命令行参数，`env_start`, `env_end` 指向该进程的环境变量。整个进程的虚拟地址空间布局如图10.3所示。

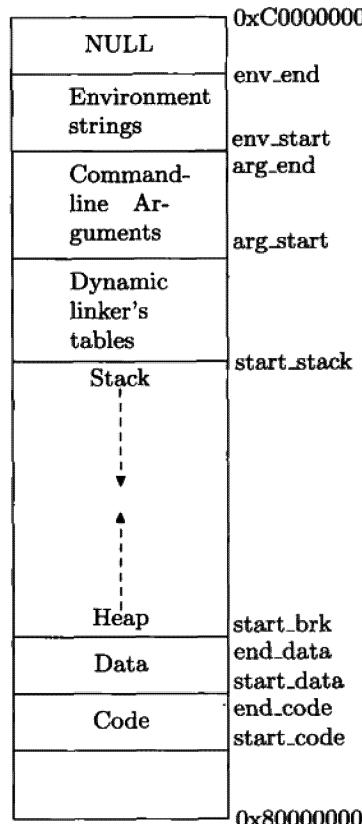


图 10.3 进程用户态虚拟地址空间

用户可以通过 `cat /proc/pid/maps4` 文件看到进程的虚拟地址空间分配情况⁵，当运行一个可指向文件时，可执行文件的装载器会根据这个布局加载可执行文件。用户态虚拟地址空间的顶端地址为 3GB，这里空出一小片区域不映射，接下来是进程的环境变量，再接下来是命令行参数，这是装载器放置好的，再往下就是动态库，对于某些库，只有一份代码存在物理内存中，但是被映射到了多个进程的虚拟地址空间。之后，就是进程的堆栈，对于 C 程序来说，`argc`, `argv` 还有 `main` 函数的返回地址压入堆栈后，就调用 `main()` 函数。堆

⁴操作时，请使用一个实际的进程 ID 替换这里的 pid。

⁵内核态进程没有 mm 结构，所以它的 maps 什么也看不到。

栈从高地址向低地址增长。从低地址 0x80000000 开始，是代码段和数据段，再往上就是堆(Heap)的起始地址，堆从低地址向高地址增长，我们知道 malloc()这一类函数就是从堆中分配内存的。这里堆和栈都是动态变化的，它们一个从上往下增长，另外一个从下往上增长。对于堆栈操作，当进程访问到一个没有被映射的虚拟地址时，缺页中断会分配物理页面并建立映射。类似地，对于堆操作，当分配到物理内存后，就映射到堆的虚拟地址空间上。由于进程对堆和栈的大小都有限制，因此缺页中断可以保证它们在虚拟地址空间上不会相交。

每一个用户态进程都有独立的页目录，pgd 就是页目录指针。通过图10.3我们看到 3G 的虚拟地址空间不可能是全部映射到物理内存中的，所以 mm 结构的 vm_area_struct 记录了虚拟地址空间的使用情况⁶，其主要成员定义如下：

代码片段 10.5 节自 include/linux/mm_types.h

```

1 struct vm_area_struct {
2     /* The address space we belong to. */
3     struct mm_struct * vm_mm;
4     /* Our start address within vm_mm. */
5     unsigned long vm_start;
6     /* The first byte after our end address within vm_mm. */
7     unsigned long vm_end;
8     /* linked list of VM areas per task, sorted by address */
9     struct vm_area_struct *vm_next;
10    /* Access permissions of this VMA. */
11    pgprot_t vm_page_prot;
12    /* Flags, listed below. */
13    unsigned long vm_flags;
14    struct rb_node vm_rb;
15    /* File we map to (can be NULL). */
16    struct file * vm_file;
17
18 };

```

在这个结构中，vm_start 和 vm_end 记录了这个进程当前使用的虚拟地址空间。假设一个进程的某段合法的起始虚拟地址为 A，大小为 2 个物理页面，就需要一个 vm_area_struct 来描述这段虚拟地址区域，通过图10.3我们看到，一个进程有多个虚拟地址区域，这些 vm_area_struct 根据虚拟地址被组织成一颗红黑树，这主要是为了加速查找的速度。假设进程最初仅仅从起始虚拟地址 A 开始映射了一个物理页面，当该进程首次访问大于 A+4KB 小于 A+8KB 的虚拟地址时，由于这个虚拟地址没有映射物理页面，因此会触发页故障，此时缺页中断处理函数 do_page_fault()将查找该进程的 vm_area_struct 结构，在本例中，这个被访问的虚拟地址在合法的范围内，所以 do_page_fault()会分配一个新的物理页面并为

⁶在第5章，我们看到 page 结构记录了物理内存的使用情况。

这个虚拟地址建立映射，此时该进程成功地访问到第二个物理页面。

我们知道 fork() 会复制当前进程的 task_struct 结构，同时会为新进程复制 mm 结构⁷。此时当前进程的 3GB~4GB 的内核态虚拟地址对应的页表项(目录项)被复制到新进程的页表项(目录项)中，所以说所有进程共享 1G 的内核态地址空间。但是对于用户态虚拟地址区域，则把它的进程页表项(目录项)设置为只读，这样当其中一个进程对其进行写入操作时，do_page_fault() 会分配新的物理页面，并建立映射，从而实现 Copy On Write 机制。

10.1.6 进程的文件信息

进程的文件信息包括 fs_struct 和 files_struct 两个结构，前者用来描述进程的根目录，以及当前目录等信息，而后者是进程的文件描述符。其中 fs_struct 结构定义如下：

代码片段 10.6 节自 include/linux/fs_struct.h

```

1 struct fs_struct {
2     atomic_t count;
3     rwlock_t lock;
4     int umask;
5     struct dentry * root, * pwd, * altroot;
6     struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
7 };

```

其中 dentry 表示 directory entry，它用来描述一个打开的目录项，root 表示当前进程的根目录，而 pwd 则表示进程的当前目录，当一个进程使用 chroot 系统调用改变自己的根目录时，altroot 就指向这个根目录，通常像 ftp, http 等服务器进程会利用 chroot 系统调用把自己的根目录改变为默认的组目录，这样通过 ftp, http 登录系统的用户就不能访问系统中的其他目录了。这些目录不可能在同一个安装点上，例如 root 可能在分区格式为 ext3 的一个分区上，而 pwd 可能在一个分区格式为 NTFS 的 USB 设备上，而系统中分区设备的每一个安装点都由一个 vfsmount 结构来描述，因此这个结构中的信息表示，root 是位于安装点 rootmnt 下的某个目录。pwd 是位于安装点 pwdmnt 下的某个目录。

files_struct 结构用来描述进程打开文件，其定义如下：

代码片段 10.7 节自 include/linux/file.h

```

1 struct files_struct {
2     .....
3     struct file * fd_array[NR_OPEN_DEFAULT];
4 };

```

它的主要成员就是 fd_array。Linux 内核中，每一个打开的文件由一个 file 结构来描述，fd_array 数组就保存着这些 file 结构的指针，数组下标就是应用程序的文件描述符。我

⁷对于轻权进程，可能会共享 mm 结构。

们都知道 0,1,2 分别表示标准输入、标准输出和标准错误的文件描述符，对应到内核中，就是 fd_array[0], fd_array[1], fd_array[2] 分别指向标准输入、标准输出和标准错误的文件描述符 file 结构。注意每次打开文件，描述符依次向上加 1。为了进一步说明，我们来看看内核启动 init 进程时，初始化的 files_struct。

代码片段 10.8 节自 init/main.c

```

1 [kernel_init() -> init_post()]
2
3 static int __init init_post(void)
4 {
5     .....
6     /* 打开 console，其文件描述符 file 的指针保存在 fd_array[0] 中。 */
7     if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
8         printk(KERN_WARNING "Warning: unable to open an initial console.\n");
9
10    /* 复制文件描述符 0，索引向上加 1，这样 fd_array[1] 就指向了 console 的 file 结构。 */
11    (void) sys_dup(0);
12    /* fd_array[2] 就指向了 console 的 file 结构。 */
13    (void) sys_dup(0);
14    .....
15    run_init_process("/sbin/init");
16    .....
17    panic("No init found. Try passing init= option to kernel.");
18 }

```

经过上面的处理后，init 进程的默认标准输入、标准输出和标准错误就都是 console 了。

另外 task_struct 结构中还包括了大量的进程信息，例如，进程的时间和定时器，优先级，用户和用户组，等等。在此不一一详述，当分析到相关内容时，再来细说。

10.2 进程的建立

创建一个进程时，父进程的一切资源都会复制到子进程中，这是一个烦琐的操作，事实上大多数进程在调用 fork() 后，都会调用 execve() 加载一个可执行文件，这样之前复制的代码、数据都会被重写。为了减小不必要的消耗，fork() 使用了 Copy On Write 机制，这样父进程和子进程共享物理页面，同时把这些物理页面设置为只读，当某个进程进行写入操作时，内核会为该进程复制一份私有的页面。另外，根据不同的需要，可以指定子进程和父进程共享某些成员。当利用系统调用 clone() 建立一个轻权进程时，子进程和父进程共享页表⁸、文件、信号等资源，这其实也就是通常所说的线程。当通过系统调用 vfork() 来建立一

⁸这样也就共享了整个地址空间，在轻权进程之间切换时，就不需要切换页表，从而减小了系统开销。

个进程时，子进程和父进程共享页表，由于内存由父子进程共享，为了阻止父进程修改内存数据，父进程将被阻塞，直到子进程结束，或者子进程调用 `execve()` 加载一个可执行文件。系统调用 `fork()`, `vfork()`, `clone()` 最终都是调用内核态的 `do_fork()` 函数。另外在内核中，还可以通过调用 `kernel_thread()` 来建立内核态线程，这个函数也是调用 `do_fork()` 的。我们先来看看 `do_fork()` 的参数：

- (1) `clone_flags`, 指定了子进程结束时，需要向父进程发送的信号，通常这个信号是 `SIGCHLD`，同时 `clone_flags` 还指定子进程需要共享父进程的哪些资源，如表 10.1 所示。

表 10.1 `fork_flags` 标志

标志	描述
<code>CLONE_VM</code>	共享内存信息及页表
<code>CLONE_FS</code>	共享 <code>fs_struct</code> 结构，这样父子进程的根目录和当前目录一致
<code>CLONE_FILES</code>	共享打开的文件
<code>CLONE_SIGHAND</code>	共享信号处理信息
<code>CLONE_PTRACE</code>	如果父进程正处于被调试状态，要求子进程也处于被调试状态
<code>CLONE_VFORK</code>	建立子进程后，父进程保持阻塞状态，直到子进程退出或者调用 <code>execve()</code>
<code>CLONE_PARENT</code>	把子进程的 <code>parent</code> 和 <code>real_parent</code> 设置为当前进程的父进程，一般情况，子进程的 <code>parent</code> 和 <code>real_parent</code> 应该设置为当前进程
<code>CLONE_THREAD</code>	子进程和父进程在同一个线程组，这样子进程的 <code>tgid</code> 和 <code>group_leader</code> 都会做相应的设置，可以把它理解为同一个进程中的多个线程
<code>CLONE_NEWNS</code>	子进程有自己的名字空间和文件系统
<code>CLONE_SYSVSEM</code>	共享 System V IPC semaphore
<code>CLONE_SETTLS</code>	为子进程设置独立的线程本地存储(TLS)
<code>CLONE_PARENT_SETTID</code>	要求把子进程的 PID 写入到 <code>do_fork()</code> 的 <code>parent_tidptr</code> 参数中，这个参数是用户态的一个指针。这样其实是通过参数传地址调用返回 PID
<code>CLONE_CHILD_SETTID</code>	要求把子进程的 PID 写入到 <code>do_fork()</code> 的 <code>child_tidptr</code> 参数中，同样这也是一个用户态指针
<code>CLONE_CHILD_CLEARTID</code>	当子进程退出或者调用 <code>execve()</code> 时，内核清除 <code>do_fork()</code> 的 <code>child_tidptr</code> ，同时唤醒等待该事件的进程
<code>CLONE_DETACHED</code>	为了兼容而设置的参数，目前几乎不被使用
<code>CLONE_UNTRACED</code>	建立一个不允许被调试的进程，通常内核态线程会设置该标志

续下页

fork_flags 标志(续表)

标志	描述
CLONE_STOPPED	建立的子进程处于 STOPPED 状态，将来由其他进程来改变这种状态
CLONE_NEWIPC ⁹	子进程用于独立的 IPC 空间
CLONE_NEWUSER	子进程拥有独立的用户名空间 ¹⁰
CLONE_NEWPID	子进程拥有独立的 PID 空间，这样在同时，可以有多个进程拥有相同的 pid，当然这些进程不在同一个 PID 空间
LONE_NEWWNET	子进程拥有独立的网络名字空间

fork_flags 标志(完)

- (2) *stack_start* 子进程用户态堆栈起始地址。通常设置为 0，父进程会复制自己的堆栈指针，当子进程对堆栈进行写入时，缺页中断处理程序会设置新的物理页面。
- (3) *regs* 这是 *pt_regs* 结构，保存了进入内核态时的寄存器的值(见第6章)。
- (4) *stack_size* 没有被使用，默认为 0。
- (5) *parent_tidptr* 用户态内存指针，当 CLONE_PARENT_SETTID 被设置时，内核会把新建立的子进程 ID 通过 *parent_tidptr* 返回。
- (6) *child_tidptr* 用户态内存指针，当 CLONE_CHILD_SETTID 被设置时，内核会把新建立的子进程 ID 通过 *child_tidptr* 返回。

10.2.1 建立子进程的 *task_struct* 对象

现在来看看 *do_fork()* 函数。

代码片段 10.9 节自 *kernel/fork.c*

```

1 long do_fork(unsigned long clone_flags,
2             unsigned long stack_start,
3             struct pt_regs *regs,
4             unsigned long stack_size,
5             int __user *parent_tidptr,
6             int __user *child_tidptr)
7 {
8     struct task_struct *p;
9     int trace = 0;

```

⁹接下来这几个标志是为虚拟化设置的，例如 OpenVZ，它允许同时在一台电脑上虚拟运行多个操作系统。

¹⁰这里指系统中登录的用户名

```
10     long nr;
11
12     /*
13      * current 是父进程，如果该进程被跟踪，那么当调试器要求跟踪每一个子进程时，
14      * 创建出来的子进程也处于跟踪状态。
15      */
16     if (unlikely(current->ptrace)) {
17         trace = fork_traceflag (clone_flags);
18         if (trace)
19             clone_flags |= CLONE_PTRACE;
20     }
21     /*
22      * 分配子进程 task_struct 结构，并复制父进程的资源，
23      * 我们接下来将对这个函数进行详细的分析。
24      */
25     p = copy_process(clone_flags, stack_start, regs,
26                      stack_size, child_tidptr, NULL);
27
28     if (!IS_ERR(p)) {
29         struct completion vfork;
30         /*
31          * 这是为进程 pid namespaces 设置的，不同的 namespaces 中，
32          * 可以建立相同的 pid 的进程。
33          */
34         nr = (clone_flags & CLONE_NEWPID) ?
35             task_pid_nr_ns(p, current->nsproxy->pid_ns) : task_pid_vnr(p);
36         /* 把进程 ID 传递到 parent_tidptr 指针指定的用户空间。*/
37         if (clone_flags & CLONE_PARENT_SETTID)
38             put_user(nr, parent_tidptr);
39         /* CLONE_VFORK 要求父进程进入子进程，现在初始化一个等待对象。*/
40         if (clone_flags & CLONE_VFORK) {
41             p->vfork_done = &vfork;
42             init_completion(&vfork);
43         }
44         /* 如果被调试，或者设置了 CLONE_STOPPED 标志，则向进程发送 SIGSTOP 信号。*/
45         if ((p->ptrace & PT_PTRACED) || (clone_flags & CLONE_STOPPED)) {
46             /*
47              * We'll start up with an immediate SIGSTOP.
48              */
49             sigaddset(&p->pending.signal, SIGSTOP);
50             set_tsk_thread_flag(p, TIF_SIGPENDING);
51         }
```

```

52     /* 如果没有设置 CLONE_STOPPED 标志，就把进程加入就绪队列。*/
53     if (!(clone_flags & CLONE_STOPPED))
54         wake_up_new_task(p, clone_flags);
55     else
56         p->state = TASK_STOPPED;
57
58     /* 如果被调试，就发送 SIGTRAP 信号。*/
59     if (unlikely(trace)) {
60         current->ptrace_message = nr;
61         ptrace_notify ((trace << 8) | SIGTRAP);
62     }
63
64     if (clone_flags & CLONE_VFORK) {
65         freezer_do_not_count();
66         /* 当前进程进入之前初始化好的等待队列。*/
67         wait_for_completion(&vfork);
68         freezer_count();
69         if (unlikely(current->ptrace & PT_TRACE_VFORK_DONE)) {
70             current->ptrace_message = nr;
71             ptrace_notify ((PTRACE_EVENT_VFORK_DONE << 8) | SIGTRAP);
72         }
73     }
74 } else {
75     nr = PTR_ERR(p);
76 }
77 return nr;
78 }

```

`copy_process()`函数为子进程分配必要内存，并把父进程的相关数据结构复制到了进程，这个函数代码很多。下面的分析省略了一些不重要的部分。

代码片段 10.10 节自 `kernel/fork.c`

```

1 static struct task_struct
2     *copy_process(unsigned long clone_flags,
3                 unsigned long stack_start,
4                 struct pt_regs *regs,
5                 unsigned long stack_size,
6                 int __user *child_tidptr,
7                 struct pid *pid)
8 {
9     int retval;
10    struct task_struct *p;
11    int cgroup_callbacks_done = 0;

```

```
12  /* 标志检查。*/
13  if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
14      return ERR_PTR(-EINVAL);
15  if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))
16      return ERR_PTR(-EINVAL);
17  if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
18      return ERR_PTR(-EINVAL);
19  /*
20  * 安全检查框架，利用它可以在进程建立前检查是否允许检查，利用这个开发框架
21  * 可以开发出进程监控功能。默认调用 dummy_task_create 函数，它什么也不做。
22  */
23  retval = security_task_create(clone_flags);
24  if (retval)
25      goto fork_out;
26  retval = -ENOMEM;
27  /*
28  * 为子进程分配一个 task_struct 和内核态堆栈，把父进程的 task_struct 结构
29  * 复制到子进程，同时设置内核态堆栈中的 thread_info 结构，见图10.2。
30  */
31  p = dup_task_struct(current);
32  if (!p)
33      goto fork_out;
34  rt_mutex_init_task(p);
35
36 #ifdef CONFIG_TRACE_IRQFLAGS
37     DEBUG_LOCKS_WARN_ON(!p->hardirqs_enabled);
38     DEBUG_LOCKS_WARN_ON(!p->softirqs_enabled);
39 #endif
40 /*
41  * 检查进程的资源限制。*/
42  retval = -EAGAIN;
43  if (atomic_read(&p->user->processes) >=
44      p->signal->rlim[RLIMIT_NPROC].rlim_cur) {
45      if (!capable(CAP_SYS_ADMIN) &&
46          !capable(CAP_SYS_RESOURCE) &&
47          p->user != current->nsproxy->user_ns->root_user)
48          goto bad_fork_free;
49  }
50
51 /* 增加当前用户建立的进程数量。*/
52 atomic_inc(&p->user->_count);
53 atomic_inc(&p->user->processes);
```

```
54     get_group_info(p->group_info);
55
56     if (nr_threads >= max_threads)
57         goto bad_fork_cleanup_count;
58
59     if (!try_module_get(task_thread_info(p)->exec_domain->module))
60         goto bad_fork_cleanup_count;
61
62     if (p->binfo && !try_module_get(p->binfo->module))
63         goto bad_fork_cleanup_put_domain;
64
65     p->did_exec = 0;
66
67     /* Must remain after dup_task_struct() */
68     delayacct_tsk_init(p);
69
70
71     /* 拷贝 clone_flags 到子进程的 task_struct. */
72     copy_flags(clone_flags, p);
73
74     INIT_LIST_HEAD(&p->children);
75     INIT_LIST_HEAD(&p->sibling);
76     p->vfork_done = NULL;
77     spin_lock_init(&p->alloc_lock);
78
79     .....
80
81     /* Perform scheduler related setup. Assign this task to a CPU. */
82     sched_fork(p, clone_flags);
83
84
85     /* 安全检查框架，默认什么也不做。*/
86     if ((retval = security_task_alloc(p)))
87         goto bad_fork_cleanup_policy;
88
89     if ((retval = audit_alloc(p)))
90         goto bad_fork_cleanup_security;
91
92
93     /* copy all the process information */
94
95     /*
96      * 下面根据 clone_flags 复制父进程的资源到子进程，对于 clone_flags 指定共享的资源，
97      * 则仅仅设置子进程的相关指针，并增加资源数据结构的引用计数。
98      */
99
100    if ((retval = copy_semundo(clone_flags, p)))
101        goto bad_fork_cleanup_audit;
102
103    if ((retval = copy_files(clone_flags, p)))
104        goto bad_fork_cleanup_semundo;
105
106    if ((retval = copy_fs(clone_flags, p)))
107        goto bad_fork_cleanup_files;
108
109    if ((retval = copy_sighand(clone_flags, p)))
110        goto bad_fork_cleanup_fs;
111
112    if ((retval = copy_signal(clone_flags, p)))
```

```
96     goto bad_fork_cleanup_sighand;
97     if ((retval = copy_mm(clone_flags, p)))
98         goto bad_fork_cleanup_signal;
99     if ((retval = copy_keys(clone_flags, p)))
100        goto bad_fork_cleanup_mm;
101    if ((retval = copy_namespaces(clone_flags, p)))
102        goto bad_fork_cleanup_keys;
103
104    /* 复制父进程的内核态堆栈到子进程。*/
105    retval = copy_thread(0, clone_flags, stack_start, stack_size, p, regs);
106    if (retval)
107        goto bad_fork_cleanup_namespaces;
108    .....
109    p->pid = pid_nr(pid);
110    p->tgid = p->pid;
111    /* 如果建立的是轻权进程，那么父子进程在同一个线程组中，就设置子进程的 tgid。*/
112    if (clone_flags & CLONE_THREAD)
113        p->tgid = current->tgid;
114
115    p->set_child_tid = (clone_flags & CLONE_CHILD_SETTID) ?
116                                child_tidptr : NULL;
117    /*
118     * Clear TID on mm_release()?
119     */
120    p->clear_child_tid = (clone_flags & CLONE_CHILD_CLEARTID) ?
121                                child_tidptr: NULL;
122    .....
123    /* 父进程是否要求子进程退出时发送信号。*/
124    /* ok, now we should be set up.. */
125    p->exit_signal = (clone_flags & CLONE_THREAD) ?
126                            -1 : (clone_flags & CSIGNAL);
127    p->pdeath_signal = 0;
128    /* 子进程默认的退出状态。*/
129    p->exit_state = 0;
130    .....
131    /* CLONE_PARENT re-uses the old parent */
132    if (clone_flags & (CLONE_PARENT|CLONE_THREAD))
133        /* 把子进程的 real_parent 设置为父进程的 real_parent. */
134        p->real_parent = current->real_parent;
135    else
136        p->real_parent = current;
137    p->parent = p->real_parent;
```

```
138
139     spin_lock(&current->sighand->siglock);
140     .....
141     if (likely(p->pid)) {
142         /* 把子进程添加到父进程的子进程链表中，这样组成了兄弟进程链表。*/
143         add_parent(p);
144         /* 如果父进程是调试器，那么设置子进程的 parent 指针为调试器的父进程。*/
145         if (unlikely(p->ptrace & PT_PTRACED))
146             __ptrace_link(p, current->parent);
147         .....
148     }
149
150     total_forks++;
151     spin_unlock(&current->sighand->siglock);
152     write_unlock_irq(&tasklist_lock);
153     proc_fork_connector(p);
154     cgroup_post_fork(p);
155     return p;
156
157 /* 出错退出。*/
158 bad_fork_free_pid:
159     if (pid != &init_struct_pid)
160         free_pid(pid);
161     .....
162 bad_fork_free:
163     free_task(p);
164 fork_out:
165     return ERR_PTR(retval);
166 }
```

`copy_process()`完成的工作主要是进行必要的检查、初始化、复制必要的数据结构。由于复制初始化的资源太多，涉及进程的方方面面，包括文件、信号、内存等，这里无法一一进行详细的分析。接下来仅仅举两个例子，分别是 `copy_mm()` 和 `copy_thread()`，选这两个函数作为例子，是因为通过前者可以了解到父子进程 Copy On Write 机制，以及共享内核虚拟地址的实现。而后者是子进程如何返回的关键。我们知道，应用程序通过 `fork()` 系统调用进入内核空间，其内核态堆栈上保存着该进程的进程上下文¹¹。通过 `copy_thread()` 将复制父进程内核态堆栈上的进程上下文，同时把堆栈上的 EAX 设置为 0。由于父子进程的代码和数据是共享的，所以返回后将接着执行，现在终于理解“`fork()` 调用一次返回两次，在父进程中返回子进程的 ID，在子进程中返回 0。”这句话了吧。

¹¹ 即该进程的各个寄存器。

10.2.2 子进程的内存区域

之后内核调用 `copy_mm()` 来建立子进程的内存区域，该函数具体分析如下：

代码片段 10.11 节自 `kernel/fork.c`

```
1 static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
2 {
3     struct mm_struct * mm, *oldmm;
4     int retval;
5
6     tsk->min_flt = tsk->maj_flt = 0;
7     tsk->nvcsw = tsk->nivcsw = 0;
8     tsk->mm = NULL;
9     tsk->active_mm = NULL;
10
11    oldmm = current->mm;
12    if (!oldmm)
13        return 0;
14
15    /* 如果要共享 mm，则增加父进程 mm 的引用计数。同时把设置 mm 为 current->mm。 */
16    if (clone_flags & CLONE_VM) {
17        atomic_inc(&oldmm->mm_users);
18        mm = oldmm;
19        goto good_mm;
20    }
21    retval = -ENOMEM;
22    mm = dup_mm(tsk);
23    if (!mm)
24        goto fail_nomem;
25 good_mm:
26    /* Initializing for Swap token stuff */
27    mm->token_priority = 0;
28    mm->last_interval = 0;
29    tsk->mm = mm;
30    tsk->active_mm = mm;
31    return 0;
32
33 fail_nomem:
34     return retval;
35 }
```

复制 `mm_struct` 的工作由 `dup_mm()` 来完成，这个函数会复制父进程的页表到子进程，这样父子进程就共享同样的物理页面，同时也共享了整个内核空间。但是对于可写

的用户空间对应的页表，`dup_mm()`会把它们设置为只读，这样当进程对它进行写入时，`do_page_fault()`函数将分配新的物理页面，为进程复制一份私有数据，这就是 Copy On Write 机制。在前面讨论 `copy_process()` 的过程中，细心的读者一定在问：为什么只看到为子进程分配内核态堆栈，但却没有分配用户态堆栈？这是由于 Copy On Write 机制，当父子进程的任何一个返回用户态首次对堆栈进行写入操作时，父子进程就会有各自独立的用户态堆栈了，但是对于代码段，它们却始终共享同一份物理页面，除非子进程调用 `execvp()` 系列函数。

代码片段 10.12 节自 `kernel/fork.c`

```

1 static struct mm_struct *dup_mm(struct task_struct *tsk)
2 {
3     struct mm_struct *mm, *oldmm = current->mm;
4     int err;
5
6     if (!oldmm)
7         return NULL;
8     /* 分配 mm_struct 结构。 */
9     mm = allocate_mm();
10    if (!mm)
11        goto fail_nomem;
12    /* 复制 mm_struct 结构。 */
13    memcpy(mm, oldmm, sizeof(*mm));
14    /* Initializing for Swap token stuff */
15    mm->token_priority = 0;
16    mm->last_interval = 0;
17    /* 初始化，同时分配页表。 */
18    if (!mm_init(mm))
19        goto fail_nomem;
20    if (init_new_context(tsk, mm))
21        goto fail_nocontext;
22
23    /* 拷贝 vm_area_struct 结构。 */
24    err = dup_mmap(mm, oldmm);
25    if (err)
26        goto free_pt;
27    mm->hiwater_rss = get_mm_rss(mm);
28    mm->hiwater_vm = mm->total_vm;
29
30    return mm;
31 free_pt:
32     .....
33     return NULL;

```

34 }

`mm_init()`初始化 `mm_struct` 结构中的自旋锁、链表等资源，然后调用 `mm_alloc_pgd()` 函数分配页表，同时把父进程的内核虚拟地址对应的页表项复制到子进程的页表，因此它们共享了内核态地址空间。

代码片段 10.13 节自 `kernel/fork.c`

```

1 static inline int mm_alloc_pgd(struct mm_struct * mm)
2 {
3     mm->pgd = pgd_alloc(mm);
4     if (unlikely(!mm->pgd))
5         return -ENOMEM;
6     return 0;
7 }
8
9 pgd_t *pgd_alloc(struct mm_struct *mm)
10 {
11     int i;
12     /* 分配页目录页面，同时调用 pgd_ctor 函数对页目录进行初始化。*/
13     pgd_t *pgd = quicklist_alloc(0, GFP_KERNEL, pgd_ctor);
14     if (PTRS_PER_PMD == 1 || !pgd)
15         return pgd;
16
17     /* 分配页内核态虚拟地址空间所需的页表12，同时把页表的物理地址设置到页目录中。*/
18     for (i = 0; i < UNSHARED_PTRS_PER_PGD; ++i) {
19         pmd_t *pmd = pmd_cache_alloc(i);
20
21         if (!pmd)
22             goto out_oom;
23         /* 虚拟化支持，我们不讨论这种情况。*/
24         paravirt_alloc_pd(__pa(pmd) >> PAGE_SHIFT);
25         set_pgd(&pgd[i], __pgd(1 + __pa(pmd)));
26     }
27     return pgd;
28 out_oom:
29     /* 只要有一个页表分配失败，就释放之前分配到的页表。*/
30     for (i--; i >= 0; i--) {
31         pgd_t pgdent = pgd[i];
32         void* pmd = (void *)__va(pgd_val(pgdent)-1);
33         paravirt_release_pd(__pa(pmd) >> PAGE_SHIFT);
34         pmd_cache_free(pmd, i);

```

¹²这里我们只考虑两级分页的情况。

```

35     }
36     quicklist_free(0, pgd_dtor, pgd);
37     return NULL;
38 }

```

pmd_cache_alloc()分配页表，并把swapper_pg_dir内核态地址空间对应的页表复制到新的页表中¹³。

代码片段 10.14 节自 arch/x86/mm/pgtable_32.c

```

1 static pmd_t *pmd_cache_alloc(int idx)
2 {
3     pmd_t *pmd;
4
5     /* 内核态虚拟地址空间，分配页表并复制 swapper_pg_dir(见第4.3节)。 */
6     if (idx >= USER_PTRS_PER_PGD) {
7         pmd = (pmd_t *)__get_free_page(GFP_KERNEL);
8
9         if (pmd)
10             memcpy(pmd, (void *)pgd_page_vaddr(swapper_pg_dir[idx]),
11                    sizeof(pmd_t) * PTRS_PER_PMD);
12     } else
13         /* 用户态虚拟地址空间，分配页表。 */
14         pmd = kmem_cache_alloc(pmd_cache, GFP_KERNEL);
15
16     return pmd;
17 }

```

现在页表分配好了，同时内核态地址空间的映射关系已经建立。接下来我们看看dup_mmap()是如何处理用户态地址空间的相关数据结构的。它主要分配并复制vm_area_struct结构，同时根据vm_area_struct结构的属性标志设置页表项，把可写入的内存片段设置为只读。这样当某个进程对其进行写入时，do_page_fault()将分配新的物理页面，为该进程建立私有数据，同时修改页表，指向新的物理页面。

代码片段 10.15 节自 kernel/fork.c

```

1 static int dup_mmap(struct mm_struct *mm, struct mm_struct *oldmm)
2 {
3     struct vm_area_struct *mpnt, *tmp, **pprev;
4     struct rb_node **rb_link, *rb_parent;
5     int retval;
6     unsigned long charge;

```

¹³对于更多级的分页，pmd_cache_alloc分配本级页表，同时会把本级页表的物理地址写到对应的上级页表中，同时调用分配下级页表的函数，最终会复制内核态地址空间的页表，从而实现共享内核态地址空间。

```
7   struct mempolicy *pol;
8
9   down_write(&oldmm->mmap_sem);
10  flush_cache_dup_mm(oldmm);
11  /*
12   * Not linked in yet - no deadlock potential:
13   */
14  down_write_nested(&mm->mmap_sem, SINGLE_DEPTH_NESTING);
15
16  mm->locked_vm = 0;
17  mm->mmap = NULL;
18  mm->mmap_cache = NULL;
19  mm->free_area_cache = oldmm->mmap_base;
20  mm->cached_hole_size = ~0UL;
21  mm->map_count = 0;
22  cpus_clear(mm->cpu_vm_mask);
23  mm->mm_rb = RB_ROOT;
24  rb_link = &mm->mm_rb.rb_node;
25  rb_parent = NULL;
26  pprev = &mm->mmap;
27
28  /* 处理每一个 vm_area_struct 结构。*/
29  for (mpnt = oldmm->mmap; mpnt; mpnt = mpnt->vm_next) {
30      struct file *file;
31
32      /* 不需要复制。*/
33      if (mpnt->vm_flags & VM_DONTCOPY) {
34          long pages = vma_pages(mpnt);
35          mm->total_vm -= pages;
36          vm_stat_account(mm, mpnt->vm_flags, mpnt->vm_file, -pages);
37          continue;
38      }
39      charge = 0;
40      /* 需要安全计数检查。*/
41      if (mpnt->vm_flags & VM_ACCOUNT) {
42          unsigned int len = (mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT;
43          if (security_vm_enough_memory(len))
44              goto fail_nomem;
45          charge = len;
46      }
47      /* 为子进程分配新的 vm_area_struct 结构。*/
48      tmp = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL);
```

```
49     if (!tmp)
50         goto fail_nomem;
51
52     /* 注意，这是整个结构复制。*/
53     *tmp = *mpnt;
54     pol = mpol_copy(vma_policy(mpnt));
55     retval = PTR_ERR(pol);
56     if (IS_ERR(pol))
57         goto fail_nomem_policy;
58     vma_set_policy(tmp, pol);
59     tmp->vm_flags &= ~VM_LOCKED;
60     tmp->vm_mm = mm;
61     tmp->vm_next = NULL;
62     anon_vma_link(tmp);
63     file = tmp->vm_file;
64     /* 如果这片内存对应的是一个文件映射，则设置文件相关信息，增加文件的引用计数等。*/
65     if (file) {
66         struct inode *inode = file->f_path.dentry->d_inode;
67         get_file(file);
68         if (tmp->vm_flags & VM_DENYWRITE)
69             atomic_dec(&inode->i_writecount);
70
71         /* insert tmp into the share list, just after mpnt */
72         spin_lock(&file->f_mapping->i_mmap_lock);
73         tmp->vm_truncate_count = mpnt->vm_truncate_count;
74         flush_dcache_mmap_lock(file->f_mapping);
75         vma_prio_tree_add(tmp, mpnt);
76         flush_dcache_mmap_unlock(file->f_mapping);
77         spin_unlock(&file->f_mapping->i_mmap_lock);
78     }
79
80     /*
81      * Link in the new vma and copy the page table entries.
82      */
83     /* 把新的 vm_area_struct 结构添加到子进程。*/
84     *pprev = tmp;
85     pprev = &tmp->vm_next;
86     /* 添加到红黑树。*/
87     __vma_link_rb(mm, tmp, rb_link, rb_parent);
88     rb_link = &tmp->vm_rb.rb_right;
89     rb_parent = &tmp->vm_rb;
90     mm->map_count++;
```

```

91     /* 分配设置页表，并不需要分配物理页面。*/
92     retval = copy_page_range(mm, oldmm, mpnt);
93     if (tmp->vm_ops && tmp->vm_ops->open)
94         tmp->vm_ops->open(tmp);
95
96     if (retval)
97         goto out;
98 }
99 /* a new mm has just been created */
100 arch_dup_mmap(oldmm, mm);
101 retval = 0;
102 out:
103     up_write(&mm->mmap_sem);
104     flush_tlb_mm(oldmm);
105     up_write(&oldmm->mmap_sem);
106     return retval;
107 fail_nomem_policy:
108     kmem_cache_free(vm_area_cachep, tmp);
109 fail_nomem:
110     retval = -ENOMEM;
111     vm_unacct_memory(charge);
112     goto out;
113 }
```

`copy_page_range()`函数，它需要为 `vm_area_struct` 结构指定的内存区域分配并设置页表，同时把页表的物理地址设置到页目录（也就是上级页表）中。

代码片段 10.16 节自 `mm/memory.c`

```

1 int copy_page_range(struct mm_struct *dst_mm,
2                     struct mm_struct *src_mm,
3                     struct vm_area_struct *vma)
4 {
5     pgd_t *src_pgd, *dst_pgd;
6     unsigned long next;
7     unsigned long addr = vma->vm_start;
8     unsigned long end = vma->vm_end;
9
10    /*
11     * Don't copy ptes where a page fault will fill them correctly.
12     * Fork becomes much lighter when there are big shared or private
13     * readonly mappings. The tradeoff is that copy_page_range is more
14     * efficient than faulting.
15     */
```

```

16    if (!(vma->vm_flags & (VM_HUGETLB| VM_NONLINEAR| VM_PFNMAP| VM_INSERTPAGE)))
17    {
18        if (!vma->anon_vma)
19            return 0;
20    }
21    if (is_vm_hugetlb_page(vma))
22        return copy_hugetlb_page_range(dst_mm, src_mm, vma);
23    dst_pgd = pgd_offset(dst_mm, addr);
24    src_pgd = pgd_offset(src_mm, addr);
25    do {
26        next = pgd_addr_end(addr, end);
27        if (pgd_none_or_clear_bad(src_pgd))
28            continue;
29        if (copy_pud_range(dst_mm, src_mm,
30                            dst_pgd, src_pgd,
31                            vma, addr, next))
32            return -ENOMEM;
33    } while (dst_pgd++, src_pgd++, addr = next, addr != end);
34    return 0;
}

```

由于为了支持多级分页，从代码上看 `copy_pud_range()` 比较烦琐，在多级分页中，它可能需要不断地为 `vm_start`, `vm_end` 指定的虚拟地址设置页表。最终它调用 `copy_one_pte()` 设置页表项：

代码片段 10.17 节自 mm/memory.c

```

1 static inline void
2     copy_one_pte(struct mm_struct *dst_mm,
3                  struct mm_struct *src_mm,
4                  pte_t *dst_pte,
5                  pte_t *src_pte,
6                  struct vm_area_struct *vma,
7                  unsigned long addr, int *rss)
8 {
9     unsigned long vm_flags = vma->vm_flags;
10    pte_t pte = *src_pte;
11    struct page *page;
12
13    /* pte contains position in swap or file, so copy. */
14    /*
15     * 虚拟地址对应的页表被交换到磁盘上。需要注意的是，缺页中断可以
16     * 从磁盘交换分区调入内存，但是缺页中断所用的内存及其页表是不可
17     * 交换的，因此内核空间使用的页表是不可交换的。

```

```
18     */
19     if (unlikely(!pte_present(pte))) {
20         .....
21     }
22     /* 如果是可写内存区域，则利用页表把该段内存区域设置为只读，以实现 Copy On Write 机制。*/
23     /*
24      * If it's a COW mapping, write protect it both
25      * in the parent and the child
26      */
27     if (is_cow_mapping(vm_flags)) {
28         ptep_set_wrprotect(src_mm, addr, src_pte);
29         pte = pte_wrprotect(pte);
30     }
31     /*
32      * If it's a shared mapping, mark it clean in
33      * the child
34      */
35     if (vm_flags & VM_SHARED)
36         pte = pte_mkcold(pte);
37     pte = pte_mkold(pte);
38
39     page = vm_normal_page(vma, addr, pte);
40     if (page) {
41         get_page(page);
42         page_dup_rmap(page, vma, addr);
43         rss[ !PageAnon(page) ]++;
44     }
45     out_set_pte:
46     set_pte_at(dst_mm, addr, dst_pte, pte);
47 }
```

10.2.3 子进程的内核态堆栈

当父进程进行系统调用时，在它的内核态保存了进程的通用寄存器，这是一个 `pt_regs` 结构，`copy_thread()` 会复制父进程的 `pt_reg` 的结构到子进程的内核态堆栈。

代码片段 10.18 节自 `arch/x86/kernel/process_32.c`

```
1 int copy_thread(int nr, unsigned long clone_flags,
2                 unsigned long esp,
3                 unsigned long unused,
4                 struct task_struct * p,
```

```
5             struct pt_regs * regs)
6 {
7     struct pt_regs * childregs;
8     struct task_struct *tsk;
9     int err;
10
11    /* 内核态堆栈。*/
12    childregs = task_pt_regs(p);
13    /* 父进程中内核态堆栈中的 pt_regs 复制到子进程的内核态堆栈。*/
14    *childregs = *regs;
15    /* 子进程 pt_regs 结构的 eax 设置为 0，所以子进程 fork() “返回” 值为 0. */
16    childregs->eax = 0;
17    /* 调整子进程的内核态堆栈指针。*/
18    childregs->esp = esp;
19    p->thread.esp = (unsigned long) childregs;
20    p->thread.esp0 = (unsigned long) (childregs+1);
21    /*
22     * 设置子进程的 thread.eip，这样当子进程被调度运行时(见10.3节)，
23     * 就从 ret_from_fork 返回。
24     */
25    p->thread.eip = (unsigned long) ret_from_fork;
26
27    savesegment(gs, p->thread.gs);
28    tsk = current;
29    /* IO 权限位。*/
30    if (unlikely(test_tsk_thread_flag(tsk, TIF_IO_BITMAP))) {
31        p->thread.io_bitmap_ptr = kmempdup(tsk->thread.io_bitmap_ptr,
32                                            IO_BITMAP_BYTES, GFP_KERNEL);
33        if (!p->thread.io_bitmap_ptr) {
34            p->thread.io_bitmap_max = 0;
35            return -ENOMEM;
36        }
37        set_tsk_thread_flag(p, TIF_IO_BITMAP);
38    }
39    /*
40     * Set a new TLS for the child thread?
41     */
42    /* 线程本地存储机制。*/
43    if (clone_flags & CLONE_SETTLS) {
44        struct desc_struct *desc;
45        struct user_desc info;
46        int idx;
```

```
47     err = -EFAULT;
48     if (copy_from_user(&info,
49                         (void __user *)childregs->esi,
50                         sizeof(info)))
51         goto out;
52     err = -EINVAL;
53     if (LDT_empty(&info))
54         goto out;
55
56     idx = info.entry_number;
57     if (idx < GDT_ENTRY_TLS_MIN || idx > GDT_ENTRY_TLS_MAX)
58         goto out;
59
60     desc = p->thread.tls_array + idx - GDT_ENTRY_TLS_MIN;
61     desc->a = LDT_entry_a(&info);
62     desc->b = LDT_entry_b(&info);
63 }
64
65     err = 0;
66 out:
67     if (err && p->thread.io_bitmap_ptr) {
68         kfree(p->thread.io_bitmap_ptr);
69         p->thread.io_bitmap_max = 0;
70     }
71 }
72     return err;
73 }
```

