提供各种IT类书籍pdf下载，如有需要，请 QQ: 2011705918
注：链接至淘宝，不喜者勿入! 整理那么多资料也不容易，请多多见谅！非诚勿扰！

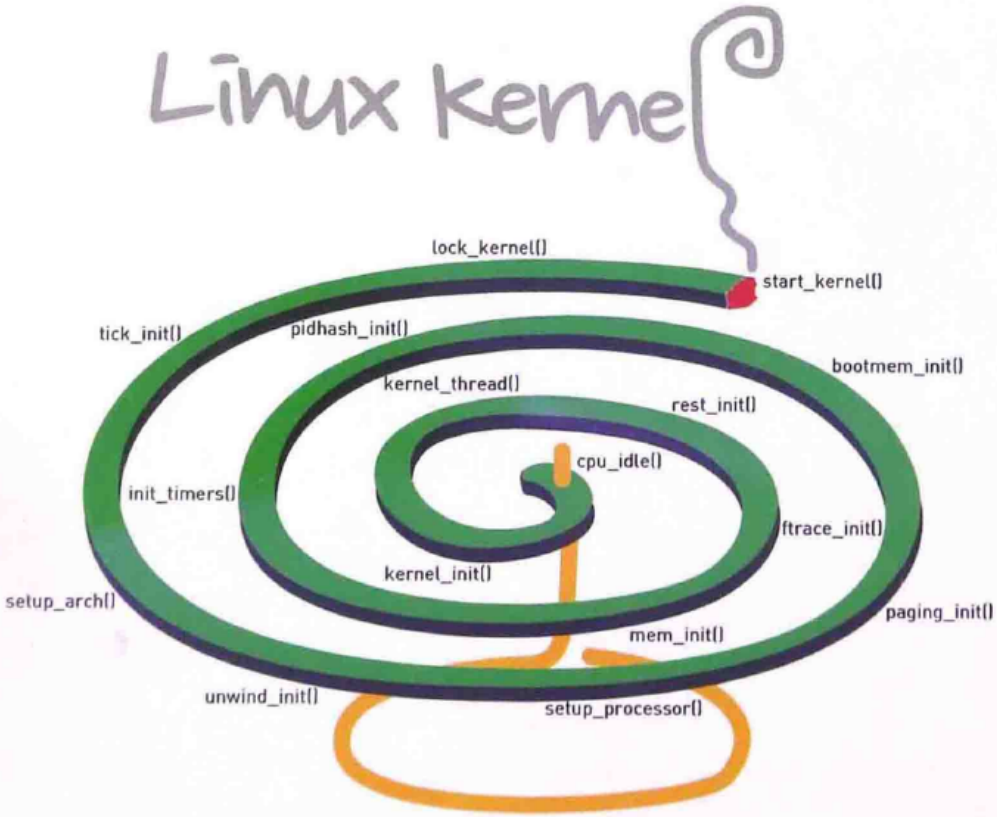
更多资源请点击

TURING 图灵程序设计丛书

2013年度韩国学术院选定 优秀学术图书

ARM Linux 内核源码剖析

【韩】尹锡训等 著　崔范松 译



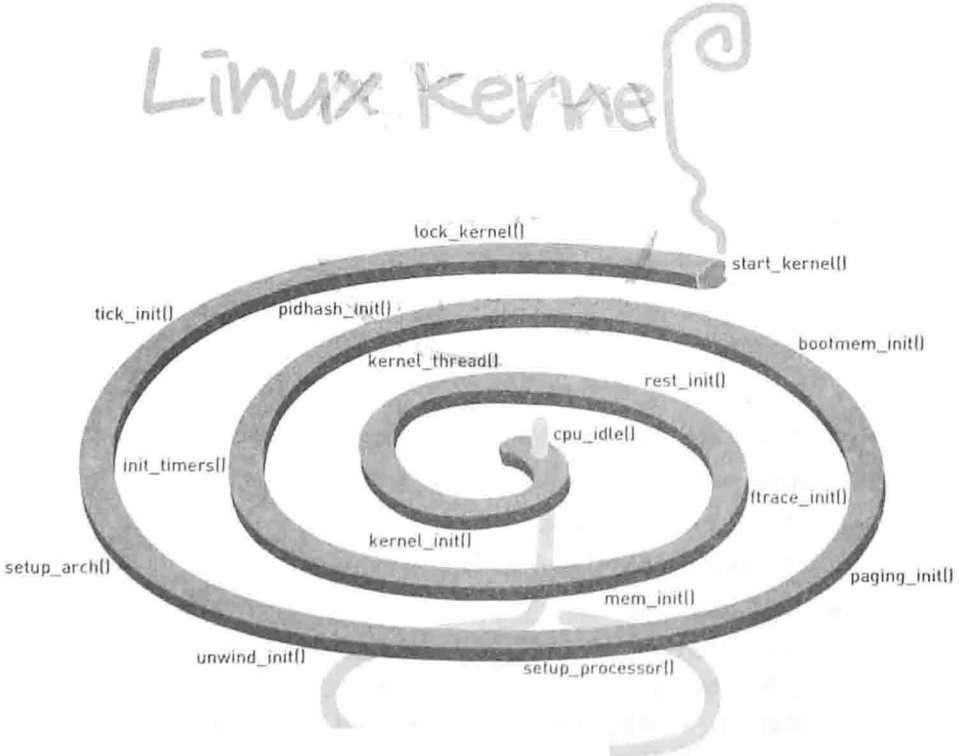
2年代码分析+1年执笔=第一本ARM Linux内核分析集大成之作!

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

ARM Linux 内核源码剖析

【韩】尹锡训等 著　崔范松 译



人民邮电出版社
北京

图书在版编目（CIP）数据

ARM Linux内核源码剖析 / （韩）尹锡训等著 ；崔范松译. -- 北京 ：人民邮电出版社，2014.7
（图灵程序设计丛书）
ISBN 978-7-115-35910-0

Ⅰ. ①A… Ⅱ. ①尹… ②崔… Ⅲ. ①微处理器－系统设计②Linux操作系统－系统设计 Ⅳ. ①TP332 ②TP316.89

中国版本图书馆CIP数据核字(2014)第117494号

内 容 提 要

本书是多位作者在3年Liunx内核分析经验和庞大资料基础上写成的，收录了其他同类书未曾讲解的内容并进行逐行分析，一扫当前市场中其他理论书带给读者的郁闷。书中详细的代码分析与大量插图能够使读者对Linux内核及ARM获得正确认识，自然而然习得如何有效分析定期发布的Linux内核。

本书适合想从Linux内核启动开始透彻分析全部启动过程的读者，因Linux代码量庞大而束手无策的人、想要了解Linux实际运行过程的人、渴求OS实操理论的人，本书必将成为他们不可或缺的参考书。

◆ 著 [韩] 尹锡训 等
译 崔范松
责任编辑 傅志红
执行编辑 陈 曦
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：33.25 插页：2
字数：786千字 2014年7月第1版
印数：1－4 000册 2014年7月北京第1次印刷
著作权合同登记号 图字：01-2013-8469号

定价：99.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第0021号

审阅者评语

整整3年的写作……

事实上，这本书的正式编写时间大约是1年。但我说用了3年，原因在于编写团队整整用了2年时间分析Linux代码，在此基础上，用1年完成了整个写作过程。为了这本书，他们在3年内不分昼夜地专注于Linux内核这一庞大领域，细心钻研，提高实力，并使辛劳努力的成果最终面世！

正是由于了解这些情况，所以，我收到原稿进行审阅时，比作者们更感慨万千。看着这本凝聚着编写团队辛勤汗水的书，我感到无比自豪。

简言之，这本书既是属于广大读者的书，更是作者们自己的书。因为书中是他们用2年时间分析Linux内核并不断提高实力的亲身经历，所以比现有的其他书籍水平更高。而且，编写团队已经完全掌握了书中内容，应该可以解答读者们的大部分疑惑。

这本书如同一把锋利的手术刀，大家读了就能感到，书中的每个部分都充满了鲜活的细节。这恰恰反映出2年间诸位作者一直保持着自己的实力。另外，由实力雄厚的编写团队逐一解开该主题的所有谜团，这使该书通顺流畅。

鉴于以上这些原因，我认为这本佳作比市面上出现的任何一本内核书都更出色。

我为这3年来付出最大努力的作者们喝彩！同时也为能够阅读此书进行学习的读者们感到幸运。因为我很久以前接触过Linux内核，但当时还没有这本书作参考，所以经历了无数失误。我向希望学习Linux内核的各位读者强烈推荐这本辅导书。

白昌佑

曾在三星电子、三星SDS、(株) Nusco公司主持开发各类RTOS，在三星综合技术院开发编译器，在(株) Nusco开发调试器和管理程序及各种系统S/W。目前在(株) Nusco担任CEO、软件开发团队指导。10年来一直负责管理离线系统S/W学习小组http://www.iamroot.org。

前　言

分析 Linux 内核并将其编撰成册，这对我们来说是一项巨大的挑战。通过 iamroot.org，我们在 2009 年 5 月举办了第一次以分析 ARM Linux 内核为目的的聚会。而且，完成第一次分析前的 2 年时间里，我们每周都是单休。身为工程师，能够亲自分析以前在书上接触过的操作系统代码，这让我们感到无比喜悦，可实际的分析过程却更要求我们耐心。

我们通过数年来对内核源代码分析的学习，依据整理好的 iamroot 的指导进行源代码分析。这本书主要讲述 ARM 使用的 Linux 内核源代码，逐行分析源代码和内核的实际工作内容，主要面向希望深层次了解内核内部机制的广大读者。逐行分析内核源代码主要是分析接通电源后如何调用内核的起始代码，以及调用起始代码内各种函数并最终显示命令行提示符的一系列过程。从这一点上讲，本书结构与其他 Linux 内核教程有所不同，涵盖了其他教材上没有涉及的内容。Linux 内核的基本概念可通过许多资料获得，但不操作内核源代码，就无法满足工程师们了解操作系统实现方法的求知欲。这正是本书最大的价值所在。