这样，子进程建立的工作就结束了，当调度到这个进程时，它将从 `ret_from_fork` 开始执行，然后跳转到 `syscall_exit`，因此仿佛子进程执行了一次系统调用。现在就要从内核空间返回到用户空间，其用户空间的返回地址保存在内核态堆栈的 `pt_regs` 结构中，这个返回地址和父进程一致。

10.2.4 0号进程的建立

建立进程就是复制父进程对象，可是第一个进程从何而来呢？内核启动时“手工”建立了0号进程，即 `swapper` 进程，这是一个内核态进程，它的页表 `swapper_pg_dir` 和内核态堆栈是在内核启动时建立的（见第4章）。这个进程定义如下：

代码片段 10.19 节自 `include/linux/init_task.h`

```
1 struct task_struct init_task = INIT_TASK(init_task);
```

```
2 EXPORT_SYMBOL(init_task);
3
4 #define INIT_TASK(tsk) \
5 { \
6     .state      = 0,           \
7     .stack      = &init_thread_info,       \
8     .usage      = ATOMIC_INIT(2),        \
9     .flags      = 0,           \
10    .lock_depth = -1,          \
11    .prio       = MAX_PRIO-20,         \
12    .static_prio = MAX_PRIO-20,         \
13    .normal_prio = MAX_PRIO-20,         \
14    .policy     = SCHED_NORMAL,        \
15    .cpus_allowed = CPU_MASK_ALL,      \
16    .mm         = NULL,           \
17    .active_mm   = &init_mm,          \
18    .run_list    = LIST_HEAD_INIT(tsk.run_list),  \
19    .ioprio      = 0,           \
20    .time_slice  = HZ,            \
21    .tasks       = LIST_HEAD_INIT(tsk.tasks),      \
22    .ptrace_children= LIST_HEAD_INIT(tsk.ptrace_children), \
23    .ptrace_list  = LIST_HEAD_INIT(tsk.ptrace_list),  \
24    .real_parent  = &tsk,           \
25    .parent      = &tsk,           \
26    .children    = LIST_HEAD_INIT(tsk.children),    \
27    .sibling     = LIST_HEAD_INIT(tsk.sibling),    \
28    .group_leader = &tsk,           \
29    .group_info   = &init_groups,        \
30    .cap_effective = CAP_INIT_EFF_SET,        \
31    .cap_inheritable = CAP_INIT_INH_SET,        \
32    .cap_permitted = CAP_FULL_SET,           \
33    .keep_capabilities = 0,           \
34    .user        = INIT_USER,          \
35    .comm        = "swapper",          \
36    .thread      = INIT_THREAD,         \
37    .fs          = &init_fs,           \
38    .files       = &init_files,         \
39    .signal      = &init_signals,        \
40    .sighand     = &init_sighand,        \
41    .nsproxy     = &init_nsproxy,        \
42    .pending     = {                 \
43        .list = LIST_HEAD_INIT(tsk.pending.list), \
```

```
44     .signal = {{0}},          \
45     .blocked = {{0}},          \
46     .alloc_lock = __SPIN_LOCK_UNLOCKED(tsk.alloc_lock), \
47     .journal_info = NULL,        \
48     .cpu_timers = INIT_CPU_TIMERS(tsk.cpu_timers),   \
49     .fs_excl = ATOMIC_INIT(0),    \
50     .pi_lock = __SPIN_LOCK_UNLOCKED(tsk.pi_lock),   \
51     .pids = {                  \
52         [PIDTYPE_PID] = INIT_PID_LINK(PIDTYPE_PID), \
53         [PIDTYPE_PGID] = INIT_PID_LINK(PIDTYPE_PGID), \
54         [PIDTYPE_SID] = INIT_PID_LINK(PIDTYPE_SID), \
55     },                      \
56     .dirtyies = INIT_PROP_LOCAL_SINGLE(dirties),      \
57     INIT_TRACE_IRQFLAGS           \
58     INIT_LOCKDEP                \
59 }
```

init_task 的各种进程资源对象都是通过 INIT_xxx 进程初始化的，在 start_kernel() 的最后由 rest_init() 函数调用 kernel_thread() 函数，以 swapper 进程为“模板”建立了 kernel_init 内核进程，之后这个进程会建立 init 进程，执行 /sbin//init 文件，从而把启动过程传递到用户态。而 swapper 进程则执行 cpu_idle() 函数让出 CPU，以后如果没有任何就绪的进程可调度执行，就会调度 swapper 进程，执行 cpu_idle() 函数，这个函数将调用 tick_nohz_stop_sched_tick() 进入 tickless 状态。

10.3 进程切换

当前运行的进程可能主动或被动地放弃 CPU，此时将发生进程切换，切换的主要工作是在 context_switch() 中完成的。

- 当前进程主动进行可能引起阻塞的 IO 操作，例如磁盘 IO 操作，或者当前进程主动在定时器对象上等待，此时当前进程被设置为等待状态，并加入到相关资源的等待队列中，并调用 schedule() 函数让出 CPU。当然，如果一个进程主动通过 exit 系统调用退出，也属于主动让出 CPU。
- 当前进程并没有执行任何可能引起阻塞的操作，由于其时间片到期，或者由于 IO 中断唤醒了在某个 IO 等待队列中的更高优先级的进程。此时当前进程被迫让出 CPU，被放到就绪队列，但是就绪队列中的更高优先级进程将抢占 CPU，由于这种情况通常发生在时钟中断或者其他 IO 中断处理函数中，而中断上下文环境下不能阻塞进程，所以通常中断处理程序通过设置 need_resched 标志（见第 6 章），请求调度，这个调度请求被延迟到中断返回前夕处理。

`context_switch()`是进程上下文切换的主体，这个函数 `prev` 是当前进程的 `task_struct` 指针，而 `next` 是新进程的 `task_struct` 指针。它的工作就是存当前进程上下文，恢复新进程的上下文，同时要设置新进程的页目录，而进程的页目录地址保存在 `mm` 结构中，如果新进程是一个内核态进程，它没有 `mm` 结构，这时就要“借用”前一个进程的 `mm` 结构，使用它的页表作为新进程的页表，由于内核态进程只访问内核态地址空间，而内核态地址空间是所有进程共享的，因此这不会有问题是。

代码片段 10.20 节自 `kernel/sched.c`

```

1 static inline void context_switch(struct rq *rq,
2                                 struct task_struct *prev,
3                                 struct task_struct *next)
4 {
5     struct mm_struct *mm, *oldmm;
6
7     prepare_task_switch(rq, prev, next);
8     mm = next->mm;
9     oldmm = prev->active_mm;
10    /* 虚拟化处理，我们不讨论。*/
11    arch_enter_lazy_cpu_mode();
12    /* 如果新进程没有 mm 结构，就增加当前进程 mm 结构，并增加引用计数。*/
13    if (unlikely(!mm)) {
14        next->active_mm = oldmm;
15        atomic_inc(&oldmm->mm_count);
16        enter_lazy_tlb(oldmm, next);
17    } else
18        /* 加载新进程 mm->pgd 到 CR3 寄存器，这样就切换到了新进程的地址空间。*/
19        switch_mm(oldmm, mm, next);
20
21    if (unlikely(!prev->mm)) {
22        prev->active_mm = NULL;
23        rq->prev_mm = oldmm;
24    }
25 #ifndef __ARCH_WANT_UNLOCKED_CTXSW
26     spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
27 #endif
28    /* Here we just switch the register state and the stack. */
29    switch_to(prev, next, prev);
30    barrier();
31    /*
32     * this_rq must be evaluated again because prev may have moved
33     * CPUs since it called schedule(), thus the 'rq' on its stack
34     * frame will be invalid.

```

```

35     */
36     finish_task_switch(this_rq(), prev);
37 }

```

具体实现进程上下文切换的工作是由 `switch_to()` 完成的，这是一段汇编代码。但是在执行 `switch_to()` 之前，在第19行就切换到了新进程的地址空间，这是由于在进程切换过程中，访问的都是内核地址空间，而内核态地址空间是由所有进程共享的，所以这不会有大问题。`switch_to()` 是一个宏，我们注意到 `switch_to()` 有三个参数，这是为什么呢？由于在 `context_switch()` 函数中，`pre` 和 `next` 都是当前进程的堆栈上的局部变量，现在假设从进程 A 切换到进程 B，此时进程 A 的内核态堆栈中的 `pre` 指向进程 A，而 `next` 指向进程 B，随着系统的运行，经过多次切换后，假设现在要从进程 C 切换到进程 A，当完成切换后，A 从 `switch_to()` 后面接着运行，保存在进程 A 的内核态堆栈上的 `prev` 指向 A，而 `next` 指向 B，但是现在正确的 `prev` 应该是 C（从 C 切换到 A），而 `next` 无关紧要，因为刚刚切换到进程 A，`next` 是哪个进程还不知道，也不需要知道，但是 `pre` 却很重要，例如第36行就访问了 `prev`，因此进程 C 在切换时，把 `prev` 也就是 C 的指针放到 EAX 寄存器中，当切换到进程 A 的内核态堆栈后，就把 EAX 写入到 `prev(switch_to 的第三个参数)` 中，由于这现在已经是进程 A 的堆栈，所以进程 A 内核态堆栈中的局部变量 `prev` 已经指向了 C。因此这里需要这样一种机制，假设现在从进程 A 切换到进程 B，切换之前，在进程 A 的内核态堆栈上 `prev` 指向 A，而 `next` 指向 B，而以前进程 B 被切换出去时，它的内核态堆栈上 `prev` 一定指向某个进程，我们把它计为 X，现在完成切换到 B 时，需要在进程 B 内核态堆栈上的 `prev` 改 X 为 A。而这个转换是通过 EAX 寄存器传递的。理解这个过程，对以后我们分析第36行的 `finish_task_switch()` 函数有重要意义。

代码片段 10.21 节自 `include/asm-x86/system_32.h`

```

1 #define switch_to(prev, next, last) do {
2     unsigned long esi, edi;
3
4     asm volatile(
5         /* 在当前进程的堆栈中保存 EFLAGS。 */
6         "pushfl\n\t"
7         /* 在当前进程的堆栈中保存 EBP。 */
8         "pushl %%ebp\n\t"
9         /* 把 esp 保存到 prev->thread.esp 中。 */
10        "movl %%esp,%0\n\t"
11        /* 把 next->thread.esp 加载到寄存器 ESP 中，现在，切换到了 next 的内核态堆栈。 */
12        "movl %5,%%esp\n\t"
13        /*
14         * 把标号为 1 的地址保存到 prev->thread.eip 中，当下一
15         * 次切换到进程 prev 时，它将从标号为 1 的地址处执行。
16         */

```

```

17     "movl $1f,%1\n\t"
18     /*
19      * 把 next->thread.eip 保存到堆栈中，并且调用
20      * __switch_to 函数，这样 __switch_to 函数的返回地址就是 next->thread.eip。
21      */
22     "pushl %6\n\t"
23     "jmp __switch_to\n"
24     /*
25      * 在第17行把这个地址保存到 prev->thread.eip 中，当再次切换到这个进程时，
26      * 通过第22行，从 next->thread.eip 取出上次保存的 EIP 作为返回地址，
27      * 进程将从这里继续执行。
28      */
29     "1:\n\t"
30     /* 执行到这里时，已经切换到当前进程的内核态堆栈了，恢复第8行保存的 EBP 寄存器。*/
31     "popl %%ebp\n\t"
32     /* 恢复第6行保存的 EFLAGS 寄存器。*/
33     "popfl"
34     : "=m" (prev->thread.esp), "=m" (prev->thread.eip),
35     "=a" (last), "=S" (esi), "=D" (edi)
36     : "m" (next->thread.esp), "m" (next->thread.eip),
37     "2" (prev), "d" (next));
38 } while (0)

```

这段代码使用 gcc 内嵌汇编代码，在第2章我们已经对 gcc 内嵌汇编进行了详细的介绍，但是 switch_to 是一个很好的内嵌汇编实例，因此我们把它当成介绍 gcc 内嵌汇编的一个例子进行详细分析。在第2章我们说过，内嵌汇编的一般格式如下：

指令部： 输出部： 输入部： 损坏部

代码片段10.21中的 switch_to 没有损坏部，在代码中的指令模板 0%表示(prev->thread.esp)，而 1%表示(prev->thread.eip)，依此类推，8%表示(next)。其中输出部的"=a"(last)，表示这个变量需要通过 EAX 输出，如果 gcc 在指令汇编阶段，last 没有被分配给 EAX 寄存器，那么最后 gcc 会插入一条指令，把 EAX 内容写入到 last 中。而在输入部中的"2"(prev)，修饰符"2"的意义类似于指令部中的 2%，表示(prev)和"=a"(last)使用同一个寄存器，这样在 gcc 生成 switch_to 汇编指令前，如果 prev 不是分配在 eax 中，gcc 会在 switch_to 指令的最前面插入 mov prev, eax，这样的指令，同样"d"表示 edx，next 如果不在 edx 中，gcc 会插入 mov next, edx 中。在第23行中跳转到 __switch_to()函数中，它是 fastcall 类型的函数，使用 eax 和 edx 传递前两个参数。这样正好，eax 指向 pre，而 edx 指向 next，而返回值就是 next->thread.eip，所以当 __switch_to()返回时，就转向执行新进程了，多数情况下，新 next->thread.eip 就是上次切换出去时保存的地址，也就是上面的标号 1 的地址，这样新进程就从 switch_to()返回了。

在上面的第23行中跳转到 `_switch_to()` 函数中继续执行，这个函数定义如下：

代码片段 10.22 节自 `arch/x86/kernel/process_32.c`

```
1 struct task_struct fastcall *
2     _switch_to(struct task_struct *prev_p,
3                 struct task_struct *next_p)
4 {
5     struct thread_struct *prev = &prev_p->thread;
6     struct thread_struct *next = &next_p->thread;
7     int cpu = smp_processor_id();
8     struct tss_struct *tss = &per_cpu(init_tss, cpu);
9
10    /* 保存进程 prev_p 的浮点、调试、MMX 寄存器，并加载进程 next_p 的相关寄存器。*/
11    __unlazy_fpu(prev_p);
12    .....
13    /*
14     * 把新进程的内核态堆栈指针设置到 tss 中，由于进程从用户态切换到内核态时，
15     * CPU 自动从 tss 中取内核态堆栈指针，所以现在要把 tss 中的内核态指针置为新
16     * 进程的见第(\ref{chap:hardbase}章)。 */load_esp0(tss, next); /* 设置变量为新进
      程。percpu_current_task*/x86_write_percpu(current_task, next_p); return
      prev_p;
```

当 `_switch_to` 返回时，就切换到了新进程，此时也就完成了进程切换的工作。我们看到 Linux 进程切换时，需要保存和重新加载的寄存器只有堆栈指针 ESP, EBP，指令指针 EIP, EFLAGS，以及 tss 中的 ESP0 等少量的寄存器，却没有保存旧进程的多数其他通用寄存器，因此也不需要加载新进程的其他通用寄存器。这就是 Linux 内核不使用 Intel 提供的 Tss 机制作为进程切换机制的原因，如果使用 Intel 提供的 Tss 机制切换进程，虽然有 CPU 硬件支持，但是 Tss 需要保存过多的寄存器，Linux 内核开发者认为这样可以节省进程切换占用的 CPU 周期，这也是这段代码的巧妙之处。

10.4 进程的退出

10.4.1 do_exit 函数

进程退出时需要销毁自己占用的资源，并且向父进程发送信号，通知父进程生命的结束。这是通过 `exit` 系统调用实现的。这个函数将一去不复返，因为进程对象都要销毁了，调度器不可能再调度这个进程了，所以 `exit` 也就不可能返回了。退出几乎是建立的逆过程，因此在掌握了进程建立的基础上，再来看退出过程就容易多了。

代码片段 10.23 节自 `kernel/exit.c`

```
1 asmlinkage long sys_exit(int error_code)
```

```
2 {
3     do_exit((error_code&0xff)<<8);
4 }
5
6 fastcall NORET_TYPE void do_exit(long code)
7 {
8     struct task_struct *tsk = current;
9     int group_dead;
10
11    profile_task_exit(tsk);
12    WARN_ON(atomic_read(&tsk->fs_excl));
13
14    /* 中断环境中不允许退出。 */
15    if (unlikely(in_interrupt()))
16        panic("Aiee, killing interrupt handler!");
17    if (unlikely(!tsk->pid))
18        panic("Attempted to kill the idle task!");
19
20    /* 如果一个被调试的进程退出，就向父进程（调试器）发送信号。 */
21    if (unlikely(current->ptrace & PT_TRACE_EXIT)) {
22        current->ptrace_message = code;
23        ptrace_notify((PTTRACE_EVENT_EXIT << 8) | SIGTRAP);
24    }
25    /*
26     * 如果当前进程已经是退出状态，那么说明它已经调用过exit一次了，
27     * 这一定是出问题了。
28     */
29    if (unlikely(tsk->flags & PF_EXITING)) {
30        printk(KERN_ALERT "Fixing recursive fault but reboot is needed!\n");
31        tsk->flags |= PF_EXITPIDONE;
32        if (tsk->io_context)
33            exit_io_context();
34        set_current_state(TASK_UNINTERRUPTIBLE);
35        schedule();
36    }
37
38    tsk->flags |= PF_EXITING;
39    .....
40    /* 设置进程退出码。 */
41    tsk->exit_code = code;
42    taskstats_exit(tsk, group_dead);
43}
```

```
44 /* 销毁进程的 mm_struct. */
45 exit_mm(tsk);
46 if (group_dead)
47     acct_process();
48 /* 信号量。*/
49 exit_sem(tsk);
50 /* 文件系统。*/
51 __exit_files(tsk);
52 __exit_fs(tsk);
53 check_stack_usage();
54 /* 进程的 thread_struct 结构。*/
55 exit_thread();
56 cgroup_exit(tsk, 1);
57 exit_keys(tsk);
58
59 if (group_dead && tsk->signal->leader)
60     disassociate_ctty(1);
61 module_put(task_thread_info(tsk)->exec_domain->module);
62 if (tsk->binfo)
63     module_put(tsk->binfo->module);
64 /* 销毁进程在 proc 文件系统中的信息。*/
65 proc_exit_connector(tsk);
66 /* 从进程的亲缘关系的链表中解除当前进程，并发送信号到父进程。*/
67 exit_notify(tsk);
68 .....
69 tsk->flags |= PF_EXITPIDONE;
70 if (tsk->io_context)
71     exit_io_context();
72 if (tsk->splice_pipe)
73     __free_pipe_info(tsk->splice_pipe);
74
75 preempt_disable();
76 /* causes final put_task_struct in finish_task_switch(). */
77 tsk->state = TASK_DEAD;
78 schedule();
79 BUG();
80 /* Avoid "noreturn function does return". */
81 for (;;)
82     cpu_relax(); /* For when BUG is null */
83 }
```

10.4.2 task_struct 结构的删除

在子进程结束后，由于内核在执行进程切换时，需要使用当前进程的内核态堆栈，因此在 do_exit() 函数中，不能删除当前进程的 task_struct 结构及它的内核态堆栈。而仅仅把当前进程的状态设置为 TASK_DEAD，借用当前进程的内核态堆栈，平稳地过渡到其他进程后，调度器会删除状态为 TASK_DEAD 的进程的相关资源。在代码片段 10.23 的第 78 行调用 schedule() 后，进程就切换到了另外一个进程，而当前进程 task_struct 资源将来由调度器删除。为了进一步分析 task_struct 是如何删除的，我们再来看看 context_switch()：

代码片段 10.24 节自 kernel/sched.c

```

1 static inline void context_switch(struct rq *rq,
2                                 struct task_struct *prev,
3                                 struct task_struct *next)
4 {
5     .....
6     switch_to(prev, next, prev);
7     .....
8     finish_task_switch(this_rq(), prev);
9 }
```

假设现在退出的是进程 A，调用 switch_to() 切换到新进程后，新进程从 switch_to() 中的标号 1 返回到 context_switch() 函数，在新进程的内核态堆栈中，prev 已经指向了进程 A。

代码片段 10.25 节自 kernel/sched.c

```

1 static void finish_task_switch(struct rq *rq,
2                                 struct task_struct *prev)
3     __releases(rq->lock)
4 {
5     struct mm_struct *mm = rq->prev_mm;
6     long prev_state;
7
8     rq->prev_mm = NULL;
9
10    /* 取进程 prev 的状态。 */
11    prev_state = prev->state;
12    finish_arch_switch(prev);
13    finish_lock_switch(rq, prev);
14    fire_sched_in_preempt_notifiers(current);
15    if (mm)
16        mmdrop(mm);
17    /*
18     * 如果 prev 进程为 TASK_DEAD 状态，则释放它的 task_struct 和内核态堆栈。

```

```
19     * 这样该进程就彻底“消失”了。
20     */
21     if (unlikely(prev_state == TASK_DEAD)) {
22         kprobe_flush_task(prev);
23         put_task_struct(prev);
24     }
25 }
```

10.4.3 通知父进程

在 do_exit() 函数中 exit_notify() 负责断开当前进程的亲缘关系的相关链表，并向父进程发送信号，

代码片段 10.26 节自 kernel/exit.c

```
1 static void exit_notify(struct task_struct *tsk)
2 {
3     int state;
4     struct task_struct *t;
5     struct pid *pgrp;
6
7     /*
8      * 当进程退出时，如果有一个信号未处理，那么需要查看当前进程是否在某个进程组中，
9      * 如果是就需要唤醒进程组中的其他进程，“委托”它来处理这个信号。什么时候会出现这
10     * 种情况呢？例如：当前进程在执行 do_exit() 过程中，产生了一个中断，而在中断处理过
11     * 程中，向该进程发送了一个信号，当中断返回再次调度到该进程时，就出现了这种情况。
12     */
13     if (signal_pending(tsk) &&
14         !(tsk->signal->flags & SIGNAL_GROUP_EXIT) &&
15         !thread_group_empty(tsk)) {
16         spin_lock_irq(&tsk->sighand->siglock);
17         for (t = next_thread(tsk); t != tsk; t = next_thread(t))
18             if (!signal_pending(t) && !(t->flags & PF_EXITING))
19                 recalcsigpending_and_wake(t);
20         spin_unlock_irq(&tsk->sighand->siglock);
21     }
22
23     /* 调整亲缘关系的相关链表。*/
24     forget_original_parent(tsk);
25     exit_task_namespaces(tsk);
26     write_lock_irq(&tasklist_lock);
27 }
```

```

28     t = tsk->real_parent;
29     .....
30     /*
31      * 向父进程发送信号, exit_signal 是进程结束时需要向父进程发送的信号,
32      * 如果为-1, 表示没有进程指定该进程结束时要发送什么信号, 那么就发送
33      * 默认信号 SIGCHLD, 如果当前进程在一个进程组中, 则要等进程组中最后一
34      * 个进程退出时, 才发送该信号。
35     */
36     if (tsk->exit_signal != -1 && thread_group_empty(tsk)) {
37         int signal = tsk->parent == tsk->real_parent ?
38                         tsk->exit_signal : SIGCHLD;
39         do_notify_parent(tsk, signal);
40     } else if (tsk->ptrace) {
41         do_notify_parent(tsk, SIGCHLD);
42     }
43
44     state = EXIT_ZOMBIE;
45     if (tsk->exit_signal == -1 && likely(!tsk->ptrace))
46         state = EXIT_DEAD;
47     tsk->exit_state = state;
48
49     /* 唤醒在该进程上等待的进程。*/
50     if (thread_group_leader(tsk) &&
51         tsk->signal->notify_count < 0 &&
52         tsk->signal->group_exit_task)
53         wake_up_process(tsk->signal->group_exit_task);
54     write_unlock_irq(&tasklist_lock);
55     /* If the process is dead, release it - nobody will wait for it */
56     if (state == EXIT_DEAD)
57         release_task(tsk);
58 }

```

当前进程要退出了, 那么它的子进程怎么办呢? 子进程的 parent 指向何处呢? forget_original_parent()就是为子进程找一个“继父”, 如当前进程在一个进程组中, 则该组中的下一个进程作为“继父”, 否则由 init 进程充当“继父”。

代码片段 10.27 节自 kernel/exit.c

```

1 static void forget_original_parent(struct task_struct *father)
2 {
3     /* 参数 father 就是当前要退出的进程。*/
4     struct task_struct *p, *n, *reaper = father;
5     struct list_head ptrace_dead;
6

```

```
7  INIT_LIST_HEAD(&ptrace_dead);
8  write_lock_irq(&tasklist_lock);
9  /*
10   * reaper 就是选定的“继父”，首先在该进程组中寻找一个标志不为 PF_EXITING
11   * 的进程作为“”继父。如果 reaper=father，则说明进程组中的所有进程都遍历
12   * 完了，还没有找到一个满足条件的“继父”，于是就调用 task_child_reaper 函数
13   * 获取默认的继父，通常这个“继父”就是 init 进程。
14   */
15 do {
16     reaper = next_thread(reaper);
17     if (reaper == father) {
18         reaper = task_child_reaper(father);
19         break;
20     }
21 } while (reaper->flags & PF_EXITING);
22
23 /* 根据子进程中的兄弟进程链表处理当前进程的所有子进程。*/
24 list_for_each_entry_safe(p, n, &father->children, sibling) {
25     int ptrace;
26     ptrace = p->ptrace;
27     /* if father isn't the real parent, then ptrace must be enabled */
28     BUG_ON(father != p->real_parent && !ptrace);
29     /*
30      * 如果子进程的 real_parent 指向当前进程，说明当前进程是真正的父进程，
31      * 那么直接调整子进程的 real_parent 为它的“继父”。
32      */
33     if (father == p->real_parent) {
34         /* reparent with a reaper, real father it's us */
35         p->real_parent = reaper;
36         reparent_thread(p, father, 0);
37     } else {
38         /*
39          * 如果子进程的 real_parent 不是当前进程，那么说明当前进程是一个调试器
40          * (调试器作为一个临时的父进程)，而子进程处于被调试状态，现在调试器要
41          * 退出了，所以调整子进程的 parent 指向它的 real_parent。如果子进程也要
42          * 退出，就向它真正的父进程发送信号。
43          */
44         /* reparent ptraced task to its real parent */
45         __ptrace_unlink (p);
46         if (p->exit_state == EXIT_ZOMBIE &&
47             p->exit_signal != -1 &&
48             thread_group_empty(p))
```

```

49         do_notify_parent(p, p->exit_signal);
50     }
51     /*
52      * if the ptraced child is a zombie with exit_signal == -1 we
53      * must collect it before we exit, or it will remain zombie
54      * forever since we prevented it from self-reap itself while
55      * it was being traced by us, to be able to see it in wait4.
56      */
57     if (unlikely(ptrace && p->exit_state == EXIT_ZOMBIE &&
58                 p->exit_signal == -1))
59         list_add(&p->ptrace_list, &ptrace_dead);
60 }
61 /*
62  * 如果当前要退出的进程，有某些子进程正在被调试，这些被调试的子进程的 real_parent
63  * 指向自己，而 parent 指向调试器进程，现在要调整它们的 real_parent 指向“继父”进程。
64  */
65 list_for_each_entry_safe(p, n, &father->ptrace_children, ptrace_list) {
66     p->real_parent = reaper;
67     reparent_thread(p, father, 1);
68 }
69 write_unlock_irq(&tasklist_lock);
70 BUG_ON(!list_empty(&father->children));
71 BUG_ON(!list_empty(&father->ptrace_children));
72
73 list_for_each_entry_safe(p, n, &ptrace_dead, ptrace_list) {
74     list_del_init(&p->ptrace_list);
75     release_task(p);
76 }
77 }

```

10.5 do_wait()函数

Linux 中，父进程可以通过 `waitpid()`, `wait4()` 调用等待子进程结束，此时父进程进入等待队列，当子进程结束时，通过 `do_notify_parent()` 向父进程发送 `SIGCHLD` 信号，并唤醒父进程。这些函数的第一个参数 `pid` 指定了等待的目标进程，有以下几种情况。

- 小于-1：等待当前进程的子进程组结束，其进程组 id 为 `pid` 的绝对值。
- -1：等待当前进程的任何一个子进程结束，无论其 `pid` 是什么，只要它是当前进程的子进程即可。

- 0: 等待进程组 ID 和当前进程组 ID 相同的子进程结束，换句话说，就是等待任何与当前进程在同一个组中的子进程结束。
- 大于 0: 等待由 pid 指定的子进程。

另外这个函数的 option 指定了等待的子进程的状态，有以下几种情况。

1. WNOHANG 如果没有满足条件的子进程，就立即返回，而不阻塞当前进程，例如子进程并没有结束，处于就绪状态。
2. WUNTRACED 如果子进程处于 TASK_STOPPED 状态(但是被调试器设置为 TASK_STOPPED 的情况除外)，也不要阻塞当前进程，立即返回¹⁴。
3. WCONTINUED 如果一个处于 TASK_STOPPED 状态的子进程被 SIGCONT 信号唤醒(通常这是调试器通知被调试程序继续执行)，此时父进程也返回。

从这里可以看出，Linux 不区分“进程”和“线程”，都统一使用一个 task_struct，在某些地方上简化了系统设计，但是在某些方面也带来了一定的负面影响，在一定程度上增加了代码的复杂性¹⁵。其中 do_wait()就是一个例子。这个函数是 waitpid()等函数的内核实现。在调用 do_wait()时，如果已经有满足条件的进程处于结束状态，那么当前进程可以直接返回，如果当前进程有满足 pid 指定的子进程，而子进程并没有结束，则当前进程进入等待队列，直到子进程结束时向父进程发送消息唤醒父进程。

代码片段 10.28 节自 kernel/exit.c

```

1 static long do_wait(pid_t pid, int options,
2                     struct siginfo __user *infop,
3                     int __user *stat_addr,
4                     struct rusage __user *ru)
5 {
6     DECLARE_WAITQUEUE(wait, current);
7     struct task_struct *tsk;
8     int flag, retval;
9     int allowed, denied;
10
11    /*
12     * 把等待队列 wait 添加到 wait_chldexit 链表上，do_notify_parent
13     * 函数根据这个链表找到等待的父进程。
14     */
15    add_wait_queue(&current->signal->wait_chldexit, &wait);
16 repeat:
17    /*

```

¹⁴默认是子进程不处于 EXIT_ZOMBIE 状态，当前进程就需要被阻塞，直到子进程进入 EXIT_ZOMBIE 状态。

¹⁵当然，如果区分“进程”和“线程”，代码会更加复杂，但是逻辑有可能会更清晰。

```
18     * 当 flag 被设置时，说明找到了由 pid 指定的子进程，但是这些子进程的
19     * 当前状态，可能不满足退出条件，所以在下面的第191行，当前进程
20     * 需要让出 CPU，直到子进程退出。
21     */
22     flag = 0;
23     allowed = denied = 0;
24
25     /* 把当前进程设置为可中断的等待状态。*/
26     current->state = TASK_INTERRUPTIBLE;
27     read_lock(&tasklist_lock);
28     tsk = current;
29     /*
30      * 如果当前进程是一个轻权进程，那么需要在进程组每一个进程的子进程链表
31      * 中寻找 pid 指定的进程。也就是说，要寻找的不仅仅是“自己的儿子”，还包括
32      * “自己兄弟的儿子”，这个 do-while 循环就处理进程组的这种情况。
33      */
34     do {
35         struct task_struct *p;
36         int ret;
37         /*
38          * 当前进程可能有多个子进程，在这些子进程中寻找 pid 指定的子进程。
39          * 例如当 pid 为 -1 时，就要循环处理每一个子进程。
40          */
41         list_for_each_entry(p, &tsk->children, sibling) {
42             ret = eligible_child(pid, options, p);
43             if (!ret)
44                 continue;
45
46             if (unlikely(ret < 0)) {
47                 denied = ret;
48                 continue;
49             }
50             allowed = 1;
51
52             switch (p->state) {
53             case TASK_TRACED:
54                 flag = 1;
55                 /*
56                  * 要等待的子进程处于被调试状态，此时子进程的信号将发送给
57                  * 调试器进程，如果当前进程就是调试器进程，则 my_ptrace_child
58                  * 返回 1，如果当前进程不是调试器进程就不能进行处理。于是就像代码
```

```
59         * 注释中说的 FALLTHROUGH，继续向下执行16。
60         */
61     if (!my_ptrace_child(p))
62         continue;
63 /*FALLTHROUGH*/
64 case TASK_STOPPED:
65     /*
66      * 如果当前进程没有指定 WUNTRACED 选项，或者子进程不是被当前进程
67      * 调试的进程，就不处理。可以看出只有调试器才会等待 TASK_STOPPED
68      * 状态的子进程。
69      */
70     flag = 1;
71     if (!(options & WUNTRACED) && !my_ptrace_child(p))
72         continue;
73     /*
74      * 子进程已经满足条件，从子进程获取必要的信息，如果成功，
75      * 当前进程就可以跳转到 end 处直接返回了。
76      */
77     retval = wait_task_stopped(p, ret == 2,
78                                (options & WNOWAIT),
79                                infop, stat_addr, ru);
80     /* 重试。 */
81     if (retval == -EAGAIN)
82         goto repeat;
83     /* 成功，父进程可以返回了。 */
84     /* He released the lock. */
85     if (retval != 0)
86         goto end;
87     break;
88 default:
89 // case EXIT_DEAD:
90     /*
91      * 如果子进程的退出状态为 EXIT_DEAD，这个进程早已退出，不需要处理了。
92      * 因此寻找下一个子进程(主要是为了处理像 pid 为 -1 这种情况)。
93      */
94     if (p->exit_state == EXIT_DEAD)
95         continue;
96 // case EXIT_ZOMBIE:
97     /*
98      * 如果子进程的退出状态为 EXIT_ZOMBIE，那么说明它正在等待调用 wait 的
```

¹⁶注意此处没有 break 语句，所以将顺序执行下一个 case 里面的语句，因此代码的作者也通过注释语句“FALLTHROUGH”特别强调了这一点。

```
99      * 父进程来“收拾残局”，那么就省事了，直接获取信息然后返回。
100     */
101     if (p->exit_state == EXIT_ZOMBIE) {
102         /*
103          * 由于 wait_task_zombie()会调用 release_task()释放某些资源，但是
104          * 如果 pid等于-1，并且子进程 p 是一个进程组中的 Group Leader 进程，
105          * 而且这个进程组中还有其他进程，这样虽然 wait()条件可能会满足，
106          * 但是却不能释放这些资源，这种情况下第42行，eligible_child 返回
107          * 回 2，从而跳转到 check_continued 处执行。
108         */
109         if (ret == 2)
110             goto check_continued;
111         if (!likely(options & WEXITED))
112             continue;
113         /* 从子进程中获取信息，如果成功从 end 处返回。*/
114         retval = wait_task_zombie(p, (options & WNOWAIT),
115                                   infop, stat_addr, ru);
116         /* He released the lock. */
117         if (retval != 0)
118             goto end;
119         break;
120     }
121     check_continued:
122     flag = 1;
123     if (!unlikely(options & WCONTINUED))
124         continue;
125     /*
126      * 从子进程中获取信息，如果成功从 end 处返回，
127      * 不会调用 release_task()释放子进程的相关资源。
128      */
129     retval = wait_task_continued(p, (options & WNOWAIT),
130                                 infop, stat_addr, ru);
131     /* He released the lock. */
132     if (retval != 0)
133         goto end;
134     /*
135      * 如果执行 check_continued 处这里，说明 pid 等于-1，
136      * 而当前子进程不满足条件，那么继续处理其他子进程。
137      */
138     break;
139 }
140 }
```

```
141     /*
142      * 如果当前进程建立了一个子进程，之后某个调试器附加到这个子进程上，这
143      * 样子进程处于被调试状态，这时子进程就临时地变成了调试器的“儿子”，但
144      * 是同时子进程被链接到当前进程的 ptrace_children 链表中。如果在第41行
145      * 对子进程链表都遍历完了(flag=0 说明在第41行的循环中，在子
146      * 进程链表中，没有一个满足条件的子进程)，还没有找到合适的子进程，那么
147      * 需要考虑查看 ptrace_children 链表上是否有合适的进程。
148      */
149     if (!flag) {
150         list_for_each_entry(p, &tsk->ptrace_children, ptrace_list) {
151             if (!eligible_child(pid, options, p))
152                 continue;
153             flag = 1;
154             break;
155         }
156     }
157     /* __WNOTHREAD 标志表示不需要到进程组中的其他进程寻找 pid 指定的进程。*/
158     if (options & __WNOTHREAD)
159         break;
160     tsk = next_thread(tsk);
161     BUG_ON(tsk->signal != current->signal);
162 } while (tsk != current);
163
164 read_unlock(&tasklist_lock);
165 /*
166  * 如果上面的 do-while 循环结束，运行到这里，并且 flag 不为 0，说明
167  * 当前进程存在由 pid 指定的子进程，但是子进程的当前状态却不是当
168  * 前进程(父进程)所期待的状态，所以当前进程可能要被阻塞了，直到
169  * 子进程进入这种状态。
170  */
171 if (flag) {
172     retval = 0;
173
174     /* WNOHANG 标志指定在这种情况下，不要阻塞当前进程，立即返回。*/
175     if (options & WNOHANG)
176         goto end;
177     retval = -ERESTARTSYS;
178     /*
179      * 再次检查当前进程是否有信号，这期间子进程可能进入了所期待的
180      * 状态，那么就不需要让出 CPU 了。例如：当前进程执行到第171行，
181      * 某个中断把子进程唤醒，并且 CPU 调度子进程运行，子进程退出，当再次调度到
182      * 当前进程从第171行继续执行时，条件就满足了。

```

```
183     */
184     if (signal_pending(current))
185         goto end;
186     /*
187      * 让出 CPU, 此时其他进程开始执行, 当有信号时, 例如子进程退出,
188      * 发送信号给父进程, 当前进程将被唤醒, 继续执行, 转到 repeat 处
189      * 重新检查, 并从子进程中获取状态, 如果成功, 就从 end 处返回了。
190      */
191     schedule();
192     goto repeat;
193 }
194 retval = -ECHILD;
195 if (unlikely(denied) && !allowed)
196     retval = denied;
197 end:
198 /* 等待结束, 当前进程置为 TASK_RUNNING 状态, 并从等待队列中删除当前进程。*/
199 current->state = TASK_RUNNING;
200 remove_wait_queue(&current->signal->wait_chldexit, &wait);
201
202 /* 复制信息到用户空间。*/
203 if (infop) {
204     if (retval > 0)
205         retval = 0;
206     else {
207         if (!retval)
208             retval = put_user(0, &infop->si_signo);
209         .....
210     }
211 }
212 return retval;
213 }
```

10.6 程序的加载

一个进程执行 fork()之后, 可能会通过 execve(), execlp()等系列的函数来装载一个新的可执行文件, 它的参数是一个可执行文件名。它们的内核实现都是系统调用 sys_execve()。由于 Linux 中存在可执行文件的格式, 例如 a.out, ELF, script 等。对于不同格式的加载过程是不一样的, 因此加载的步骤就是先根据文件头部信息判断出这个文件的类别, 然后调用类别相关的加载函数。为此内核定义了一个装载器对象 linux_binfmt, 其定义如下:

代码片段 10.29 节自 include/linux/binfmts.h

```

1 struct linux_binfmt {
2     struct list_head lh;
3     /* 装载器可以是一个内核模块动态加载，module 为模块指针。*/
4     struct module *module;
5     /* 加载函数。*/
6     int (*load_binary)(struct linux_binprm *, struct pt_regs * regs);
7     /* 库加载函数。*/
8     int (*load_shlib)(struct file *);
9     /*
10      * dump 函数，当一个程序异常退出时，调用该函数生成一个 core 文件，
11      * 它把程序的内存镜像 dump(复制) 到磁盘上，之后可以利用 gdb 等调试
12      * 器对这个文件进行分析。
13      */
14     int (*core_dump)(long signr, struct pt_regs *regs,
15                      struct file *file, unsigned long limit);
16     /* minimal dump size */
17     unsigned long min_coredump;
18     int hasvdso;
19 };

```

每一个装载器模块在初始化时，都会调用 register_binfmt() 函数，注册一个 linux_binfmt 对象，这个函数很简单，就是把新的 linux_binfmt 连接到 lh 链表中，全局变量 formats 指向链表的头部。

另外，由于装载可执行文件之前，需要准备好相关信息，传递给装载函数，但是这些信息太多，不方便一个个通过参数传递，所以内核定义了一个辅助结构 linux_binprm：

代码片段 10.30 节自 include/linux/binfmts.h

```

1 struct linux_binprm{
2     char buf[BINPRM_BUF_SIZE];
3 #ifdef CONFIG_MMU
4     struct vm_area_struct *vma;
5 #else
6     # define MAX_ARG_PAGES 32
7     struct page *page[MAX_ARG_PAGES];
8 #endif
9     struct mm_struct *mm;
10    unsigned long p; /* current top of mem */
11    int sh_bang;
12    struct file * file;
13    int e_uid, e_gid;
14    kernel_cap_t cap_inheritable, cap_permitted;

```

```

15     bool cap_effective;
16     void *security;
17     int argc, envc;
18     char * filename;
19     char * interp;
20
21     unsigned interp_flags;
22     unsigned interp_data;
23     unsigned long loader, exec;
24     unsigned long argv_len;
25 };

```

通过下面的分析，将会明白这个结构的作用，其中 buf 是要执行的文件头部，首先从文件头部读取一少量信息，就可以判断出这个可执行文件的类别，例如脚本文件以 #!开头，而 ELF 和其他格式的文件都有自己的标志，对于 ELF 这一类的可执行文件来说，filename 和 interp 都是一样的，而对于脚本程序来书，filename 指向脚本的名字，而 interp 指向脚本解释器的名字。

`sys_execve()` 最终调用 `do_execve()`：

代码片段 10.31 节自 `fs/exec.c`

```

1 int do_execve(char * filename, char __user * __user * argv,
2                 char __user * __user * envp, struct pt_regs * regs)
3 {
4     struct linux_binprm *bprm;
5     struct file *file;
6     unsigned long env_p;
7     int retval;
8
9     retval = -ENOMEM;
10    bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
11    if (!bprm)
12        goto out_ret;
13
14    file = open_exec(filename);
15    retval = PTR_ERR(file);
16    if (IS_ERR(file))
17        goto out_kfree;
18
19    sched_exec();
20    bprm->file = file;
21    /* 现在还不知道可执行文件的格式，所以 filename 和 interp 都初始化为 filename。 */
22    bprm->filename = filename;