在分析初期，甚至到第一次分析结束时，写书这件事都不在我们的计划之中。大家开始分析内核时还兴致勃勃，逐步深入后才明白，我们的能力和知识是有限的，而编书这一重任更是远远超出了自己的能力范围。我们明知自身还有很多不足之处却依然坚持编写此书，就是希望将分析过程中遇到的困惑和辛辛苦苦理解到的内容与广大喜欢 Linux 的工程师、对内核感兴趣的读者分享。因为我们相信，正是有了许多工程师对 Linux 做出的贡献，才能使大家学习 Linux 内核代码。

我们通过本书与大家分享经验并不是因为自己掌握的知识多，也不因为这本书涉及的内容完美无缺。就像腼腆的林纳斯（Linus）1991 年用邮件与世界分享自己的内核代码那样，我们也只希望自己整理的内容能够给一些人带来帮助，也希望另一些人以这本书为基础，写出更多、更好的著作。所以，其实是我们在大家的帮助下更加深入地理解 Linux 内核。希望这本书能为想要了解 Linux 内核的广大读者带来些许帮助。

全体作者

2012 年 8 月

目　录

第一部分　ARM Linux 内核——分析内核前需要做的准备

第 1 章　内核介绍及 2.6 版和 3.2 版 之间的差异 ····· 2
1.1　内核的诞生、作用以及内部结构 ····· 2
1.1.1　Linus 创造的 Linux ····· 2
1.1.2　由多种子系统集成运行的单内核 ····· 3
1.1.3　全世界最著名的通用操作系统 ····· 5
1.2　内核 2.6 版和 3.2 版之间的差异 ····· 5
第 2 章　内核构建系统 ····· 8
2.1　内核初始化 ····· 8
2.2　内核配置 ····· 9
2.3　内核构建 ····· 11
2.4　内核安装 ····· 17
第 3 章　了解 ARM 处理器 ····· 19
3.1　处理器概要和特征 ····· 19
3.2　处理器架构与核心 ····· 19
3.3　处理器命名规则 ····· 21
3.4　处理器内部结构 ····· 21
3.5　处理器模式和寄存器 ····· 23
3.6　处理器异常 ····· 25
3.7　硬件扩展功能 ····· 26
3.7.1　缓存 ····· 26
3.7.2　内存管理装置 ····· 26
3.7.3　协处理器 ····· 26
第 4 章　构建分析环境 ····· 28
4.1　下载并安装 Linux 源内核 ····· 28

4.1.1　下载源内核 ……28
4.1.2　安装源内核 ……30
4.2　安装 ctags+cscope ……31
4.2.1　用 ctags 制作源代码标签 ……31
4.2.2　制作 cscope 标签数据库 ……33
4.3　vim 插件下载及环境设置 ……34
4.3.1　下载 vim 插件 ……34
4.3.2　vim+plugin 的环境结构 ……37
4.3.3　vim 环境设置 ……38
4.4　查看源码分析环境工具 ……40

第二部分　内核的启动——start_kernel 调用方法

第 5 章　准备解压内核 ……48
5.1　进入启动加载后结束首个启动——start 标签 ……49
5.2　BSS 系统域初始化——not_relocated 标签 ……50
5.3　激活缓存——cache_on 标签 ……53
5.4　页目录项初始化——__setup_mmu 标签 ……56
5.5　指令缓存激活及缓存策略适用——__common_mmu_cache_on 标签 ……58
第 6 章　从压缩的内核 zImage 还原内核映像 ……60
6.1　解压内核并避免覆写——wont_overwrite、decompress_kernel 标签 ……61
6.2　调用已解压内核——call_kernel 标签 ……62
6.3　缓存清理及清除——cache_clean_flush 标签 ……62
6.4　缓存禁用——cache_off 标签 ……64
第 7 章　调用 start_kernel() ……65
7.1　初始化指向——stext 标签 ……65
7.2　处理器信息搜寻——__look_processor_type ……69
7.2.1　__lookup_processor_type 标签 ……69
7.2.2　__proc_info_begin 和__proc_info_end 中保存的信息 ……71
7.2.3　在 MMU 禁用状态下将虚拟地址转换为物理地址 ……73
7.2.4　查找 proc_info_list 结构体并比较处理器信息 ……74
7.3　搜寻我的机型——__lookup_machine_type ……75
7.3.1　__lookup_machine_type 标签 ……75
7.3.2　保存在__arch_info_begin 和__arch_info_end 中的 machine_desc 信息及访问路径 ……76
7.3.3　查找 machine_desc 结构体并比较机器信息 ……77
7.4　源自启动加载项的 atags——__vet_atags 标签 ……78

7.5 对虚拟内存进行基础创建——__create_page_tables 标签 ····· 81
7.6 设置核心（core）——v6_setup 标签 ····· 85
7.7 打开 MMU 并使用虚拟地址——__enable_mmu/__turn_mmu_on 标签 ····· 86
7.8 跳转至 start_kernel——__mmap_switched 标签 ····· 90

第三部分 内核的执行——内核的起始与结束位置

第 8 章 start_setup_processor_id()~lock_kernel() ····· 94
8.1 smp_setup_processor_id()、lockdep_init()、debug_objects_early_init() ····· 95
8.1.1 smp_setup_processor_id() ····· 95
8.1.2 lockdep_init() ····· 95
8.1.3 debug_objects_early_init() ····· 96
8.2 栈溢出感应——__boot_init_stack_canary ····· 98
8.3 初始化提供进程集成方法的 cgroup——__cgroup_init_early() ····· 98
8.3.1 cgroupfs_root 和 cgroup 的关联初始化——init_cgroup_root() ····· 102
8.3.2 初始化子系统——cgroup_init_subsys() ····· 103
8.4 禁用 IRQ ····· 104
8.5 early_boot_irqs_off()、early_init_irq_lock_class() ····· 104
8.6 大内核锁——lock_kernel() ····· 106
第 9 章 注册针对时钟事件的处理器 ····· 111
9.1 函数的声明和定义——tick_init() ····· 111
9.2 注册处理事件的处理器——_clockevents_register_notifier() ····· 113
9.2.1 为 clockevents_lock 添加自旋锁 ····· 114
9.2.2 clockevents_chain 生成原理 ····· 115
9.2.3 在 clockevents_chain 中注册 tick_notifier 的方法 ····· 116
9.2.4 对 clockevents_lock 解除自旋锁的原理 ····· 117
第 10 章 在 CPU 位图中注册当前运行 CPU/初始化 HIGHMEM 管理 ····· 119
10.1 在包含热插拔信息的位图上添加执行 init_task 的 CPU——boot_cpu_init() ····· 119
10.2 管理高端内存——page_address_init() ····· 121
第 11 章 整体指向——setup_arch ····· 123
第 12 章 unwind_init()~early_trap_init() ····· 126
12.1 栈回溯——unwind_init() ····· 126
12.2 求出包含机器信息的 machine_desc 结构体——setup_machine() ····· 126
12.3 处理 ATAG 信息——setup_arch() ····· 127
12.4 处理启动参数——parse_cmdline() ····· 129

12.5 构建源代码树——request_standard_resources()……131
12.6 初始化 cpu possible 位图——smp_init_cpus()……136
12.7 用栈指定各 ARM 异常模式——cpu_init()……137
12.8 初始化以处理异常——early_trap_init()……138
12.9 查看中断处理器函数……143
12.9.1 调用 IRQ 处理器——asm_do_IRQ()……147
12.9.2 返回中断之前——ret_to_user 标签……147

第 13 章 设置处理器—— setup_processor()……150

13.1 查看 setup_processor()结构……150
13.2 查找 CPU ID——read_cpuid_id()……151
13.3 查找处理器信息——lookup_processor_type()……153
13.4 查找处理器结构信息——cpu_architecture()……153
13.5 查找处理器缓存类型_cacheid_init()……156
13.6 调用处理器初始化函数——cpu_proc_init()……160

第 14 章 准备内存分页—— paging_init()……163

14.1 查看 paging_init()的整体结构……163
14.2 设置内存类型表——build_mem_type_table()……165
14.3 检验内存信息——sanity_check_meminfo()……166
14.4 准备页表——prepare_page_table()……168
14.4.1 prepare_page_table()……168
14.4.2 Linux 的分页结构……170
14.4.3 求出页目录项……170
14.4.4 pmd_clear()……172
14.5 设备区域映射准备——devicemaps_init()……174
14.6 准备使用高端内存——kmap_init()……177
14.7 初始化零页……178
14.7.1 分配内存——__alloc_bootmem_nopanic()……179
14.7.2 在指定节点使用 fallback 分配内存——alloc_bootmem_core……180
14.7.3 将虚拟地址变换为 page 结构体——virt_to_page……182
14.8 保持数据缓存一致性——flush_dcache_page()……182

第 15 章 在启动时初始化内存分配器……184

15.1 bootmem 函数流和数据结构……185
15.2 查看 bootmem_init()结构……188
15.3 查找虚拟内存盘位置——check_initrd()……189
15.4 将节点的 BANK 信息反映到页目录——bootmem_init_node()……191
15.4.1 map_memory_bank()……192
15.4.2 bootmem_bootmap_pages()……195

15.4.3 find_bootmap_pfn() ····· 196
15.4.4 node_set_online() ····· 197
15.4.5 NODE_DATA 宏 ····· 198
15.4.6 init_bootmem_node() ····· 200
15.4.7 free_bootmem_node() ····· 202
15.4.8 reserve_bootmem_node() ····· 202
15.5 排除 0 号节点——reserve_node_zero() ····· 203
15.6 排除虚拟内存盘节点——bootmem_reserve_initrd() ····· 204
15.7 设置为无可用页——bootmem_free_node() ····· 205
15.8 初始化 free_area 区域 ····· 207
15.8.1 free_area 结构体 ····· 207
15.8.2 free_area_init_node() ····· 208
15.8.3 free_area_init_core() ····· 209
15.8.4 init_currently_empty_zone() ····· 211
15.8.5 memmap_init() ····· 212