```

```
23     bprm->interp = filename;
24     /* 分配一个 mm_struct. */
25     retval = bprm_mm_init(bprm);
26     if (retval)
27         goto out_file;
28     /* 计算参数的数量。*/
29     bprm->argc = count(argv, MAX_ARG_STRINGS);
30     if ((retval = bprm->argc) < 0)
31         goto out_mm;
32     /* 计算环境变量的数量。*/
33     bprm->envc = count(envp, MAX_ARG_STRINGS);
34     if ((retval = bprm->envc) < 0)
35         goto out_mm;
36
37     /* 提供安全检查，默认为空函数。*/
38     retval = security_bprm_alloc(bprm);
39     if (retval)
40         goto out;
41
42     /* 主要作用是读取文件头部部分内容到 bprm->buf 中。*/
43     retval = prepare_binprm(bprm);
44     if (retval < 0)
45         goto out;
46     /* 把用户态可执行文件名拷贝到内核 bprm 中。*/
47     retval = copy_strings_kernel(1, &bprm->filename, bprm);
48     if (retval < 0)
49         goto out;
50     /* 把用户态的环境变量复制到 bprm 中。*/
51     bprm->exec = bprm->p;
52     retval = copy_strings(bprm->envc, envp, bprm);
53     if (retval < 0)
54         goto out;
55     /* 用户态参数复制到 bprm 中。*/
56     env_p = bprm->p;
57     retval = copy_strings(bprm->argc, argv, bprm);
58     if (retval < 0)
59         goto out;
60     bprm->argv_len = env_p - bprm->p;
61     retval = search_binary_handler(bprm, regs);
62     if (retval >= 0) {
63         /* execve success */
64         free_arg_pages(bprm);
```

```

65     security_bprm_free(bprm);
66     acct_update_integrals(current);
67     kfree(bprm);
68     return retval;
69 }
70 out:
71     .....
72 }
```

do_execve()函数把信息收集到 linux_binprm 结构中之后，就调用 search_binary_handler()函数，这个函数根据 linux_binprm 结构，依次“询问”内核中注册的每一个加载器，你能加载这个文件吗？

代码片段 10.32 节自 fs/exec.c

```

1 int search_binary_handler(struct linux_binprm *bprm, struct pt_regs *regs)
2 {
3     int try, retval;
4     struct linux_binfmt *fmt;
5
6     /* 安全检查，当前默认为空函数。*/
7     retval = security_bprm_check(bprm);
8     if (retval)
9         return retval;
10    /* kernel module loader fixup */
11    /* so we don't try to load run modprobe in kernel space. */
12    set_fs(USER_DS);
13    retval = audit_bprm(bprm);
14    if (retval)
15        return retval;
16
17    retval = -ENOENT;
18    for (try=0; try<2; try++) {
19        read_lock(&binfmt_lock);
20        /* formats 执行加载器对象链表头，依次调用各个加载器的 load_binary() 函数。*/
21        list_for_each_entry(fmt, &formats, lh) {
22            int (*fn)(struct linux_binprm *, struct pt_regs *);
23            fn = fmt->load_binary;
24            if (!fn)
25                continue;
26            if (!try_module_get(fmt->module))
27                continue;
28            read_unlock(&binfmt_lock);
29            retval = fn(bprm, regs);
```

```
30     if (retval >= 0) {
31         put_binfmt(fmt);
32         allow_write_access(bprm->file);
33         if (bprm->file)
34             fput(bprm->file);
35         bprm->file = NULL;
36         current->did_exec = 1;
37         proc_exec_connector(current);
38         return retval;
39     }
40     read_lock(&binfmt_lock);
41     put_binfmt(fmt);
42     if (retval != -ENOEXEC || bprm->mm == NULL)
43         break;
44     if (!bprm->file) {
45         read_unlock(&binfmt_lock);
46         return retval;
47     }
48 }
49 read_unlock(&binfmt_lock);
50 .....
51 }
52 return retval;
53 }
```

依次调用各个加载器的 `load_binary()` 函数，如果返回成功，则说明被成功加载了。在第18行的 `for` 循环中，需要尝试两次，这是因为可能某个加载器模块当前没有被装载，第一次会装载对应的内核模块，第二次就能正确地加载可执行文件了。对于 ELF 格式的可执行文件来说，它的加载器定义如下：

代码片段 10.33 节自 `elf_format`

```
1 static struct linux_binfmt elf_format = {
2     .module    = THIS_MODULE,
3     .load_binary = load_elf_binary,
4     .load_shlib = load_elf_library,
5     .core_dump  = elf_core_dump,
6     .min_coredump = ELF_EXEC_PAGESIZE,
7     .hasvdso   = 1
8 };
```

ELF 格式的文件由 `load_elf_binary()` 函数加载，由于涉及到 ELF 文件格式的细节，以及 ELF 程序链接、加载的相关知识，如果继续分析该函数，要么就是“浅尝辄止”，要么

就把话题扯远了。因此建议感兴趣的读者可以在阅读了相关书籍的基础上再来分析这个函数。但是，就代码分析来说，彻底搞清楚程序编译、链接以及加载的过程之后，在实践中常常可以做到事半功倍。由于在调用 `fork()` 时，子进程拷贝或者共享了父进程的某些资源，现在子进程要“另起炉灶”了，因此 `load_elf_binary()` 还需要完成子进程“独立自主”的某些工作，例如设置子进程 `task_struct` 结构中的 `comm` 成员等，如果必要，还需要分配并复制某些资源。

第11章 调度器

Linux 是一个优先级驱动、抢占式的分时操作系统。在每一个 CPU 上都有一个运行队列，队列中优先级最高的进程总是被调度运行。CPU 的运行时间被划分为基本的时间片分配给就绪进程，依靠时钟中断机制，当时钟中断发生时，就会检查当前运行的进程是否用完了它的时间片，如果是，时钟中断就会请求调度器重新调度，以便使其他进程得到运行。但是某些情况下，一个运行中的进程也可能还没有用完自己的时间片，也没有主动让出 CPU 的“举动”¹，就被剥夺了 CPU 资源，这被称为抢占。

例如：当前运行着一个优先级相对较低的后台程序，此时用户在终端按下键盘，键盘中断处理程序唤醒终端控制台程序，在唤醒操作中，发现了一个更高优先级的进程加入了就绪队列，于是请求重新调度，当中断处理结束时，更高优先级的控制台程序将被投入运行。调度器的算法也在不断的发展之中。本章先简要回顾早期调度器的优缺点，然后再对当前最新的 CFS 调度器进行深入介绍。

11.1 早期的调度器

Linux 把进程分为以下三类：

- 实时进程要求尽快响应，处理紧急任务。
- 交互进程处理和用户的交互，这一类进程大部分时间在等待鼠标、键盘等输入设备，当有输入数据产生时，需要尽快响应，否则用户会感觉到延迟。例如前台的 shell 进程。
- 后台进程不需要和用户交互，在后台运行，因此不需要很快响应，例如后台的 make 进程。

在 2.6 版本之前的内核中，就绪状态的进程都在一个队列中，这样选择下一个优先级最高的进程的代价是 $O(n)^2$ ，在 2.6 版本的内核中，采用了时间开销为 $O(1)$ 的调度器。该调

¹ 例如当前进程退出，进行可阻塞的 IO 操作，或者在定时器等资源对象上面等待。

² 在数据结构与算法中， O 标记法用来表示算法时间复杂度与输入集合的关系。例如 $O(n)$ 表示，算法所需时间与输入集合为线性关系，在这里，当系统中的进程越多的时候，从队列中挑选出下一个最高优先级进程的时间代价就越大， $O(1)$ 表示算法时间复杂度与输入集合没有关系，也就是说，无论输入集合中的元素是增加还是减少，算法的时间花费都是固定的。

度器为每一个 CPU 设置一个运行队列 `rq`, 这个队列中有 140 个优先级链表, 如图 11.1 所示。不同优先级的进程被链接到不同的队列中, 其中 0~99 是保留给实时优先级的进程,

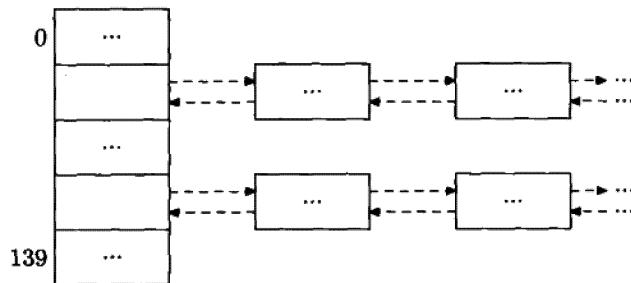


图 11.1 进程优先级队列

100~139 是保留给普通优先级的进程, 每一个就绪进程总是被链接到对应的优先级链表中, 0~139 被称为进程的静态优先级, 数值越大优先级越小。调度器从 0 开始处理, 这里说的实时进程, 仅仅是优先级高于普通进程, 只有处理完 0~99 的链表中的实时进程后, 才轮到普通优先级的进程。

普通进程的静态优先级范围是 100~139, 默认优先级是 120, 可以通过 `nice` 系统调用来改变它的静态优先级, `nice` 参数范围是 -20~19。Linux 内核根据静态优先级为普通优先级的进程划分基本的时间片, 时间片从 5ms~800ms 不等, 优先级越高的进程时间片越大。内核根据一些经验公式调整进程的优先级, 例如对于 IO 消耗型进程给予适当的时间片奖励。因为交互进程通常需要等待输入输出设备的 IO, 这样可以提高交互进程的响应性。另外还根据进程的睡眠时间来调整奖励或者惩罚响应的进程。但是这些经验公式不但增加了代码的复杂性, 而且在某些条件下也是无效的。

例如对于一个后台的数据库查询备份系统, 它需要频繁地等待磁盘 IO, 但不需要提高响应性。另外为一个优先级相对较高的进程分配一个较长的时间片, 比如 400ms, 在这 40ms 中, 低优先级的进程得不到调度, 对于低优先级的进程来说, 不太公平。为此提出了 CFS 调度器, CFS 的全称是 Completely Fair Scheduling, 即完全公平的调度器。现实中理想的完全公平的调度是不存在的, CFS 仍然是一个相对公平的调度器。对于前面的例子来说, CFS 的公平之处就在于, 它不会把 400ms 分配给一个高优先级的进程, 而让低优先级的进程在这段时间内得不到调度, CFS 把 400ms 划分为更小的时间片, 例如 20ms, 然后根据优先级按比例分配这些时间片给不同的进程, 因此 CFS 在一定程度上提高了调度的公平性。另外 CFS 在动态优先级的处理上, 放弃了以往的经验公式, 不需要跟踪进程的睡眠时间, 同时实现了简单有效的动态优先级方案。另外 CFS 还能够根据用户来分配时间片, 例如系统中登录了 N 个用户, 调度器首先“公平”地为这些用户分配时间片, 然后再调度相应用户的进程。

Linux 的调度器经过了长期的发展，但是到目前为止，还没有任何一个调度器能够适用所有的应用，无论从理论上的分析，还是从大量的实际测试结果来看，CFS 无疑是目前最为简单有效的，经过大量的论证之后，CFS 已经被内核采纳为主流的调度器。由于其他的调度器逐渐被 CFS 取代，因此笔者不想花费大量的篇幅深入讨论这些调度器，对它们进行简要的回顾，是为了说明 CFS 发展的缘由，如果想深入了解以前的调度器，请参考《Understanding the Linux Kernel》的相关章节，本章剩余章节将对 CFS 进行深入的探讨。

11.2 CFS 调度器的虚拟时钟

CFS 把进程的优先级转化为权重(weight)，每个进程的优先级对应一个权重，优先级越高，权重越大。每一个 CPU 有一个 `cfs_rq` 对象，它维护着就绪队列中进程的总权重。由于普通进程的静态优先级为 100~139，进程描述符 `task_struct` 中的 `prio` 保存着进程的优先级，通过 `nice` 可以调整进程的静态优先级(-20~19)，CFS 把静态优先级转化为对应的权重，因此第一步需要解决的问题就是静态优先级到 `nice` 的转化。为此内核定义了以下数据：

代码片段 11.1 节自 `include/linux/sched.h`

```
1 /* 实时进程的最大优先级。 */
2 #define MAX_USER_RT_PRIO 100
3 #define MAX_RT_PRIO    MAX_USER_RT_PRIO
4
5 /* 最大优先级。 */
6 #define MAX_PRIO      (MAX_RT_PRIO + 40)
7 /* 普通进程的默认优先级(120)。 */
8 #define DEFAULT_PRIO   (MAX_RT_PRIO + 20)
9 /* nice 到静态优先级的转换，-20 对应 100，19 对应 139。 */
10 #define NICE_TO_PRIO(nice)  (MAX_RT_PRIO + (nice) + 20)
11 /* 静态优先级到 nice 的转换，100 对应-20，139 对应 19。 */
12 #define PRIO_TO_NICE(prio) ((prio) - MAX_RT_PRIO - 20)
13 /*
14 * 静态优先级到权重的转换，nice 值-20 的权重为 88761,
15 * 19 的权重为 15，权重越大优先级越高。
16 */
17 static const int prio_to_weight[40] = {
18     /* -20 */ 88761, 71755, 56483, 46273, 36291,
19     /* -15 */ 29154, 23254, 18705, 14949, 11916,
20     /* -10 */ 9548, 7620, 6100, 4904, 3906,
21     /* -5 */ 3121, 2501, 1991, 1586, 1277,
22     /* 0 */ 1024, 820, 655, 526, 423,
23     /* 5 */ 335, 272, 215, 172, 137,
24     /* 10 */ 110, 87, 70, 56, 45,
```

```
25 /* 15 */ 36,      29,      23,      18,      15,
26 };
```

CFS 把一个时间片，按照权重比例分配给各个进程，其中内核默认时间片的大小根据式11.1计算：

$$\text{slice} = \begin{cases} 20\text{ms} & (\text{NR} \leq 5) \\ 20\text{ms} * \frac{\text{NR}}{5} & (\text{NR} > 5) \end{cases} \quad (11.1)$$

其中 NR 表示就绪队列中的进程总数。时间片 slice 应该根据权重按比例分配，假设有 n 个进程，那么进程 i 应该占用的时间片 T_i 为：

$$T_i = \text{slice} * \frac{W_i}{\sum_{i=0}^n W_i} \quad (11.2)$$

其中 T_i 就是进程 i 应该占用的时间片， W_i 表示进程 i 的权重， $\sum_{i=0}^n W_i$ 表示就绪队列中的进程权重的总和。在理想情况下，进程 i 应该在执行 T_i ms 后，立刻调度另外一个进程运行。例如有进程 A 和 B， $W_a=2$, $W_b=5$ ，现根据式11.1可以知道默认的 slice 为 20ms ($\text{NR}<5$)，于是进程 B 首先执行 $(20 * \frac{5}{7})\text{ms}$ (这里总权重为 7, 进程 B 的权重为 5.)，然后进程 A 执行 $(20 * \frac{2}{7})\text{ms}$ ，接着下一轮的 slice 分配。然而这仅仅是理想的 CPU 分配方式，在现实中，周期性的时钟中断不能保证恰好在各个进程的执行时间 T_i 的边缘立刻发出，因此相对于理想状况来说，在一个 slice 中，总会有进程多执行了一小段时间，而其他的一些进程少执行了一小段时间。这样就需要在下一轮的 slice 分配过程中，“奖励”少执行的进程，而“惩罚”多执行的进程。为了记住历史运行时间，提出了虚拟时钟的概念。

cfs_rq 对象维护着一个虚拟时钟，时钟前进的步伐和总权重成反比，也就是说，权重越大，虚拟时钟前进的步伐越慢。同时每一个进程也有一个虚拟时钟，它记录着进程已经占用 CPU 的虚拟时间，它前进的步伐和本进程的权重成反比。当一个进程受到了完全公平的调度待遇时，它的虚拟时钟应该和调度对象 cfs_rq 的虚拟时钟一致，如果某个进程的虚拟时钟快于 cfs_rq 的虚拟时钟，则在将来的调度过程中需要接受“惩罚”，而对于虚拟时钟比 cfs_rq 的虚拟时钟慢的，则需要适当的“奖励”。CFS 调度的原则就是：总是把 CPU 分配给虚拟时钟最慢的那个进程。为了更好地理解这个结论，我们先看一个简化的例子，假设进程 A 和 B 的权重分别为 2 和 5，那么 CFS 的调度序列如表11.1所示。

可以看到，对于高优先级的进程，由于其权重比较大，因此它的虚拟时钟是按比例缩小的，为了更加形象地说明这个原理，我们假设实际的时钟单元为 10ms，根据表11.1，进程 B 实际运行了 10ms，但是其权重为 5，CFS 认为它运行了 $1/5$ 个虚拟时钟单元，而对于进程 A，实际运行 10ms，CFS 认为它运行了 $1/2$ 个虚拟时钟单元。因此，对于同一个实际时钟单元(10ms)，高优先级的时钟相对缩小，而低优先级进程的时钟被相对放大($1/2 > 1/5$)。我们假设 10ms 就是时钟中断周期，在本例中，根据式11.1，一个时间分配单元 20ms 结束时，A 和 B 并没有享受到完全公平的待遇。但是根据 CFS 总是把 CPU 分配给虚拟时钟最慢的进程，这样就绪队列中最慢的时钟，因受到“奖励”而向 cfs 的虚拟时钟逼近，而快的时钟因受“惩罚”而停滞。随着时间的推移，所有的虚拟时钟逐渐相等，这样所有进

表 11.1 CFS 的虚拟时钟调度策略

实际时钟	cfs 的虚拟时钟	进程 A 的虚拟时钟	进程 B 的虚拟时钟	备注
0	0	0	0	选择进程 B 运行
1	1/7	0	1/5	1/5 > 1/7, B 受到“惩罚”，调度 A
2	2/7	1/2	1/5	1/2 > 2/7, A 受到“惩罚”，调度 B
3	3/7	1/2	2/5	2/5 < 3/7, “奖励” B, B 继续
4	4/7	1/2	3/5	3/5 > 4/7, “惩罚” B, 调度 A
5	5/7	1	3/5	1 > 5/7, “惩罚” A, 调度 B
6	6/7	1	4/5	4/5 < 6/7, “奖励” B, B 继续
7	1	1	1	1 = 1 = 1, “完全公平”

程就受到完全公平的待遇。在下一个周期，各个虚拟时钟又将逐渐拉开距离，然后再次相等。因此整体上是完全公平的。

CFS 的使用红黑树来组织就绪队列，而树的键值就是进程的虚拟时间，红黑树的最左边的进程的虚拟时钟最慢。CFS 中的实际时钟以纳秒为单位，而虚拟时钟根据下面的公式计算：

$$VT = T * \frac{1024}{W} \quad (11.3)$$

这里 T 表示实际时钟的时间， W 是进程优先级对应的权重，而 1024 是静态优先级为 120(nice 值为 0) 对应的权重，当静态优先级为 120 时，其比例因子为 1，此时虚拟时钟和实际时钟一致，这时由于内核中默认的静态优先级为 120，这是最普遍的情况，可以避免对虚拟时钟的计算。当 $W > 1024$ 时(进程静态优先级大于 120)， $\frac{1024}{W} < 1$ ，当该进程实际执行了 T_{ns} 后，记录它的虚拟执行时间却小于 T 。因此当进入完全公平待遇时，尽管所有的虚拟时钟相等，但是优先级高的进程执行的实际时间相对较大。由于实际时钟的单位是 ns，为了防止溢出，虚拟时钟变量的位宽是 64 位，而除法运算的消耗相对较大，为了优化代码，内核对公式 11.3 进行等价变换：

$$VT = 1024 * T * \frac{2^{32}}{W} * \frac{1}{2^{32}} \quad (11.4)$$

由于静态优先级对应的权重只有 40 种，因此 $\frac{2^{32}}{W}$ 可以事先计算好，而乘以 $\frac{1}{2^{32}}$ 可以转化为右移 32 位。为此内核定义了一个数组，保存事先计算的 $\frac{2^{32}}{W}$ 的结果：

代码片段 11.2 节自 kernel/sched.c

```

1 static const u32 prio_to_wmult[40] = {
2     /* -20 */    48388,      59856,      76040,      92818,      118348,
3     /* -15 */    147320,     184698,     229616,     287308,     360437,
4     /* -10 */    449829,     563644,     704093,     875809,     1099582,
5     /* -5 */     1376151,    1717300,    2157191,    2708050,    3363326,
```

```

6  /* 0 */ 4194304, 5237765, 6557202, 8165337, 10153587,
7  /* 5 */ 12820798, 15790321, 19976592, 24970740, 31350126,
8  /* 10 */ 39045157, 49367440, 61356676, 76695844, 95443717,
9  /* 15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
10 };

```

每一个进程有一个 `load_weight` 结构，专门用来保存当前进程优先级对应的权重 W，以及 $\frac{2^{32}}{W}$ 的值，其定义如下：

代码片段 11.3 节自 `include/linux/sched.h`

```

1 struct load_weight {
2     unsigned long weight;
3     unsigned long inv_weight;
4 };

```

每当通过 nice 系统调用改变进程的权重 `weight` 时，`inv_weight` 也随之改变。

最后，函数 `calc_delta_fair()` 就是根据上面的原理，计算虚拟时钟，其定义如下：

代码片段 11.4 节自 `kernel/sched.c`

```

1 static inline unsigned long
2 calc_delta_fair(unsigned long delta_exec, struct load_weight *lw)
3 {
4     return calc_delta_mine(delta_exec, NICE_0_LOAD, lw);
5 }

```

每当进程被调度运行时，CFS 记录了进程的起始运行时间，每次时钟中断，CFS 根据当前系统时间，计算出进程已经运行的实际时间 `delta_exec`，然后调用 `calc_delta_fair` 函数，根据进程的 `load_weight` 结构计算虚拟时钟的时间。其中 `NICE_0_LOAD` 被定位为 1024，这个函数以 1024 为基准，调用 `calc_delta_mine()` 根据式 11.4 计算虚拟时间：

代码片段 11.5 节自 `kernel/sched.c`

```

1 # define WMULT_CONST (~0UL)
2 #define WMULT_SHIFT 32
3
4 define SRR(x, y) (((x) + (1UL << ((y) - 1))) >> (y))
5
6 /* delta_exec 是实际运行时间，weight 是基准因子 1024，lw 是进程的 load_weight 结构。*/
7 static unsigned long
8 calc_delta_mine(unsigned long delta_exec,
9                 unsigned long weight,
10                struct load_weight *lw)
11 {
12     u64 tmp;

```

```

13  /*
14   * 万一 inv_weight 没有初始化好，就根据 weight 计算 inv_weight 的值，
15   * 这种情况很少发生。这个式子本应该为 (WMULT_CONST + lw->weight/2) / lw->weight3，
16   * 但是由于 WMULT_CONST 已经是 32 位数的最大值，再加就要溢出了，故做等价变换。
17   */
18  if (unlikely(!lw->inv_weight))
19      lw->inv_weight = (WMULT_CONST - lw->weight/2) / lw->weight + 1;
20  /* 基准因子 weight 为 1024。相当于式11.4中的 1024*T。*/
21  tmp = (u64)delta_exec * weight;
22  /*
23   * 由于 tmp 可能超过 32 位，这时再乘以一个 32 位的数，可能超出 64 位，所以在
24   * 这种情况下，先把 tmp 右移 16 位，然后再乘以 inv_weight，最后再右移 16 位。
25   * 这也是右移 32 位的等价变换。否则直接计算。
26   */
27  if (unlikely(tmp > WMULT_CONST))
28      tmp = SRR(SRR(tmp, WMULT_SHIFT/2) * lw->inv_weight, WMULT_SHIFT/2);
29  else
30      tmp = SRR(tmp * lw->inv_weight, WMULT_SHIFT);
31
32  return (unsigned long)min(tmp, (u64)(unsigned long)LONG_MAX);
33 }

```

这段代码虽然很简短，但是却相当精巧，希望读者能够用心体会。

11.3 CFS 调度器的基本管理结构

CFS 为每一个 CPU 维护一个调度队列 `cfs_rq`，其定义如下：

代码片段 11.6 节自 `kernel/sched.c`

```

1 struct cfs_rq {
2     /* 当前就绪队列进程的总权重。*/
3     struct load_weight load;
4     /* 当前就绪队列中的总进程数。*/
5     unsigned long nr_running;
6     /* 实时时钟。*/
7     u64 exec_clock;
8     /* 当前队列中运行时间最小的虚拟时间。*/
9     u64 min_vruntime;
10
11    struct rb_root tasks_timeline;

```

³为什么不是 $\frac{WMULT_CONST}{lw->weight}$ 呢？这是由于 `inv_weight` 是 $\frac{2^{32}}{weight}$ 的近似值，读者动手计算便能体会其中的缘由。

```
12  /* 红黑树最左子树节点。*/
13  struct rb_node *rb_leftmost;
14  struct rb_node *rb_load_balance_curr;
15  /*
16   * 'curr' points to currently running entity on this cfs_rq.
17   * It is set to NULL otherwise (i.e when none are currently running).
18   */
19  struct sched_entity *curr;
20  unsigned long nr_spread_over;
21 #ifdef CONFIG_FAIR_GROUP_SCHED
22  .....
23#endif
24 };
```

每一个进程的 task_struct 中，有一个 sched_entity 结构，而 csf_rq 中的 curr 指向当前运行中的进程的 sched_entity 结构：

代码片段 11.7 节自 include/linux/sched.h

```
1 struct sched_entity {
2     /* 当前进程的权重。*/
3     struct load_weight load;
4     /* 红黑树的节点。*/
5     struct rb_node run_node;
6     /* 当前进程在就绪队列中，on_rq 被设置为 1. */
7     unsigned int on_rq;
8
9     /* 进程开始执行的实际时间。*/
10    u64 exec_start;
11    /* 进程投入运行的总实际时间。*/
12    u64 sum_exec_runtime;
13    /* 进程总共执行的虚拟时间。*/
14    u64 vruntime;
15    /* 进程前一次投入运行的总实际时间。*/
16    u64 prev_sum_exec_runtime;
17 #ifdef CONFIG_SCHEDSTATS
18    u64 wait_start;
19    u64 wait_max;
20    .....
21#endif
22
23 #ifdef CONFIG_FAIR_GROUP_SCHED
24    struct sched_entity *parent;
25    /* rq on which this entity is (to be) queued: */
```

```

26 struct cfs_rq *cfs_rq;
27 /* rq "owned" by this entity/group: */
28 struct cfs_rq *my_rq;
29 #endif
30 };

```

在这个结构中 `exec_start`, `sum_exec_runtime` 和 `prev_sum_exec_runtime` 都是实际时间, 但是却有着不同的意义。它们之间的区别和联系, 可以用下面的简化代码来说明。

代码片段 11.8 进程计时操作

```

1 /* 每当加载一个进程开始执行时, 执行以下操作。*/
2 se->exec_start = now;
3 se->prev_sum_runtime = se->sum_exec_runtime.
4
5 /* 在每一次时钟中断中, 执行以下操作。*/
6 delta_exec = now - se->exec_start;
7 se->sum_exec_runtime += delta_exec;
8 se->exec_start = now;

```

其中 `exec_start` 记录进程开始运行的时间, 每次时钟中断利用当前时间减去 `exec_start`, 就得到了进程运行了多长时间, 同时再把当前时间赋值给 `exec_start`⁴。但是由于一个进程被调度运行后, 可能要经过一个或者多个时钟中断才用完它的时间片, 因此 `prev_sum_runtime` 记录的是当前运行的进程占用 CPU 的总时间, 不包括本次调入 CPU 运行的时间, 而 `sum_exec_runtime` 记录的是当前运行的进程占用的 CPU 的总时间, 包括本次调入 CPU 运行的时间, 这样在进程被换出前的任何时刻, 都可以通过它们的差值, 计算出一个进程本次调入运行的时间。

11.4 CFS 调度器对象

新的内核对调度器采用了模块化的设计, 每一个调度器对象由一个 `sched_class` 结构表示, 其中 CFS 调度器对象定义如下:

代码片段 11.9 节自 `kernel/sched_fair.c`

```

1 static const struct sched_class fair_sched_class = {
2     /* 指向下一个调度器对象。*/
3     .next      = &idle_sched_class,
4     /* 把一个进程加入就绪队列。*/
5     .enqueue_task = enqueue_task_fair,
6     /* 把一个进程从就绪队列中移除。*/

```

⁴当然, 在其他情况下也有类似的统计, 例如进程被阻塞时。

```

7   .dequeue_task    = dequeue_task_fair,
8
9   /* 当前进程让出 CPU, 从就绪队列中选择其他进程投入运行。*/
10  .yield_task     = yield_task_fair,
11  /* 检查就绪队列中是否存在一个优先级高于当前进程的进程。*/
12  .check_preempt_curr = check_preempt_wakeup,
13  /* 从就绪队列中选择一个最高优先级的进程。*/
14  .pick_next_task  = pick_next_task_fair,
15  /* 把当前进程换出前, 更新统计信息。*/
16  .put_prev_task   = put_prev_task_fair,
17
18 #ifdef CONFIG_SMP
19  /* 多处理器就绪队列负载均衡。*/
20  .load_balance    = load_balance_fair,
21  /* 把一个进程从其他处理器的就绪队列转移到当前处理器。*/
22  .move_one_task   = move_one_task_fair,
23 #endif
24  .set_curr_task   = set_curr_task_fair,
25  /* 调度器 tick 函数, 由时钟中断调用。*/
26  .task_tick       = task_tick_fair,
27  /* 把一个新建立的进程加入到队列中。*/
28  .task_new        = task_new_fair,
29 };

```

内核中的多个调度器对象通过 next 链接, 当前除了 CFS 调度器对象 fair_sched_class 之外, 还存在 rt_sched_class 和 idle_sched_class, 前者是实时进程的调度器对象, 后者是 idle 进程调度器对象。

11.5 CFS 调度操作

11.5.1 update_curr()函数

很多函数都会调用 update_curr(), 它用来更新相关的统计信息。该函数定义如下:

代码片段 11.10 节自 kernel/sched_fair.c

```

1 static void update_curr(struct cfs_rq *cfs_rq)
2 {
3     /* 当前进程对应的 sched_entity 结构。*/
4     struct sched_entity *curr = cfs_rq->curr;
5     u64 now = rq_of(cfs_rq)->clock;
6     unsigned long delta_exec;

```

```
7
8     if (unlikely(!curr))
9         return;
10    /* 进程已经执行的时间。*/
11    delta_exec = (unsigned long)(now - curr->exec_start);
12    __update_curr(cfs_rq, curr, delta_exec);
13    /* 把 exec_start 设置为当前时间。*/
14    curr->exec_start = now;
15    if (entity_is_task(curr)) {
16        struct task_struct *curtask = task_of(curr);
17        /* 分组调度支持，我们不讨论这种情况。*/
18        cpuacct_charge(curtask, delta_exec);
19    }
20 }
21
22 static inline void
23 __update_curr(struct cfs_rq *cfs_rq,
24                 struct sched_entity *curr,
25                 unsigned long delta_exec)
26 {
27     unsigned long delta_exec_weighted;
28     u64 vruntime;
29
30     /* 设置 exec_max。*/
31     schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));
32     /* 统计进程运行的总时间。*/
33     curr->sum_exec_runtime += delta_exec;
34     schedstat_add(cfs_rq, exec_clock, delta_exec);
35     delta_exec_weighted = delta_exec;
36     /*
37      * 如果当前进程的静态优先级为 120，则虚拟时钟和时间时钟是 1:1 的关系，否则调
38      * 用 calc_delta_fair 函数计算实际时间 delta_exec 对应的虚拟时间(见11.2节)。
39      */
40     if (unlikely(curr->load.weight != NICE_0_LOAD)) {
41         delta_exec_weighted=calc_delta_fair(delta_exec_weighted, &curr->load);
42     }
43     curr->vruntime += delta_exec_weighted;
44
45     if (first_fair(cfs_rq)) {
46         vruntime = min_vruntime(curr->vruntime,
47                               __pick_next_entity(cfs_rq)->vruntime);
48     } else
```

```

49     vruntime = curr->vruntime;
50     cfs_rq->min_vruntime = max_vruntime(cfs_rq->min_vruntime, vruntime);
51 }

```

由于 cfs_rq 中的 min_vruntime 成员总是保存最慢的虚拟时钟，当前进程因为虚拟时钟最慢而被调度运行，但是当它执行一段时间后，它的虚拟时钟又前进了，此时它的虚拟时钟可能不是最慢的了，所以在第45行，如果就绪队列中还有其他进程，就比较当前进程的虚拟时钟和红黑树中的最左节点的虚拟时钟，取它们的最小值。

11.5.2 scheduler_tick()函数

scheduler_tick()是调度器的核心，每一次时钟中断都会调用这个函数，它需要调用 update_curr()函数更新当前进程的信息，之后还要检查是否需要把下一个时间片分配给当前进程，如果不能，就需要请求调度，但是因为中断上下文环境中不能发生进程切换，因此这个调度请求被延迟到中断返回(见第6章)。

代码片段 11.11 节自 kernel/sched.c

```

1 void scheduler_tick(void)
2 {
3     int cpu = smp_processor_id();
4     struct rq *rq = cpu_rq(cpu);
5     struct task_struct *curr = rq->curr;
6     u64 next_tick = rq->tick_timestamp + TICK_NSEC;
7
8     spin_lock(&rq->lock);
9     __update_rq_clock(rq);
10    /*
11     * Let rq->clock advance by at least TICK_NSEC:
12     */
13    if (unlikely(rq->clock < next_tick))
14        rq->clock = next_tick;
15    rq->tick_timestamp = rq->clock;
16    update_cpu_load(rq);
17
18    /* CFS 的 task_tick() 函数为 task_tick_fair()。 */
19    if (curr != rq->idle) /* FIXME: needed? */
20        curr->sched_class->task_tick(rq, curr);
21    spin_unlock(&rq->lock);
22
23 #ifdef CONFIG_SMP
24     rq->idle_at_tick = idle_cpu(cpu);

```

```
25     trigger_load_balance(rq, cpu);
26 #endif
27 }
28
29 static void task_tick_fair(struct rq *rq, struct task_struct *curr)
30 {
31     struct cfs_rq *cfs_rq;
32     struct sched_entity *se = &curr->se;
33
34     /* 如果不考虑分组调度，就只处理当前进程。*/
35     for_each_sched_entity(se) {
36         cfs_rq = cfs_rq_of(se);
37         entity_tick(cfs_rq, se);
38     }
39 }
40
41 static void entity_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
42 {
43     /*
44      * Update run-time statistics of the 'current'.
45      */
46     update_curr(cfs_rq);
47     /* 如果就绪队列中的进程大于 1，需要检查当前进程的时间片是否到期。*/
48     if (cfs_rq->nr_running > 1 || !sched_feat(WAKEUP_PREEMPT))
49         check_preempt_tick(cfs_rq, curr);
50 }
51
52 static void
53 check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
54 {
55     unsigned long ideal_runtime, delta_exec;
56
57     ideal_runtime = sched_slice(cfs_rq, curr);
58     delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
59     if (delta_exec > ideal_runtime)
60         resched_task(rq_of(cfs_rq)->curr);
61 }
```

check_preempt_tick 函数调用 sched_slice() 计算理想状况下应该分配的时间片。delta_exec 表示本次调度当前进程已经执行的时间，如果它比理想时间片 ideal_runtime 还要大，则说明当前进程的虚拟时钟将要超过 cfs_rq 的虚拟时钟了。因此调用 resched_task() 函数为当前进程设置 TIF_NEED_RESCHED 标志，请求调度。其中 sched_slice() 定义如下：

代码片段 11.12 节自 kernel/sched_fair.c

```

1 static u64 sched_slice(struct cfs_rq *cfs_rq, struct sched_entity *se)
2 {
3     u64 slice = __sched_period(cfs_rq->nr_running);
4     slice *= se->load.weight;
5     do_div(slice, cfs_rq->load.weight);
6     return slice;
7 }
8
9 static u64 __sched_period(unsigned long nr_running)
10 {
11     u64 period = sysctl_sched_latency;
12     unsigned long nr_latency = sched_nr_latency;
13
14     if (unlikely(nr_running > nr_latency)) {
15         period *= nr_running;
16         do_div(period, nr_latency);
17     }
18     return period;
19 }

```

`sched_slice()`首先调用 `__sched_period` 根据公式 11.1 获取总的 slice，然后根据公式 11.2 按照权重比例计算当前进程应该获取的执行时间。

11.5.3 put_prev_task_fair() 函数

当调度请求被受理，或进程主动让出 CPU 时，会调用 `schedule()` 函数，`scheduler()` 先调用 `put_prev_task()` 把当前进程加入到就绪队列，然后调用 `pick_next_task()` 从就绪队列中选择下一个进程，最后切换到该进程。

代码片段 11.13 节自 kernel/sched_fair.c

```

1 static void put_prev_task_fair(struct rq *rq, struct task_struct *prev)
2 {
3     struct sched_entity *se = &prev->se;
4     struct cfs_rq *cfs_rq;
5
6     for_each_sched_entity(se) {
7         cfs_rq = cfs_rq_of(se);
8         put_prev_entity(cfs_rq, se);
9     }
10 }
11

```

```

12 static void put_prev_entity(struct cfs_rq *cfs_rq,
13                             struct sched_entity *prev)
14 {
15     /* 更新统计信息。*/
16     if (prev->on_rq)
17         update_curr(cfs_rq);
18     check_spread(cfs_rq, prev);
19     if (prev->on_rq) {
20         update_stats_wait_start(cfs_rq, prev);
21         /* Put 'current' back into the tree. */
22         __enqueue_entity(cfs_rq, prev);
23     }
24     cfs_rq->curr = NULL;
25 }
```

其中 `__enqueue_entity()` 函数就是根据进程的 `vruntime`, 把当前进程插入红黑树。

代码片段 11.14 节自 `kernel/sched_fair.c`

```

1 static inline s64 entity_key(struct cfs_rq *cfs_rq, struct sched_entity *se
2 )
3 {
4     return se->vruntime - cfs_rq->min_vruntime;
5 }
6
7 static void __enqueue_entity(struct cfs_rq *cfs_rq,
8                             struct sched_entity *se)
9 {
10    struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
11    struct rb_node *parent = NULL;
12    struct sched_entity *entry;
13    s64 key = entity_key(cfs_rq, se);
14    int leftmost = 1;
15
16    /* vruntime 越小的进程越靠左。*/
17    while (*link) {
18        parent = *link;
19        entry = rb_entry(parent, struct sched_entity, run_node);
20
21        if (key < entity_key(cfs_rq, entry)) {
22            link = &parent->rb_left;
23        } else {
24            link = &parent->rb_right;
25            leftmost = 0;
```

```

25     }
26 }
27
28 if (leftmost)
29     cfs_rq->rb_leftmost = &se->run_node;
30 rb_link_node(&se->run_node, parent, link);
31 rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
32 }

```

11.5.4 pick_next_task()函数

利用 put_prev_task 函数把当前进程加入就绪队列后, scheduler 函数调用 pick_next_task()从就绪队列中选择下一个进程, 然后切换到该进程。

代码片段 11.15 节自 kernel/sched.c

```

1 static inline struct task_struct *
2 pick_next_task(struct rq *rq, struct task_struct *prev)
3 {
4     const struct sched_class *class;
5     struct task_struct *p;
6
7     /*
8      * Optimization: we know that if all tasks are in
9      * the fair class we can call that function directly:
10     */
11    if (likely(rq->nr_running == rq->cfs.nr_running)) {
12        p = fair_sched_class.pick_next_task(rq);
13        if (likely(p))
14            return p;
15    }
16    class = sched_class_highest;
17    for ( ; ; ) {
18        p = class->pick_next_task(rq);
19        if (p)
20            return p;
21        class = class->next;
22    }
23 }

```

这里做了个优化, 在多数情况下, 内核只有一个调度器对象 CFS, 所以如果就绪队列总进程数等于 CFS 的就绪队列中的总进程数, 说明全部进程都在 CFS 的调度队列中, 此时

直接调用 CFS 调度对象的 pick_next_task() 函数。

代码片段 11.16 节自 kernel/sched_fair.c

```

1 static struct task_struct *pick_next_task_fair(struct rq *rq)
2 {
3     struct cfs_rq *cfs_rq = &rq->cfs;
4     struct sched_entity *se;
5
6     if (unlikely(!cfs_rq->nr_running))
7         return NULL;
8     /* 这个 do-while 循环是为了处理分组调度的情况。*/
9     do {
10         se = pick_next_entity(cfs_rq);
11         cfs_rq = group_cfs_rq(se);
12     } while (cfs_rq);
13
14     return task_of(se);
15 }
16
17 static struct sched_entity *pick_next_entity(struct cfs_rq *cfs_rq)
18 {
19     struct sched_entity *se = NULL;
20
21     /* 如果红黑树不为空。*/
22     if (first_fair(cfs_rq)) {
23         se = __pick_next_entity(cfs_rq);
24         set_next_entity(cfs_rq, se);
25     }
26     return se;
27 }
```

在第22行，如果就绪队列的红黑树最左节点不为空，就调用 __pick_next_entity() 取最左节点的进程对应的 sched_entity 结构。然后调用 set_next_entity() 把新进程从红黑树移出来。

代码片段 11.17 节自 kernel/sched_fair.c

```

1 /* 返回最左子节点。*/
2 static inline struct rb_node *first_fair(struct cfs_rq *cfs_rq)
3 {
4     return cfs_rq->rb_leftmost;
5 }
6
7 static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
8 {
```

```

9     return rb_entry(first_fair(cfs_rq), struct sched_entity, run_node);
10 }
11
12 set_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
13 {
14     /* se 对应的进程即将被调度运行，如果 on_rq 被设置，说明它在就绪队列中，首先出队。*/
15     /* 'current' is not kept within the tree. */
16     if (se->on_rq) {
17         update_stats_wait_end(cfs_rq, se);
18         __dequeue_entity(cfs_rq, se);
19     }
20
21     /* 更新统计信息。*/
22     update_stats_curr_start(cfs_rq, se);
23     /* cfs_rq->curr 被设置为新进程。*/
24     cfs_rq->curr = se;
25 #ifdef CONFIG_SCHEDSTATS
26     /* 统计信息。*/
27     .....
28 #endif
29     se->prev_sum_exec_runtime = se->sum_exec_runtime;
30 }

```

11.5.5 等待和唤醒操作

在前面我们看到，当一个进程投入运行时，要调用 `__dequeue_entity()` 把进程从就绪队列中删除，当进程被阻塞时，进程被加入到等待队列，直到进程被唤醒时才再次进入就绪队列。需要注意的是，等待队列不是由调度器维护的，而是由等待对象的相关程序维护的。以 `do_wait()` 为例（见第10章）：

```

1 static long do_wait(pid_t pid, int options,
2                     struct siginfo __user *infop,
3                     int __user *stat_addr,
4                     struct rusage __user *ru)
5 {
6     /* 定义一个等待队列节点 wait。*/
7     DECLARE_WAITQUEUE(wait, current);
8     .....
9     /* 把节点 wait 加入到等待队列中，其中第一个参数是等待队列头。*/
10    add_wait_queue(&current->signal->wait_chldexit, &wait);
11    /* 设置当前进程为可中断等待状态。*/

```

```
12     current->state = TASK_INTERRUPTIBLE;
13     .....
14     /* 当前进程主动让出 CPU, 调度其他进程运行。*/
15     schedule();
16
17     /* 设置进程状态为运行态。*/
18     current->state = TASK_RUNNING;
19     /* 把当前进程从等待队列中移除。*/
20     remove_wait_queue(&current->signal->wait_chldexit, &wait);
21     .....
22 }
```

`do_wait()`函数用来等待子进程结束，当子进程结束时，会向父进程发送信号，同时唤醒父进程，此时父进程进入就绪队列，当父进程被调度时，从第18行继续执行。其中 `DECLARE_WAITQUEUE` 定义如下：

```
1 #define __WAITQUEUE_INITIALIZER(name, tsk) {
2     /* 指向当前进程。*/
3     .private = tsk,
4     /* 唤醒时调用的参数。*/
5     .func = default_wake_function,
6     .task_list = { NULL, NULL } }
7
8 #define DECLARE_WAITQUEUE(name, tsk)
9     wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)
```