第 16 章 mm_init_owner()~preempt_disable() ····· 217

16.1 设置内存拥有者——mm_init_owner() ····· 217
16.2 保存命令行——setup_command_line() ····· 218
16.3 初始化 per-cpu 数据——setup_per_cpu_areas() ····· 219
16.4 求 CPU 个数——setup_nr_cpu_ids() ····· 221
16.5 注册 SMP 上的启动进程——smp_prepare_boot_cpu() ····· 222
16.6 初始化数据结构以使用调度程序——sched_init() ····· 224
16.6.1 为集合调度中使用的 task_group 的 sched_entity 结构体和 runqueue 结构体分配内存 ····· 224
16.6.2 初始化 root_domain、rt_bandwidth、task_group 相关数据结构 ····· 227
16.6.3 初始化系统上所有可用 CPU 的就绪队列 ····· 229
16.6.4 初始化当前任务的调度相关值与注册针对负载均衡的中断处理器 ····· 230
16.7 允许内核抢占和阻止抢占——preempt_enable()/preempt_disable() ····· 231

第 17 章 构建借用内存的后台 ····· 233

17.1 在 build_all_zonelists()中操作的一些数据结构 ····· 233
17.2 查看 build_all_zonelists()结构 ····· 235
17.3 决定 zone 的列表方式——set_zonelist_order() ····· 236
17.4 构建备用列表和备用位图——__build_all_zonelists() ····· 238
17.4.1 build_zonelists() ····· 239
17.4.2 build_zonelist_in_node_order() ····· 241
17.4.3 build_zonelists_in_zone_order() ····· 243
17.4.4 build_thisnode_zonelists() ····· 244
17.4.5 build_zonelists_cache() ····· 244

17.5　输出备用列表信息——mminit_verify_zonelist()……246
17.6　指定处理页分配请求的节点——cpuset_init_current_mems_allowed()……246
17.7　求空页数——nr_free_pagecache_pages()……247
17.8　页移动性……250

第 18 章　page_alloc_init()~pidhash_init()……253

18.1　处理用于热插拔 CPU 的页——page_alloc_init()……253
18.2　处理 console 参数——parse_early_param()……255
18.3　处理特殊参数——parse_args()……257
18.4　确认中断处理是否激活——irqs_disable()……260
18.5　内核异常列表定义——sort_main_extable()……261
18.6　初始化 RCU 机制——rcu_init()……263
18.7　准备使用 IRQ——early_irq_init()……266
18.8　初始化中断——init_IRQ()……269
18.9　构建迅速搜寻进程信息的结构——pidhash_init()……271

第 19 章　init_timers()~page_cgroup _init()……273

19.1　初始化计时器——init_timers()……274
19.1.1　timers_cpu_notify()……275
19.1.2　register_cpu_notifier()……275
19.1.3　open_softirq()……276
19.2　初始化高分辨率计时器——hrtimers_init()……277
19.3　注册 softirq 的回调函数——softirq_init()……280
19.4　设置 xtime——timekeeping_init()……282
19.5　初始化硬件计时器——time_init()……285
19.6　初始化时钟时间——sched_clock_init()……286
19.7　激活 CPU 的中断处理——local_irq_enable()……288
19.8　检测用作根文件系统的 init 虚拟内存盘……288
19.9　初始化以分配动态内存——vmalloc_init()……289
19.10　预先初始化目录项和索引节点缓存——vfs_caches_init_early()……290
19.11　初始化 cpuset 子系统——cpuset_init_early()……293
19.12　初始化内存子系统——page_cgroup_init()……294

第 20 章　终止 bootmem 分配器并替换为伙伴系统……297

20.1　mem_init()函数的调用关系及其与数据结构的相互关系……297
20.2　查看 mem_init()结构……298
20.3　记录到不存在的内存位图——free_unused_memmap_node()……300
20.4　移交至普通空白页伙伴系统——free_all_bootmem_node()……301
20.4.1　register_page_bootmem_info_node()……301
20.4.2　free_all_bootmem_core()……303