`add_wait_queue()`非常简单，它的第一个参数是队列头，第二个参数是 `wait_queue_t` 结构，这个函数把该结构加入到队列头部中。当进程被唤醒时，会调用 `default_wake_function()` 函数。

```
1 int default_wake_function(wait_queue_t *curr,
2                             unsigned mode,
3                             int sync,
4                             void *key)
5 {
6     return try_to_wake_up(curr->private, mode, sync);
7 }
8
9 static int try_to_wake_up(struct task_struct *p,
10                           unsigned int state,
11                           int sync)
12 {
13     .....
```

```

14     activate_task(rq, p, 1);
15     check_preempt_curr(rq, p);
16     .....
17     return success;
18 }

```

其中 `activate_task()` 就是把进程 `p` 加入就绪队列，对于 CFS 来说，它最终会调用到 `enqueue_task_fair()` 函数。在进程唤醒时，如果 `check_preempt_curr()` 检查到被唤醒的进程优先级高于当前进程，那么就会设置 `TIF_NEED_RESCHED` 标志，请求调度。`check_preempt_curr()` 会调用 CFS 的 `check_preempt_wakeup()` 函数，这在前面已经讨论过，现在来看看 `enqueue_task_fair()`：

```

1 static void enqueue_task_fair(struct rq *rq,
2                               struct task_struct *p,
3                               int wakeup)
4 {
5     struct cfs_rq *cfs_rq;
6     struct sched_entity *se = &p->se;
7
8     for_each_sched_entity(se) {
9         /* 如果进程 p 已经在就绪队列中，就什么也不用做了。 */
10        if (se->on_rq)
11            break;
12        cfs_rq = cfs_rq_of(se);
13        enqueue_entity(cfs_rq, se, wakeup);
14        wakeup = 1;
15    }
16 }
17
18 static void enqueue_entity(struct cfs_rq *cfs_rq,
19                           struct sched_entity *se,
20                           int wakeup)
21 {
22     /*
23      * Update run-time statistics of the 'current'.
24      */
25     update_curr(cfs_rq);
26
27     if (wakeup) {
28         /* 如果必要调整进程的虚拟时钟 vruntime. */
29         place_entity(cfs_rq, se, 0);
30     }

```

```

31     /* 更新统计信息。*/
32     enqueue_sleeper(cfs_rq, se);
33 }
34
35 update_stats_enqueue(cfs_rq, se);
36 check_spread(cfs_rq, se);
37
38 /* 把进程加入红黑树就绪队列中。*/
39 if (se != cfs_rq->curr)
40     _enqueue_entity(cfs_rq, se);
41 account_entity_enqueue(cfs_rq, se);
42 }
43
44 static void account_entity_enqueue(struct cfs_rq *cfs_rq,
45                                     struct sched_entity *se)
46 {
47     update_load_add(&cfs_rq->load, se->load.weight);
48     /* 增加就绪进程数目。*/
49     cfs_rq->nr_running++;
50     /* 设置on_rq，表示当前进程在就绪队列中。*/
51     se->on_rq = 1;
52 }
```

当一个进程睡眠一段时间后，其他进程得到了运行，这样这些进程的虚拟时钟都在前进，而当这个进程被唤醒时，它的虚拟时间可能远远小于其他进程。如果直接把这个进程加入就绪队列，那么在今后它在CPU的竞争中将占据很大的优势，如果不做特殊处理，在唤醒之后的一段时间间隔之内别的进程都得不到调度，所以对于曾经主动放弃CPU的进程，在唤醒时，要根据当前最慢的虚拟时钟来调整该进程的虚拟时钟，第29行调用place_entity()的作用：

```

1 place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
2 {
3     u64 vruntime;
4
5     /* 注意这是时钟最慢的虚拟时间。*/
6     vruntime = cfs_rq->min_vruntime;
7
8     /* 省略一些默认没有被配置的处理代码。*/
9     .....
10    /* 对于新建立的进程的处理。*/
11    if (initial && sched_feat(START_DEBIT))
12        vruntime += sched_vslice_add(cfs_rq, se);
```

```

13  /* 唤醒操作，而不是新建立的进程。*/
14  if (!initial) {
15      /* 如果允许则减少 vruntime，这相当于通过减慢进程的虚拟时钟来对它进行小小的“奖励”。*/
16      if (sched_feat(NEW_FAIR_SLEEPERS) && entity_is_task(se))
17          vruntime -= sysctl_sched_latency;
18      /* ensure we never gain time by being placed backwards. */
19      vruntime = max_vruntime(se->vruntime, vruntime);
20  }
21  se->vruntime = vruntime;
22 }

```

当建立一个新进程后，在这个进程第一次唤醒时，需要调用 task_new 函数，对 CFS 来说，这个函数就是 task_new_fair()。前面说过，CFS 根据就绪队列的虚拟时钟，把就绪进程组织成一颗红黑树，红黑树最左边子树是虚拟时钟最慢的进程。而对新进程来说，它从来没有得到运行，因此需要为它设置合适的虚拟时钟初值，以决定它在红黑树中的位置：

```

1 static void task_new_fair(struct rq *rq, struct task_struct *p)
2 {
3     struct cfs_rq *cfs_rq = task_cfs_rq(p);
4     struct sched_entity *se = &p->se, *curr = cfs_rq->curr;
5     int this_cpu = smp_processor_id();
6
7     sched_info_queued(p);
8     /* 更新当前进程的相关信息。*/
9     update_curr(cfs_rq);
10    /* 为新进程设置虚拟时钟的初始值。*/
11    place_entity(cfs_rq, se, 1);
12    /* 如果返回后子进程要先于当前进程（父进程）运行。*/
13    if (sysctl_sched_child_runs_first &&
14        this_cpu == task_cpu(p) &&
15        curr && curr->vruntime < se->vruntime) {
16        swap(curr->vruntime, se->vruntime);
17    }
18    /* 把子进程加入就绪队列。*/
19    enqueue_task_fair(rq, p, 0);
20    /* 请求调度。*/
21    resched_task(rq->curr);
22 }

```

task_new_fair()也调用 place_entity()来设置虚拟时钟，只不过它的 initial 参数为 1：

```

1 static void
2 place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)

```

```

3  {
4      u64 vruntime;
5
6      vruntime = cfs_rq->min_vruntime;
7      .....
8      if (initial && sched_feat(START_DEBIT))
9          vruntime += sched_vslice_add(cfs_rq, se);
10     .....
11     se->vruntime = vruntime;
12 }

```

`sched_vslice_add()`根据进程的权重计算出进程应得的时间片，加到 `cfs_rq->vruntime` 中，这就是新进程虚拟时钟的默认初始值。

11.5.6 nice 系统调用

通过 `nice` 系统调用可以改变进程的静态优先级，范围从-20~19，但是普通进程的静态优先级的范围是 100~139，对于 CFS 来说，因此 `nice` 需要根据参数做必要的转换。

代码片段 11.18 节自 `kernel/sched.c`

```

1 asmlinkage long sys_nice(int increment)
2 {
3     long nice, retval;
4     /*
5      * 普通进程的静态优先级范围是 100~139，共 40 个级别，
6      * 当 increment 为负数时，表示增大优先级，为正数时表示减小优先级。
7      */
8     if (increment < -40)
9         increment = -40;
10    if (increment > 40)
11        increment = 40;
12
13    /* static_prio - 120 + increment 得到 nice 值，范围从-20~19。 */
14    nice = PRIO_TO_NICE(current->static_prio) + increment;
15    if (nice < -20)
16        nice = -20;
17    if (nice > 19)
18        nice = 19;
19    /* 检测当前进程是否有权限提高进程优先级。 */
20    if (increment < 0 && !can_nice(current, nice))
21        return -EPERM;
22    /* 安全检查函数，默认为空函数。 */

```

```
23     retval = security_task_setnice(current, nice);
24     if (retval)
25         return retval;
26     /* 设置 nice. */
27     set_user_nice(current, nice);
28     return 0;
29 }
```

set_user_nice() 定义如下代码片段11.19所示：

代码片段 11.19 节自 kernel/sched.c

```
1 void set_user_nice(struct task_struct *p, long nice)
2 {
3     int old_prio, delta, on_rq;
4     unsigned long flags;
5     struct rq *rq;
6
7     if (TASK_NICE(p) == nice || nice < -20 || nice > 19)
8         return;
9     /*
10      * We have to be careful, if called from sys_setpriority(),
11      * the task might be in the middle of scheduling on another CPU.
12      */
13     rq = task_rq_lock(p, &flags);
14     update_rq_clock(rq);
15     /*
16      * The RT priorities are set via sched_setscheduler(), but we still
17      * allow the 'normal' nice value to be set - but as expected
18      * it wont have any effect on scheduling until the task is
19      * SCHED_FIFO/SCHED_RR;
20      */
21     if (task_has_rt_policy(p)) {
22         p->static_prio = NICE_TO_PRIO(nice);
23         goto out_unlock;
24     }
25     on_rq = p->se.on_rq;
26
27     /* 如果进程在就绪队列中，就从就绪队列摘下来。*/
28     if (on_rq) {
29         dequeue_task(rq, p, 0);
30         dec_load(rq, p);
31     }
32 }
```

```
33     /* 设置新的静态优先级。*/
34     p->static_prio = NICE_TO_PRIO(nice);
35     /* 根据新优先级设置权重。*/
36     set_load_weight(p);
37     old_prio = p->prio;
38     p->prio = effective_prio(p);
39     delta = p->prio - old_prio;
40
41     /* 如果以前在就绪队列中，就再次入队，对于在等待队列中的进程，则不必要进行。*/
42     if (on_rq) {
43         enqueue_task(rq, p, 0);
44         inc_load(rq, p);
45         /*
46          * If the task increased its priority or is running and
47          * lowered its priority, then reschedule its CPU:
48          */
49         /*
50          * 优先级改变后，请求重新调度，例如目标进程的优先级比当前进程的
51          * 优先级还高，就需要把CPU让给目标进程了。
52          */
53         if (delta < 0 || (delta > 0 && task_running(rq, p)))
54             resched_task(rq->curr);
55     }
56 out_unlock:
57     task_rq_unlock(rq, &flags);
58 }
```

第12章 文件系统

文件系统是操作系统，文件是数据的逻辑管理单位，它可能在内存、磁盘等存储介质中。而文件系统则定义了数据在这些存储介质上的存储格式和存储方式。如果把存储介质比喻为是白纸，布匹或者竹简，如果要在白纸，布匹或竹简上记录一本书的内容，就必须确定页的大小，存储方式(例如是现代的横排从左到右的习惯，或者是中国古代的竖排从上到下的习惯)，书籍目录的排列方式，等等。不同的文件系统，其内容在存储介质上的组织方式不同，但是其原理都是一样的。

本章以 Ext2 为例，深入介绍 Ext2 文件系统在磁盘上的组织结构。不同的文件系统，其操作方式也不同，但同时操作系统又必须要为应用程序建立一个统一的接口，为此 Linux 内核设计了虚拟文件系统(VFS)。本章从底向上，首先介绍 Ext2 文件系统的操作方式，之后在这个基础上对虚拟文件进行介绍。由于具体文件系统和虚拟文件系统紧密相关，因此笔者把它们安排在同一章进行介绍，尽量做到一气呵成，例如：对文件的打开操作，可以直接从 `open()` 系统调用，到 VFS 层，到 Ext2 层进行介绍。

12.1 Ext2 的磁盘结构

磁盘以扇区作为基本存储单位，目前每个扇区的大小是 512 字节¹。可以通过硬盘分区把一个物理硬盘划分为多个逻辑硬盘，每一个逻辑硬盘就是一个分区，每个分区可以使用独立的文件系统格式。硬盘的第一个扇区，保存了 MBR²、分区表，和结束标志 55AA。其中前 446 个字节为 MBR 代码，其后的 64 个字节是分区表，最后是 4 个字节的标志 55AA。每个分区表占用 16 个字节，所以一块硬盘最多只能有 4 个分区，随着硬盘容量的增加，4 个分区可能不够用，为此提出了扩展分区，在扩展分区上可以划分出多个逻辑分区，这就类似于分区表的二级索引。其中分区表项格式如表12.1所示。

需要注意的是，磁盘控制器通过磁头号、柱面号和扇区号来对扇区进行寻址，而逻辑扇区号，就是把全部的扇区看做一个线性地址空间上的扇区编号。当使用 `fdisk` 等磁盘分区工具对磁盘进行分区时，操作的就是分区表。

¹随着硬盘容量的不断增加，现在也提出了把扇区的大小提升为 1KB，2KB 或者 4KB。

²Master Boot Record，主引导记录，开机后，BIOS 会加载 MBR 并执行 MBR 的代码。

表 12.1 分区表项结构

偏移	说明
0	0x80 表示该分区为活动分区，否则为 00
1	该分区的起始磁头号
2	低 6 位表示该分区的起始扇区号，高两位表示该分区的起始柱面号
3	该分区的起始柱面号的低 8 位，起始柱面号一共 10 位
4	文件系统标志，0x00 表示未使用，0x05 表示扩展分区，0x07 表示 NTFS，0x83 表示 Ext2，等等
5	该分区的结束磁头号
6	低 6 位表示该分区的结束扇区号，高两位表示结束柱面号
7	该分区结束柱面号的低 8 位，总共 10 位
8-11	该分区起始的相对逻辑扇区号，高位在前
12-15	该分区的扇区数，高位在前

由于磁盘是机械式结构，在进行读写时，磁盘控制器首先需要根据指定的磁头号、柱面号和扇区号，控制磁头移动到指定磁道，这被称为寻道时间。在定位好磁道后，还要等待盘片的相关扇区旋转到磁头下面，这被称为旋转延时，最后才通过 DMA 把指定扇区的数据传输到内存中。可以看出，如果文件的内容是连续存放的，那么只需要经过一次寻道时间和一次旋转延迟，就可以把文件内存全部传输到内存中。相反，如果文件不连续存放，则需要经过反复的寻道和旋转延迟。然而实际中，是很难保证每一个文件都是连续存放的。例如一个进程新建立一个 1KB 大小的文件，随后另外一个进程又建立了一个文件，之后如果第一个文件增大，这些内容就有可能位于第二个文件之后。虽然可以通过磁盘碎片整理来改善这种情况，但是我们希望能够在第一个文件的末尾保留几个连续的扇区，这样当文件增长时就可以减少碎片的情况，为此 Linux 提出了块(Block)³的概念，块大小是扇区的 2^n 倍，通常块的大小为 1024B, 2048B 或者 4096B。一个分区上的每个块都有块号，块是文件系统分配的基本单位，以 4096B 为例，就算一个文件只有一个字节的内容，也会占用 8 个扇区，这样当文件内容增加时就可以减少文件碎片。但是如果块太大，就会造成空间的浪费，这就是时间和空间的矛盾。

文件的内容可能散落在不连续的块中，这样每个文件都需要一个结构统一记录该文件的数据块位置，在 Linux 中，这个结构被称为索引节点(inode)，Ext2 磁盘上的 inode 结构定义如下：

代码片段 12.1 节自 `include/linux/ext2_fs.h`

```

1 struct ext2_inode {
2     /* 文件模式，可以通过 chmod 命令修改。*/
3     __le16 i_mode;    /* File mode */
4

```

³ 在 Windows 中被称为簇(Cluster)。

```
5  /* 文件拥有者的用户 ID, 可以通过 chown 命令修改。*/
6  __le16 i_uid;      /* Low 16 bits of Owner Uid */
7  __le32 i_size;     /* Size in bytes */
8  __le32 i_atime;    /* Access time */
9  __le32 i_ctime;    /* Creation time */
10 __le32 i_mtime;    /* Modification time */
11 __le32 i_dtime;    /* Deletion Time */
12 __le16 i_gid;      /* Low 16 bits of Group Id */
13 __le16 i_links_count; /* Links count */
14
15 /* 文件数据区占用的块数。*/
16 __le32 i_blocks;   /* Blocks count */
17 __le32 i_flags;    /* File flags */
18 .....
19 /* 数据块地址。*/
20 __le32 i_block[ EXT2_N_BLOCKS];/* Pointers to blocks */
21 __le32 i_generation; /* File version (for NFS) */
22 __le32 i_file_acl; /* File ACL */
23 __le32 i_dir_acl; /* Directory ACL */
24 __le32 i_faddr;   /* Fragment address */
25 union {
26     struct {
27         __u8 l_i_frag; /* Fragment number */
28         __u8 l_i_fsize; /* Fragment size */
29         __u16 i_pad1;
30         __le16 l_i_uid_high; /* these 2 fields */
31         __le16 l_i_gid_high; /* were reserved2[0] */
32         __u32 l_i_reserved2;
33     } linux2;
34     struct {
35         __u8 h_i_frag; /* Fragment number */
36         __u8 h_i_fsize; /* Fragment size */
37         __le16 h_i_mode_high;
38         __le16 h_i_uid_high;
39         __le16 h_i_gid_high;
40         __le32 h_i_author;
41     } hurd2;
42     struct {
43         __u8 m_i_frag; /* Fragment number */
44         __u8 m_i_fsize; /* Fragment size */
45         __u16 m_pad1;
46         __u32 m_i_reserved2[ 2];
```

```

47     } masix2;
48 } osd2;      /* OS dependent 2 */
49 };

```

这个结构中的 `i_flags` 是文件的标志，其中 `EXT2_SECRM_FL` 表示安全删除，在删除该标志时，其数据块会被随机数填充，`EXT2_UNRM_FL` 表示可恢复删除，删除该文件后，其数据块，`inode` 等内容会保存一段时间，在这段时间内，仍然可以恢复该文件。`EXT2_SYNC_FL` 表示同步写，由于磁盘上的内容通常被缓冲在内存中，修改操作都是对内存的内容进行的，必要时才会被写到磁盘，`EXT2_SYNC_FL` 表示在写入操作过程中，同步更新磁盘块的内容。另外还有许多其他标志，在此不一一列举。

由于块是文件系统分配的基本单位，当块设置得较大时，可能会存在一定程度的空间浪费，假设块大小为 4KB，一个 5KB 大小的文件会占用 8KB 的数据块，这样就浪费了 3KB。为此又提出了片(Fragment)的概念，片的本意是让不同的文件可以使用同一个块存储数据，从而达到节省空间的目的，但是目前没有使用。

`i_blocks` 表示一个文件所占用的数据块个数，数组 `i_block` 存储数据块的地址，总共 15 项，假设块大小为 4KB 时，一个文件的大小不能超过 60KB，这显然是不符合要求的，为此前 12 项直接存储块地址，而剩余的 3 项，依次为“一次间接块指针”，“二次间接块指针”，“三次间接块指针”。其结构如图 12.1 所示。

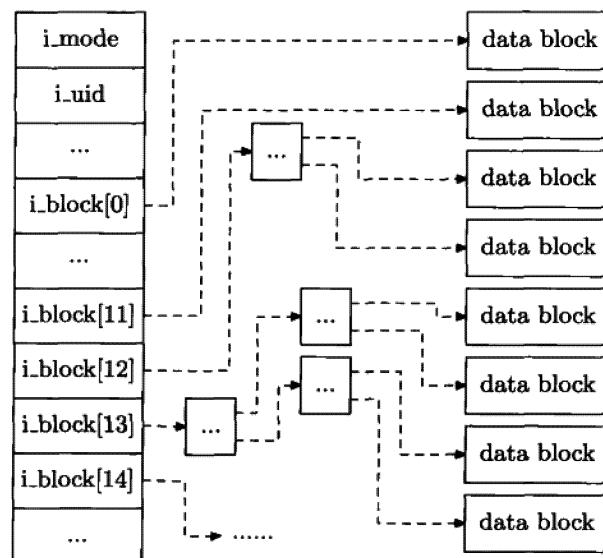


图 12.1 Ext2 inode 结构示意图

`i_block` 数组中的前 12 项保存直接块地址，假设块大小为 BS ，所以直接块中可以存储的文件大小为 $12 * BS$ 字节。由于块地址占 4 个字节，一个“一次间接块”中可以存储 $\frac{BS}{4}$

个块地址，于是通过一次间接块存储的内容大小为 $BS * \frac{BS}{4}$ 字节，一个“二次间接块”存储着 $\frac{BS}{4}$ 个块地址，每一个块又存储着 $\frac{BS}{4}$ 个块地址，所以一个“三次间接块”的内容大小为 $BS * \frac{BS}{4} * \frac{BS}{4}$ 字节，同理通过一个“三次间接块”存储的内容为 $BS * \frac{BS}{4} * \frac{BS}{4} * \frac{BS}{4}$ 的内容。假设块大小为 4096 字节，那么“一次间接块”存储的内容就是 4MB，“二次间接块”存储的内容就是 4GB，“三次间接块”存储的内容就是 4TB。

每一个文件都有一个对应的 inode 结构，通过 i_block 数组就能访问到该文件的数据，然而在 inode 中却没有文件名，那么文件名保存在哪里呢？实际上文件名保存在目录中，从磁盘结构来看，目录也是一个文件，它也占用一个 inode 结构，它的数据块存储的是该目录下所有文件的文件名，以及各个文件对应的 inode 号。Ext2 的目录项结构定义如下：

代码片段 12.2 节自 include/linux/ext2_fs.h

```

1 #define EXT2_NAME_LEN 255
2
3 struct ext2_dir_entry_2 {
4     __le32    inode;      /* Inode number */
5     __le16    rec_len;    /* Directory entry length */
6     __u8     name_len;   /* Name length */
7     __u8     file_type;
8     char    name[EXT2_NAME_LEN]; /* File name */
9 };

```

其中 name 保存的是文件名，最大长度为 255 字节，inode 保存着这个文件对应的 inode 号，file_type 保存着这个文件的类型，目前有以下几种类型：

代码片段 12.3 节自 include/linux/ext2_fs.h

```

1 enum {
2     EXT2_FT_UNKNOWN,
3     EXT2_FT_REG_FILE, /* 普通文件。 */
4     EXT2_FT_DIR,      /* 目录。 */
5     EXT2_FT_CHRDEV,   /* 字符设备文件。 */
6     EXT2_FT_BLKDEV,   /* 块设备文件。 */
7     EXT2_FT_FIFO,     /* 管道文件。 */
8     EXT2_FT SOCK,     /* Unix 套接字文件。 */
9     EXT2_FT_SYMLINK,  /* 符号链接文件。 */
10    EXT2_FT_MAX
11 };

```

在 Ext2 文件系统中，根目录的 inode 号被定义为 EXT2_ROOT_INO，总是固定为 2。通过命令 ls -ai 可以查看到各个文件的 inode 号。Ext2 目录项在磁盘上的结构如图 12.2 所示。

inode	rec_len	name_len	file_type									
			name									
0	2	12	1	2	.	0	0	0				
12	2	12	2	2	.	.	0	0				
24	58	16	6	2	i	n	i	t	r	d	0	0
40	54	12	3	2	l	i	b	0				
52	72	16	5	2	m	e	d	i	a	0	0	0
68	64	12	3	2	b	i	n	0				

图 12.2 Ext2 目录数据示意图

文件的定位是一个简单烦琐的过程，假设现在要打开的文件是/bin/bash，需要经过以下步骤：

- (1) 根目录的 Inode 编号总是固定为 2，读取根目录的 Inode，其 file_type 成员说明这是一个目录，根据这个 Inode 的 i_block 数组，找到它的数据块，从数据块中读取该目录下的文件列表，找到文件 bin 的 inode 号，如图12.2所示，此处为 64。
- (2) 根据 64 找到 bin 文件对应的 Inode，根据 file_type 和 i_block 读取数据块中的文件列表，找到 bash 文件的 Inode 号。
- (3) 根据 bash 的 Inode 号，找到对应的 Inode 结构，根据 i_block 读取 bash 文件的数据块。

因为每一次都从根目录开始查找文件过于烦琐，所以每一个目录项中都保存了两个特殊目录项“.”和“..”分别记录了当前目录和上级目录的 Inode 号。

在 Linux 中，可以为一个文件建立别名，这被称为链接文件。链接文件又分为软链接和硬链接。假设/tmp 目录下有一个名为 target 的文件，/tmp 的目录项中必然 target 和它的 Inode 号的对应项，假设 target 对应的 Inode 号为 I。现在在/tmp 目录下通过 ln target hard_link，命令为 target 文件建立一个硬链接，那么/tmp 的目录项中会多出来一个 hard_link 的条目，它所指向的 Inode 编号也为 I。所以无论是打开 target 文件还是 hard_link 文件，定位到的都是同一个 Inode，同理进一步定位到的也是相同的数据块。硬链接的一个缺点是不能跨越文件系统，因为内核找到目录项中文件名对应的 Inode 编号后，总是在当前的文件系统中打开这个 Inode，对它进行操作。因此即便对于两个 Ext2 格式文件系统分区，也不能在它们之间建立硬链接。

软链接又被称为符号链接，如果到/tmp 目录下中通过 ln -s target soft_link 命令，可以为 target 建立一个名为 soft_link 的软链接。这样在/tmp 的目录项中，就会有一个 soft_link 的条目，它的 Inode 号为 J(J≠I)，同时这个目录项条目中的 type 为 EXT2_FT_SYMLINK。

在这种情况下，内核就会打开编号为 J 的 Inode，从它的数据块中读取到它链接的目标文件路径。在本例中为 target，然后内核转去打开这个路径，由于 Inode 的 i_block 数组长度 15，一共是 60 个字节大小，如果目标文件的路径长度小于 60，那么就不需要使用额外的数据块了，直接把这个路径保存在 i_block 数组中。如果目标文件的路径长度大于 60，那么就需要额外的数据块了。在内核中，前者被称为 fast symbolic link，后者被称为 symbolic link。

可以看出硬链接共用 Inode，因此不需要额外的 Inode 空间，而软链接会占用一个额外的 Inode 存储空间，甚至可能需要额外的数据块空间。但是软链接克服了硬链接的缺点，它可以跨越文件系统。下面的试验展示了软链接和硬链接的区别。

代码片段 12.4 软连接和硬连接的区别

```

1 /tmp$ touch target
2 /tmp$ ln target hard_link
3 /tmp$ ln -s target soft_link
4
5 /tmp$ ls -i
6 915631 hard_link  915632 soft_link  915631 target

```

文件名前面的数字是 Inode 号，硬链接的 Inode 编号和目标文件的一致，而软链接则不一致。

Inode 建立了目录、文件、以及数据之间的联系。但是这还不够，在每一个分区上，还需要记录 Inode 和数据块分别从哪里开始，从哪里结束，哪些 Inode 和数据块是空闲的。这样才能正确的分配文件。随着磁盘容量的不断增加，分区的大小也在不断增加，为了使文件的数据块尽可能地放在连续的磁盘空间上，Ext2 又提出了组，组是一个逻辑概念，把一个分区划分成若干个组。组用来限制文件的数据块不要太过于散落。Ext2 会尽量保证一个文件的数据块在同一个组，这样可以提高读写效率。Ext2 文件系统在磁盘分区上的整体结构如图12.3所示。通常对磁盘格式化完成后，使用 mkfs.ext2 格式化建立文件系统时，就是在磁盘上建立如图12.3所示的结构。

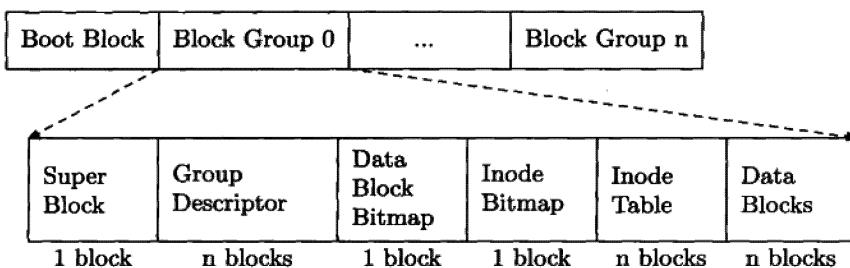


图 12.3 Ext2 磁盘布局示意图

超级块记录着一个分区的整体信息，例如这个分区上有多少个 Inode，多少个组，多少个数据块，多少个空闲块等。组描述符记录了这个组中的 Inode 数量、空闲块数量等信息。

数据块 Bitmap 和 Inode Bitmap 分别是数据块和 Inode 的位图。每一个 bit 表示一个块或者一个 Inode 的状态。这样假设数据块 Bitmap 是 N 个字节。那么最多只能有 $8 * N$ 个数据块。每一个分区中，只要有一个超级块和一个组描述符就可以了，但是实际上，一个分区中有多个组，每一个组都有一个超级块和组描述符。磁盘检测工具 e2fsck 会把第一个超级块复制到各个组中，以后当 e2fsck 检测到某些错误时，可以根据备份的超级块来恢复。Ext2 磁盘超级块结构如下：

代码片段 12.5 节自 include/linux/ext2_fs.h

```
1 struct ext2_super_block {
2     __le32 s_inodes_count;    /* Inodes count */
3     __le32 s_blocks_count;   /* Blocks count */
4     __le32 s_r_blocks_count; /* Reserved blocks count */
5     __le32 s_free_blocks_count; /* Free blocks count */
6     __le32 s_free_inodes_count; /* Free inodes count */
7     __le32 s_first_data_block; /* First Data Block */
8     __le32 s_log_block_size; /* Block size */
9     __le32 s_log_frag_size; /* Fragment size */
10    __le32 s_blocks_per_group; /* # Blocks per group */
11    __le32 s frags_per_group; /* # Fragments per group */
12    __le32 s_inodes_per_group; /* # Inodes per group */
13    __le32 s_mtime; /* Mount time */
14    __le32 s_wtime; /* Write time */
15    __le16 s_mnt_count; /* Mount count */
16    __le16 s_max_mnt_count; /* Maximal mount count */
17    __le16 s_magic; /* Magic signature */
18    __le16 s_state; /* File system state */
19    __le16 s_errors; /* Behaviour when detecting errors */
20    __le16 s_minor_rev_level; /* minor revision level */
21    __le32 s_lastcheck; /* time of last check */
22    __le32 s_checkinterval; /* max. time between checks */
23    __le32 s_creator_os; /* OS */
24    __le32 s_rev_level; /* Revision level */
25    __le16 s_def_resuid; /* Default uid for reserved blocks */
26    __le16 s_def_resgid; /* Default gid for reserved blocks */
27 /*
28  * These fields are for EXT2_DYNAMIC_REV superblocks only.
29  *
30  * Note: the difference between the compatible feature set and
31  * the incompatible feature set is that if there is a bit set
32  * in the incompatible feature set that the kernel doesn't
33  * know about, it should refuse to mount the filesystem.
34  *
```

```

35     * e2fsck's requirements are more strict; if it doesn't know
36     * about a feature in either the compatible or incompatible
37     * feature set, it must abort and not try to meddle with
38     * things it doesn't understand...
39
40     __le32 s_first_ino;      /* First non-reserved inode */
41     __le16 s_inode_size;     /* size of inode structure */
42     __le16 s_block_group_nr; /* block group # of this superblock */
43     __le32 s_feature_compat; /* compatible feature set */
44     __le32 s_feature_incompat; /* incompatible feature set */
45     __le32 s_feature_ro_compat; /* readonly-compatible feature set */
46     __u8 s_uuid[16];        /* 128-bit uuid for volume */
47     char s_volume_name[16];  /* volume name */
48     char s_last_mounted[64]; /* directory where last mounted */
49     __le32 s_algorithm_usage_bitmap; /* For compression */
50
51     .....
52 };

```

`s_inodes_count` 和 `s_blocks_count` 分别表示这个分区中的 Inode 和 block 的总数量。`s_free_inodes_count` 和 `s_free_blocks_count` 分别表示空闲的 Inode 和空闲块的总数。`s_r_blocks_count` 表示保留的块数量，当空闲块的数量等于保留块数量的时候，普通用户就不能申请到空闲块了。这些块是保留给超级用户使用的。`s_first_data_block` 是第一个数据块的编号。

`s_log_block_size` 是块的大小，一般是 1KB~4KB。`s_log_frag_size` 表示片的大小，目前默认等于 `s_log_block_size`，也就是不分片。

`s_blocks_per_group` 表示每个组包含多少个块。`s_frags_per_group` 表示每个组中有多少个片，由于不分片，因此 `s_blocks_per_group` 等于 `s_frags_per_group`。

`s_mnt_count` 表示被 mount 的次数，系统启动的时候，e2fsck 会检查 `s_mnt_count` 和 `s_max_mnt_count`，当 `s_mnt_count` 达到 `s_max_mnt_count` 时，就会进行文件系统检测。`s_lastcheck` 是上一次进行文件系统检测的时间，`s_checkinterval` 是检测的最大间隔，这样也可以配置 e2fsck 进行文件系统检测最大的间隔。`s_state` 表示文件系统的状态，当 `s_state` 为 0 时，表示这个分区被 mount 了，磁盘被 mount 之前，如果检测到它的 `s_state` 为 0，就说明上一次这个磁盘没有被 unmount，那么有可能出现错误，因此 e2fsck 会进行磁盘检测。这主要是由于块设备使用了缓存，如果没有被正确地 unmount，可能缓存中的数据并每有被正确地写入磁盘。

`s_inode_size` 表示分区中，Inode 占用的空间大小。`s_volume_name` 和 `s_uuid` 分别表示这个分区的名字和 UID。

组是一个逻辑概念，组的目的是为了尽量保证一个文件的数据块在磁盘上的位置不要

太分散。Ext2 文件系统的组描述符在磁盘上的格式如下：

代码片段 12.6 节自 `include/linux/ext2_fs.h`

```

1 struct ext2_group_desc
2 {
3     __le32  bg_block_bitmap; /* Blocks bitmap block */
4     __le32  bg_inode_bitmap; /* Inodes bitmap block */
5     __le32  bg_inode_table; /* Inodes table block */
6     __le16  bg_free_blocks_count; /* Free blocks count */
7     __le16  bg_free_inodes_count; /* Free inodes count */
8     __le16  bg_used_dirs_count; /* Directories count */
9     __le16  bg_pad;
10    __le32  bg_reserved[3];
11 };

```

`bg_block_bitmap` 表示这个组的位图在哪一个块，`bg_inode_bitmap` 表示这个组的 Inode 位图所在的块号，`bg_inode_table` 表示这个组的 inode 所在的起始块号。`bg_free_blocks_count` 和 `bg_free_inodes_count` 分别表示这个组中空闲块的数量和空闲 Inode 的数量。每一个组描述符占用 24 个字节，所以和超级块一样，每一个组中的组描述符都保存了一份这个分区上的所有的组描述符。

根据分区表可以知道一个分区的总扇区数，根据超级块中的 `s_log_block_size` 可以知道块的大小，假设 `s_log_block_size` 为 bs ，则每一个块为 $1024 * 2^{bs}$ 个字节。例如 0 表示块大小为 1024 字节，1 表示块大小为 2048 个字节。但是一个分区上，有多少个组呢？一个组又包含了多少个块呢？这是由分区大小和块大小决定的。从图12.3中可以看出，每一个组只有一个逻辑块作为组的位图，由于位图中，每一个 bit 表示对应的一个块的状态，0 表示空闲，1 表示占用。假设块大小为 bs 字节，一个组中最多只能有 $8 * bs$ 个块。根据一个分区中的总块数和每一个组的块数，就能确定一个分区有几个组了。假设分区大小为 ps 字节，块大小为 bs 字节，则每组中的块数为 $8 * bs$ ，每组中的字节数为 $8 * bs * bs$ ，所以这个分区中一共有 $\frac{ps}{8 * bs * bs}$ 个组。同理可以算出每个组的大小。

12.2 Ext2 的内存结构

上一节介绍的是 Ext2 磁盘上的布局，在使用过程中，内核需要频繁地访问某些结构，因此当磁盘驱动程序把相关数据从磁盘上读出来后，内核会建立相应的内存中的结构。

Ext2 在内存中的超级块结构定义如下：

代码片段 12.7 节自 `include/linux/ext2_fs_sb.h`

```

1 struct ext2_sb_info {
2     /* Size of a fragment in bytes */

```

```
3     unsigned long s_frag_size;
4     /* Number of fragments per block */
5     unsigned long s_frags_per_block;
6     /* Number of inodes per block */
7     unsigned long s_inodes_per_block;
8     /* Number of fragments in a group */
9     unsigned long s_frags_per_group;
10    /* Number of blocks in a group */
11    unsigned long s_blocks_per_group;
12    /* Number of inodes in a group */
13    unsigned long s_inodes_per_group;
14    /* Number of inode table blocks per group */
15    unsigned long s_itb_per_group;
16    /* Number of group descriptor blocks */
17    unsigned long s_gdb_count;
18    /* Number of group descriptors per block */
19    unsigned long s_desc_per_block;
20    /* Number of groups in the fs */
21    unsigned long s_groups_count;
22    /* Last calculated overhead */
23    unsigned long s_overhead_last;
24    /* Last seen block count */
25    unsigned long s_blocks_last;
26
27    /* Buffer containing the super block */
28    struct buffer_head * s_sbh;
29    /* Pointer to the super block in the buffer */
30    struct ext2_super_block * s_es;
31    struct buffer_head ** s_group_desc;
32    unsigned long s_mount_opt;
33    unsigned long s_sb_block;
34    .....
35 };
```

这个结构中的大部分成员和磁盘结构的成员一致，注释得也比较清楚。在初始化阶段，内核利用磁盘驱动程序把磁盘上面的超级块和组描述符全部读入内存，由于内核需要频繁地使用这些结构，因此它们是常驻内存的。`ext2_sb_info` 结构中的 `s_sbh` 和 `s_group_desc` 分别指向包含磁盘超级块和组描述符的缓冲区头部。`s_es` 成员则指向包含磁盘超级块结构的内存首地址。`ext2_sb_info` 的建立是由 `ext2_fill_super()` 函数完成的。我们将在后面进一步介绍 `ext2_fill_super()` 函数(见第391页代码片段12.13)。

Ext2 在内存中的 Inode 结构定义如下：

代码片段 12.8 节自 fs/ext2/ext2.h

```
1 struct ext2_inode_info {
2     __le32 i_data[15];
3     __u32 i_flags;
4     __u32 i_faddr;
5     __u8 i_frag_no;
6     __u8 i_frag_size;
7     __u16 i_state;
8     __u32 i_file_acl;
9     __u32 i_dir_acl;
10    __u32 i_dtime;
11
12    /*
13     * i_block_group is the number of the block group which contains
14     * this file's inode. Constant across the lifetime of the inode,
15     * it is used for making block allocation decisions - we try to
16     * place a file's data blocks near its inode block, and new inodes
17     * near to their parent directory's inode.
18     */
19    __u32 i_block_group;
20
21    /* block reservation info */
22    struct ext2_block_alloc_info *i_block_alloc_info;
23    __u32 i_dir_start_lookup;
24    struct inode vfs_inode;
25    .....
```

这个结构中的多数成员也和磁盘上的类似。磁盘上的块位图和 Inode 位图也会根据需要被读取到内存中。对于符号链接来说，如果目标文件的路径长度小于 60，就把路径直接存储在 `i_data` 中。

需要注意的是 `ext2_inode_info` 结构中包含了一个 `vfs_inode` 成员，这是虚拟文件系统的 `inode` 结构，我们将在下一节对 VFS 的 `inode` 结构进行讨论。

12.3 虚拟文件系统的管理结构

Linux 支持各种不同的文件系统，同时对上层抽象出一个统一的接口，例如上层应用程序无须关心文件系统的细节，只需要利用 `open()`, `read()`, `write()` 等系统调用就能对文件进行操作。为此提出了虚拟文件系统(Virtual Filesystem)。对于不同的文件系统，它的磁盘结构的布局肯定是不一样的，但是虚拟文件系统屏蔽了不同的文件系统之间的差异，向上层提供一个统一接口。虚拟文件系统的管理结构包括超级块、Inode、目录项等。这些结构

都是根据不同文件系统的超级块、Inode、目录项目构造的。例如，无论两个不同的具体文件系统的超级块的布局有什么差异，它们的 VFS 的超级块布局都是一致的。

12.3.1 文件系统对象

每一个文件系统驱动程序都有一个文件系统对象，其定义如下：

代码片段 12.9 节自 `include/linux/fs.h`

```

1 struct file_system_type {
2     /* 文件系统名称。*/
3     const char *name;
4     int fs_flags;
5     /* 超级块初始化函数指针。*/
6     int (*get_sb) (struct file_system_type *, int,
7                     const char *, void *,
8                     struct vfsmount *);
9     /* 释放超级块函数指针。*/
10    void (*kill_sb) (struct super_block *);
11    struct module *owner;
12    struct file_system_type * next;
13    struct list_head fs_supers;
14    .....
15 };

```

各个文件系统驱动都需要调用 `register_filesystem()` 注册文件系统对象。所有的文件系统对象通过 `next` 指针链接成链表，全局变量 `file_systems` 指向链表的头部。`register_filesystem()` 函数很简单，就是保证不重复的前提下，把新的文件系统对象加入到链表中。由于一个文件系统驱动可能对应多个分区，因此 `fs_supers` 链表链接了各个分区的超级块。

Ext2 的 `file_system_type` 结构定义如下：

代码片段 12.10 节自 `fs/ext2/super.c`

```

1 static struct file_system_type ext2_fs_type = {
2     .owner      = THIS_MODULE,
3     .name       = "ext2",
4     .get_sb    = ext2_get_sb,
5     .kill_sb   = kill_block_super,
6     .fs_flags  = FS_REQUIRES_DEV,
7 };
8
9 static int __init init_ext2_fs(void)
10 {
11     int err = init_ext2_xattr();

```

```
12     .....
13     err = register_filesystem(&ext2_fs_type);
14     if (err)
15         goto out;
16     return 0;
17     .....
18 }
```

12.3.2 VFS 的超级块

VFS 超级块是内核根据具体文件系统的超级块建立的内存结构，其定义如下：

代码片段 12.11 节自 `include/linux/fs.h`

```
1 struct super_block {
2     /* 每一个分区都有一共超级块，所有的超级块通过 s_list 链接。*/
3     /* Keep this first */
4     struct list_head s_list;
5
6     /* search index; _not_ kdev_t */
7     dev_t      s_dev;
8     unsigned long   s_blocksize;
9     unsigned char    s_blocksize_bits;
10    unsigned char   s_dirt;
11
12    /* Max file size */
13    unsigned long long s_maxbytes;
14
15    struct file_system_type *s_type;
16    const struct super_operations *s_op;
17
18    struct dquot_operations *dq_op;
19    struct quotactl_ops *s_qcop;
20    const struct export_operations *s_export_op;
21    unsigned long   s_flags;
22    unsigned long   s_magic;
23    struct dentry   *s_root;
24    struct rw_semaphore s_umount;
25    struct mutex     s_lock;
26    int           s_count;
27    int           s_syncing;
28    int           s_need_sync_fs;
29    atomic_t      s_active;
```

```

30     .....
31     /* Informational name */
32     char s_id[32];
33     /* Filesystem private info */
34     void     *s_fs_info;
35     .....
36 };

```

结构中的 `s_type` 指针指向这个分区对应的文件系统对象。当内核需要挂载(mount)一个块设备的时候，会根据分区表中的信息分析这个块设备的文件系统类型，然后从 `file_systems` 链表中找到对应的文件系统驱动程序的文件系统对象，调用它的 `get_sb()` 函数获取具体文件系统的超级块的信息，然后根据这些信息初始化 VFS 超级块。结构中的 `s_fs_info` 就指向具体文件系统的超级块内存对象。如果这个分区的文件系统类型是 Ext2，那么这个结构就是 `ext2_sb_info`。

由于各种文件系统的超级块是不同，对操作超级块也不一样。为此内核定义了一个 `super_operations` 结构，用 C++ 的观念来说，这就是 `super_block` 对象的虚函数表，只不过在 C++ 中这个虚函数表是依靠编译器自动建立的，而在这里是由程序员手工建立的。`super_operations` 定义如下：

代码片段 12.12 节自 `include/linux/fs.h`

```

1 struct super_operations {
2     /* 分配一个 Inode 结构。*/
3     struct inode *(*alloc_inode)(struct super_block *sb);
4     /* 释放 Inode 结构。*/
5     void (*destroy_inode)(struct inode *);
6     /*
7      * 从磁盘上读取指定的 Inode 结构，并初始化 Inode 结构。
8      * 磁盘的 Inode 号由 inode 的 i_ino 成员指定。
9      */
10    void (*read_inode) (struct inode *);
11    /* 当 Inode 被修改后被调用。*/
12    void (*dirty_inode) (struct inode *);
13    /* 把 Inode 写入到磁盘。*/
14    int (*write_inode) (struct inode *, int);
15    /* 减少 Inode 的引用计数。*/
16    void (*put_inode) (struct inode *);
17    /* 当 Inode 的引用计数为 0 时，用来删除 Inode 在内存中的结构。*/
18    void (*drop_inode) (struct inode *);
19    /* 删除 Inode，包括内存中的 Inode 结构和磁盘上的结构。*/
20    void (*delete_inode) (struct inode *);
21    /* 释放超级块。*/

```

```

22 void (*put_super) (struct super_block *);
23 /* 写磁盘上的超级块。 */
24 void (*write_super) (struct super_block *);
25 /* 把文件系统写入磁盘时，更新文件系统的特点结构(日志文件系统使用，例如 Ext3)。 */
26 int (*sync_fs)(struct super_block *sb, int wait);
27 void (*write_super_lockfs) (struct super_block *);
28 void (*unlockfs) (struct super_block *);
29 /* 获取文件系统信息。 */
30 int (*statfs) (struct dentry *, struct kstatfs *);
31 /*
32  * 重新挂载这个块设备，通常在 shell 中运行 mount 命令，
33  * 并指定 remount 属性时会调用该函数。
34  */
35 int (*remount_fs) (struct super_block *, int *, char *);
36 void (*clear_inode) (struct inode *);
37 void (*umount_begin) (struct vfsmount *, int);
38 /* 获取文件系统属性。 */
39 int (*show_options)(struct seq_file *, struct vfsmount *);
40 int (*show_stats)(struct seq_file *, struct vfsmount *);
41 #ifdef CONFIG_QUOTA
42     ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
43     ssize_t (*quota_write)(struct super_block *,
44                           int, const char *,
45                           size_t, loff_t);
46 #endif
47 };

```

可以看到 VFS 超级块中的 `super_operations` 指针决定了如何操作下层文件系统，因此对于不同的文件系统来说，其 `super_operations` 是不同的。那么又是根据什么来设置 VFS 超级块中的 `s_op` 指针呢？前面我们说过，当内核挂载块设备时，会根据分区表中的信息，分析出文件系统类型，然后找到对应的文件系统对象，并调用它的 `get_sb()` 函数，`get_sb()` 会设置 `s_op` 指针。以 Ext2 为例，这个函数就是 `ext2_get_sb()`，`ext2_get_sb()` 请求磁盘驱动程序把相应的块读取出来后，调用 `ext2_fill_super()` 根据磁盘上的 `ext2_super_block` 结构，初始化内存中的 `ext2_sb_info` 及 VFS 的 `super_block` 结构，同时把 `s_op` 设置为 `ext2_sops`。`ext2_fill_super()` 是一个烦琐的函数，这里我们提取出主要部分。

代码片段 12.13 节自 `fs/ext2/super.c`

```

1 static int ext2_fill_super(struct super_block *sb, void *data, int silent)
2 {
3     /* 参数 sb 指向 VFS 的超级块。 */
4     struct buffer_head * bh;
5

```

```
6  /*
7   * 下面的 sbi 指向内存中的 Ext2 超级块，es 指向从磁盘读取到的 Ext2 超级块4。
8   * 这个函数主要的工作就是根据 es 结构初始化 sbi 和 sb 结构。
9   */
10  struct ext2_sb_info * sbi;
11  struct ext2_super_block * es;
12
13  struct inode *root;
14  unsigned long block;
15  unsigned long sb_block = get_sb_block(&data);
16  unsigned long logic_sb_block;
17  unsigned long offset = 0;
18  unsigned long def_mount_opts;
19
20  /* BLOCK_SIZE 默认为 1KB. */
21  int blocksize = BLOCK_SIZE;
22  int db_count;
23  int i, j;
24  __le32 features;
25  int err;
26
27  /* 分配内存的 ext2_sb_info 结构。*/
28  sbi = kzalloc(sizeof(*sbi), GFP_KERNEL);
29  if (!sbi)
30      return -ENOMEM;
31
32  /* VFS 超级块的 s_fs_info 指向 Ext2 超级块 ext2_sb_info 结构。*/
33  sb->s_fs_info = sbi;
34  /* Ext2 超级块的 s_sb_block 指向 VFS 超级块的 super_block 结构。*/
35  sbi->s_sb_block = sb_block;
36  /*
37   * See what the current blocksize for the device is, and
38   * use that as the blocksize. Otherwise (or if the blocksize
39   * is smaller than the default) use the default.
40   * This is important for devices that have a hardware
41   * sectorsize that is larger than the default.
42   */
43  blocksize = sb_min_blocksize(sb, BLOCK_SIZE);
44  if (!blocksize) {
45      printk ("EXT2-fs: unable to set blocksize\n");
```

⁴es 指向的超级块也被读取到内存中了，但是它和磁盘上的结构一致，而这里说的内存中的 Ext2 超级块是指 ext2_sb_info 结构。请读者注意区分。

```
46     goto failed_sbi;
47 }
48 /*
49  * If the superblock doesn't start on a hardware sector boundary,
50  * calculate the offset.
51 */
52 if (blocksize != BLOCK_SIZE) {
53     logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
54     offset = (sb_block*BLOCK_SIZE) % blocksize;
55 } else {
56     logic_sb_block = sb_block;
57 }
58
59 /* 请求磁盘驱动程序读取超级块。*/
60 if (!(bh = sb_bread(sb, logic_sb_block))) {
61     printk ("EXT2-fs: unable to read superblock\n");
62     goto failed_sbi;
63 }
64 /*
65  * Note: s_es must be initialized as soon as possible because
66  *       some ext2 macro-instructions depend on its value
67  */
68 /* bh->b_data 指向读取缓冲区的首地址, offset 是超级块的偏移。*/
69 es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
70
71 /* Ext 内存超级块结构的 s_es 指向 Ext2 磁盘超级块 es (现在这个结构在内存中)。*/
72 sbi->s_es = es;
73 sb->s_magic = le16_to_cpu(es->s_magic);
74
75 if (sb->s_magic != EXT2_SUPER_MAGIC)
76     goto cantfind_ext2;
77 .....
78 /* 根据磁盘上的 s_log_block_size 计算逻辑块大小 (1024 * 2s)。*/
79 blocksize = BLOCK_SIZE << le32_to_cpu(sbi->s_es->s_log_block_size);
80
81 if ((ext2_use_xip(sb)) && ((blocksize != PAGE_SIZE) ||
82     (sb->s_blocksize != blocksize))) {
83     if (!silent)
84         printk("XIP: Unsupported blocksize\n");
85     goto failed_mount;
86 }
87 .....
```

```
88  /* 片大小，当前没有实现分片，片大小等于块大小。*/
89  sbi->s_frag_size = EXT2_MIN_FRAG_SIZE <<
90  1e32_to_cpu(es->s_log_frag_size);
91  if (sbi->s_frag_size == 0)
92  goto cantfind_ext2;
93  sbi->s frags_per_block = sb->s_blocksize / sbi->s_frag_size;
94  /* 每个组的块个数。*/
95  sbi->s_blocks_per_group = 1e32_to_cpu(es->s_blocks_per_group);
96  /* 每个组的片个数。*/
97  sbi->s frags_per_group = 1e32_to_cpu(es->s frags_per_group);
98  /* 每个组的 Inode 个数。*/
99  sbi->s_inodes_per_group = 1e32_to_cpu(es->s_inodes_per_group);
100 if (EXT2_INODE_SIZE(sb) == 0)
101  goto cantfind_ext2;
102 /* 一个块中的 Inode 数量为块大小除以 Inode 大小。*/
103 sbi->s_inodes_per_block = sb->s_blocksize / EXT2_INODE_SIZE(sb);
104 if (sbi->s_inodes_per_block == 0 ||
105     sbi->s_inodes_per_group == 0)
106  goto cantfind_ext2;
107 /*
108  * 一个组的 Inode 数量除以一个块的 Inode 数量,
109  * 就得到一个组中有几个块是用来存储 Inode 的。
110 */
111 sbi->s_itb_per_group = sbi->s_inodes_per_group /
112  sbi->s_inodes_per_block;
113 /* 块大小除以组描述符大小，得到一个块中最多有几个组描述符。*/
114 sbi->s_desc_per_block = sb->s_blocksize/sizeof(struct ext2_group_desc);
115 .....
116 /* 由于还没有实现分片，如果片大小不等于块大小就报错。*/
117 if (sb->s_blocksize != sbi->s_frag_size) {
118  printk ("EXT2-fs: fragsize %lu != blocksize %lu (not supported yet)\n",
119         sbi->s_frag_size, sb->s_blocksize);
120  goto failed_mount;
121 }
122 .....
123 if (EXT2_BLOCKS_PER_GROUP(sb) == 0)
124  goto cantfind_ext2;
125 /*
126  * 计算当前分区组的个数。前面已经讨论过，组的个数由分区大小和块大小决定,
127  * 见第385页。这里需要处理分区中块的边界不能凑成一个组的情况。
128 */
129 sbi->s_groups_count = ((le32_to_cpu(es->s_blocks_count) -
```

```
130         le32_to_cpu(es->s_first_data_block) - 1) /  
131             EXT2_BLOCKS_PER_GROUP(sb)) + 1;  
132 db_count = (sbi->s_groups_count + EXT2_DESC_PER_BLOCK(sb) - 1) /  
133             EXT2_DESC_PER_BLOCK(sb);  
134 /* 为组描述符分配 buffer_head 结构。 */  
135 sbi->s_group_desc = kmalloc (db_count * sizeof(struct buffer_head *),  
136                                     GFP_KERNEL);  
137 if (sbi->s_group_desc == NULL) {  
138     printk ("EXT2-fs: not enough memory\n");  
139     goto failed_mount;  
140 }  
141 bgl_lock_init(&sbi->s_blockgroup_lock);  
142 sbi->s_debts = kcalloc(sbi->s_groups_count,  
143                         sizeof(*sbi->s_debts),  
144                         GFP_KERNEL);  
145 if (!sbi->s_debts) {  
146     printk ("EXT2-fs: not enough memory\n");  
147     goto failed_mount_group_desc;  
148 }  
149 /* 请求磁盘驱动程序，把组描述符读取出来，并设置对应的 s_group_desc 数组中的指针。 */  
150 for (i = 0; i < db_count; i++) {  
151     /*  
152      * 根据超级块中的 s_first_data_block, 块大小, 以及组描述符大小,  
153      * 计算第 i 个组描述符的逻辑块号。  
154      */  
155     block = descriptor_loc(sb, logic_sb_block, i);  
156     /* 请求磁盘驱动程序读取第 block 块。 */  
157     sbi->s_group_desc[i] = sb_bread(sb, block);  
158     /* 一个块中包含了多个组描述符。 */  
159     if (!sbi->s_group_desc[i]) {  
160         for (j = 0; j < i; j++)  
161             brelse (sbi->s_group_desc[j]);  
162         printk ("EXT2-fs: unable to read group descriptors\n");  
163         goto failed_mount_group_desc;  
164     }  
165 }  
166 if (!ext2_check_descriptors (sb)) {  
167     printk ("EXT2-fs: group descriptors corrupted!\n");  
168     goto failed_mount2;  
169 }  
170 sbi->s_gdb_count = db_count;  
171 .....
```

```

172  /*
173   * set up enough so that it can read an inode
174   */
175 /* 设置 VFS 超级块的 super_operations 指针为 ext2_sops. */
176 sb->s_op = &ext2_sops;
177 sb->s_export_op = &ext2_export_ops;
178 sb->s_xattr = ext2_xattr_handlers;
179 /* 获取根目录的 inode. */
180 root = iget(sb, EXT2_ROOT_INO);
181 /* 初始化根的目录结构.*/
182 sb->s_root = d_alloc_root(root);
183 if (!sb->s_root) {
184     iput(root);
185     printk(KERN_ERR "EXT2-fs: get root inode failed\n");
186     goto failed_mount3;
187 }
188 if (!S_ISDIR(root->i_mode) ||
189     !root->i_blocks ||
190     !root->i_size) {
191     dput(sb->s_root);
192     sb->s_root = NULL;
193     printk(KERN_ERR "EXT2-fs: corrupt root inode, run e2fsck\n");
194     goto failed_mount3;
195 }
196 if (EXT2_HAS_COMPAT_FEATURE(sb, EXT3_FEATURE_COMPAT_HAS_JOURNAL))
197     ext2_warning(sb, __FUNCTION__, "mounting ext3 filesystem as ext2");
198 ext2_setup_super (sb, es, sb->s_flags & MS_RDONLY);
199 return 0;
200
201 .....
202 return -EINVAL;
203 }

```

这个函数很长，但是并不复杂，在最后的第176行，将 VFS 超级块的 super_operations 设置为 ext2_sops，这样就建立了抽象的 VFS 超级块对象和具体的 ext2 超级块对象之间的关联。ext2_sops 定义如下：

代码片段 12.14 节自 fs/ext2/super.c

```

1 static const struct super_operations ext2_sops = {
2     .alloc_inode = ext2_alloc_inode,
3     .destroy_inode = ext2_destroy_inode,
4     .read_inode = ext2_read_inode,
5     .write_inode = ext2_write_inode,

```

```

6   .delete_inode = ext2_delete_inode,
7   .put_super = ext2_put_super,
8   .write_super = ext2_write_super,
9   .statfs = ext2_statfs,
10  .remount_fs = ext2_remount,
11  .clear_inode = ext2_clear_inode,
12  .show_options = ext2_show_options,
13 #ifdef CONFIG_QUOTA
14  .quota_read = ext2_quota_read,
15  .quota_write = ext2_quota_write,
16 #endif
17 };

```

可以看到 ext2_sops 中，只设置了一部分函数指针。这些函数的作用前面已经讨论过了。我们不能深入讨论每一个函数，这里以 ext2_read_inode() 为例，它从磁盘上读取一个指定的 Inode，然后设置 VFS 的 Inode 和 Ext2 文件系统的内存 Inode 结构。从前面的讨论中我们看到，ext2_fill_super() 函数的最后会请求读取根目录的 Inode，这个工作是由 iget() 函数完成的，iget() 会调用 ext2_read_inode() 请求从磁盘上读取根目录的 Inode。

代码片段 12.15 节自 fs/ext2/super.c

```

1 static int ext2_fill_super(struct super_block *sb, void *data, int silent)
2 {
3     struct inode *root;
4     .....
5     sb->s_op = &ext2_sops;
6     root = iget(sb, EXT2_ROOT_INO);
7     .....
8 }

```

iget 函数定义如下：

代码片段 12.16 节自 fs/ext2/super.c

```

1 static inline struct inode *iget(struct super_block *sb, unsigned long ino)
2 {
3     struct inode *inode = iget_locked(sb, ino);
4     if (inode && (inode->i_state & I_NEW)) {
5         sb->s_op->read_inode(inode);
6         unlock_new_inode(inode);
7     }
8     return inode;
9 }

```

`iget()`函数从超级块 `sb` 指定的分区中读取出编号为 `ino` 的 `Inode` 结构。并根据磁盘上的 `Inode` 结构，初始化 VFS 的 `Inode` 结构和 Ext2 内存 `Inode` 结构，返回 VFS 的 `Inode` 结构。由于根据 `Inode` 号取 `Inode` 结构是一个频繁的操作，因此对于已经读取出来的 `Inode` 结构，内核根据 `ino` 把它们保存在 `hash` 链表中。`iget()` 函数首先调用 `iget_locked()` 从 `hash` 缓存中查找 `ino` 指定的 `Inode` 结构，如果找到了，就不需要读取磁盘上的 `inode` 结构了。如果没有找到，`iget_locked()` 会分配一个新的 `inode` 结构，这种情况下它的 `i_state` 被设置为 `I_NEW`。在本例中，是在 `mount` 根目录时候调用的，所以在 `hash` 缓存中肯定找不到。于是 `iget_locked()` 会分配一个新的 `inode` 结构。需要注意的是，在 12.2 节中，我们看到 `ext2_inode_info` 结构中包含了 VFS `inode` 结构。因此 `iget_locked` 返回后，VFS `inode` 结构和 Ext2 内存 `inode` 结构都已经分配好了。于是根据超级块的 `s_op` 调用具体文件系统的 `read_inode()`。对于 Ext2 来说，这个函数就是 `ext2_read_inode()`。

代码片段 12.17 节自 `fs/ext2/inode.c`

```

1 void ext2_read_inode (struct inode * inode)
2 {
3     /* 根据 VFS inode 获取 ext2_inode_info 结构。*/
4     struct ext2_inode_info *ei = EXT2_I(inode);
5     /* 获取 inode 号。*/
6     ino_t ino = inode->i_ino;
7     struct buffer_head * bh;
8     /* 请求磁盘驱动程序读取指定的 inode 结构，保存在 raw_inode 中。*/
9     struct ext2_inode * raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);
10    int n;
11    .....
12    /* 根据 raw_inode 设置 VFS inode 和 Ext2 内存 inode. */
13    inode->i_mode = le16_to_cpu(raw_inode->i_mode);
14    inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
15    inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
16    if (! (test_opt (inode->i_sb, NO_UID32))) {
17        inode->i_uid |= le16_to_cpu(raw_inode->i_uid_high) << 16;
18        inode->i_gid |= le16_to_cpu(raw_inode->i_gid_high) << 16;
19    }
20    inode->i_nlink = le16_to_cpu(raw_inode->i_links_count);
21    inode->i_size = le32_to_cpu(raw_inode->i_size);
22    inode->i_atime.tv_sec = (signed)le32_to_cpu(raw_inode->i_atime);
23    inode->i_ctime.tv_sec = (signed)le32_to_cpu(raw_inode->i_ctime);
24    inode->i_mtime.tv_sec = (signed)le32_to_cpu(raw_inode->i_mtime);
25    .....
26    /*
27     * NOTE! The in-memory inode i_data array is in little-endian order
28     * even on big-endian machines: we do NOT byteswap the block numbers!

```

```
29      */
30     for (n = 0; n < EXT2_N_BLOCKS; n++)
31         ei->i_data[n] = raw_inode->i_block[n];
32     /* 设置 i_op 和 i_fop 指针。*/
33     /* 普通文件。*/
34     if (S_ISREG(inode->i_mode)) {
35         inode->i_op = &ext2_file_inode_operations;
36         if (ext2_use_xip(inode->i_sb)) {
37             inode->i_mapping->a_ops = &ext2_aops_xip;
38             inode->i_fop = &ext2_xip_file_operations;
39         } else if (test_opt(inode->i_sb, NOBH)) {
40             inode->i_mapping->a_ops = &ext2_nobh_aops;
41             inode->i_fop = &ext2_file_operations;
42         } else {
43             inode->i_mapping->a_ops = &ext2_aops;
44             inode->i_fop = &ext2_file_operations;
45         }
46     /* 目录。*/
47     } else if (S_ISDIR(inode->i_mode)) {
48         inode->i_op = &ext2_dir_inode_operations;
49         inode->i_fop = &ext2_dir_operations;
50         if (test_opt(inode->i_sb, NOBH))
51             inode->i_mapping->a_ops = &ext2_nobh_aops;
52         else
53             inode->i_mapping->a_ops = &ext2_aops;
54     /* 符号链接。*/
55     } else if (S_ISLNK(inode->i_mode)) {
56         /* fast symbol link. 目标路径小于 60 个字符。*/
57         if (ext2_inode_is_fast_symlink(inode))
58             inode->i_op = &ext2_fast_symlink_inode_operations;
59         else {
60             /* 目标路径大于 60 个字符的符号链接。*/
61             inode->i_op = &ext2_symlink_inode_operations;
62             if (test_opt(inode->i_sb, NOBH))
63                 inode->i_mapping->a_ops = &ext2_nobh_aops;
64             else
65                 inode->i_mapping->a_ops = &ext2_aops;
66         }
67     /* 特殊文件系统。*/
68     } else {
69         inode->i_op = &ext2_special_inode_operations;
70         if (raw_inode->i_block[0])
```

```

71     init_special_inode(inode, inode->i_mode,
72         old_decode_dev(le32_to_cpu(raw_inode->i_block[0])));
73     else
74         init_special_inode(inode, inode->i_mode,
75             new_decode_dev(le32_to_cpu(raw_inode->i_block[1])));
76     }
77     brelse (bh);
78     ext2_set_inode_flags(inode);
79     return;
80
81 bad_inode:
82     make_bad_inode(inode);
83     return;
84 }
```

这个函数在第9行调用 `ext2_get_inode()` 函数读取磁盘上的 `inode` 结构, `ext2_get_inode()` 首先根据 `inode` 号和每个组的 `inode` 个数(保存在超级块中的 `s_inodes_per_group` 成员), 就可以计算出这个 `inode` 所在的组(全部的组描述符已经在 `ext2_fill_super()` 函数中读入内存, 并初始化完毕), 然后根据 `inode` 编号, 组中保存 `inode` 的首块号(保存在组描述符中的 `bg_inode_table` 成员), 以及块大小和 `inode` 大小, 计算出包含这个 `inode` 的块号和块内偏移, 最后请求磁盘驱动程序读取这个块。这个函数比较简单, 读者可以自行分析。

在 `ext2_fill_super()` 函数返回后, `ext2_get_inode()` 的工作就是根据读取出来的信息设置 VFS `inode` 和 Ext2 内存 `inode` 结构了。这些结构前面已经讨论过了, 直到第34行开始, 它设置 VFS 结构中的另外两个重要的成员, `i_op` 和 `i_fop`, 和 VFS 超级块中的 `s_op` 指针一样, `i_op` 和 `i_fop` 都是具体文件系统的虚函数表, 它们决定着 VFS 如何操作下层的具体文件系统。我们将在 VFS Inode 的讨论中进一步讲述这两个函数指针。

12.3.3 VFS 的 `inode` 结构

和超级块一样, 不同文件系统的 `inode` 结构是有差别的, 因此对它们的操作也是有差别的。VFS `inode` 的作用就是隐藏各种具体文件系统的 `inode` 的差别, 向上层软件提供一个统一的 `inode` 接口。VFS 的 `inode` 定义如下:

代码片段 12.18 节自 `include/linux/fs.h`

```

1 struct inode {
2     struct hlist_node i_hash;
3     struct list_head i_list;
4     struct list_head i_sb_list;
5     struct list_head i_dentry;
6 }
```

```
7  /* inode 编号。*/
8  unsigned long    i_ino;
9
10 /* 引用计数。*/
11 atomic_t        i_count;
12 unsigned int    i_nlink;
13 uid_t          i_uid;
14 gid_t          i_gid;
15 dev_t          i_rdev;
16 unsigned long   i_version;
17 loff_t         i_size;
18 #ifdef __NEED_I_SIZE_ORDERED
19     seqcount_t   i_size_seqcount;
20#endif
21     struct timespec  i_atime;
22     struct timespec  i_mtime;
23     struct timespec  i_ctime;
24     .....
25     struct list_head i_devices;
26
27 /* 为特殊文件的 inode 设置的，例如字符设备。*/
28 union {
29     struct pipe_inode_info *i_pipe;
30     struct block_device *i_bdev;
31     struct cdev    *i_cdev;
32 };
33     int      i_cindex;
34     .....
35 /* fs or device private pointer */
36     void    *i_private;
37 };
```

这个结构中的大部分成员都是根据磁盘上的 inode 结构初始化的，它们所代表的意义都一样，在这里仅仅对几个重要的结构进行说明。

根据 inode 编号获取对应的 inode 结构是一个很频繁的操作，因此内核使用了 hash 表，每一个 inode 都通过 i_hash 链接在对应的 hash 链表中。

每一个 inode 可能唯一地出现在以下的链表中，通过 i_list 成员链接。它们分别是：

- inode_unused, 位于这条链表上的 inode，已经不再被任何进程使用，但是还没有被释放，以后如果某个进程需要读的 inode 在 inode_unused 中，就可以直接使用。
- inode_in_use, 位于这条链表上的 inode，正在被某些进程使用，但是内存中的结构和

磁盘上对应的结构一致，也就是说，如果当进程不再使用这个 inode 时，如果有必要，可以直接释放，不用考虑内存中的 inode 和磁盘上的 inode 结构的同步问题。

- dirty，链接在对应超级块的 `s_dirty` 链表中，这些 inode 结构和磁盘上的 inode 结构不一致，适当的时候，需要调用超级块中 `s_op` 的 `write_inode()` 函数，把 inode 写入磁盘。

在 `ext2_read_inode()` 函数中，我们看到每当从磁盘上读取出一个 inode 结构时，会设置 VFS inode 的 `i_op` 和 `i_fop` 指针，其中 `i_op` 包含了一组操作具体文件系统的 inode 的函数。在 12.1 节中我们看到，从磁盘格式的角度来看，目录和文件是没有区别的，但是 VFS 提供给上层的程序接口是区分文件和目录的。所以 `ext2_read_inode()` 最后要根据文件的类别为 `i_op` 设置不同的 `inode_operations`。目前分别有以下 4 种情况。

1. 普通文件的 `inode_operations` 结构

普通 Ext2 文件的 `inode_operations` 虚函数表是 `ext2_file_inode_operations`，其定义如下：

代码片段 12.19 节自 `fs/ext2/file.c`

```

1 const struct inode_operations ext2_file_inode_operations = {
2     .truncate = ext2_truncate,
3 #ifdef CONFIG_EXT2_FS_XATTR
4     .setxattr = generic_setxattr,
5     .getxattr = generic_getxattr,
6     .listxattr = ext2_listxattr,
7     .removexattr = generic_removexattr,
8 #endif
9     .setattr = ext2_setattr,
10    .permission = ext2_permission,
11 };

```

其中 `ext2_truncate()` 函数的作用是修改一个文件的大小，这主要是需要修改磁盘 inode 的 `size` 等成员。`ext2_permission()` 函数是用来检测权限的，例如当前用户是否能够对这个 inode 进行某项操作。为什么没有看到分配磁盘 inode 等操作呢？在磁盘上分配一个 inode 的操作由 `ext2_new_inode()` 函数完成。别着急，继续往下看，很快就会见分晓。

2. 目录的 `inode_operations` 结构

目录的 `inode_operations` 虚函数表是 `ext2_dir_inode_operations`，其定义如下：

代码片段 12.20 节自 `fs/ext2/namei.c`

```

1 const struct inode_operations ext2_dir_inode_operations = {

```

```

2   .create    = ext2_create,
3   .lookup    = ext2_lookup,
4   .link      = ext2_link,
5   .unlink    = ext2_unlink,
6   .symlink   = ext2_symlink,
7   .mkdir     = ext2_mkdir,
8   .rmdir     = ext2_rmdir,
9   .mknod    = ext2_mknod,
10  .rename    = ext2_rename,
11 #ifdef CONFIG_EXT2_FS_XATTR
12  .setxattr  = generic_setxattr,
13  .getxattr  = generic_getxattr,
14  .listxattr = ext2_listxattr,
15  .removexattr = generic_removexattr,
16 #endif
17  .setattr   = ext2_setattr,
18  .permission = ext2_permission,
19 };

```

在12.1节中我们看到，普通文件和目录都有一个对应的 inode 结构，如果要在某个目录下建立一个文件，会调用目录对应的 inode 结构的 ext2_create()函数，ext2_create()函数会调用 ext2_new_inode()从磁盘上分配一个空闲的 inode，同时初始化 Ext2 内存中的 inode 结构。

同样 ext2_mkdir()是在当前目录下建立一个子目录，此时 ext2_mkdir()也会调用 ext2_new_inode()函数在磁盘上为子目录分配一个空闲的 inode，并初始化。ext2_mknod()负责在这个目录下建立一个设备节点，它也会调用 ext2_new_inode()函数。ext2_link()和 ext2_symlink()分别用于建立硬链接和软链接(见第12.1)。

3. Fast symbol link 的 inode_operations 结构

对于一个普通文件的定位步骤是，找到它的目录项，然后从目录项中定位到这个文件的 inode 号，对于硬链接来说，可以直接定位到目标文件的 inode 号(见第12.1)，对于软链接文件则需要特殊处理，它在这里定位到的是链接文件本身的 inode，此时需要根据这个 inode 号，读取目标文件的路径，然后根据这个路径重新定位目标文件。如果目标文件的路径小于 60 个字节，这被称为 Fast symbol link，此时这个路径可以保存在链接文件 inode 的 i_block 数组中，此时不需要使用数据块。它的 inode_operations 定义如下：

代码片段 12.21 节自 fs/ext2/symlink.c

```

1 const struct inode_operations ext2_fast_symlink_inode_operations = {
2   .readlink = generic_readlink,

```

```

3   .follow_link = ext2_follow_link,
4 #ifdef CONFIG_EXT2_FS_XATTR
5   .setxattr = generic_setxattr,
6   .getxattr = generic_getxattr,
7   .listxattr = ext2_listxattr,
8   .removexattr = generic_removexattr,
9 #endif
10 };

```

这里的 `readlink()` 函数的作用是到 `i_block` 数组中读出目标文件的路径，而 `follow_link()` 则是把对链接文件的操作（打开）直接转到目标文件上。

4. 普通符号链接的 `inode_operations` 结构

普通符号链接的目标文件路径大于 60，需要根据 `i_block` 数组中的数据块地址读取目标文件的路径。它的 `inode_operations` 结构定义如下：

代码片段 12.22 节自 `fs/ext2/symlink.c`

```

1 const struct inode_operations ext2_symlink_inode_operations = {
2   .readlink = generic_readlink,
3   .follow_link = page_follow_link_light,
4   .put_link = page_put_link,
5 #ifdef CONFIG_EXT2_FS_XATTR
6   .setxattr = generic_setxattr,
7   .getxattr = generic_getxattr,
8   .listxattr = ext2_listxattr,
9   .removexattr = generic_removexattr,
10 #endif
11 };

```

它比 `fast symbol link` 多了一个 `put_link()` 函数，这是由于普通符号链接需要使用数据块，因此 `follow_link()` 函数需要把数据块中的内容读取出来，放到一个内存中，当处理完成时，`put_link()` 负责释放内存。

讨论完了 VFS `inode` 的 `i_op` 后，现在我们来看看它的 `i_fop`，`i_op` 指定一组对 `inode` 的操作函数，而 `i_fop` 则指定了对文件内容本身操作的函数。在前面 `ext2_read_inode()` 函数的分析中我们看到，文件和目录的 `i_fop` 也是不一样的，Ext2 文件的 `i_fop` 定义如下：

代码片段 12.23 节自 `fs/ext2/file.c`

```

1 const struct file_operations ext2_file_operations = {
2   .llseek = generic_file_llseek,
3   .read = do_sync_read,

```

```

4     .write      = do_sync_write,
5     .aio_read   = generic_file_aio_read,
6     .aio_write   = generic_file_aio_write,
7     .ioctl      = ext2_ioctl,
8 #ifdef CONFIG_COMPAT
9     .compat_ioctl = ext2_compat_ioctl,
10 #endif
11    .mmap      = generic_file_mmap,
12    .open       = generic_file_open,
13    .release    = ext2_release_file,
14    .fsync      = ext2_sync_file,
15    .splice_read = generic_file_splice_read,
16    .splice_write = generic_file_splice_write,
17 };

```

其中 llseek()的作用是调整文件读写指针。read()和 write()分别是同步读写指针，当进程进行读时，如果数据不在缓冲区内，内核向磁盘驱动程序发送这个请求，同时阻塞当前进程。aio_read()和 aio_write()函数分别是异步读写函数，如果一个进程调用异步读函数，即便是数据不在缓冲区内，也可以不阻塞当前进程，立即返回，将来数据准备好之后，通过其他方式通知该进程。ioctl()用来向设备文件发送 ioctl 命令。mmap()函数把一个文件的内容映射到进程的虚拟地址空间中(利用页表)，这样就可以通过内存指针来访问文件内容，假设一个大小为 N 个字节的文件被映射到进程的起始虚拟地址为 p 的地方，以后可以通过 p[n]来访问文件内容，当 p[n]不在物理内存中时，会触发缺页中断，此时缺页中断会把对应的内容从磁盘上读取出来，并建立相关映射。open()打开文件操作函数，这个函数会建立相关的内存管理结构，如 inode 对象等。release()减少文件的引用计数，当引用计数为 0 时，会关闭文件，同时释放相关管理结构。fsync()把内存中的内容写入磁盘。splice_read()和 splice_write()用于管道操作。

Ext2 目录的 i_fop 结构为 ext2_dir_operations，其定义如下：

代码片段 12.24 节自 fs/ext2/dir.c

```

1 const struct file_operations ext2_dir_operations = {
2     .llseek    = generic_file_llseek,
3     .read      = generic_read_dir,
4     .readdir   = ext2_readdir,
5     .ioctl     = ext2_ioctl,
6 #ifdef CONFIG_COMPAT
7     .compat_ioctl = ext2_compat_ioctl,
8 #endif
9     .fsync      = ext2_sync_file,
10 };

```

这里的 `generic_read_dir()` 是一个空函数，`ext2_readdir()` 从目录的数据块中把目录项读取出来，其他的函数和前面的文件操作类似。

每当初始化一个 `inode` 时，设置好它的 `i_op` 和 `i_fop` 后，就能够正确地找到对这个文件的操作函数了。我们将在以后进一步对这些函数进行分析。

12.3.4 VFS 的文件对象

无论下层的具体文件系统的差异如何，VFS 通过 `file` 结构向上层提供一个统一的文件对象，VFS 的文件对象定义如下：

代码片段 12.25 节自 `include/linux/fs.h`

```

1 struct file {
2     union {
3         struct list_head fu_list;
4         struct rcu_head fu_rcuhead;
5     } f_u;
6     struct path f_path;
7 #define f_dentry f_path.dentry
8 #define f_vfsmnt f_path.mnt
9
10    /* 文件操作函数指针。*/
11    const struct file_operations *f_op;
12    /* 引用计数。*/
13    atomic_t f_count;
14    unsigned int f_flags;
15    mode_t f_mode;
16    /* 文件的读写指针。*/
17    loff_t f_pos;
18    struct fown_struct f_owner;
19    unsigned int f_uid, f_gid;
20    struct file_ra_state f_ra;
21
22    u64 f_version;
23    .....
24    struct address_space *f_mapping;
25 };

```

这个结构中的大部分成员都比较容易理解，其中最重要的是 `f_op` 指针，它指向一个 `file_operations` 结构，这个结构定义如下：

代码片段 12.26 节自 `include/linux/fs.h`

```

1 struct file_operations {

```

```
2 struct module *owner;
3 /* 调整文件读写指针位置。*/
4 loff_t (*llseek) (struct file *, loff_t, int);
5 /* 文件读操作函数。*/
6 ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
7 /* 文件写操作函数。*/
8 ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
9 /* 文件异步读操作函数。*/
10 ssize_t (*aio_read) (struct kiocb *,
11                      const struct iovec *,
12                      unsigned long, loff_t);
13 /* 文件异步写操作函数。*/
14 ssize_t (*aio_write) (struct kiocb *,
15                      const struct iovec *,
16                      unsigned long, loff_t);
17 /* 读取目录操作。*/
18 int (*readdir) (struct file *, void *, filldir_t);
19 /* 在一个文件上查询一个指定的事件，当指定的事件发生时，唤醒相关进程。*/
20 unsigned int (*poll) (struct file *, struct poll_table_struct *);
21 /* 向文件发送 IOCTL 命令，通常用于设备文件。*/
22 int (*ioctl) (struct inode *,
23               struct file *,
24               unsigned int,
25               unsigned long);
26 long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
27 /* 64 位内核调用 32 位的 ioctl。*/
28 long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
29 /* 建立文件映射。*/
30 int (*mmap) (struct file *, struct vm_area_struct *);
31 /* 打开文件。*/
32 int (*open) (struct inode *, struct file *);
33 /* 刷新磁盘文件，将内存写入磁盘。*/
34 int (*flush) (struct file *, fl_owner_t id);
35 /* 减少文件引用计数，当引用计数为 0 时，释放文件。*/
36 int (*release) (struct inode *, struct file *);
37 /* 把文件在内存中的内容写入磁盘。*/
38 int (*fsync) (struct file *, struct dentry *, int datasync);
39 /* 把文件在内存中的内容写入磁盘，异步操作。*/
40 int (*aio_fsync) (struct kiocb *, int datasync);
41 /* 开启或者关闭文件的某些异步事件。*/
42 int (*fasync) (int, struct file *, int);
43 /* 锁定一个文件。*/
```

```
44     int (*lock) (struct file *, int, struct file_lock *);  
45     /* 把文件的内容传送到 page 中去。 */  
46     ssize_t (*sendpage) (struct file *,  
47                           struct page *,  
48                           int, size_t,  
49                           loff_t *, int);  
50     /*  
51      * 获取一个映射文件的未映射区域，映射文件并不是一次性为整个文件内容  
52      * 建立内存映射，而是利用缺页中断实现“延迟读”。  
53      */  
54     unsigned long (*get_unmapped_area) (struct file *,  
55                                         unsigned long,  
56                                         unsigned long,  
57                                         unsigned long,  
58                                         unsigned long);  
59     /* 用来检测文件的某些标志，例如 fcntl() 系统调用。 */  
60     int (*check_flags) (int);  
61     /*  
62      * 通过 fcntl() 系统调用，可以要求一个目录改变时向应用程序发出通知。  
63      * 例如 Common Internet File System (CIFS) network 文件系统会使用这个机制。  
64      */  
65     int (*dir_notify) (struct file *filp, unsigned long arg);  
66     int (*flock) (struct file *, int, struct file_lock *);  
67     /* 用于管道操作。 */  
68     ssize_t (*splice_write) (struct pipe_inode_info *,  
69                             struct file *,  
70                             loff_t *,  
71                             size_t,  
72                             unsigned int);  
73     ssize_t (*splice_read) (struct file *,  
74                             loff_t *,  
75                             struct pipe_inode_info *,  
76                             size_t,  
77                             unsigned int);  
78     int (*setlease) (struct file *, long, struct file_lock **);  
79 };
```

由于 VFS 的 file 对象要顾及到各种文件系统，因此就某个具体文件系统而言，其 file_operations 结构中的函数指针难免是多余的，例如某些函数指针是为网络文件系统设置的，因此具体文件系统的 file_operation 结构一般都没这么复杂。比如 Ext2 的 file_operations 结构定义如下：

代码片段 12.27 节自 fs/ext2/file.c

```

1 const struct file_operations ext2_file_operations = {
2     .llseek    = generic_file_llseek,
3     .read      = do_sync_read,
4     .write     = do_sync_write,
5     .aio_read  = generic_file_aio_read,
6     .aio_write = generic_file_aio_write,
7     .ioctl     = ext2_ioctl,
8 #ifdef CONFIG_COMPAT
9     .compat_ioctl = ext2_compat_ioctl,
10 #endif
11    .mmap     = generic_file_mmap,
12    .open      = generic_file_open,
13    .release   = ext2_release_file,
14    .fsync     = ext2_sync_file,
15    .splice_read = generic_file_splice_read,
16    .splice_write = generic_file_splice_write,
17 };

```

从这里可以看到 VFS 的 file 对象中的 f_op 和 VFS 的 inode 对象中的 i_fop 是一致的，inode 中的 i_fop 指针是在 ext2_read_inode() 函数中设置的。同理 file 对象中的 f_op 指针肯定也是在 file 对象初始化的时候设置的，这个过程我们留到以后分析。

12.3.5 VFS 的目录对象

在 VFS 中，每一个目录项都对应一个 dentry⁵ 对象，这个结构是下层具体文件系统目录项的抽象，dentry 结构定义在 include/linux/dcache.h 文件中，其定义如下：

代码片段 12.28 节自 include/linux/dcache.h

```

1 struct dentry {
2     /* 引用计数。*/
3     atomic_t d_count;
4     /* protected by d_lock */
5     unsigned int d_flags;
6     /* per dentry lock */
7     spinlock_t d_lock;
8     /* 目录对应的 inode. */
9     /* Where the name belongs to - NULL is * negative */
10    struct inode *d_inode;
11    /*

```

⁵Directory Entry.

```
12     * The next three fields are touched by __d_lookup. Place them here
13     * so they all fit in a cache line. .
14     */
15     /* lookup hash list */
16     struct hlist_node d_hash;
17     /* parent directory */
18     struct dentry *d_parent;
19     /* 目录名。*/
20     struct qstr d_name;
21     /* LRU list */
22     struct list_head d_lru;
23     /*
24      * d_child and d_rcu can share memory
25      */
26     union {
27         /* child of parent list */
28         struct list_head d_child;
29         struct rcu_head d_rcu;
30     } d_u;
31     /* our children */
32     struct list_head d_subdirs;
33     /* inode alias list */
34     struct list_head d_alias;
35     /* used by d_revalidate */
36     unsigned long d_time;
37     /* 目录操作函数指针。*/
38     struct dentry_operations *d_op;
39     /* The root of the dentry tree */
40     struct super_block *d_sb;
41     /* fs-specific data */
42     void *d_fsbdata;
43 #ifdef CONFIG_PROFILING
44     /* cookie, if any */
45     struct dcookie_struct *d_cookie;
46 #endif
47     int d_mounted;
48     /* small names */
49     unsigned char d_iname[DNAME_INLINE_LEN_MIN];
50 };
```

通常的文件路径是通过目录树来组织的，例如 /bin/bash，首先根目录/对应一个 dentry 对象，然后是 bin 目录对应一个 dentry 对象，每个目录项的名字保存在 d_name 成员中。这

些目录通过 `d_child` 链表组织成树形结构，例如根目录下的所有目录的 `dentry` 都链接在根目录中的 `d_child` 链表中，同时各个 `dentry` 的 `d_parent` 成员指向父目录的 `dentry` 对象。需要注意的是不要认为只有目录才有 `dentry` 对象，`dentry` 表示的是目录项，也就是目录中的条目，因此 `bash` 同样对应一个 `dentry` 对象。

从 VFS 的层面来看，我们以 `/bin/bash` 为例，来看看如何根据一个路径定位一个文件过程。首先需要定位到 `/` 的 `dentry` 对象，这个对象是在 `mount` 根目录时建立的，然后从它的 `d_child` 中找到 `bin` 目录的 `dentry` 对象，最后从它的 `d_child` 中定位到 `bash` 的 `dentry` 对象。如果没有找到一个目录项的 `dentry` 对象，则需要尝试从磁盘上读取这个目录项。在这个过程中，根据目录名的字符串比较是很频繁的操作，假设 `bin` 目录项 `dentry` 对象的 `d_child` 链上有 500 个 `dentry` 对象，那么从这 500 个 `dentry` 对象中挑选出 `bash` 的 `dentry` 对象，需要通过多次字符串比较。为了加快速度，就根据目录名对它们进行 hash 处理。