20.4.3 __free_pages_bootmem() ····· 305
20.4.4 __free_pages() ····· 308
20.4.5 free_hot_cold_page() ····· 308
20.4.6 __free_pages_ok() ····· 309
20.5 移交到高端内存空白页伙伴系统——free_area() ····· 314
第 21 章 初始化以支持 CPU 热插拔 ····· 315
21.1 初始化 cpu_hotplug 成员变量——cpu_hotplug_init() ····· 315
21.2 CPU 的联机→脱机转换处理 ····· 316
第 22 章 激活 slab 内存分配器——kmem_cache_init() ····· 318
22.1 slab 分配器的概念及结构体 ····· 318
22.2 slab 分配器的重要结构体——kmem_cache 和 kmem_list3 ····· 319
22.3 查看 kmem_cache_init()结构 ····· 322
22.4 初始化 initkmem_list3[]、cache_cache、nodelist[] ····· 326
22.5 连接 kmem_list3 数组并决定 cache 压缩时间——set_up_list3s() ····· 328
22.6 求出用于 cache 扩展/压缩的页顺序——cache_estimate() ····· 329
22.7 malloc_sizes 和 cache_names ····· 332
22.8 生成 cache——kmem_cache_create() ····· 334
22.8.1 kmem_cache_zalloc() ····· 336
22.8.2 calculate_slab_order() ····· 337
22.8.3 setup_cpu_cache() ····· 337
22.8.4 enable_cpucache() ····· 339
22.9 生成 arraycache_init,kmem_list3 cache ····· 340
22.10 用 kmalloc()函数分配的内存替代静态分配的内存 ····· 342
第 23 章 kmem_trace_init()~security_init() ····· 344
23.1 生成 ID alloccator 缓存——idr_init_cache() ····· 345
23.2 初始化 pageset——setup_per_cpu_pageset() ····· 345
23.3 指定交叉节点——numa_policy_init() ····· 350
23.4 结束计时器初始化——late_time_init() ····· 353
23.5 测定 BogoMIPS——calibrate_delay() ····· 353
23.6 制作位图以分配进程识别符（ID）——pidmap_init() ····· 354
23.7 初始化优先树的数据结构——prio_tree_init() ····· 356
23.8 生成 anon_vma slab 缓存——anon_vma_init() ····· 356
23.9 为对象的每个用户赋予资格——cred_init() ····· 357
23.10 初始化数据结构以使用 fork()函数——fork_init() ····· 358
23.11 初始化生成进程的缓存——proc_caches_init() ····· 359
23.12 初始化缓冲缓存——buffer_init() ····· 361
23.13 准备密钥——key_init() ····· 364

第 24 章　初始化 VFS 中使用的多种缓存——vfs_cache_init()……367

第 25 章　radix_tree_init()~ftrace_init()……382

25.1　基数树相关数据结构初始化——radix_tree_init()……383
25.2　准备使用信号——signals_init()……383
25.3　注册并挂载 proc 文件系统——proc_root_init()……384
25.4　注册未能初始化的子系统——cgroup_init()……385
25.5　重置 top_cpuset 并注册 cpuset 文件系统——cpuset_init()……386
25.6　初始化任务统计信息接口——delayacct_init()……387
25.7　为管理延迟信息做准备——delayacct_init()……388
25.7.1　延迟审计……388
25.7.2　delayacct_init……389
25.7.3　task_delay_info 结构体和 delayacct_tsk_init()……390
25.8　检查写缓冲一致性——check_bugs()……392

第 26 章　同步内存与后备存储——page write back……394

26.1　页回写机制……394
26.2　激活页回写——pdflush_init()……395
26.3　pdflush 线程……397
26.4　指定页回写函数……398
26.5　周期性页回写和强制性页回写回调函数调用方法……399
26.5.1　周期性页回写函数——wb_kupdate()……399
26.5.2　强制性页回写函数——background_writeout()……401
26.6　初始化周期性页回写……403

第 27 章　查看启动内核的最终函数结构——rest_init()……405

第 28 章　生成执行函数的内核线程——kernel_thread()……407

28.1　查看 kernel_thread()结构……407
28.2　生成处理器的网关——do_fork()……408
28.3　复制父进程——copy_process()……411

第 29 章　唤醒新生成的任务……419

29.1　查看 wake_up_new_task 结构……419
29.2　获取任务的就绪队列——task_rq_lock()……421
29.3　改善任务的优先顺序——effective_prio()……422

第 30 章　准备使用内核……426

30.1　将当前进程转移到其他 CPU——sched_init_smp()……427
30.2　结束系统整体初始化——do_basic_setup()……429
30.2.1　生成执行 rcu_sched_grace_period()的线程——rcu_init_sched()……430

30.2.2 生成 events 工作队列——init_workqueues ····· 430
30.2.3 初始化 cpuset 子系统的 top_cpuset——cpuset_init_smp() ····· 437
30.2.4 生成 khelper 工作队列——usermodehelper_init() ····· 437
30.2.5 初始化 Linux 的设备模型——driver_init() ····· 439
30.2.6 在 proc 文件系统注册 irq 信息——init_irq_proc() ····· 446
30.2.7 调用内核未知子系统——do_initcalls() ····· 448
30.3 为初始化之后的操作做准备——init_post() ····· 451
第 31 章 内核线程守护进程 ····· 453
31.1 内核线程守护进程——kthreadd() ····· 453
31.2 忽略信号——ignore_signals() ····· 456
31.3 设置 nice 值——set_user_nice() ····· 458
31.4 搜索执行任务的 CPU——set_cpus_allowed_prt() ····· 463
31.5 搜索包含列表的实际结构体位置——list_entry() ····· 464
31.6 生成内核线程——create_kthread() ····· 466
第 32 章 find_task_by_pid_ns()~cpu_idle() ····· 469
32.1 用 PID 搜索任务——find_task_by_pid_ns() ····· 470
32.2 解除 BKL——unlock_kernel() ····· 472
32.3 将调度类变更为 idle——init_idle_bootup_task() ····· 473
32.4 RCU 机制激活完成通知——rcu_scheduler_starting() ····· 473
32.5 激活内核抢占——preempt_enable_no_resched() ····· 473
32.6 执行进程调度表——schedule() ····· 474
32.7 Linux 启动万里长征的终点——cpu_idle() ····· 476