各个具体文件系统对目录的操作可能不一样，因此 `dentry` 中的 `d_op` 就指向底层文件系统的目录操作函数表。这是一个 `dentry_operations` 结构，其定义如下：

代码片段 12.29 节自 `include/linux/dcache.h`

```
1 struct dentry_operations {
2     /* 在使用一个 dentry 前判断 dentry 是否有效，通常网络文件系统使用。 */
3     int (*d_revalidate) (struct dentry *, struct nameidata *);
4     /* 根据名字计算 hash 值，各个文件系统可以采用不同的 hash 算法。 */
5     int (*d_hash) (struct dentry *, struct qstr *);
6     /*
7      * 名字比较，通常这就是一个字符串比较，但是对于不同的文件系统可
8      * 能采取不同的比较算法，例如 MS DOS 可能忽略大小写。
9      */
10    int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
11    /* 引用计数为 0 时，删除 dentry 对象。 */
12    int (*d_delete) (struct dentry *);
13    /* 减小引用计数。 */
14    void (*d_release) (struct dentry *);
15    /* 释放目录项的 inode。 */
16    void (*d_iput) (struct dentry *, struct inode *);
17    /* 为特殊文件系统构造路径。 */
18    char *(*d_dname) (struct dentry *, char *, int);
19 }
```

当 `dentry_operations` 为 `NULL` 时，意味着需要按照 Linux 默认的 `dentry` 操作方式，Ext2 使用默认的 `dentry` 操作，所以它的 `dentry_operations` 为 `NULL`。

12.3.6 VFS 在进程中的文件结构

前面讨论了各种管理结构，这么多的管理结构，现在我们来看看这些管理结构之间的整体关系，如图12.4所示。

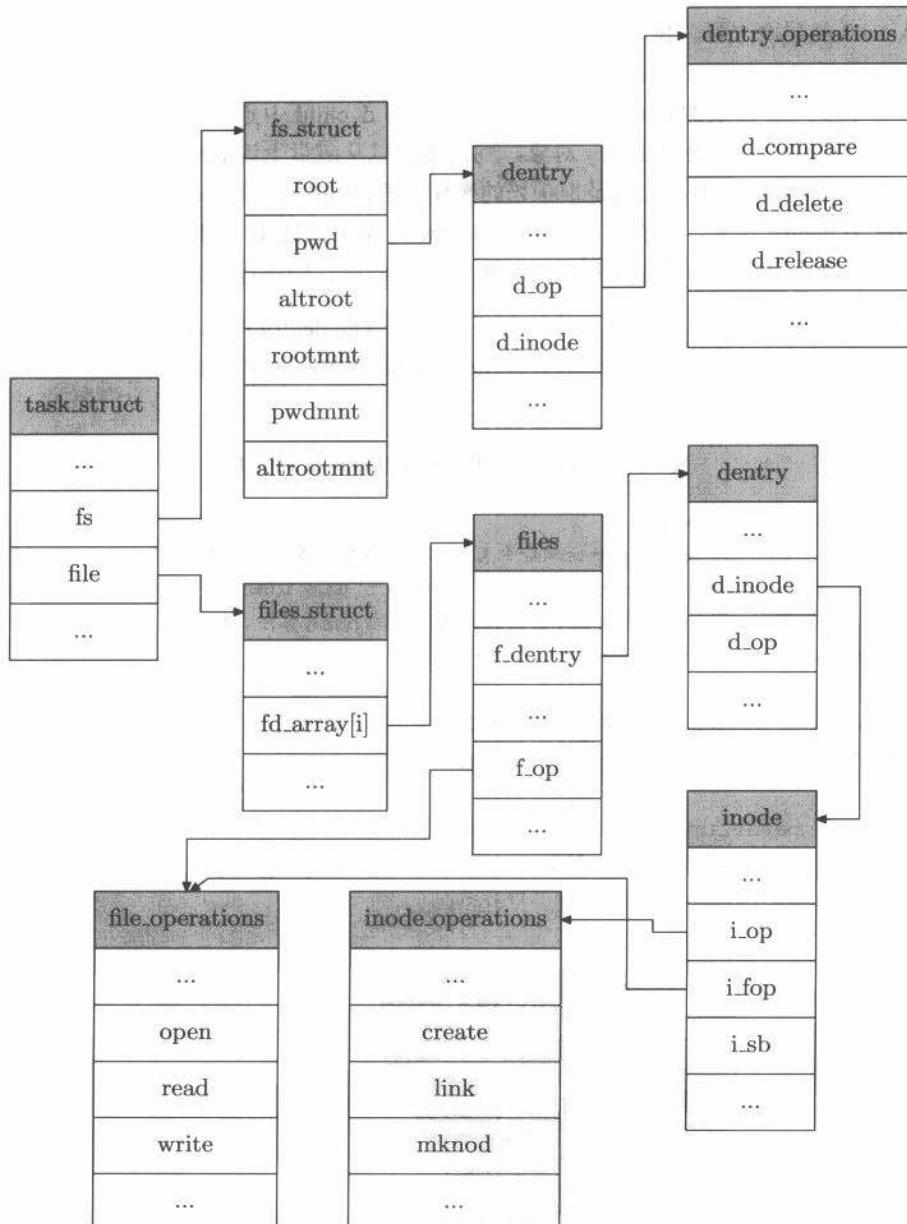


图 12.4 文件系统整体结构示意图

其中进程中的 fs 和 files 结构我们在第10章中已经介绍过了，fs 中的 root 指向进程根目录的 dentry 结构，而 pwd 指向进程当前目录的 dentry 结构，如果进程通过 chroot 调用改变了进程的根目录，那么 chroot 就指向这个根目录。files_struct 结构中的 fd_array 数组保存着进程打开的文件对象指针，其中 0, 1, 2 分别是标准输入、标准输出和标准错误的 files 结构。

从这里可以看到，无论下层的具体文件系统之间有什么差异，进程统一看到的是 VFS 的文件对象。当挂载一个新的块设备时，内核根据文件系统的类型，找到它的 file_system_type 对象，并调用该文件系统的 get_sb() 函数，get_sb() 函数初始化超级块对象，设置了 s_op 函数指针，这样 VFS 通过这个指针就知道如何操作这个块设备了。同时 get_sb() 会初始化块设备根目录的 inode 对象，并设置 inode 对象的 i_op 和 i_fop 函数指针。以后我们会看到文件对象中的 f_op 成员，及 inode 对象中的 i_fop 成员都指向同一个 file_operations 结构。这样 VFS 就知道如何操作它们了。现在知道为什么移动硬盘一定要在 mount 后才能使用了吧？

12.4 文件系统的挂载

在 Linux 文件系统中，只有一颗目录树，一个新的块设备必须 mount 到现有的目录下，我们把它称为安装点。每一个安装点都对应一个 vfsmount 对象，其定义如下：

代码片段 12.30 节自 include/linux/mount.h

```
1 struct vfsmount {
2     struct list_head mnt_hash;
3     /* 父节点的 vfsmount 对象。 */
4     /* fs we are mounted on */
5     struct vfsmount *mnt_parent;
6     /* 安装点的 dentry 对象。 */
7     /* dentry of mountpoint */
8     struct dentry *mnt_mountpoint;
9     /* 本设备根目录的 dentry 对象。 */
10    /* root of the mounted tree */
11    struct dentry *mnt_root;
12    /* pointer to superblock */
13    struct super_block *mnt_sb;
14    /* list of children, anchored here */
15    struct list_head mnt_mounts;
16    /* and going through their mnt_child */
17    struct list_head mnt_child;
18    int mnt_flags;
19    /* 4 bytes hole on 64bits arches */
20    /* Name of device e.g. /dev/dsk/hd1 */
```

```
1  char *mnt_devname;
2  struct list_head mnt_list;
3  /* link in fs-specific expiry list */
4  struct list_head mnt_expire;
5  /* circular list of shared mounts */
6  struct list_head mnt_share;
7  /* list of slave mounts */
8  struct list_head mnt_slave_list;
9  /* slave list entry */
10 struct list_head mnt_slave;
11 /* slave is on master->mnt_slave_list */
12 struct vfsmount *mnt_master;
13 /* containing namespace */
14 struct mnt_namespace *mnt_ns;
15
16 /* true if marked for expiry */
17 atomic_t mnt_count;
18 int mnt_expiry_mark;
19 int mnt_pinned;
20 };
```

每当 mount 一个新的块设备时，就会调用 alloc_vfsmnt()函数分配一个 vfsmnt 对象，然后根据文件系统类型找到 file_system_type 对象，并调用它的 get_sb()函数初始化 vfsmnt 对象。这里有一个问题，假设现在的根文件系统对应块设备/dev/sda1，那么 mount -t ext3 /dev/sda1 / 时，/目录还不存在，那么 dev 和 sda1 又从何而来呢？内核在启动时 mount 一个临时的根目录。这个临时的根目录的文件系统类别为 rootfs，它不对应任何磁盘文件，也就是我们通常说的 ramfs。这个函数是在 start_kernel()中调用的。我们就以这个函数为例子，讨论内核是如何挂载文件系统的。

代码片段 12.31 节自 fs/namespace.c

```
13                         0, SLAB_HWCACHE_ALIGN | SLAB_PANIC,
14                         NULL);
15     mount_hashtable = (struct list_head *) __get_free_page(GFP_ATOMIC);
16     if (!mount_hashtable)
17         panic("Failed to allocate mount hash table\n");
18     .....
19     err = sysfs_init();
20     if (err)
21         printk(KERN_WARNING "%s: sysfs_init error: %d\n",
22                 __FUNCTION__, err);
23     err = subsystem_register(&fs_subsys);
24     if (err)
25         printk(KERN_WARNING "%s: subsystem_register error: %d\n",
26                 __FUNCTION__, err);
27     init_rootfs();
28     init_mount_tree();
29 }
```

mnt_init()函数首先对相关的缓存及文件系统进行初始化。然后调用 init_rootfs()注册 rootfs 文件系统对象。

代码片段 12.32 节自 fs/ramfs/inode.c

```
1 static struct file_system_type rootfs_fs_type = {
2     .name      = "rootfs",
3     .get_sb    = rootfs_get_sb,
4     .kill_sb   = kill_litter_super,
5 };
6
7 int __init init_rootfs(void)
8 {
9     int err;
10
11     err = bdi_init(&ramfs_backing_dev_info);
12     if (err)
13         return err;
14     err = register_filesystem(&rootfs_fs_type);
15     if (err)
16         bdi_destroy(&ramfs_backing_dev_info);
17
18     return err;
19 }
```

这个函数很简单，它调用 `register_filesystem()` 注册 `rootfs` 文件系统对象。而挂载根文件系统的工作是由 `init_mount_tree()` 来完成的。

代码片段 12.33 节自 fs/namespace.c

```

1 static void __init init_mount_tree(void)
2 {
3     struct vfsmount *mnt;
4     struct mnt_namespace *ns;
5
6     /* 调用 do_kern_mount() 挂载 rootfs. */
7     mnt = do_kern_mount("rootfs", 0, "rootfs", NULL);
8     if (IS_ERR(mnt))
9         panic("Can't create rootfs");
10    ns = kmalloc(sizeof(*ns), GFP_KERNEL);
11    if (!ns)
12        panic("Can't allocate initial namespace");
13    atomic_set(&ns->count, 1);
14    INIT_LIST_HEAD(&ns->list);
15    init_waitqueue_head(&ns->poll);
16    ns->event = 0;
17    list_add(&mnt->mnt_list, &ns->list);
18    ns->root = mnt;
19    mnt->mnt_ns = ns;
20
21    init_task.nsproxy->mnt_ns = ns;
22    get_mnt_ns(ns);
23
24    set_fs_pwd(current->fs, ns->root, ns->root->mnt_root);
25    set_fs_root(current->fs, ns->root, ns->root->mnt_root);
26 }

```

`do_kern_mount()` 定义如下：

代码片段 12.34 节自 fs/super.c

```

1 struct vfsmount *
2 do_kern_mount(const char *fstype,
3                 int flags,
4                 const char *name,
5                 void *data)
6 {
7     /* 获取 fstype 指定的文件系统对象。 */
8     struct file_system_type *type = get_fs_type(fstype);
9     struct vfsmount *mnt;

```

```
10  if (!type)
11      return ERR_PTR(-ENODEV);
12  mnt = vfs_kern_mount(type, flags, name, data);
13
14  if (!IS_ERR(mnt) &&
15      (type->fs_flags & FS_HAS_SUBTYPE) &&
16      !mnt->mnt_sb->s_subtype)
17      mnt = fs_set_subtype(mnt, fstype);
18  put_filesystem(type);
19  return mnt;
20 }
21
22 struct vfsmount * vfs_kern_mount(struct file_system_type *type,
23                                     int flags,
24                                     const char *name,
25                                     void *data)
26 {
27     struct vfsmount *mnt;
28     char *secdatas = NULL;
29     int error;
30
31     if (!type)
32         return ERR_PTR(-ENODEV);
33
34     error = -ENOMEM;
35     /* 分配 vfsmnt 对象。*/
36     mnt = alloc_vfsmnt(name);
37     if (!mnt)
38         goto out;
39     .....
40     /* 调用文件系统对象的 get_sb() 函数。*/
41     error = type->get_sb(type, flags, name, data, mnt);
42     if (error < 0)
43         goto out_free_secdatas;
44     BUG_ON(!mnt->mnt_sb);
45
46     /* 安全检查，默认为空。*/
47     error = security_sb_kern_mount(mnt->mnt_sb, secdatas);
48     if (error)
49         goto out_sb;
50
51     mnt->mnt_mountpoint = mnt->mnt_root;
```

```

52     mnt->mnt_parent = mnt;
53     up_write(&mnt->mnt_sb->s_umount);
54     free_secdata(secdata);
55     return mnt;
56     .....
57 }

```

`vfsmnt` 对象的初始化是由具体文件系统的 `get_sb()` 函数完成的，`rootfs` 对应的 `get_sb()` 函数为 `rootfs_get_sb()`，其定义如下：

代码片段 12.35 节自 `fs/ramfs/inode.c`

```

1 static int rootfs_get_sb(struct file_system_type *fs_type,
2                           int flags,
3                           const char *dev_name,
4                           void *data,
5                           struct vfsmount *mnt)
6 {
7     return get_sb_nodev(fs_type,
8                         flags| MS_NOUSER,
9                         data,
10                        ramfs_fill_super,
11                        mnt);
12 }

```

顾名思义，`get_sb_nodev()` 函数为那些不对应于任何设备文件的文件系统设置超级块。需要注意的是，这里的 `ramfs_fill_super()` 函数做为参数传递到 `get_sb_nodev()` 函数中。`get_sb_nodev()` 定义如下：

代码片段 12.36 节自 `fs/super.c`

```

1 int get_sb_nodev(struct file_system_type *fs_type,
2                   int flags,
3                   void *data,
4                   int (*fill_super) (struct super_block *, void *, int),
5                   struct vfsmount *mnt)
6 {
7     int error;
8     struct super_block *s = sget(fs_type, NULL, set_anon_super, NULL);
9
10    if (IS_ERR(s))
11        return PTR_ERR(s);
12
13    s->s_flags = flags;
14

```

```
15     error = fill_super(s, data, flags & MS_SILENT ? 1 : 0);
16     if (error) {
17         up_write(&s->s_umount);
18         deactivate_super(s);
19         return error;
20     }
21     s->s_flags |= MS_ACTIVE;
22     return simple_set_mnt(mnt, s);
23 }
```

在第8行，首先调用 `sget()` 获取一个 `super_block` 对象。`sget()` 首先查找指定的 `super_block` 对象是否存在，如果不存在就需要分配一个。在我们的这个例子中，肯定是不存在的。最后调用 `fill_super()` 函数，函数指针 `fill_super` 是 `rootfs_get_sb()` 传递过来的参数，这里为 `ramfs_fill_super()`，定义如下：

代码片段 12.37 节自 `fs/ramfs/inode.c`

```
1 static int ramfs_fill_super(struct super_block * sb,
2                             void * data,
3                             int silent)
4 {
5     struct inode * inode;
6     struct dentry * root;
7
8     sb->s_maxbytes = MAX_LFS_FILESIZE;
9     sb->s_blocksize = PAGE_CACHE_SIZE;
10    sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
11    sb->s_magic = RAMFS_MAGIC;
12
13    /* 设置 s_op 函数指针。*/
14    sb->s_op = &ramfs_ops;
15    sb->s_time_gran = 1;
16    inode = ramfs_get_inode(sb, S_IFDIR | 0755, 0);
17    if (!inode)
18        return -ENOMEM;
19    root = d_alloc_root(inode);
20    if (!root) {
21        iput(inode);
22        return -ENOMEM;
23    }
24    sb->s_root = root;
25    return 0;
26 }
```

可以看到，它的超级块的设置也非常简单，`ramfs_fill_super()`函数分配并初始化根目录的`inode`和`dentry`结构。并且把超级块的`s_op`设置为`ramfs_ops`。这个结构定义如下：

代码片段 12.38 节自 `fs/ramfs/inode.c`

```

1 static const struct super_operations ramfs_ops = {
2     .statfs    = simple_statfs,
3     .drop_inode = generic_delete_inode,
4 };

```

对`inode`的设置我们在前面已经讨论过了，其他特殊文件系统和`ramfs`的原理都是一样的，这里再来看看`ramfs_get_inode()`函数，然后在此基础上对特殊文件系统进行一个简单的总结。

代码片段 12.39 节自 `fs/ramfs/inode.c`

```

1 struct inode *ramfs_get_inode(struct super_block *sb,
2                                int mode,
3                                dev_t dev)
4 {
5     struct inode *inode = new_inode(sb);
6
7     if (inode) {
8         inode->i_mode = mode;
9         inode->i_uid = current->fsuid;
10        inode->i_gid = current->fsgid;
11        inode->i_blocks = 0;
12        inode->i_mapping->a_ops = &ramfs_aops;
13        inode->i_mapping->backing_dev_info = &ramfs_backing_dev_info;
14        mapping_set_gfp_mask(inode->i_mapping, GFP_HIGHUSER);
15        inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
16        switch (mode & S_IFMT) {
17            default:
18                init_special_inode(inode, mode, dev);
19                break;
20            case S_IFREG:
21                inode->i_op = &ramfs_file_inode_operations;
22                inode->i_fop = &ramfs_file_operations;
23                break;
24            case S_IFDIR:
25                inode->i_op = &ramfs_dir_inode_operations;
26                inode->i_fop = &simple_dir_operations;
27
28                /* directory inodes start off with i_nlink == 2 (for ".," entry) */
29                inc_nlink(inode);

```

```

30     break;
31     case S_IFLNK:
32         inode->i_op = &page_symlink_inode_operations;
33         break;
34     }
35 }
36 return inode;
37 }
```

ramfs_get_inode()函数首先分配一个 inode 对象，然后根据文件的类别设置它的 i_op 和 i_fop 指针。将来 file 对象中的 f_op 也是取自 i_fop。

dentry 对象是由 d_alloc_root()建立的，定义如下：

代码片段 12.40 节自 fs/dcache.c

```

1 struct dentry * d_alloc_root(struct inode * root_inode)
2 {
3     struct dentry *res = NULL;
4
5     if (root_inode) {
6         static const struct qstr name = { .name = "/", .len = 1 };
7
8         /* 分配 dentry 对象。*/
9         res = d_alloc(NULL, &name);
10        if (res) {
11            res->d_sb = root_inode->i_sb;
12            res->d_parent = res;
13            d_instantiate(res, root_inode);
14        }
15    }
16    return res;
17 }
```

至此，VFS 所需的管理结构就建立起来了，这里 rootfs 是完全虚拟的文件系统。从这个例子中，我们可以看出特殊文件系统的原理，仅仅是在内存中建立 VFS 所需的管理结构，向上层导出一个统一的接口。其他的 proc 文件系统也是一样的。例如通过 create_proc_entry()在/prco 文件系统中建立一个目录时，它只需要按照 VFS 的规范建立 inode, dentry 等对象就可以了。

我们看到这里 mount 根目录的主要工作是由 do_kern_mount()完成的，现在看看 sys_mount()系统调用。当用户使用形如 mount -t ext2 /dev/sda3 /mnt 时，会调用 sys_mount()函数，这个函数先检查/mnt 这个路径是否存在，如果存在就获取它的 dentry 结构，然后调用 do_kern_mount()。

代码片段 12.41 节自 fs/namespace.c

```
1 [sys_mount() -> do_mount()]
2
3 long do_mount(char *dev_name,
4                 char *dir_name,
5                 char *type_page,
6                 unsigned long flags,
7                 void *data_page)
8 {
9     struct nameidata nd;
10    int retval = 0;
11    int mnt_flags = 0;
12
13    /* Discard magic */
14    if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
15        flags &= ~MS_MGC_MSK;
16
17    /* Basic sanity checks */
18    if (!dir_name ||
19        !*dir_name ||
20        !memchr(dir_name, 0, PAGE_SIZE))
21        return -EINVAL;
22    if (dev_name && !memchr(dev_name, 0, PAGE_SIZE))
23        return -EINVAL;
24
25    if (data_page)
26        ((char *)data_page)[PAGE_SIZE - 1] = 0;
27    .....
28    /* 检查安装点的路径。*/
29    /* ... and get the mountpoint */
30    retval = path_lookup(dir_name, LOOKUP_FOLLOW, &nd);
31    if (retval)
32        return retval;
33
34    retval = security_sb_mount(dev_name, &nd, type_page, flags, data_page);
35    if (retval)
36        goto dput_out;
37    if (flags & MS_REMOUNT)
38        retval = do_remount(&nd, flags & ~MS_REMOUNT, mnt_flags, data_page);
39
40    else if (flags & MS_BIND)
41        retval = do_loopback(&nd, dev_name, flags & MS_REC);
```

```

42     else if (flags & (MS_SHARED | MS_PRIVATE | MS_SLAVE | MS_UNBINDABLE))
43         retval = do_change_type(&nd, flags);
44     else if (flags & MS_MOVE)
45         retval = do_move_mount(&nd, dev_name);
46     else
47         retval = do_new_mount(&nd, type_page, flags, mnt_flags, dev_name,
48                               data_page);
48 dput_out:
49     path_release(&nd);
50     return retval;
51 }

```

由于 mount 命令的参数可以指定各种标志，例如-o remount 时表示重新 mount 一个已经 mount 了的设备，主要用于改变某些 mount 标志，比如以前是只读的，现在改成可写的。在这种情况下将调用第38行的 do_remount 函数，其他的函数也类似。这里我们主要讨论挂载新设备的情况，所以调用的是第47行的 do_new_mount()。

代码片段 12.42 节自 fs/namespace.c

```

1 [sys_mount() -> do_mount() -> do_new_mount()]
2
3 static int do_new_mount(struct nameidata *nd,
4                         char *type,
5                         int flags,
6                         int mnt_flags,
7                         char *name,
8                         void *data)
9 {
10    struct vfsmount *mnt;
11    .....
12    mnt = do_kern_mount(type, flags, name, data);
13    if (IS_ERR(mnt))
14        return PTR_ERR(mnt);
15
16    return do_add_mount(mnt, nd, mnt_flags, NULL);
17 }

```

可以看到这个函数很简单，其中 do_kern_mount()在前面我们已经分析过了(见第416页代码片段12.34)。差别就是本例中 do_kern_mount()函数将调用 ext2 的文件系统对象的 get_sb()函数，那就是 ext2_get_sb()。

代码片段 12.43 节自 fs/ext2/super.c

```

1 [sys_mount() -> do_mount() -> do_new_mount() -> do_kern_mount() ->
ext2_get_sb()]

```

```

2
3 static int ext2_get_sb(struct file_system_type *fs_type,
4                         int flags,
5                         const char *dev_name,
6                         void *data,
7                         struct vfsmount *mnt)
8 {
9     return get_sb_bdev(fs_type, flags, dev_name, data, ext2_fill_super, mnt);
10 }

```

get_sb_bdev 函数定义如下：

代码片段 12.44 节自 fs/ext2/super.c

```

1 /* 注意函数指针 fill_super 指向 ext2_fill_super()。 */
2 int get_sb_bdev(struct file_system_type *fs_type,
3                  int flags,
4                  const char *dev_name,
5                  void *data,
6                  int (*fill_super)(struct super_block *, void *, int),
7                  struct vfsmount *mnt)
8 {
9     struct block_device *bdev;
10    struct super_block *s;
11    int error = 0;
12
13    bdev = open_bdev_excl(dev_name, flags, fs_type);
14    if (IS_ERR(bdev))
15        return PTR_ERR(bdev);
16
17    down(&bdev->bd_mount_sem);
18    /*
19     * 在 super_block 链表中搜索指定的 super_block 对象,
20     * 如果搜索失败, 就重新分配一个。
21     */
22    s = sget(fs_type, test_bdev_super, set_bdev_super, bdev);
23    up(&bdev->bd_mount_sem);
24    if (IS_ERR(s))
25        goto error_s;
26
27    /* 如果返回的 super_block 对象的 s_root 成员非空, 则说明已经 mount 过了。 */
28    if (s->s_root) {
29        if ((flags ^ s->s_flags) & MS_RDONLY) {
30            up_write(&s->s_umount);

```

```
31     deactivate_super(s);
32     error = -EBUSY;
33     goto error_bdev;
34 }
35 close_bdev_excl(bdev);
36 } else {
37     char b[BDEVNAME_SIZE];
38
39     s->s_flags = flags;
40     strlcpy(s->s_id, bdevname(bdev, b), sizeof(s->s_id));
41     sb_set_blocksize(s, block_size(bdev));
42     error = fill_super(s, data, flags & MS_SILENT ? 1 : 0);
43     if (error) {
44         up_write(&s->s_umount);
45         deactivate_super(s);
46         goto error;
47     }
48     s->s_flags |= MS_ACTIVE;
49 }
50
51 return simple_set_mnt(mnt, s);
52 .....
53 }
```

这个函数在第22行调用 `sget()` 函数，它首先尝试从超级块链表中搜索指定的 `super_block` 对象，如果失败就分配一个 `super_block` 对象。如果 `sget()` 函数返回的 `super_block` 对象已经设置了 `s_root`(`s_root` 的类型为 `dentry`)，则说明这个块设备已经 `mount` 过了，那么内存中已经有现成的 `super_block` 的完整信息，不需要经过磁盘 IO 操作来设置了。否则在第42行调用 `ext2_fill_super()`，这个函数我们在第391页代码片段12.13中已经讨论过了，它利用磁盘驱动程序读取 Ext2 在磁盘上的超级块，然后根据这些信息设置 `super_block` 对象，在此不再重复。

12.5 路径定位

根据一个路径定位到一个文件的 `inode` 和数据块，是一个很频繁的操作。这个过程表面上看起来很简单，但是其代码却比较复杂，为了方便后面代码的分析，这里我们先大概地介绍它的整体过程，在有了整体认识的基础之后，再来看代码就简单得多了。

路径可以是绝对路径，也可以是相对路径，进程的 `fs->root` 和 `fs->pwd` 这两个 `dentry` 对象分别指向进程的根目录和当前目录。当要定位的目标路径以/开头时，表示绝对路径，使用 `fs->root` 作为起始点，否则表示相对路径，使用 `fs->pwd` 作为起始点。分析的过程就

是依次从 dentry 对象的子节点中匹配下一级目录，直到定位到最终的节点。以 /bin/bash 为例，由于这个路径以 / 开头，因此取当前进程的 fs->root 所指向的 dentry 对象作为起点，到它的子节点链表中搜索 bin 的 dentry 对象，然后再对 bin 的 dentry 对象的子节点链表中搜索 bash 的 dentry 对象⁶。在这个过程中，如果失败，就需要尝试到磁盘上读取目录项。这个例子是最简单的情况，在接下来的分析过程中，我们会看到，实际上的定位过程还需要考虑各种情况。

在定位过程中，内核用到几个辅助结构，分别是 nameidata、qstr 及 path，先来看看 qstr。

代码片段 12.45 节自 include/linux/dcache.h

```

1 struct qstr {
2     unsigned int hash;
3     unsigned int len;
4     const unsigned char *name;
5 };

```

这个结构表示路径字符串中的项目，以 /bin/bash 为例，内核首先从这个路径中解析出 bin，此时 qstr 对象的 name 指向路径字符串中的 bin，其 len 为 3，hash 用于加速搜索速度。第二次 qstr 对象的 name 指向路径字符串中的 bash，其 len 为 4。下面是它的解析代码片段。

代码片段 12.46 节自 fs/namei.c

```

1     .....
2     while (*name=='/')
3         name++;
4
5     for (;;) {
6         unsigned long hash;
7         struct qstr this;
8         unsigned int c;
9         this.name = name;
10        c = *(const unsigned char *)name;
11        hash = init_name_hash();
12        do {
13            name++;
14            hash = partial_name_hash(c, hash);
15            c = *(const unsigned char *)name;
16        } while (c && (c != '/'));
17        this.len = name - (const char *) this.name;

```

⁶ 这里再次提醒，不要认为 dentry 仅仅代表目录，它代表一个目录项，也可以说是目录条目，因此文件也有它的 dentry 对象。例如这个例子中，bash 一定是 bin 目录中的一个目录项。

```

18     this.hash = end_name_hash(hash);
19     if (!c)
20         goto last_component;
21     while (*++name == '/')
22     ;
23     if (!*name)
24         goto last_with_slashes;
25 }

```

上面这段代码节自 `_link_path_walk()` 函数的一个循环中，还是以 `/bin/bash` 为例，`name` 指向这个字符串，在这段代码中的 `for` 循环之前，`while` 循环忽略符号`/`，把 `name` 调整为指向 `bin/bash`，之后定义了一个 `qstr` 变量 `this`，在第9行把 `this->name` 调整指向 `bin/bash`。然后在第12行的 `do-while` 循环中，调整 `name` 指针依次指向 `b,i,n`，直到遇到`/`或者字符串末尾结束，同时计算字符串的 `hash` 值。如果到了末尾就跳转到 `last_component` 处。此时 `name` 指向 `bash` 字符串。否则在第21行中的 `while` 循环中，跳过符号`/`。然后一直重复这个过程，最后，如果到了路径末尾是以`/`号结束，那么就跳转到 `last_with_slashes` 处，这要求这个路径指定的是一个目录。

我们再来看看 `nameidata`，其定义如下：

代码片段 12.47 节自 `include/linux/namei.h`

```

1 struct nameidata {
2     struct dentry *dentry;
3     struct vfsmount *mnt;
4     struct qstr last;
5     unsigned int flags;
6     int last_type;
7     unsigned depth;
8     char *saved_names[MAX_NESTED_LINKS + 1];
9
10    /* Intent data */
11    union {
12        struct open_intent open;
13    } intent;
14 };

```

这个结构中的 `dentry` 指向当前的 `dentry` 对象，`mnt` 指向当前对应 `vfsmount` 对象。为了说明这两个结构的用途，我们假设有一个块设备 `mount` 在 `/media/disk` 目录中，现在要定位的路径为 `/media/disk/test.c`。初始化时，`nameidata` 结构中的 `dentry` 指向根目录对应的 `dentry`

⁷从这里可以看出，内核在路径定位的过程中会自动忽略多余的`/`号，也就是说，`/bin/bash` 和 `//bin//bash` 都可以正确地定位到 `bash` 文件。

对象，`mnt` 指向根目录对应的 `vfsmount` 对象，当定位到 `disk` 时，`nameidata` 中的 `mnt` 需要调整为 `mount` 的块设备 `disk` 对应的 `vfsmount` 对象，`dentry` 指向 `disk` 对应的 `dentry` 对象。

由于在定位过程中，可能遇到符号链接，指针数组 `saved_names` 指向每一级别的符号链接，最大为 8，这是为了避免在定位符号链接的过程中陷入死循环，注意这是一个指针数组，每一项指向一个路径字符串路径。当定位一个普通路径时，`depth` 为 0，每当进入一级符号链接时，`depth` 会加 1，最大为 8。举例来说，假设现在定位的是 `a`，此时 `depth` 为 0，如果发现 `a` 是 `b` 的一个符号链接，此时转向定位 `b`，同时 `depth` 被设置为 1。

`open_intent` 主要保存一个 `file` 对象指针，我们将在介绍文件打开操作时，介绍它的作用。另外一个结构就是 `path`，它的定义如下：

代码片段 12.48 节自 `include/linux/namei.h`

```

1 struct path {
2     struct vfsmount *mnt;
3     struct dentry *dentry;
4 };

```

这个结构主要用于返回在一个目录中搜索子目录项的结构，其中 `mnt` 指向子目录项的 `vfsmount` 对象，`dentry` 指向子目录项的 `dentry` 对象。

现在我们先来看看具体的定位过程吧。

代码片段 12.49 节自 `fs/namei.c`

```

1 static int fastcall do_path_lookup(int dfd,
2                                     const char *name,
3                                     unsigned int flags,
4                                     struct nameidata *nd)
5 {
6     int retval = 0;
7     int fput_needed;
8     struct file *file;
9     struct fs_struct *fs = current->fs;
10
11    /* if there are only slashes... */
12    nd->last_type = LAST_ROOT;
13    nd->flags = flags;
14    nd->depth = 0;
15
16    if (*name=='/') {
17        read_lock(&fs->lock);
18        if (fs->altroot && !(nd->flags & LOOKUP_NOALT)) {
19            nd->mnt = mntget(fs->altrootmnt);
20            nd->dentry = dget(fs->altroot);

```

```

21     read_unlock(&fs->lock);
22
23     /* found in altroot */
24     if (_emul_lookup_dentry(name, nd))
25         goto out;
26     read_lock(&fs->lock);
27 }
28 nd->mnt = mntget(fs->rootmnt);
29 nd->dentry = dget(fs->root);
30 read_unlock(&fs->lock);
31 } else if (dfd == AT_FDCWD) {
32     read_lock(&fs->lock);
33     nd->mnt = mntget(fs->pwdmnt);
34     nd->dentry = dget(fs->pwd);
35     read_unlock(&fs->lock);
36 }
37 .....
38 retval = path_walk(name, nd);
39 .....
40 return retval;
41 }

```

如果路径以/开头，说明是一个绝对路径，此时就需要把 nd->mnt 和 nd->dentry 设置为根目录对应的 vfstype 对象和 dentry 对象。但是某些服务器进程由于安全原因，可能通过 chroot 系统调用改变了进程的根目录，这被称为 altroot。这样做的好处是即便是利用服务器漏洞也只能操作到这个根目录中的文件，而不能获取这个目录之外的其他文件，例如/etc/passwd。因此这里检查如果设置了 altroot，并且也没有指定 LOOKUP_NOALT 标志，那么就需要把 nd->mnt 和 nd->dentry 分别设置为当前进程的 fs->altrootmnt 和 fs->altroot。

如果 dfd 为 AT_FDCWD，就需要从进程的当前目录开始定位，此时需要把 nd->mnt 和 nd->dentry 分别设置为 fs->pwdmnt 和 fs->pwd。最后这个进程调用 path_walk() 函数开始搜索定位工作。

代码片段 12.50 节自 fs/namei.c

```

1 static int fastcall path_walk(const char * name, struct nameidata *nd)
2 {
3     current->total_link_count = 0;
4     return link_path_walk(name, nd);
5 }

```

link_path_walk() 函数定义如下：

代码片段 12.51 节自 fs/namei.c

```

1 static int fastcall link_path_walk(const char *name, struct nameidata *nd)
2 {
3     /* 保存一个 nameidata 对象的备份。*/
4     struct nameidata save = *nd;
5     int result;
6
7     /* 增加引用计数。*/
8     /* make sure the stuff we saved doesn't go away */
9     dget(save.dentry);
10    mntget(save.mnt);
11    /* 开始定位，结果保存在 nd 中。*/
12    result = __link_path_walk(name, nd);
13    if (result == -ESTALE) {
14        *nd = save;
15        dget(nd->dentry);
16        mntget(nd->mnt);
17        nd->flags |= LOOKUP_REVAL;
18        result = __link_path_walk(name, nd);
19    }
20    dput(save.dentry);
21    mntput(save.mnt);
22
23    return result;
24 }

```

这个函数首先调用 `__link_path_walk()` 尝试在内存中定位目标路径，第一次不会进行磁盘操作，如果失败，就设置 `LOOKUP_REVAL` 标志，再次调用 `__link_path_walk()`，这一次可能会进行磁盘操作。`__link_path_walk()` 函数定义如下：

代码片段 12.52 节自 fs/namei.c

```

1 static fastcall int __link_path_walk(const char *name,
2                                     struct nameidata *nd)
3 {
4     struct path next;
5     struct inode *inode;
6     int err;
7     unsigned int lookup_flags = nd->flags;
8
9     /* 跳过路径前面的 / 号。*/
10    while (*name=='/')
11        name++;
12    /* 如果跳过 / 后得到一个空的路径字符串就，直接返回。*/
13    if (!*name)

```

```
14     goto return_reval;
15  /* 当前起点的 dentry 对象对应的 inode. */
16  inode = nd->dentry->d_inode;
17  /* 如果 depth 不为 0, 设置 LOOKUP_FOLLOW 标志。*/
18  if (nd->depth)
19      lookup_flags = LOOKUP_FOLLOW | (nd->flags & LOOKUP_CONTINUE);
20  /*
21   * 下面这个循环就是根据路径中的 /, 一级级地分离出目录名,
22   * 然后处理。见第426页代码片段12.45。
23   */
24  /* At this point we know we have a real path component. */
25  for(;;) {
26      unsigned long hash;
27      struct qstr this;
28      unsigned int c;
29
30      nd->flags |= LOOKUP_CONTINUE;
31      err = exec_permission_lite(inode, nd);
32      if (err == -EAGAIN)
33          err = vfs_permission(nd, MAY_EXEC);
34      if (err)
35          break;
36      this.name = name;
37      c = *(const unsigned char *)name;
38
39      hash = init_name_hash();
40      do {
41          name++;
42          hash = partial_name_hash(c, hash);
43          c = *(const unsigned char *)name;
44      } while (c && (c != '/'));
45      this.len = name - (const char *) this.name;
46      this.hash = end_name_hash(hash);
47
48      /* 执行到这里, this 代表本次要解析的名字。*/
49
50      /* 到达路径字符串的末尾了, 跳转到 last_componet. */
51      if (!c)
52          goto last_component;
53      /* 如果路径字符串以 / 结束, 说明被定位的必须是一个目录, 跳到 last_with_slashes. */
54      while (*++name == '/');
55      if (!*name)
```

```
56     goto last_with_slashes;
57     /*
58      * 如果这个 qstr 项长度为 1，并且 name 为一个点，则表示当前目录，
59      * 于是执行 continue，进行下一次 for 循环。
60      * 如果这个 qstr 项长度为 2，并且 name 为连续的两个点，则表示上级目录，
61      * 于是调用 follow_dotdot() 调整 nd 中的相关成员。
62      */
63     if (this.name[0] == '.') switch (this.len) {
64         default:
65             break;
66         case 2:
67             if (this.name[1] != '.')
68                 break;
69             follow_dotdot(nd);
70             inode = nd->dentry->d_inode;
71             /* fallthrough */
72         case 1:
73             continue;
74     }
75     /* 某些文件系统可能提供了自己的 hash 算法，那么就调用它的 d_hash. */
76     if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {
77         err = nd->dentry->d_op->d_hash(nd->dentry, &this);
78         if (err < 0)
79             break;
80     }
81     /*
82      * 在 nd 的 dentry 成员中搜索由 this 指定的子目录项节点，
83      * 并在 next 中返回子目录项节点的 dentry.
84      */
85     err = do_lookup(nd, &this, &next);
86     if (err)
87         break;
88     err = -ENOENT;
89
90     /* 调整 inode 指向子节点目录项节点对应的 inode. */
91     inode = next.dentry->d_inode;
92     if (!inode)
93         goto out_dput;
94     err = -ENOTDIR;
95     if (!inode->i_op)
96         goto out_dput;
97     /*

```

```
98     * 在对 ext2_read_inode() 函数的分析过程中(见第397页代码片段12.17),  
99     * 我们看到 ext2_read_inode() 为目录, 普通文件或者符号链接设置不同的 i_op,  
100    * 所以在这里, 如果 i_op 的 follow_link 不为空, 那就说明这一定是一个符号链接,  
101    * 于是调用 do_follow_link() 处理符号链接。  
102    */  
103    if (inode->i_op->follow_link) {  
104        err = do_follow_link(&next, nd);  
105        if (err)  
106            goto return_err;  
107        err = -ENOENT;  
108        inode = nd->dentry->d_inode;  
109        if (!inode)  
110            break;  
111        err = -ENOTDIR;  
112        if (!inode->i_op)  
113            break;  
114    }  
115    /* 如果不是符号链接, nd 中的 dentry 成员就可以前进一步了。 */  
116    else  
117        path_to_nameidata(&next, nd);  
118    err = -ENOTDIR;  
119    if (!inode->i_op->lookup)  
120        break;  
121  
122    /* 继续 for 循环, 处理下一项。 */  
123    continue;  
124    /* here ends the main loop */  
125    /*  
126     * 如果从第54行跳转到这里, 说明目标路径一定  
127     * 是一个目录, 设置 LOOKUP_DIRECTORY 标志。  
128     */  
129 last_with_slashes:  
130     lookup_flags |= LOOKUP_FOLLOW | LOOKUP_DIRECTORY;  
131  
132     /*  
133      * 现在 qstr 对象 this 中保存这路径中最后一项, 而 nd 的 dentry 成员也指向  
134      * 了这一项的父节点。为了方便理解, 我们假设路径为 /bin/bash, 则 this  
135      * 指向字符 bash, nd->dentry 指向 bin 对应的 dentry 对象, 只要在这里从  
136      * nd->dentry 的目录项中搜索到 bash, 就大功告成了。  
137      */  
138 last_component:  
139     /* Clear LOOKUP_CONTINUE iff it was previously unset */
```

```
140     nd->flags &= lookup_flags | ~LOOKUP_CONTINUE;
141     if (lookup_flags & LOOKUP_PARENT)
142         goto lookup_parent;
143
144     /* 这里还是要考虑路径项仅仅为一个点或者两个点的情况，见第63行。*/
145     if (this.name[0] == '.') switch (this.len) {
146         default:
147             break;
148         case 2:
149             if (this.name[1] != '.')
150                 break;
151             follow_dotdot(nd);
152             inode = nd->dentry->d_inode;
153             /* fallthrough */
154         case 1:
155             goto return_reval;
156     }
157     if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {
158         err = nd->dentry->d_op->d_hash(nd->dentry, &this);
159         if (err < 0)
160             break;
161     }
162     /* 调用 do_lookup 搜索 this 指定的目录项。*/
163     err = do_lookup(nd, &this, &next);
164     if (err)
165         break;
166     inode = next.dentry->d_inode;
167
168     /* 考虑符号链接的情况。*/
169     if ((lookup_flags & LOOKUP_FOLLOW) &&
170         inode && inode->i_op &&
171         inode->i_op->follow_link) {
172         err = do_follow_link(&next, nd);
173         if (err)
174             goto return_err;
175         inode = nd->dentry->d_inode;
176     }
177     /*
178      * 如果不是符号链接，nd 中的 dentry 成员就可以前进一步了，
179      * 这时它指向了最终要定位的 dentry 对象了。
180      */
181     else
```

```
182     path_to_nameidata(&next, nd);
183     err = -ENOENT;
184     if (!inode)
185         break;
186     if (lookup_flags & LOOKUP_DIRECTORY) {
187         err = -ENOTDIR;
188         if (!inode->i_op || !inode->i_op->lookup)
189             break;
190     }
191     goto return_base;
192 lookup_parent:
193     nd->last = this;
194     nd->last_type = LAST_NORM;
195     if (this.name[0] != '.')
196         goto return_base;
197     if (this.len == 1)
198         nd->last_type = LAST_DOT;
199     else if (this.len == 2 && this.name[1] == '.')
200         nd->last_type = LAST_DOTDOT;
201     else
202         goto return_base;
203 return_reval:
204     /*
205      * We bypassed the ordinary revalidation routines.
206      * We may need to check the cached dentry for staleness.
207      */
208     if (nd->dentry && nd->dentry->d_sb &&
209         (nd->dentry->d_sb->s_type->fs_flags & FS_REVAL_DOT)) {
210         err = -ESTALE;
211         /* Note: we do not d_invalidate() */
212         if (!nd->dentry->d_op->d_revalidate(nd->dentry, nd))
213             break;
214     }
215     /* 大功告成。*/
216 return_base:
217     return 0;
218 out_dput:
219     dput_path(&next, nd);
220     break;
221 }
222 path_release(nd);
223 return_err:
```

```
224     return err;
225 }
```

总的来说，首先 `nd` 被设置为起始目录的相关信息，这个函数就是以/号为分隔，从路径中分离出一个目录项，然后到 `nd->dentry` 的子目录项中去搜索，如果搜索到了，就调用 `path_to_nameidata()` 函数让 `nd` 前进一步。`path_tonameidata()` 定义如下：

代码片段 12.53 节自 `fs/namei.c`

```
1 static inline void path_to_nameidata(struct path *path,
2                                     struct nameidata *nd)
3 {
4     dput(nd->dentry);
5
6     /* 如果子目录项对应一个新的 mount point，那么需要改变 nd->mnt。 */
7     if (nd->mnt != path->mnt)
8         mntput(nd->mnt);
9     nd->mnt = path->mnt;
10    /* nd->dentry 前进一步。 */
11    nd->dentry = path->dentry;
12 }
```

在解析过程中，如果遇到一目录项字符串是一个点，这表示当前目录，这种情况就维持 `nd` 不变，继续处理下一项。如果遇到两个点，它表示上级目录，那么 `nd` 就需要后退一步，这需要考虑以下几种情况：

- (1) 当前 `nd` 已经对应于文件的根目录，此时维持 `nd` 不变。
- (2) 当前 `nd` 对应于某个 `mount point` 的根目录，此时需要把 `nd->dentry` 设置为上级目录的 `dentry`，同时调整 `nd->mnt` 指向上级目录对应的 `vfsmount` 对象。
- (3) 当前 `nd` 对应于某个子目录，且上级目录和子目录位于同一个 `mount point`，直接调整 `nd->dentry` 指向上级目录的 `dentry` 对象。

这个工作是由 `follow_dotdot()` 函数完成的，弄清楚了它要完成的工作后，其代码也很容易理解了，因此感兴趣的读者可以自行分析它的代码。

现在我们来看看符号链接的情况，如果子目录项对应的 `inode->i_op->follow_link` 不为空，就说明这是一个符号链接，那么就需要暂时停止前进的步伐，先读取出 `inode` 保存的符号链接的目标路径，然后定位这个路径对应的 `dentry` 对象，这将再次调用 `_link_path_walk()` 函数，当链接路径处理结束后，才能返回 `_link_path_walk()` 函数继续处理。为了方便理解，我们举个例子，假设现在要定位的目标路径是 /opt/bin/bash，而 /opt 目录中的 bin 是一个符号链接，它对应的目标文件为 /bin，当 `_link_path_walk()` 函数定位到 /opt/bin 时，发现 bin 是一个符号链接，就从它的 `inode` 中读取出符号链接的目标路径 /bin，于是以 /bin 为目标

路径，再次调用 `_link_path_walk()` 函数，当它返回时 `nd->dentry` 就指向了 `/bin` 目录对应的 `dentry` 对象，这个时候，再继续前进定位 `bash`。

代码片段 12.54 节自 fs/namei.c

```

1 static inline int do_follow_link(struct path *path,
2                                 struct nameidata *nd)
3 {
4     int err = -ELOOP;
5     /* 最大的 link 深度为 8. */
6     if (current->link_count >= MAX_NESTED_LINKS)
7         goto loop;
8     if (current->total_link_count >= 40)
9         goto loop;
10    BUG_ON(nd->depth >= MAX_NESTED_LINKS);
11
12    cond_resched();
13    err = security_inode_follow_link(path->dentry, nd);
14    if (err)
15        goto loop;
16    current->link_count++;
17    current->total_link_count++;
18
19    nd->depth++;
20    err = __do_follow_link(path, nd);
21    current->link_count--;
22    nd->depth--;
23    return err;
24 loop:
25     dput_path(path, nd);
26     path_release(nd);
27     return err;
28 }
```

`__do_follow_link()` 函数定义如下：

代码片段 12.55 节自 fs/namei.c

```

1 static __always_inline int __do_follow_link(struct path *path,
2                                             struct nameidata *nd)
3 {
4     int error;
5     void *cookie;
6     struct dentry *dentry = path->dentry;
7 }
```

```

8   touch_atime(path->mnt, dentry);
9   nd_set_link(nd, NULL);
10  if (path->mnt != nd->mnt) {
11      path_to_nameidata(path, nd);
12      dget(dentry);
13  }
14  mntget(path->mnt);
15  cookie = dentry->d_inode->i_op->follow_link(dentry, nd);
16  error = PTR_ERR(cookie);
17  .....
18  dput(dentry);
19  mntput(path->mnt);
20
21  return error;
22 }

```

在第15行，调用具体文件系统的 follow_link()函数，对于 ext2 的就是 ext2_follow_link()，其定义如下：

代码片段 12.56 节自 fs/ext2/symlink.c

```

1 static void *ext2_follow_link(struct dentry *dentry,
2                               struct nameidata *nd)
3 {
4     struct ext2_inode_info *ei = EXT2_I(dentry->d_inode);
5     nd_set_link(nd, (char *)ei->i_data);
6     return NULL;
7 }
8 static inline void nd_set_link(struct nameidata *nd, char *path)
9 {
10    nd->saved_names[nd->depth] = path;
11 }

```

对于符号链接来说，ext2 的内存中的 inode 对象的 i_data 指向目标路径字符串，这是在 inode 的初始化时设置好的，现在 ext2_follow_link()函数从这里取出这个路径。之后使用这个路径作为参数调用 __vfs_follow_link()。

代码片段 12.57 节自 fs/namei.c

```

1 static __always_inline int __vfs_follow_link(struct nameidata *nd,
2                                              const char *link)
3 {
4     int res = 0;
5     char *name;
6     if (IS_ERR(link))

```

```
7     goto fail;
8
9     if (*link == '/') {
10        path_release(nd);
11        if (!walk_init_root(link, nd))
12            goto out;
13    }
14    res = link_path_walk(link, nd);
15    .....
16 }
```

这个函数再次调用 `link_path_walk()`, 见第429页代码片段12.50, 当这个函数返回时, 对符号链接的定位已经完成。

现在来看看 `do_lookup()` 函数, 这个函数负责从参数 `nd->dentry` 中寻找 `qstr` 指定的目录项, 它的定义如下:

代码片段 12.58 节自 `fs/namei.c`

```
1 static int do_lookup(struct nameidata *nd,
2                     struct qstr *name,
3                     struct path *path)
4 {
5     struct vfsmount *mnt = nd->mnt;
6     struct dentry *dentry = __d_lookup(nd->dentry, name);
7
8     if (!dentry)
9         goto need_lookup;
10    if (dentry->d_op && dentry->d_op->d_revalidate)
11        goto need_revalidate;
12 done:
13    path->mnt = mnt;
14    path->dentry = dentry;
15    __follow_mount(path);
16    return 0;
17
18 need_lookup:
19    dentry = real_lookup(nd->dentry, name, nd);
20    if (IS_ERR(dentry))
21        goto fail;
22    goto done;
23    .....
24 fail:
25    return PTR_ERR(dentry);
```

26 }

参数 `nd->dentry` 就是当前定位到的目录，而 `name` 就是要定位的目录项，以`/bin/bash`为例，假设当前定位到了`/bin`，那么 `nd->dentry` 就指向`/bin`对应的 `dentry` 对象，而 `name` 就指向 `bash`，这个函数就是要在 `bin` 目录下搜索 `bash`。在第6行首先调用 `_d_lookup()` 在缓存中搜索，如果失败，就在第19行调用 `real_lookup()` 尝试从磁盘上搜索。`_d_lookup()` 函数比较简单，我们来看看 `real_lookup()` 函数。

代码片段 12.59 节自 `fs/namei.c`

```

1 static struct dentry * real_lookup(struct dentry * parent,
2                                     struct qstr * name,
3                                     struct nameidata *nd)
4 {
5     struct dentry * result;
6     struct inode *dir = parent->d_inode;
7
8     mutex_lock(&dir->i_mutex);
9     result = d_lookup(parent, name);
10    if (!result) {
11        struct dentry * dentry = d_alloc(parent, name);
12        result = ERR_PTR(-ENOMEM);
13        if (dentry) {
14            result = dir->i_op->lookup(dir, dentry, nd);
15            if (result)
16                dput(dentry);
17            else
18                result = dentry;
19        }
20        mutex_unlock(&dir->i_mutex);
21        return result;
22    }
23    .....
24 }
```

由于在第8行 `mutex_lock()` 可能引起阻塞，当进程被唤醒时，要搜索的目录项可能已经在内存中了，所以这里再次调用 `d_lookup()` 尝试在内存中搜索，如果还是返回失败，那么调用 `d_alloc()` 分配一个 `dentry` 对象，然后调用具体文件系统的 `i_op->lookup` 函数，在 Ext2 中，这个函数是 `ext2_lookup()`。读者可以自己阅读 `d_alloc()` 的代码，这里要提醒的是，`d_alloc()` 会把 `name` 复制到新分配的 `dentry` 对象中。现在来看看 `ext2_lookup()`。

代码片段 12.60 节自 `fs/ext2/namei.c`

```
1 static struct dentry *ext2_lookup(struct inode * dir,
```

```

2                         struct dentry *dentry,
3                         struct nameidata *nd)
4 {
5     struct inode * inode;
6     ino_t ino;
7
8     if (dentry->d_name.len > EXT2_NAME_LEN)
9         return ERR_PTR(-ENAMETOOLONG);
10    ino = ext2_inode_by_name(dir, dentry);
11    inode = NULL;
12    if (ino) {
13        inode = igrab(dir->i_sb, ino);
14        if (!inode)
15            return ERR_PTR(-EACCES);
16    }
17    return d_splice_alias(inode, dentry);
18 }

```

假设要定位的路径/bin/bash，当前已经定位到/bin，现在要在 bin 目录中搜索 bash，那么调用 ext2_lookup()时，参数 dir 指向/bin 对应的 inode，参数 dentry 指向新分配的 dentry 对象，ext2_lookup()会根据磁盘上的信息，进一步设置 dentry 对象。在第10行调用 ext2_inode_by_name()函数，从/bin 目录项中获取 bash 对应的 inode 号(见第380页代码片段12.2)，然后在第13行调用 igrab()，根据 inode 号获取 inode 对象。

12.6 文件打开与关闭

文件打开由 sys_open()函数实现，它主要是建立进程和文件的联系。sys_open()根据路径字符串搜索相应文件的 dentry 和 inode 结构，并且初始化 VFS 的 file 对象，同时把 file 对象的指针保存在进程的 files_struct 结构的一个数组中，数组的下标就是我们通常说的文件号 fd。这样应用程序的 fd 就对应内核中的一个 file 对象。现在来看看 sys_open()函数。

代码片段 12.61 节自 `fs/open.c`

```

1 asmlinkage long sys_open(const char __user *filename,
2                           int flags,
3                           int mode)
4 {
5     long ret;
6
7     if (force_o_largefile())
8         flags |= O_LARGEFILE;

```

```

9   ret = do_sys_open(AT_FDCWD, filename, flags, mode);
10  /* avoid REGPARM breakage on x86: */
11  prevent_tail_call(ret);
12  return ret;
13 }

```

这个函数主要调用 `do_sys_open()`, 其中 `AT_FDCWD`, 表示以进程的当前目录作为起始目录, 来搜索由 `filename` 指定的文件。

代码片段 12.62 节自 `fs/open.c`

```

1 long do_sys_open(int dfd,
2                   const char __user *filename,
3                   int flags,
4                   int mode) {
5
6     char *tmp = getname(filename);
7     int fd = PTR_ERR(tmp);
8
9     if (!IS_ERR(tmp)) {
10        fd = get_unused_fd_flags(flags);
11        if (fd >= 0) {
12            struct file *f = do_filp_open(dfd, tmp, flags, mode);
13            if (IS_ERR(f)) {
14                put_unused_fd(fd);
15                fd = PTR_ERR(f);
16            } else {
17                fsnotify_open(f->f_path.dentry);
18                fd_install(fd, f);
19            }
20        }
21        putname(tmp);
22    }
23    return fd;
24 }

```

首先在第10行调用 `get_unused_fd_flags()` 获取一个空闲的文件描述符, 之后在第12行调用 `do_filp_open()`, 它返回一个 `file` 对象的指针, 如果返回成功, 在第18行调用 `fd_install()`, `fd_install()` 把 `file` 指针保存到 `fdtable` 数组中。`fd_install()` 很简单, 这里我们来看看 `do_filp_open()`。

代码片段 12.63 节自 `fs/open.c`

```
1 static struct file *do_filp_open(int dfd,
```

```

2                     const char *filename,
3                     int flags,
4                     int mode)
5 {
6     int namei_flags, error;
7     struct nameidata nd;
8
9     namei_flags = flags;
10    if (((namei_flags+1) & O_ACCMODE)
11        namei_flags++;
12    error = open_namei(dfd, filename, namei_flags, mode, &nd);
13    if (!error)
14        return nameidata_to_filp(&nd, flags);
15    return ERR_PTR(error);
16 }

```

`do_filp_open()`定义了一个`nameidata`结构`nd`(关于`nameidata`结构的描述, 见第427页代码片段12.47), 然后调用`open_namei()`函数, 其结果通过`nd`返回。`open_namei()`函数比较复杂, 它需要根据调用`open()`时的标志做不同的处理, 例如, 如果设置了`O_CREAT`, 那么当文件不存在时就要创建文件。由于该函数需要根据不同的标志进行不同的处理, 也要调用不同的函数, 而笔者也没有好的办法同时展开多条线路进行讨论, 这里我们以打开一个已经存在的为主线, 来分析`open_namei()`。

代码片段 12.64 节自 fs/namei.c

```

1 int open_namei(int dfd, const char *pathname,
2                 int flag, int mode,
3                 struct nameidata *nd)
4 {
5     int acc_mode, error;
6     struct path path;
7     struct dentry *dir;
8     int count = 0;
9
10    acc_mode = ACC_MODE(flag);
11    /* O_TRUNC implies we need access checks for write permissions */
12    if (flag & O_TRUNC)
13        acc_mode |= MAY_WRITE;
14    /*
15     * Allow the LSM permission hook to distinguish append
16     * access from general write access.
17     */
18    if (flag & O_APPEND)

```

```
19     acc_mode |= MAY_APPEND;
20
21     /*
22      * The simplest case - just a plain lookup.
23      */
24
25     /* 如果没有设置 O_CREAT 标志。*/
26     if (!(flag & O_CREAT)) {
27         error = path_lookup_open(dfd, pathname,
28                               lookup_flags(flag),
29                               nd, flag);
30
31         if (error)
32             return error;
33         goto ok;
34     }
35
36     .....
37
38 ok:
39
40     error = may_open(nd, acc_mode, flag);
41     if (error)
42         goto exit;
43
44     return 0;
45 }
```

`path_lookup_open()` 定义如下：

代码片段 12.65 节自 fs/namei.c

path lookup intent open() 定义如下：

代码片段 12.66 节自 fs/namei.c

```

7  /* 获取一个新的 file 对象。*/
8  struct file *filp = get_empty_filp();
9  int err;
10
11 if (filp == NULL)
12     return -ENFILE;
13 /* 把 file 对象指针保存到 nd 中。*/
14 nd->intent.open.file = filp;
15 nd->intent.open.flags = open_flags;
16 nd->intent.open.create_mode = create_mode;
17 /* 路径定位。*/
18 err = do_path_lookup(dfd, name, lookup_flags|LOOKUP_OPEN, nd);
19 if (IS_ERR(nd->intent.open.file)) {
20     if (err == 0) {
21         err = PTR_ERR(nd->intent.open.file);
22         path_release(nd);
23     }
24 } else if (err != 0)
25     release_open_intent(nd);
26 return err;
27 }
```

这个函数分配一个 file 对象并保存在 nd 中，然后调用 do_path_lookup() 函数，根据路径字符串 name 定位到目标文件，do_path_lookup() 函数在 12.5 节已经讨论过了，再这里不再重复，当这个函数返回时，结果保存在 nd 中。现在回到 do_filp_open()。

代码片段 12.67 节自 fs/open.c

```

1 static struct file *do_filp_open(int dfd, const char *filename,
2                                 int flags, int mode)
3 {
4     .....
5     error = open_namei(dfd, filename, namei_flags, mode, &nd);
6     if (!error)
7         return nameidata_to_filp(&nd, flags);
8     return ERR_PTR(error);
9 }
```

open_namei() 的结果保存在 nd 中，当它返回后就调用 nameidata_to_filp() 获取 file 对象的指针。

代码片段 12.68 节自 fs/open.c

```

1 struct file *nameidata_to_filp(struct nameidata *nd, int flags)
2 {
```

```

3   struct file *filp;
4
5   /* Pick up the filp from the open intent */
6   filp = nd->intent.open.file;
7   /* 如果下层具体文件系统没有初始化 file 对象。 */
8   /* Has the filesystem initialised the file for us? */
9   if (filp->f_path.dentry == NULL)
10    filp = __dentry_open(nd->dentry, nd->mnt, flags, filp, NULL);
11 else
12    path_release(nd);
13 return filp;
14 }

```

假设要打开的文件路径为 /bin/bash，现在 nd->dentry 已经指向 bash 对应的目录项了，而如果下层具体文件系统没有初始化 file 对象，就调用 __dentry_open() 函数来初始化。

代码片段 12.69 节自 fs/open.c

```

1 /* 注意在上面的调用中，最后一个参数为 NULL. */
2 static struct file *__dentry_open(struct dentry *dentry,
3                                     struct vfsmount *mnt,
4                                     int flags, struct file *f,
5                                     int (*open)(struct inode *, struct file *));
6 {
7   struct inode *inode;
8   int error;
9
10  f->f_flags = flags;
11  f->f_mode = ((flags+1) & O_ACCMODE) | FMODE_LSEEK |
12      FMODE_PREAD | FMODE_PWRITE;
13  inode = dentry->d_inode;
14  if (f->f_mode & FMODE_WRITE) {
15    error = get_write_access(inode);
16    if (error)
17      goto cleanup_file;
18  }
19
20  f->f_mapping = inode->i_mapping;
21  f->f_path.dentry = dentry;
22  f->f_path.mnt = mnt;
23  f->f_pos = 0;
24
25  /* 把 file 对象的 f_op 设置为 inode 对象中的 i_fop. 见图12.4。 */
26  f->f_op = fops_get(inode->i_fop);

```

```

27     file_move(f, &inode->i_sb->s_files);
28     error = security_dentry_open(f);
29     if (error)
30         goto cleanup_all;
31     /* 如果参数中的 open 函数指针为 NULL, 就调用下层具体文件系统提供的 open() 函数。 */
32     if (!open && f->f_op)
33         open = f->f_op->open;
34     if (open) {
35         error = open(inode, f);
36         if (error)
37             goto cleanup_all;
38     }
39
40     f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);
41     file_ra_state_init(&f->f_ra, f->f_mapping->host->i_mapping);
42     /* NB: we're sure to have correct a_ops only after f_op->open */
43     if (f->f_flags & O_DIRECT) {
44         if (!f->f_mapping->a_ops ||
45             ((!f->f_mapping->a_ops->direct_IO) &&
46              (!f->f_mapping->a_ops->get_xip_page))) {
47             fput(f);
48             f = ERR_PTR(-EINVAL);
49         }
50     }
51
52     return f;
53 cleanup_all:
54     .....
55 }

```

文件关闭由 `sys_close()` 函数实现:

代码片段 12.70 节自 `fs/open.c`

```

1 asmlinkage long sys_close(unsigned int fd)
2 {
3     struct file * filp;
4
5     /* 获取当前进程的 files_struct 结构。 */
6     struct files_struct *files = current->files;
7     struct fdtable *fdt;
8     int retval;
9     spin_lock(&files->file_lock);
10    /* 获取 fdtable. */

```

```

11    fdt = files_fdtable(files);
12    if (fd >= fdt->max_fds)
13        goto out_unlock;
14    /* 以 fd 为数组下标，从 fdtable 中获取 file 对象指针。 */
15    filp = fdt->fd[fd];
16    if (!filp)
17        goto out_unlock;
18
19    /* 清空 fd 和对应的位图项。 */
20    rCU_assign_pointer(fdt->fd[fd], NULL);
21    FD_CLR(fd, fdt->close_on_exec);
22    __put_unused_fd(files, fd);
23    spin_unlock(&files->file_lock);
24
25    /* 关闭文件。 */
26    retval = filp_close(filp, files);
27    /* can't restart close syscall because file table entry was cleared */
28    if (unlikely(retval == -ERESTARTSYS ||
29        retval == -ERESTARTNOINTR ||
30        retval == -ERESTARTNOHAND ||
31        retval == -ERESTART_RESTARTBLOCK))
32        retval = -EINTR;
33
34    return retval;
35 out_unlock:
36    spin_unlock(&files->file_lock);
37    return -EBADF;
38 }

```

根据 fd 获取到对应的 file 对象后，由 filp_close() 执行文件关闭操作，应用程序可以通过 dup(fd) 调用复制一个文件描述符，它对应内核中的 sys_dup() 系统调用，这个函数会在 fdtable 数组中找到一个空闲项，然后会复制 fd 对应的 file 对象的指针，同时返回新的数组下标，这样就有两个不同的 fd 对应同一个 file 对象。所以当进程调用 sys_close() 时，fd 是可以清除了，但是对应的 file 对象却不一定被释放。

代码片段 12.71 节自 fs/open.c

```

1 int filp_close(struct file *filp, fl_owner_t id)
2 {
3     int retval = 0;
4
5     if (!file_count(filp)) {
6         printk(KERN_ERR "VFS: Close: file count is 0\n");

```

```
7     return 0;
8 }
9 /* 调用下层具体文件系统的 flush() 函数，它负责把内存中的数据更新到磁盘上。*/
10 if (filp->f_op && filp->f_op->flush)
11     retval = filp->f_op->flush(filp, id);
12 dnotify_flush(filp, id);
13 locks_remove_posix(filp, id);
14 /* 减小 file 对象的引用计数，当引用计数为 0 时就释放它。*/
15 fput(filp);
16 return retval;
17 }
```

12.7 文件读写

在打开文件时，已经根据磁盘上的 inode 结构设置了内存的 inode 结构，文件的数据块地址就保存在 inode 对象的 `i_block` 数组中，同时 VFS 的 `file` 对象中有一个 `f_pos` 成员，它保存着文件的当前位置指针，因此读操作就根据 `i_block`, `f_pos` 和要读取的字节数，计算出需要读取的数据块地址，然后向磁盘驱动程序发送读操作命令。由于磁盘操作相对较慢，因此根据局部性原理，引入了缓冲区管理和预读机制。假设现在要读取 n 个字节，而缓冲层实际上会读取 m 个字节 ($m > n$) 到缓冲区，此后当进程再次进行读操作时，缓冲区管理层首先查看读取的目标数据是否已经在缓冲区中，只有当缓冲不命中时，缓冲区管理层才会向磁盘驱动程序发出读操作。

同样对于写操作来说，也要根据 `i_block`, `f_pos` 和要写入的字节数，计算出磁盘的数据块地址，如果磁盘上的数据块不够，还需要在磁盘上分配新的数据块，然后把这个数据块地址填入对应的 `i_block` 中，最后向磁盘驱动程序发出写入操作命令。为了提高性能，在文件系统层和磁盘驱动程序之间加入了缓冲区管理层。当写请求到达缓冲区管理层，缓存管理模块仅仅是把要写入的内容复制到相应的缓存区中，并把缓冲区标记为 `dirty`，就返回了。缓冲区管理模块将在适当的时候统一把缓存的内容写到磁盘。

12.7.1 缓冲区管理

文件内容和缓存在逻辑上是一一对应的映射关系，假设一个文件大小为 1GB，假设当前文件指针 `f_pos` 的偏移是 512MB 字节处，现在需要读取 1MB 字节的内容，把 1GB 的内容映射到一个 1GB 的虚拟地址空间是不现实的，因此我们使用 radix tree 来管理文件内容和缓冲区之间的对应关系。文件 `inode` 和 radix tree 的关系如图 12.5 所示。每一个文件的 `inode` 中，有一个类型为 `address_space` 的指针 `i_mapping`, `address_space` 结构中的 `page_tree` 是一个 `radix_tree_root` 的指针，`radix_tree_root` 定义如代码片段 12.73。

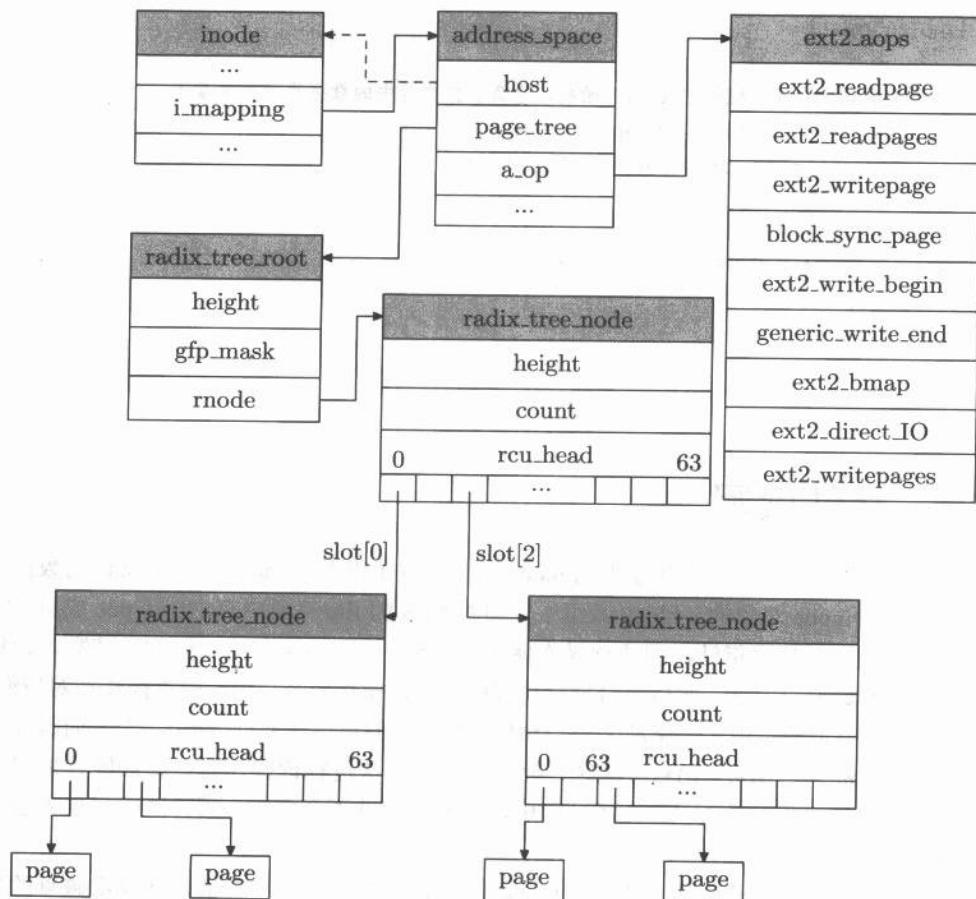


图 12.5 文件缓存管理示意图

代码片段 12.72 (节自 include/linux/radix-tree.h)

```

1 struct radix_tree_root {
2     /* 树的高度。 */
3     unsigned int    height;
4     gfp_t          gfp_mask;
5     struct radix_tree_node *rnode;
6 };

```

`radix_tree_root` 中的 `height` 表示树的高度，在图12.5中，`radix tree` 的高度为 2。`mode` 指向树的节点，节点的类型为 `radix_tree_node`，定义如下：

代码片段 12.73 (节自 include/linux/radix-tree.h)

```
1 struct radix_tree_node {
```

```

2  /* Height from the bottom */
3  unsigned int height;
4  unsigned int count;
5  struct rCU_head rCU_head;
6  void *slots[RADIX_TREE_MAP_SIZE];
7  unsigned long tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
8 };

```

`radix_tree_node` 中的 `height` 表示底部到当前节点的高度，例如图12.5中，第一级 `radix_tree_node` 的高度为 1，第二级 `radix_tree_node` 的高度为 0(叶子节点)。`slots` 是一个指针数组，数组大小为默认 64。在非叶子节点中，`slots[n]` 指向下一级的 `radix_tree_node` 结构，在叶子节点中，`slots[n]` 指向一个 `page` 结构。每一个 `page` 结构表示一个有效的页面。

让我们用一个例子来说明 `radix tree` 和文件内容之间的关系。假设 `radix tree` 的高度为 1，页大小为 4KB，由于 `slots` 大小为 64，因此高度为 1 的 `radix tree` 能够映射的最大值是 256KB($4KB \times 64$)。如果当前的文件指针 `f_pos` 为 9KB，现在要读取 1KB 的内容，其过程如下：

- (1) 计算 $9KB / 4KB$ ，结果为 2，于是从 `radix_tree_node` 中找到 `slots[2]`。
- (2) 如果 `slots[2]` 为 NULL，说明缓存中没有对应的内容，于是需要请求磁盘驱动程序读取这部分内容。
- (3) 如果 `slots[2]` 不为 NULL，则缓存命中，根据 `slots[2]` 中的 `page` 读取对应的内容。

需要注意的是在 `radix tree` 的映射中，是允许存在“空洞”的。例如在这个例子中，`slots[0]` 指向第一个 4KB，`slots[1]` 为 NULL，而 `slots[2]` 却指向了第三个 4KB。因此 `radix tree` 可以处理一个很大的文件，而不受物理内存大小的限制。如果高度为 2，那么在第一级节点中，每一个 `slots` 指针对应一个 256KB 的映射空间，其计算公式为 $64 * (\text{height}-1) * 4KB$ 。在页大小为 4 的情况下，`radix tree` 的高度和文件最大尺寸的如表12.2所示。

表 12.2 `radix` 树的高度与文件最大尺寸对于关系

<code>radix tree</code> 的高度	最大文件尺寸
0	0
1	256KB
2	16MB
3	1GB
4	64GB
5	4TB
6	16TB

下面我们来看看在 radix tree 中查找指定页面的过程。这个工作由 `find_get_page()` 函数完成，函数定义如下：

代码片段 12.74 (节自 mm/filemap.c)

```

1 struct page * find_get_page(struct address_space *mapping,
2                               pgoff_t offset)
3 {
4     struct page *page;
5
6     read_lock_irq(&mapping->tree_lock);
7     page = radix_tree_lookup(&mapping->page_tree, offset);
8     if (page)
9         page_cache_get(page);
10    read_unlock_irq(&mapping->tree_lock);
11    return page;
12 }
```

`find_get_page()` 接收两个参数，`mapping` 指向文件的 `address_space` 对象，`offset` 指向要查找的页面号，如果要读取的起始位置为 5KB，那么 `offset` 为 1，($5\text{KB} \% 4\text{KB} = 1$)。`find_get_page()` 首先调用 `radix_tree_lookup()` 在 radix tree 中搜索，如果成功，`radix_tree_lookup()` 将返回对应的 `page` 结构。`radix_tree_lookup()` 定义如下：

代码片段 12.75 (节自 lib/radix-tree.c)

```

1 void *radix_tree_lookup(struct radix_tree_root *root,
2                         unsigned long index)
3 {
4     unsigned int height, shift;
5     struct radix_tree_node *node, **slot;
6
7     node = rcu_dereference(root->rnode);
8     if (node == NULL)
9         return NULL;
10
11    if (!radix_tree_is_indirect_ptr(node)) {
12        if (index > 0)
13            return NULL;
14        return node;
15    }
16    /* 获取子节点指针。*/
17    node = radix_tree_indirect_to_ptr(node);
18    height = node->height;
19    /*
```

```

20     * radix_tree_maxindex() 根据高度 height, 计算出能够映射的最大
21     * 尺寸(见表12.2)的页面号, 如果 index 超过这个页面号,
22     * 就返回 NULL.
23     */
24     if (index > radix_tree_maxindex(height))
25         return NULL;
26     /* RADIX_TREE_MAP_SHIFT 默认为 6. */
27     shift = (height-1) * RADIX_TREE_MAP_SHIFT;
28     do {
29         /* RADIX_TREE_MAP_MASK 默认为  $2^6 - 1$ , 也就是 0x3F, 低 6 位有效. */
30         slot = (struct radix_tree_node **) (node->slots +
31                                         ((index>>shift) & RADIX_TREE_MAP_MASK));
32         node = rcu_dereference(*slot);
33         if (node == NULL)
34             return NULL;
35         shift -= RADIX_TREE_MAP_SHIFT;
36         height--;
37     } while (height > 0);
38
39     return node;
40 }

```

为了更加容易理解, 我们使用一个例子来分析这段代码。假设 radix tree 的高度为 1, 这时在第27行中, shift 为 0, 进入 do-while 循环, 第30行中 node->slots[index]就是 index 指定的页面指针。之后 height--为 0, do-while 循环结束。

当 radix tree 的高度为 2 时, 在第27行中, shift 为 6, 进入 do-while 循环, 第30行, node->slots[(index>>6)&0x3F]就是下一级的 radix_tree_node 指针, 这意味着低 6 位用于下一级的 slots 数组索引, 之后 shift-6, 结构为 0, height--为 1, 进入下一轮循环, 此时 node->slots[(index>>0)&0x3F]就是 index 指定的页面指针, 实际上就是取 index 的低 6 位作为这一级的 slots 数组索引, 之后循环结束。这样我们理解高度为 n 的计算过程了。

当进行文件读操作时, 首先利用 find_get_page()在缓冲区中查找, 如果缓冲区命中失败, 就会请求磁盘设备驱动程序从磁盘介质上读取相应的内容。此后, 需要把读取出来的内容添加到 radix tree 中, 这个工作由 add_to_page_cache()函数完成。函数定义如下:

代码片段 12.76 (节自 mm/filemap.c)

```

1 int add_to_page_cache(struct page *page,
2                     struct address_space *mapping,
3                     pgoff_t offset,
4                     gfp_t gfp_mask)
5 {
6     int error = radix_tree_preload(gfp_mask & ~__GFP_HIGHMEM);

```

```

7
8     if (error == 0) {
9         write_lock_irq(&mapping->tree_lock);
10        error = radix_tree_insert(&mapping->page_tree, offset, page);
11        if (!error) {
12            page_cache_get(page);
13            SetPageLocked(page);
14            page->mapping = mapping;
15            page->index = offset;
16            mapping->nrpages++;
17            __inc_zone_page_state(page, NR_FILE_PAGES);
18        }
19        write_unlock_irq(&mapping->tree_lock);
20        radix_tree_preload_end();
21    }
22    return error;
23 }

```

把一个页面添加到 radix tree 中的过程中可能需要增加树的高度，在增加树的高度时，需要分配新的 radix_tree_node 结构。在调整树的高度过程中，可能进行到一定阶段，由于内存资源不足，不能再分配 radix_tree_node 结构，这时调整失败，同时还改变了以前的状态，为了保证内核的健壮性，就必须能够回退到进行高度调整之前的状态，这样会使代码变得复杂。因此为每一个 CPU 设置了一个 percpu 变量 radix_tree_preloads，它保存了一定数量预先分配好的 radix_tree_node 结构，在调整之前，就调用 radix_tree_preload() 函数检查预分配的 radix_tree_node 数量是否低于某个阈值，如果低于这个阈值，radix_tree_preload() 就会分配一定数量的 radix_tree_node，从而保证后续步骤中不会出现上述极端的情况。radix_tree_preload() 函数定义如下：

代码片段 12.77 (节自 mm/filemap.c)

```

1 int radix_tree_preload(gfp_t gfp_mask)
2 {
3     struct radix_tree_preload *rtp;
4     struct radix_tree_node *node;
5     int ret = -ENOMEM;
6
7     preempt_disable();
8     rtp = &__get_cpu_var(radix_tree_preloads);
9     /*
10      * 如果预分配的 radix_tree_node 的数量低于 ARRAY_SIZE(rtp->nodes),
11      * 就预分配足量的 radix_tree_node 结构。
12      */
13     while (rtp->nr < ARRAY_SIZE(rtp->nodes)) {

```

```
14     preempt_enable();
15     node = kmem_cache_alloc(radix_tree_node_cachep,
16                             set_migrateflags(gfp_mask, __GFP_RECLAMABLE));
17     if (node == NULL)
18         goto out;
19     preempt_disable();
20     rtp = &__get_cpu_var(radix_tree_preloads);
21     if (rtp->nr < ARRAY_SIZE(rtp->nodes))
22         rtp->nodes[rtp->nr++] = node;
23     else
24         kmem_cache_free(radix_tree_node_cachep, node);
25     }
26     ret = 0;
27 out:
28     return ret;
29 }
```

如果 radix_tree_preload()返回成功, add_to_page_cache()就可以放心地调用 radix_tree_insert(), 把指定的 page 插入 radix tree 中。radix_tree_insert()定义如下:

代码片段 12.78 (节自 lib/radix-tree.c)

```
1 int radix_tree_insert(struct radix_tree_root *root,
2                         unsigned long index,
3                         void *item)
4 {
5     struct radix_tree_node *node = NULL, *slot;
6     unsigned int height, shift;
7     int offset;
8     int error;
9
10    BUG_ON(radix_tree_is_indirect_ptr(item));
11
12    /*
13     * 如果当前要增加的 page 结构(由 index 指定)对应的映射地址超过了当前
14     * 高度能表示的最大地址, 就调用 radix_tree_extend() 调整树的高度。
15     */
16
17    /* Make sure the tree is high enough. */
18    if (index > radix_tree_maxindex(root->height)) {
19        error = radix_tree_extend(root, index);
20        if (error)
21            return error;
22    }
23    slot = radix_tree_indirect_to_ptr(root->rnode);
24    height = root->height;
```

```

23     shift = (height-1) * RADIX_TREE_MAP_SHIFT;
24     offset = 0;
25     /* uninitialised var warning */
26     while (height > 0) {
27         if (slot == NULL) {
28             /* Have to add a child node. */
29             if (!(slot = radix_tree_node_alloc(root)))
30                 return -ENOMEM;
31             slot->height = height;
32             if (node) {
33                 rcu_assign_pointer(node->slots[offset], slot);
34                 node->count++;
35             } else
36                 rcu_assign_pointer(root->rnode,
37                     radix_tree_ptr_to_indirect(slot));
38         }
39
40         /* Go a level down */
41         offset = (index >> shift) & RADIX_TREE_MAP_MASK;
42         node = slot;
43         slot = node->slots[offset];
44         shift -= RADIX_TREE_MAP_SHIFT;
45         height--;
46     }
47     .....
48     return 0;
49 }

```

高度调整完成之后，就找到 index 对应的 slots，这个定位过程和前面的 radix_tree_lookup() 函数中的查找过程类似，但是这里要多一个步骤，如果在到达叶子节点之前，slots[n] 为 NULL，这时就需要分配一个 radix_tree_node 结构，作为中间节点。例如：树的高度为 2，当前 index 对应的页面定位过程为，先定位第一级的 slots[n]，然后根据 slots[n] 定位到第二级的 slots[m]，此时如果发现 slots[n] 为 NULL，就需要分配一个新的 radix_tree_node，并设置相应的指针。最后再把 index 指定的 page 添加到叶子节点的 slots[m] 中。

12.7.2 文件读写操作分析

由于加入了缓冲区管理层，因此文件读写被可分为以下几种方式：

1. 普通模式

对于读取操作，如果要读取的数据不在缓存中，则阻塞当前进程，直到磁盘驱动程序

准备好数据。对于写操作，把数据复制到缓冲区之后，立即返回，由缓冲区管理模块将在适当的时候向磁盘驱动程序发起写入请求。

2. 同步模式

对于读取操作，如果要读取的数据不在缓存中，则阻塞当前进程，直到磁盘驱动程序准备好数据。对于写操作，把数据复制到缓冲区后，立即向磁盘驱动程序发出写入请求，并阻塞当前进程，当磁盘驱动程序完成写入操作之后，再唤醒该进程。

3. 直接 IO 模式

对于读写操作，直接跳过缓冲区管理层，直接向磁盘驱动发出读写命令。

4. 异步模式

读磁盘进行读写操作时，如果数据没有准备好，向磁盘驱动程序发出读写命令后，立即返回，并不阻塞当前进程，当驱动程序完成操作时，会通过事件的方式通知相应的进程。在 Linux 内核中，异步 IO 操作有两套机制，一套是 c 库中实现的，和内核没有关系(c 库提供 aio_read()/aio_write()等函数)。另外一套是在内核中实现的(通过系统 c 库提供的系统调用接口 io_setup()/io_submit()等函数实现的)。c 库实现的不能算真正的异步 IO，它其实是在 aio_read()这一类的库函数中调用 fork()之类的函数建立一个新的进程，然后子进程被阻塞，父进程可以立即返回继续执行。

5. 文件映射方式

通过 mmap()建立文件映射，这样就可以通过数组一样直接访问内存，而不需要调用 read(), write() 等函数，对于读操作，如果内容不在内存中，则触发缺页异常，由异常处理程序读入相应的数据，之后进程继续执行。

明白了上面的这个过程之后，现在我们来看看文件读取操作的具体过程。读操作的入口是 sys_read()，定义如下：

代码片段 12.79 (节自 fs/read_write.c)

```
1 asmlinkage ssize_t sys_read(unsigned int fd,
2                               char __user * buf,
3                               size_t count)
4 {
5     struct file *file;
6     ssize_t ret = -EBADF;
7     int fput_needed;
8
9     /* 根据 fd 获取文件对象 file 结构。 */
10    file = fget_light(fd, &fput_needed);
11    if (file) {
12        /* 获取文件当前读取位置指针 f_pos. */
```

```

13     loff_t pos = file_pos_read(file);
14     /* 调用 VFS 层的读操作函数。*/
15     ret = vfs_read(file, buf, count, &pos);
16     /* 调整文件当前读取位置指针 f_pos。*/
17     file_pos_write(file, pos);
18     fput_light(file, fput_needed);
19 }
20 return ret;
21 }
```

vfs_read()函数定义如下：

代码片段 12.80 (节自 fs/read_write.c)

```

1 [sys_read() -> vfs_read()]
2
3 ssize_t vfs_read(struct file *file,
4                   char __user *buf,
5                   size_t count,
6                   loff_t *pos)
7 {
8     ssize_t ret;
9
10    if (!(file->f_mode & FMODE_READ))
11        return -EBADF;
12    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
13        return -EINVAL;
14    /* 参数 buf 是用户态的内存指针, access_ok() 检查这片内存区域是否可以访问。*/
15    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
16        return -EFAULT;
17    /* 检查文件是否可读。*/
18    ret = rw_verify_area(READ, file, pos, count);
19    if (ret >= 0) {
20        count = ret;
21        /* 安全检查接口, 默认为空操作。*/
22        ret = security_file_permission(file, MAY_READ);
23        if (!ret) {
24            if (file->f_op->read)
25                ret = file->f_op->read(file, buf, count, pos);
26            else
27                ret = do_sync_read(file, buf, count, pos);
28            if (ret > 0) {
29                fsnotify_access(file->f_path.dentry);
30                add_rchar(current, ret);
31            }
32        }
33    }
34 }
```

```
31     }
32     inc_syscr(current);
33 }
34 }
35 return ret;
36 }
```