附 录

附录 A 汇编语言、gas 关键词总结 ····· 480
附录 B 内核分析常见 API ····· 485
附录 C 浅谈 ext2 文件系统 ····· 487
附录 D Linux 线程模型 ····· 497
附录 E 链接器脚本文件结构 ····· 500
后记 ····· 510
索引 ····· 513

第一部分

ARM Linux内核

——分析内核前需要做的准备

Linux 内核源代码即使压缩后也超过了 70 MB。对如此庞大的源代码进行逐行分析往往需要付出莫大的努力。但是，需要倾注的努力与我们具备的 Linux 知识、对 ARM 处理器的理解、代码分析工具的使用经验等或许是一种反比关系。也就是说，如果我们具备了这些丰富的知识和经验，将会大大缩短分析代码的时间。因此，正式进入源代码分析之前，需要做好充分准备。

第一部分是内核代码分析的准备阶段，主要包括对 Linux 内核的介绍、如何编译内核并生成可加载到内存的图像、必须掌握的 ARM 处理器相关内容、为提高代码分析效率而搭建分析环境等代码分析必备常识。

真正进入代码分析之前，我们利用第一部分做个热身吧！

第1章

内核介绍及2.6版和3.2版之间的差异

本书对启动Linux内核过程中调用的代码进行分析，并着重讲解Linux是怎样运行的，以及Linux运行时哪些部分需要初始化等内容。

本章在分析之前，先讲解两个内容：第一，Linux是由林纳斯发起的当前最成功的操作系统之一；第二，Linux内核是由多个子系统构成的一个单内核。

最后，了解本书中分析的内核版本和内核3.2版之间的差异。在这一过程中大家会发现，内核从2.4版升级到2.6版的过程中，结构上并没有发生太大变化。所以本书并没有把重点放在内核3.2版上，但对其的讲解有助于分析最新版的内核。

1.1 内核的诞生、作用以及内部结构

1.1.1 Linus 创造的 Linux

Linux[①]是由赫尔辛基大学的研究生林纳斯·托瓦兹（Linus Torvalds）在1991年发布的操作系统。林纳斯在大学期间对MINIX的许可授权政策感到不满，从而打算开发Linux。虽然开发初期是以MINIX为基础的，但进展到一定程度后，他在运行Linux内核的Linux系统上进行了开发。[②]

之后，与MINIX相关的组件全部被GNU应用程序取代[③]。由于从GNU阵营中得到了许多支持和无数内核黑客的贡献，Linux成为全世界应用最广、最为成功的开放源操作系统。

---

① Linux的早期名称是不一样的。刚开始林纳斯将自己制作的组件命名为Freax，后来接受了上传其内核的网站管理员的提议，更名为Linux，并使用至今。Linux的意思是“林纳斯的UNIX”（Linus's UNIX）或“Linux不是UNIX”（Linux is not UNIX），或者是“林纳斯的MINIX”（Linus's MINIX）。

② 构建Linux的编译器是linux-gcc。那么linux-gcc又是用什么编译器构建的呢？某位内核黑客与林纳斯对此进行过对话，参见如下URL：http://groups.google.com/group/comp.os.linux/msg/4ae6db18d3f49b0e。