在文件打开操作过程中，ext2文件系统的file->f_op指向ext2_file_operations，该结构定义如下：

代码片段 12.81 (节自fs/ext2/file.c)

```
1 const struct file_operations ext2_file_operations = {
2     .llseek    = generic_file_llseek,
3     .read      = do_sync_read,
4     .write     = do_sync_write,
5     .aio_read  = generic_file_aio_read,
6     .aio_write = generic_file_aio_write,
7     .....
8 };
```

因此，vfs_read()中将调用do_sync_read()，do_sync_write()定义如下：

代码片段 12.82 (节自fs/read_write.c)

```
1 [sys_read() -> vfs_read() -> do_sync_read()]
2
3 ssize_t do_sync_read(struct file *filp,
4                       char __user *buf,
5                       size_t len,
6                       loff_t *ppos)
7 {
8     struct iovec iov = { .iov_base = buf, .iov_len = len };
9     struct kiocb kiocb;
10    ssize_t ret;
11
12    init_sync_kiocb(&kiocb, filp);
13    kiocb.ki_pos = *ppos;
14    kiocb.ki_left = len;
15
16    for (;;) {
17        ret = filp->f_op->aio_read(&kiocb, &iov, 1, kiocb.ki_pos);
18        if (ret != -EIOCBRETRY)
19            break;
20        wait_on_retry_sync_kiocb(&kiocb);
```

```

21     }
22     if (-EIOCBQUEUED == ret)
23         ret = wait_on_sync_kiocb(&kiocb);
24     *ppos = kiocb.ki_pos;
25     return ret;
26 }

```

`do_sync_read()`把文件的读缓存区地址和长度保存到 `iovec` 结构中，把目标文件指针和文件位置指针等信息保存到 `kiocb` 结构中，然后调用 `f_op` 中的 `aio_read()`，在前面我们看到 `ext2_file_operation` 结构中的 `aio_read` 函数指针指向 `generic_file_aio_read()` 函数。`generic_file_aio_read()` 函数定义如下：

代码片段 12.83 (节自 mm/filemap.c)

```

1 [sys_read() -> vfs_read() -> do_sync_read() -> generic_file_aio_read()]
2
3 ssize_t generic_file_aio_read(struct kiocb *kiocb,
4                                const struct iovec *iov,
5                                unsigned long nr_segs,
6                                loff_t pos)
7 {
8     struct file *filp = kiocb->ki_filp;
9     ssize_t retval;
10    unsigned long seg;
11    size_t count;
12    loff_t *ppos = &kiocb->ki_pos;
13
14    count = 0;
15    retval = generic_segment_checks(iov, &nr_segs, &count, VERIFY_WRITE);
16    if (retval)
17        return retval;
18    /*
19     * 对于设置了 O_DIRECT 标志的文件，调用 generic_file_direct_IO()
20     * 跳过缓冲区管理模块。
21     */
22    /* coalesce the iovecs and go direct-to-BIO for O_DIRECT */
23    if (filp->f_flags & O_DIRECT) {
24        .....
25        size = i_size_read(inode);
26        if (pos < size) {
27            retval = generic_file_direct_IO(READ, kiocb, iov, pos, nr_segs);
28            if (retval > 0)
29                *ppos = pos + retval;

```

```
30     }
31     if (likely(retval != 0)) {
32         file_accessed(filp);
33         goto out;
34     }
35 }
36
37 retval = 0;
38 if (count) {
39     for (seg = 0; seg < nr_segs; seg++) {
40         read_descriptor_t desc;
41
42         desc.written = 0;
43         desc.arg.buf = iov[seg].iov_base;
44         desc.count = iov[seg].iov_len;
45         if (desc.count == 0)
46             continue;
47         desc.error = 0;
48         do_generic_file_read(filp, ppos, &desc, file_read_actor);
49         retval += desc.written;
50         if (desc.error) {
51             retval = retval ?: desc.error;
52             break;
53         }
54         if (desc.count > 0)
55             break;
56     }
57 }
58 out:
59 return retval;
60 }
```

如果目标文件在打开操作时设置了 O_DIRECT 标志，在第27行调用 generic_file_direct_IO() 函数，跳过缓冲区管理层，直接把读请求传送到块设备驱动层。否则在第48行，调用 do_generic_file_read()，这是缓冲区管理层的通用函数，在默认情况下，任何文件系统的读请求都是通过这个函数传递到缓冲区管理层的。在这里 sys_read() 传递的地址、长度等信息又被复制到 read_descriptor_t 结构中，同时 do_generic_file_read() 的最后一个参数为 file_read_actor，这是一个函数指针，将来缓冲区管理层把数据准备好之后，会调用 file_read_actor() 把相关内容从内核态复制到用户态空间。do_generic_file_read() 函数定义如下：

代码片段 12.84 (节自 include/linux/fs.h)

```

1 [sys_read() -> vfs_read() -> do_sync_read() -> generic_file_aio_read() ->
   do_generic_file_read()]

2
3 static inline void do_generic_file_read(struct file * filp,
4                                         loff_t * ppos,
5                                         read_descriptor_t * desc,
6                                         read_actor_t actor)
7 {
8     do_generic_mapping_read(filp->f_mapping,
9                             &filp->f_ra,
10                            filp,
11                            ppos,
12                            desc,
13                            actor);
14 }

```

do_generic_file_read()是do_generic_mapping_read()的一个包装，第一个参数f_mapping就是前面讨论过的address_space结构(见449页，图12.5)。

do_generic_mapping_read()是一个很庞大的函数，为了方便阅读，在这里先对它的基本工作做一个大概的介绍，有了整体认识的基础后再对具体的代码进行分析。

首先，do_generic_mapping_read()在radix tree中查找指定的页面，如果找到并且内容是最新的，就可以直接从缓存中读取。如果查找失败，就需要分配新的页面，并把它加入到radix tree中，然后再请求块设备驱动层读取相应内容。这里有两点需要注意：

- (1) 由于是先把新页面加入到radix tree中，然后再向下层块设备驱动发出请求，因此在radix tree的页面查找过程中，必须判断找到的页面内容是否为最新的，如果缺少这个判断，可能会读取到错误的信息。例如：进程A发起读取某个页面请求，且缓存命中失败，于是分配新的页面加入到radix tree中，然后请求下层块设备驱动层把内容读取到这个页面中，在读操作结束前，进程B可能会读取同样的一个页面。
- (2) 根据局部性原理，缓冲区管理层通常提供了预读机制，Linux内核中有着非常复杂的预读算法，这里我们不对预读算法进行探讨，请感兴趣的读者参考其他相关资料。

现在，我们来分段阅读do_generic_mapping_read()的代码：

代码片段 12.85 (节自 mm/filemap.c)

```

1 [sys_read() -> vfs_read() -> do_sync_read() ->
2  generic_file_aio_read() -> do_generic_file_read() ->
3  do_generic_mapping_read()]

4
5 void do_generic_mapping_read(struct address_space *mapping,

```

```
6                         struct file_ra_state *ra,
7                         struct file *filp,
8                         loff_t *ppos,
9                         read_descriptor_t *desc,
10                        read_actor_t actor)
11 {
12     struct inode *inode = mapping->host;
13     pgoff_t index;
14     pgoff_t last_index;
15     pgoff_t prev_index;
16     /* offset into pagecache page */
17     unsigned long offset;
18     unsigned int prev_offset;
19     int error;
20
21     /*
22      * 把文件位置指针转换为对应的页面号，这就是在 radix tree 中
23      * 用到的页面号，见第12.7.1节。
24      */
25     index = *ppos >> PAGE_CACHE_SHIFT;
26     prev_index = ra->prev_pos >> PAGE_CACHE_SHIFT;
27     prev_offset = ra->prev_pos & (PAGE_CACHE_SIZE-1);
28     last_index = (*ppos + desc->count + PAGE_CACHE_SIZE-1) >>
29                  PAGE_CACHE_SHIFT;
30     offset = *ppos & ~PAGE_CACHE_MASK;
31     for (;;) {
32         struct page *page;
33         pgoff_t end_index;
34         loff_t isize;
35         unsigned long nr, ret;
36
37         cond_resched();
38     find_page:
39         page = find_get_page(mapping, index);
40         if (!page) {
41             page_cache_sync_readahead(mapping, ra, filp, index,
42                                     last_index - index);
43             page = find_get_page(mapping, index);
44             if (unlikely(page == NULL))
45                 goto no_cached_page;
46         }
47         if (PageReadahead(page)) {
```

```

48     page_cache_async_readahead(mapping, ra, filp, page, index,
49                               last_index - index);
50 }
51 if (!PageUptodate(page))
52     goto page_not_up_to_date;
53 page_ok:

```

首先调用 `find_get_page()` 函数(见第451页代码片段12.74)，根据页面号 `index`，从文件的 radix tree 中搜索对应的页面，如果搜索失败，就调用 `page_cache_sync_readahead()` 函数请求预读管理模块进行读取，这个函数根据预读算法会读取更多的页面，之后再次调用 `find_get_page()` 函数在 radix tree 中搜索，如果还是失败就跳转到 `no_cached_page`(见第468页代码片段12.91)处，我们在后面将继续讨论 `no_cached_page`。在第47行，如果搜索到指定的页面是在预读窗口中，那么需要使预读窗口向前推进。

之后如果 `PageUptodate()` 判断出这个页面内容不是最新的，就跳转到 `page_not_up_to_date` 处。例如其他进程正在对这个页面进行写入操作，或者设备正在把磁盘内容传送到这个页面中，在这种情况下，就跳转到 `page_not_up_to_date` 处。

现在我们先看看 `page_ok` 是怎么处理的。

代码片段 12.86 (节自 mm/filemap.c)

```

1 page_ok:
2     /*
3      * i_size must be checked after we know the page is Uptodate.
4      *
5      * Checking i_size after the check allows us to calculate
6      * the correct value for "nr", which means the zero-filled
7      * part of the page is not copied back to userspace (unless
8      * another truncate extends the file-this is desired though).
9      */
10
11    isize = i_size_read(inode);
12    end_index = (isize - 1) >> PAGE_CACHE_SHIFT;
13    if (unlikely(!isize || index > end_index)) {
14        page_cache_release(page);
15        goto out;
16    }
17    /* nr is the maximum number of bytes to copy from this page */
18    nr = PAGE_CACHE_SIZE;
19    if (index == end_index) {
20        nr = ((isize - 1) & ~PAGE_CACHE_MASK) + 1;
21        if (nr <= offset) {
22            page_cache_release(page);

```

```
23         goto out;
24     }
25 }
26 nr = nr - offset;
27
28 /* If users can be writing to this page using arbitrary
29 * virtual addresses, take care about potential aliasing
30 * before reading the page on the kernel side.
31 */
32 if (mapping_writably_mapped(mapping))
33     flush_dcache_page(page);
34
35 /*
36 * When a sequential read accesses a page several times,
37 * only mark it as accessed the first time.
38 */
39 if (prev_index != index || offset != prev_offset)
40     mark_page_accessed(page);
41 prev_index = index;
42
43 /*
44 * Ok, we have the page, and it's up-to-date, so
45 * now we can copy it to user space...
46 *
47 * The actor routine returns how many bytes were actually used..
48 * NOTE! This may not be the same as how much of a user buffer
49 * we filled up (we may be padding etc), so we can only update
50 * "pos" here (the actor routine has to update the user buffer
51 * pointers and the remaining count).
52 */
53 ret = actor(desc, page, offset, nr);
54 offset += ret;
55 index += offset >> PAGE_CACHE_SHIFT;
56 offset &= ~PAGE_CACHE_MASK;
57 prev_offset = offset;
58
59 page_cache_release(page);
60 if (ret == nr && desc->count)
61     continue;
62 goto out;
```

这段代码的主要作用就是检查读取的文件是否超过了实际大小，例如在页大小为4KB的情况下，一个5KB的文件占用两个页面，第二个页面中，只有前1KB是有效数据。然后通过函数指针actor调用file_read_actor()，把文件内容从内核态复制到用户态。

现在我们来看看page_not_up_to_date的处理过程：

代码片段12.87 (节自mm/filemap.c)

```

1 page_not_up_to_date:
2     /* Get exclusive access to the page ... */
3     lock_page(page);
4     /* Did it get truncated before we got the lock? */
5     if (!page->mapping) {
6         unlock_page(page);
7         page_cache_release(page);
8         continue;
9     }
10    /* Did somebody else fill it already? */
11    if (PageUptodate(page)) {
12        unlock_page(page);
13        goto page_ok;
14    }

```

首先调用lock_page()获得对这个页面的使用权，如果其他进程正在使用该页面，那么当前进程可能会在lock_page()中被阻塞。等当前进程获取到对该页面的使用权时，因为其他进程可能通过truncate()这一类的系统调用把目标文件的实际长度缩短了，因此需要进一步判断当前页面是否有效。之后如果PageUptodate()检查到当前页面中的内容已经是最新的了，就跳转到page_ok处(见第464页代码片段12.86)。如果此时这个页面中的内容还不是最新的，就需要启动一个读请求了。

代码片段12.88 (节自mm/filemap.c)

```

1 readpage:
2     /* Start the actual read. The read will unlock the page. */
3     error = mapping->a_ops->readpage(filp, page);
4
5     if (unlikely(error)) {
6         if (error == AOP_TRUNCATED_PAGE) {
7             page_cache_release(page);
8             goto find_page;
9         }
10        goto readpage_error;
11    }
12    if (!PageUptodate(page)) {
13        lock_page(page);

```

```

14     if (!PageUptodate(page)) {
15         if (page->mapping == NULL) {
16             /*
17              * invalidate_inode_pages got it
18              */
19             unlock_page(page);
20             page_cache_release(page);
21             goto find_page;
22         }
23         unlock_page(page);
24         error = -EIO;
25         shrink_readahead_size_eio(filp, ra);
26         goto readpage_error;
27     }
28     unlock_page(page);
29 }
30 goto page_ok;

```

如果执行到 `readpage` 标号处，就需要调用 `a_ops->readpage` 进行实际的读操作，这个函数计算出页面在文件中的块号，然后向块设备驱动发出读取请求。其中 ext2 的 `a_ops` 结构定义如下：

代码片段 12.89 (节自 fs/ext2/inode.c)

```

1 const struct address_space_operations ext2_aops = {
2     .readpage    = ext2_readpage,
3     .readpages   = ext2_readpages,
4     .writepage   = ext2_writepage,
5     .sync_page   = block_sync_page,
6     .write_begin  = ext2_write_begin,
7     .write_end   = generic_write_end,
8     .bmap        = ext2_bmap,
9     .direct_IO   = ext2_direct_IO,
10    .writepages  = ext2_writepages,
11    .migratepage = buffer_migrate_page,
12 };

```

因此对 ext2 文件系统来说，这里调用的函数是 `ext2_readpage()`，进程可能会因此而被阻塞，当读取完成后，会跳转到 `page_ok` 处(见第464页代码片段12.86)。如果在读操作失败了，就跳转到 `readpage_error` 处。

代码片段 12.90 (节自 mm/filemap.c)

```

1 readpage_error:

```

```

2      /* UHHUH! A synchronous read error occurred. Report it */
3      desc->error = error;
4      page_cache_release(page);
5      goto out;

```

通过前面的讨论讨论，我们知道，如果在 radix tree 中没有搜索到指定的页面，就会调用 `page_cache_sync_readahead()`，之后如果还是搜索不到指定页面，就说明在缓存中没有这个页面，并且预读窗口中也没有这个页面，因此就跳转到 `no_cached_page` 处(见第462页代码片段12.85)。

代码片段 12.91 (节自 mm/filemap.c)

```

1 no_cached_page:
2     /*
3      * Ok, it wasn't cached, so we need to create a new
4      * page..
5     */
6     page = page_cache_alloc_cold(mapping);
7     if (!page) {
8         desc->error = -ENOMEM;
9         goto out;
10    }
11    error = add_to_page_cache_lru(page, mapping,
12        index, GFP_KERNEL);
13    if (error) {
14        page_cache_release(page);
15        if (error == -EEXIST)
16            goto find_page;
17        desc->error = error;
18        goto out;
19    }
20    goto readpage;
21 }

```

在这里，首先分配一个页面，如果分配失败，就跳转到 `out` 处退出。否则调用 `add_to_page_cache_lru()`把这个页面添加到 radix tree 中，如果这个函数返回-EEXIST，说明对应的页面已经在 radix tree 中了，于是就释放新分配到的页面，跳转到 `find_page` 处(见第462页代码片段12.85)，这是由于其他进程或者预读模块已经提前把数据读取出来，并且添加到 radix tree 中。如果成功把新分配到的页面添加到 radix tree 中，就必须跳转到 `readpage`(见第466页代码片段12.88)处，向块设备发送读操作请求。

最后 `do_generic_mapping_read()`函数从 `out` 处返回，调整 `ra` 和文件位置指针 `ppos` 和 `ra`，其中 `ra` 是一个 `file_ra_state` 结构，用于跟踪记录当前实际读取位置，以及预读窗口位置

等相关信息。

代码片段 12.92 (节自 mm/filemap.c)

```
1 out:
2     ra->prev_pos = prev_index;
3     ra->prev_pos <= PAGE_CACHE_SHIFT;
4     ra->prev_pos |= prev_offset;
5
6     *ppos = ((loff_t)index << PAGE_CACHE_SHIFT) + offset;
7     if (filp)
8         file_accessed(filp);
9 }
```

前面我们看到，在缓存命中失败的情况下，会调用 `ext2_readpage()` 把读请求发送到块设备驱动层，这个函数会根据 ext2 文件系统的相关信息(`i_block`, 数据块位图等)计算出要读取的内容位于磁盘的扇区号，然后把请求传递到块设备驱动层。我们不打算对块设备驱动进行讨论，因此对于文件读取的分析到此为止，在这里我们最后总结一下文件读请求要经过的相关模块：

(1) 虚拟文件系统层

即 VFS 层，它的作用是屏蔽各种下层具体文件系统的差异，向上层应用程序提供一个统一的操作接口，通过 `sys_read()`/`sys_write()` 等系统调用把读写请求传递到虚拟文件系统层。

(2) 具体文件系统层

提供各种具体文件系统相关操作的具体实现，并通过 `register_filesystem()` 向虚拟文件系统层注册自己的接口，各种具体文件系统的差异在具体文件系统层得到体现。

(3) 缓冲区管理层

相对来说，块设备属于慢速设备，为了提高系统性能，因此提供了缓冲区管理。同时还提供了预读管理机制。块设备缓存和预读一直是内核中的一个重要模块，其算法也随着内核的发展而不断更新。

(4) 块设备驱动抽象层

它的作用是屏蔽各种下层块设备的差异，向上层提供一个统一的操作接口。

(5) 块设备 IO 调度层

由于块设备和 CPU 的操作是异步进行的，当 CPU 向块设备发出操作请求后，就继续执行其他任务，之后在操作块设备完成操作之前，CPU 可能再次发送操作请求。而 CPU 和块设备在处理速度上的巨大差异会导致块设备请求队列中积压大量的操作请求。另一方面，许多块设备都是机械式结构，顺序读写操作的速度将比反复跳跃式的

读写操作块，因此引入块设备 IO 调度层，它的工作是对块设备中请求队列中的相关请求进行重排序，尽可能地将乱序操作转化为按序操作。

(6) 具体块设备驱动层

提供各种具体块设备驱动的具体实现，通过控制具体设备中的控制寄存器，命令寄存器等实现对设备的控制。例如 CPU 向磁盘的控制寄存器发送柱面号、磁道号、扇区号，向命令寄存器发送读操作命令，读取长度，向 DMA 相关寄存器发送相关的内存地址，之后磁盘就开始向内存中传送相关信息，CPU 就可以调度其他进程了，等到磁盘操作结束时，会发出中断请求，然后 CPU 进行中断响应会调用具体驱动程序的中断服务例程。

需要注意的是，从严格意义上来说，文件的读写操作请求，并不是按照上面的过程从上到下依次传递的。例如在读操作过程中，请求到达缓冲区管理层时，如果缓存命中失败，需要向块设备驱动抽象层传递这一操作请求，但是它首先必须计算出文件在磁盘上的具体扇区等信息，而这一计算又是具体文件系统层的相关操作，因为各种文件系统的 inode 等信息是有差异的。

在看懂读操作过程之后，写操作过程就很容易理解了，它们的大体操作过程是相似的，唯一的区别就在于，读操作时有预读过程，而写操作可能会面临文件的现有块空间不足的问题，因此需要分配新的逻辑块，这个操作是在具体文件系统层提供的。对于没有设置 O_SYNC 标志的文件来说，写入操作仅仅是分配必要的页面，把页面添加到 radix tree 中，同时把内容复制到相应的页面，并且为页面设置 PG_dirty 标志。而实际上把内容传输到块设备的存储介质上的工作则由内核线程 pdflush 完成，这个线程将被定期唤醒或者空闲页面低于一定阈值时被唤醒。

第13章 常用内核分析方法

Linux 内核支持各种硬件平台和各种配置选项，因此在阅读代码时常常会遇到多个同名的宏、结构体定义，以及同名函数，对于内核新手来说，常常需要花费大量的时间和精力来定位这些同名的宏或函数。此外，Linux 内核处处可见面向对象的设计思想，很多代码都是通过函数指针的方式来调用某个函数的，当对于一个函数 func() 来说，是很难通过字符串搜索来找到它的调用者的。本章将讨论如何解决这些问题。

13.1 准确定位同名宏及结构体

当遇到下面这样的代码时：

```
1 #ifdef XXX
2     .....
3 #else XXX
4     .....
5#endif
```

如果 XXX 是配置选项，那么可以直接打开.config 文件搜索，就可以看出究竟了。但是有时候遇到多个嵌套的 #ifdef #else 时，手工搜索就不是那么直观了。在编译的过程中编译器一定知道究竟定义了哪个宏，因此把它修改成下面的样子：

```
1 #ifdef XXX
2 #error XXX is defined
3     .....
4 #else XXX
5 #error XXX is defined
6     .....
7#endif
```

当编译上面的代码时，编译器肯定会报错，根据它的错误信息，就可以精确地定位哪个宏被定义了，哪个宏没有被定义。

上面的方法可以确定某个宏是否被定义，但是某些时候，还需要进一步确定一个宏到底是定义在哪个文件的哪一行，我们同样可以问编译器。例如在 kernel/sched.c 中调用了

barrier(), 但是发现有很多同名的 barrier() 宏定义，于是我们到 kernel/sched.c 的最前面加入下面的代码：

```

1 #define barrier() foo()
2
3 #include <linux/mm.h>
4 #include <linux/module.h>
5 #include <linux/nmi.h>
6 .....

```

之后再进行编译，于是编译器会准确地报告这个宏定义在哪个文件中，信息如下：

```

1 In file included from include/linux/compiler-gcc4.h:6,
2     from include/linux/compiler.h:40,
3     from include/linux/stddef.h:4,
4     from include/linux posix_types.h:4,
5     from include/linux/types.h:11,
6     from include/linux/thread_info.h:10,
7     from include/linux/preempt.h:9,
8     from include/linux/spinlock.h:49,
9     from include/linux/mmzone.h:7,
10    from include/linux/gfp.h:4,
11    from include/linux/mm.h:8,
12    from kernel/sched.c:28:
13 include/linux/compiler-gcc.h:12:1: warning: "barrier" redefined
14 kernel/sched.c:26:1: warning: this is the location of the
15     previous definition
16
17 .....

```

从 gcc 报告的警告信息中，我们可以准确地定位到 barrier() 这个宏是在 include/linux/compiler-gcc.h 文件的第 12 行定义的。这个技巧非常实用，或许这个例子不具说服力，因为可以猜到用 gcc 编译时，肯定是使用 compiler-gcc.h 中的宏。事实上，在代码阅读过程中，还常常遇到大量的不是这么直观的宏定义，使用这个方法是非常方便的。

对于结构体定义，如法炮制，同样可以让编译器准确地报告出结构体，等各种变量是在哪里定义的，我们来看一个不容易猜出来的结构体定义。例如：page 结构是一个常用的结构体，但是内核中以 page 命名的结构，变量实在太多了，于是在相关文件的最前面加入下面的内容：

```

1 /* 重新定义一个 page 结构。*/
2 struct page { int a;};
3
4 #include <linux/stddef.h>

```

```
5 #include <linux/mm.h>
6 .....
```

然后再编译一次，得到下面的编译错误：

```
1 In file included from include/linux/mm.h:14,
2   from mm/page_alloc.c:19:
3   include/linux/mm_types.h:36: error: redefinition of 'struct' page
```

通过这个错误信息，我们可以准确地定位到这里使用的 page 结构位于 include/linux/mm_types.h 的第 36 行。

13.2 准确定位同名函数

同样，对于同名的函数，虽然有时可以“猜”出来，但是有时“猜测”的结果却不一定准确，我们可以利用 addr2line 这个工具来准确定位同名的函数。我们以 init_IRQ()这个函数为例，内核中有很多个名为 init_IRQ()的函数，现在我们来介绍如何利用这个方法来定位它。首先编译一个带调试信息的内核(确保选中了 CONFIG_DEBUG_INFO 这个编译选项)。然后输入下面的命令：

```
1 # vmlinux 是编译号的内核文件。
2 % nm vmlinux | grep init_IRQ
3
4 # 得到的结果如下：
5 c0493050 W init_IRQ
6 c0493050 T native_init_IRQ
7
8 # 现在我们 init_IRQ 的地址为 0xc0493050。输入下面的命令：
9 % addr2line -e vmlinux 0xc0493050
10
11 # 得到的结果如下：
12 arch/x86/kernel/i8259_32.c:391
```

于是我们知道，当前内核中的 init_IRQ()这个函数位于 arch/x86/kernel/i8259_32.c 文件中的 391 行。其定义如下：

代码片段 13.1 (节自 arch/x86/kernel/i8259_32.c)

```
1 /* Overridden in paravirt.c */
2 void init_IRQ(void) __attribute__((weak, alias("native_init_IRQ")));
3
4 void __init native_init_IRQ(void)
5 {
```

```

6     .....
7 }

```

从上面可以看出，`init_IRQ()`还有一个别名为`native_init_IRQ()`。

13.3 利用 link map 文件定位全局变量

有时候很难确定一个全局变量到底是在哪一个文件中定义的，在内核分析过程中，经常遇到这种情况，典型的例子就是`interrupt`数组，这个数组一共有256个成员，保存着中断向量(0~255)对应的中断入口函数的地址。`init_IRQ()`函数会把数组`interrupt`保存的地址设置到对应的中断描述符表中，代码如下：

代码片段 13.2 (节自 `arch/x86/kernel/i8259_32.c`)

```

1 /* Overridden in paravirt.c */
2 void init_IRQ(void) __attribute__((weak, alias("native_init_IRQ")));
3
4 void __init native_init_IRQ(void)
5 {
6     .....
7     for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
8         int vector = FIRST_EXTERNAL_VECTOR + i;
9         if (i >= NR_IRQS)
10             break;
11         /* SYSCALL_VECTOR was reserved in trap_init. */
12         if (!test_bit(vector, used_vectors))
13             set_intr_gate(vector, interrupt[i]);
14     }
15     .....
16 }

```

这里调用`set_intr_gate()`把数组`interrupt`的地址设置到中断描述符表中，但是却很难找到`interrupt`到底是在哪里定义的，在哪里初始化的呢？但是有一点可以肯定，`interrupt`是一个全局变量，编译后位于内核执行文件的数据区。于是我们可以利用`link map`文件来定位它。

首先打开`vmlinux.o.cmd`文件，这里面保存着链接`vmlinux`的命令，现在把它复制出来，加上`-M`或者`--print-map`参数，命令如下：

```
% ld -M -m elf_i386 -m elf_i386 -r -o vmlinux.o \
    arch/x86/kernel/head_32.o arch/x86/kernel/init_task.o \
    init/built-in.o --start-group usr/built-in.o \

```

```

arch/x86/kernel/built-in.o  arch/x86/mm/built-in.o \
arch/x86/mach-default/built-in.o \
arch/x86/crypto/built-in.o  kernel/built-in.o \
mm/built-in.o   fs/built-in.o  ipc/built-in.o \
security/built-in.o  crypto/built-in.o \
block/built-in.o  lib/lib.a  arch/x86/lib/lib.a \
lib/built-in.o  arch/x86/lib/built-in.o \
drivers/built-in.o  sound/built-in.o \
arch/x86/pci/built-in.o  net/built-in.o \
--end-group .tmp_kallsyms2.o > /tmp/vmlinux.map

```

现在打开/tmp/vmlinux.map，在这个文件中搜索 interrupt，得到的结果如下：

```

.data 0x0000000000004000    0x3474 arch/x86/kernel/built-in.o
      0x0000000000004184    data_resource
      0x0000000000004180    pci_mem_start
      0x00000000000044a0    pit_clockevent
      .....
      0x0000000000004010    interrupt

```

从上面的信息我们可以知道数据段中的 interrupt 来自于 arch/x86/kernel/built-in.o 这个文件，现在利用同样的方法对 arch/x86/kernel/.built-in.o.cmd 进行处理，得到它的 link map 文件，再次搜索 interrupt，得到结果如下：

```

.data 0x0000000000000010    0x40 arch/x86/kernel/entry_32.o
      0x0000000000000010    interrupt

```

现在我们知道 interrupt 来自 arch/x86/kernel/entry_32.o，打开目录下的 entry_32.o.cmd，我们发现 entry_32.o 是由 entry_32.S 这个文件编译而来的，于是从 entry_32.S 文件中找到 interrupt，定义如下：

代码片段 13.3 (节自 arch/x86/kernel/entry_32.S)

```

1 /*
2  * Build the entry stubs and pointer table with
3  * some assembler magic.
4  */
5
6 # interrupt 放置在数据段。
7 .data
8 ENTRY(interrupt)
9

```

```

10 # 代码段。
11 .text
12
13 ENTRY(irq_entries_start)
14     RING0_INT_FRAME
15 vector=0
16 .rept NR_IRQS
17     ALIGN
18     .if vector
19         CFI_ADJUST_CFA_OFFSET -4
20     .endif
21
22 # 在代码段放置中断入口代码。
23 1: pushl $~(vector)
24     CFI_ADJUST_CFA_OFFSET 4
25     jmp common_interrupt
26
27 # 切换到最近的数据段，在 interrupt 中依次放置标号为 1 的地址。
28 .previous
29     .long 1b
30 .text
31 vector=vector+1
32 .endr
33 END(irq_entries_start)
34
35 .previous
36 END(interrupt)

```

这里 NR_IRQS 为 224(256-32)，首先在数据段中定义了 interrupt 标号，然后在代码段依次放置中断入口代码，再把代码的首地址依次保存到数据段的 interrupt 中(参见第6章中断和异常处理)。

13.4 准确定位函数调用线索

前面介绍的方法在自上而下的代码分析过程非常有用，但是有时我们需要采取自下而上的分析方法。这是常见的分析过程，例如在文件系统相关代码分析过程中，有下面这样的代码：

代码片段 13.4 (节自 fs/ext2/super.c)

```

1 static struct file_system_type ext2_fs_type = {
2     .owner      = THIS_MODULE,

```

```

3   .name    = "ext2",
4   .get_sb   = ext2_get_sb,
5   .kill_sb  = kill_block_super,
6   .fs_flags = FS_REQUIRES_DEV,
7 };
8
9 static int __init init_ext2_fs(void)
10 {
11     .....
12     err = register_filesystem(&ext2_fs_type);
13     .....
14 }

```

在这段代码中，我们看到 `register_filesystem()` 注册了一个 ext2 文件系统对象 `ext2_fs_type`，函数指针 `get_sb` 指向 `ext2_get_sb()`。我们希望进一步分析，`ext2_get_sb()` 这个函数是如何被调用的，遗憾的是这个函数是通过函数指针 `get_sb()` 进行调用的，因此我们不能直接通过 grep 等文本搜索的方式来找到 `ext2_get_sb()` 是在哪里调用的。但是我们可以在 `ext2_get_sb()` 中把它的返回地址打印出来，然后使用 `addr2line` 来获取调用函数的信息。

代码片段 13.5 (节自 `fs/ext2/super.c`)

```

1 #include <linux/kallsyms.h>
2
3 static int ext2_get_sb(struct file_system_type *fs_type,
4                         int flags, const char *dev_name,
5                         void *data, struct vfsmount *mnt)
6 {
7     printk("call %s from 0x%lx\n", __FUNCTION__,
8           (unsigned int) __builtin_return_address(0));
9
10    printk("call %s from ", __FUNCTION__);
11    print_fn_descriptor_symbol("%s()\n",
12                               (unsigned long) __builtin_return_address(0));
13
14    dump_stack();
15    return get_sb_bdev(fs_type, flags, dev_name, data, ext2_fill_super, mnt);
16 }

```

- (1) 在第7行中，我们利用 gcc 的内部函数，`__builtin_return_address()` 来获取 `ext2_get_sb()` 的返回地址。在 x86 平台上，函数的返回地址保存在堆栈中，返回地址距离栈顶的偏移受函数的参数的影响，gcc 在编译的时候可以计算出这个偏移量，从而为 `__builtin_return_address()` 生成对应的代码。这样当调用 `ext2_get_sb()` 时，可以得到下

面的输出。

```
call ext2_get_sb from 0xc0173bf7
```

现在我们可以利用 `addr2line` 来根据返回地址 `0xc0173bf7` 确定调用 `ext2_get_sb()` 的函数。

```
% addr2line -e vmlinux 0xc0173bf7
```

得到的结果如下：

```
fs/super.c:882
```

- (2) 利用 `__builtin_address()` 获取返回地址后，还需要 `addr2line` 命令才能得到结果。实际上内核中的 `print_fn_descriptor_symbol()` 可以完成与 `addr2line` 类似的功能。这个函数的原型定义在 `linux/kallsyms.h` 文件中，因此需要包含这个头文件。在第10行中，我们把 `__builtin_address()` 的结果传递给 `print_fn_descriptor_symbol()`，于是可以直接得到下面的结果：

```
call ext2_get_sb from vfs_kern_mount+0x37/0x90()
```

- (3) 上面两种方法只能获取到上一级的调用者的信息，而且需要重新编译并且运行修改后的内核，假设在这个例子中，我们进一步使用这个方法确定 `vfs_kern_mount()` 的调用者，那么需要再次重新编译内核。在这种情况下，我们可以利用 `dump_stack()` 这个函数一次就得到结果，在第11行调用 `dump_stack()` 得到的结果如下：

```
Pid: 1134, comm: mount Not tainted 2.6.24.4 #20
[<c01cf628>] ext2_get_sb+0x58/0x90
[<c0173bf7>] vfs_kern_mount+0x37/0x90
[<c0173bf7>] vfs_kern_mount+0x37/0x90
[<c0173cad>] do_kern_mount+0x3d/0xe0
[<c0187d9c>] do_mount+0x4dc/0x670
[<c01550a6>] get_page_from_freelist+0x1b6/0x3e0
[<c017bdb9>] __user_walk_fd+0x49/0x60
[<c0155360>] __alloc_pages+0x60/0x390
[<c017b86d>] do_unlinkat+0x4d/0x150
[<c0186840>] copy_mount_options+0x40/0x140
[<c0187fa7>] sys_mount+0x77/0xc0
[<c0104352>] syscall_call+0x7/0xb
```

从上面的信息中，我们可以大概地知道整个函数调用链的情况，需要注意的是 `dump_stack()` 的原理是依次比较堆栈中的值，是否位于内核代码段，如果是，就打印出来。因此它的输出结果不完全是函数调用链，例如上面 `vfs_kern_mount()` 出现了两次，这是由于堆栈中恰巧有一个值和 `vfs_kern_mount()` 的地址一致。另外在代码编译过程中，`gcc` 可能会把某些函数变为内联函数，因此在代码中看到的函数调用关系，从返回地址来分析，却找不到这个关系。但是我们只要明白这两点，就能够很好的利用这个方法来进行代码分析了。

13.5 SystemTap 在代码分析中的使用

在内核分析中，我们经常需要在代码中加入打印信息，然后重新编译内核，再启动这个内核，最后根据打印的结果对内核进行分析。有时候，我们发现大量的时间被浪费在编译、重启内核这个过程中。SystemTap 可以帮助我们避免这个问题。关于 SystemTap 的安装配置请读者参考官方网站的相关介绍，这里我们先通过一个例子来展示它的强大功能。

代码片段 13.6 (SystemTap 脚本)

```
1 #!/usr/bin/stap -v
2
3 probe kernel.function("do_timer").return {
4     printf("call do_timer from 0x%x\n", caller_addr());
5 }
```

以超级用户运行这个脚本，我们得到输出如下：

```
# 对 SystemTap 脚本进行语法检查。
Pass 1: parsed user script and 38 library script(s)
in 270usr/20sys/296real ms.
```

```
# 分析统计 SystemTap 脚本需要完成哪些工作。
Pass 2: analyzed script: 1 probe(s), 1 function(s),
0 embed(s), 0 global(s) in 250usr/90sys/34lreal ms.
```

```
# 把 SystemTap 脚本转换为等价的 C 代码。
Pass 3: translated to C into "/tmp/stapXOivEX/
stap_6a92f989451816923450027da3a803f4_483.c"
in 0usr/0sys/0real ms.
```

```
# 把得到的 C 代码编译为内核模块。
```

```
Pass 4: compiled C into
"stap_6a92f989451816923450027da3a803f4_483.ko"
in 3020usr/240sys/3205real ms.
```

加载运行这个内核模块。

```
Pass 5: starting run.
```

```
call do_timer from 0xfffffffffc014a595
```

上面这个脚本运行之后，就能够打印出 `do_timer()` 函数的返回地址，你可以把 `do_timer` 修改为其他的函数，然后再次运行这个脚本就可以得到对应的结果，在这个过程中，不需要重新编译内核，也不需要重启内核。脚本中的 `probe` 被称为一个监测点。SystemTap 的原理类似于 Runtime Hook 技术。在本例中，它首先找到 `do_timer()` 函数的入口地址，然后保存 `do_timer()` 入口的代码，再修改这部分代码，当内核调用 `do_timer()` 函数时，就会跳转到 SystemTap 的代码处执行监测点指定的代码，从而打印出我们指定的结果，最后再恢复 `do_timer()` 的原始代码，然后执行 `do_timer()` 函数。

现在我们只需要修改 SystemTap 脚本，就可以获取到前面提到的 `dump_stack()` 函数完成的功能。

代码片段 13.7 (SystemTap 脚本)

```
1 #!/usr/bin/stap -v
2
3 probe kernel.function("do_timer").return {
4     print_backtrace();
5     printf("\n");
6 }
```

运行这个脚本，可以得到下面的结果：

```
Returning from: 0xc0135e40 : do_timer+0x0/0xe0 []
Returning to : 0xc014a595 : tick_do_update_jiffies64+0xf5/0x110 []
0xc0144598 : ktime_get+0x18/0x40 []
0xc014a684 : tick_sched_timer+0xd4/0xf0 []
0xc014493f : hrtimer_interrupt+0x15f/0x1f0 []
0xc014a5b0 : tick_sched_timer+0x0/0xf0 []
0xc0118840 : smp_apic_timer_interrupt+0x50/0x80 []
0xc01054d8 : apic_timer_interrupt+0x28/0x30 []
0xc012007b : __is_prefetch+0x9b/0x240 []
0xf88775ff : acpi_idle_enter_simple+0x14c/0x1b9 [processor]
```

```
0xc029530c : cpuidle_idle_call+0x7c/0xb0 []
0xc0102695 : cpu_idle+0x45/0xd0 []
```

现在，我们可以在不重新编译内核，不重启系统的情况下，使用更加简单的脚本语言来修改内核，这不得不说是内核代码分析人员的福音。笔者在使用 SystemTap 之后，已经基本不再使用调试器了。本书不是一本 SystemTap 脚本的专著，关于这方面的资料，请参考《SystemTap Language Reference》。为了进一步介绍 SystemTap 在代码分析中的强大功能，我们再来看几个例子。

代码片段 13.8 (SystemTap 脚本)

```
1 #!/usr/bin/stap -v
2
3 probe kernel.function("__switch_to") {
4     if (task_execname($next_p) == "vim") {
5         printf("switch from [%s] \t to \t [%s]\n",
6                task_execname($prev_p),
7                task_execname($next_p));
8     }
9 }
10
11 probe end {
12     printf("\nDONE\n")
13 }
```

`__switch_to()`是内核中的一个函数，这个函数有两个参数，`prev_p`指向前一个进程的`task_struct`结构，而`next_p`指向下一个要切换的目标进程的`task_struct`结构。`task_execname()`是 SystemTap 的内置函数，它接收一个`task_struct`结构指针，返回进程的名字。当调度到 vim 这个进程时，这个脚本输出如下：

```
switch from [Xorg]    to    [ vim]
switch from [Xorg]    to    [ vim]
switch from [Xorg]    to    [ vim]
switch from [swapper] to    [ vim]
....
```

SystemTap 的强大功能不局限于内核分析，在分析 XServer 时，利用下面的这个脚本可以确定 Xorg 这个进程需要打开哪些文件，它可以帮助我们迅速找到 Xorg 输入/输出来源于/dev/中的哪个设备文件。

代码片段 13.9 (SystemTap 脚本)

```
1 #!/usr/bin/stap -v
2
```

```
3 probe kernel.function("sys_open") {
4     if (task_execname(task_current()) == "Xorg") {
5         printf("open %s\n", user_string($filename));
6     }
7 }
```

在对 XServer 进行代码的分析过程中，笔者首先启动这个脚本，然后再启动 Xorg，几乎不费吹灰之力，就确定了 XServer 的键盘输入是来自于从/dev/tty7 这个设备文件。

同样，如果要监测某个进程是否需要打开某个文件，可以使用下面的这个脚本：

代码片段 13.10 (SystemTap 脚本)

```
1 #!/usr/bin/stap -v
2 probe kernel.function("sys_open") {
3     if (task_execname(task_current()) == "Xorg") {
4         if (user_string($filename) == "libm.so") {
5             printf("%s open %s\n", task_execname(task_current()), user_string(
6                 $filename));
7         }
8     }
9 }
```

好了，SystemTap 的抛砖引玉就到这里，相信你已经见识到了 SystemTap 的强大功能了，笔者建议任何一个代码分析人员在代码分析过程中多参考《SystemTap Language Reference》，写出实用的 SystemTap 脚本来，这样可以做到事半功倍。

《独辟蹊径品内核：Linux 内核源代码导读》

读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传 真：（010）88254397

E-mail：dbqq@phei.com.cn

通信地址：北京市万寿路173信箱

电子工业出版社总编办公室

邮 编：100036