③ 通过GNU获得了什么灵感或感动呢？在几乎与此同时的1992年年初，林纳斯用GPL取代了Linux的许可。

1.1.2 由多种子系统集成运行的单内核

Linux内核由许多子系统构成。图1-1表示各子系统之间的关系，请通过该图了解各子系统的作用。

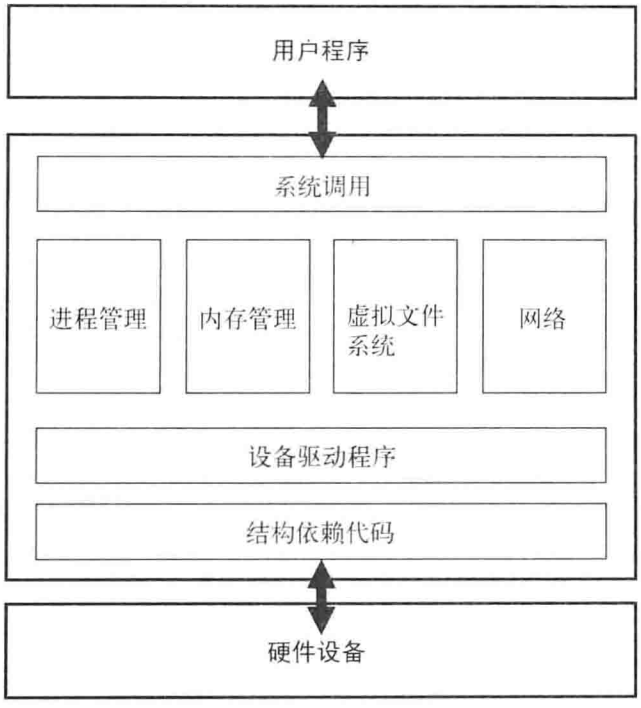


图1-1 Linux子系统结构

结构依赖代码

无论为何种结构类型，Linux均提供相同服务。但在其底层，每个结构都有需要另行控制的部分。这些部分是指用于CPU、MMU以及机载（on board）状态的低级（low-level）驱动程序，而且这些代码均位于arch目录下。

设备驱动程序

我们使用的电脑中，有很多周边设备（LCD、按钮、蓝牙、Wi-Fi等）通过多种方式（J2C、SCSI、EIDE）进行通信。设备驱动程序是指为这些高层设备制订的代码，位于drivers目录下。Linux内核中有一半以上是设备驱动程序代码，Linux通常最先封装最新设备。

进程管理

执行进程前必须先分配CPU资源。同时执行多个进程时，根据制定的政策公平分配资源，并且要管理进程的生成及状态转移的过程。进程管理就是执行这些任务的子系统，位于kernel目录下。

内存管理

与CPU同样重要的系统资源是内存。内存管理子系统负责内存分配、释放及共享，其相关代

码位于mm目录下。

虚拟文件系统

虚拟文件系统在Linux中是个很有意思的部分。该系统将多种文件系统抽象化，从而提供共通的接口。因此，我们能够在Linux中使用相同的文件操作访问多个文件系统或设备。

网络子系统

网络子系统就像TCP/IP通信协议和套接字（SOCKET）接口那样，对网络装置提供抽象化概念，可通过一致的方法使用多种网络装置。

系统调用接口

提供接口，以在用户空间级调用已在内核中实现的特定功能。根据结构的不同，其实现方法可能有所不同。其相关代码位于linux/kernel或linux/arch目录下。

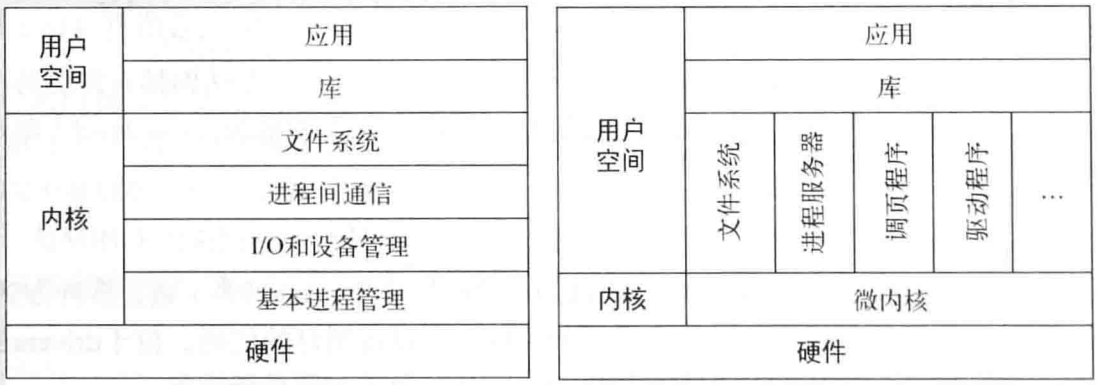


补充知识点　内核结构的区分：单内核（宏内核）与微内核

单内核与微内核是具有代表性的内核结构。其特征如下。

- **单内核**——所有内核服务均在内核地址空间中存在并运行。直接调出内核服务。
- **微内核**——内核服务中一部分位于用户空间。利用消息传送方式调用内核服务。

内核结构的区分标准是，所有内核服务是否集成在内核地址空间运行。由于2.6版Linux内核具有支持模块的模块化内核（Modular kernel）的特性，偶尔会让人产生误解。但模块也是在内核地址空间中运行的，因此可将Linux内核视作单内核（请参考图1-2）。



(a) 基于单内核的操作系统　　(b) 基于微内核的操作系统

图1-2　单内核vs.微内核

※ 参考文献：Monolithic kernel vs. Microkernel（http://www.vmars.tuwien.ac.at/courses/akti12/journal/04ss/article_04ss_Roch.pdf）

1.1.3 全世界最著名的通用操作系统

也许林纳斯没有想到，现在的Linux比其他任何一种操作系统都得到了更广泛的应用。重视可靠性和性能的超级计算机中也体现出了Linux的真正价值（请参考图1-3）。此外，其用途已扩大到网络设备、电视机、游戏机领域。目前，安卓系统在手机市场上占据近一半的席位，这再次证实了Linux的实用性。

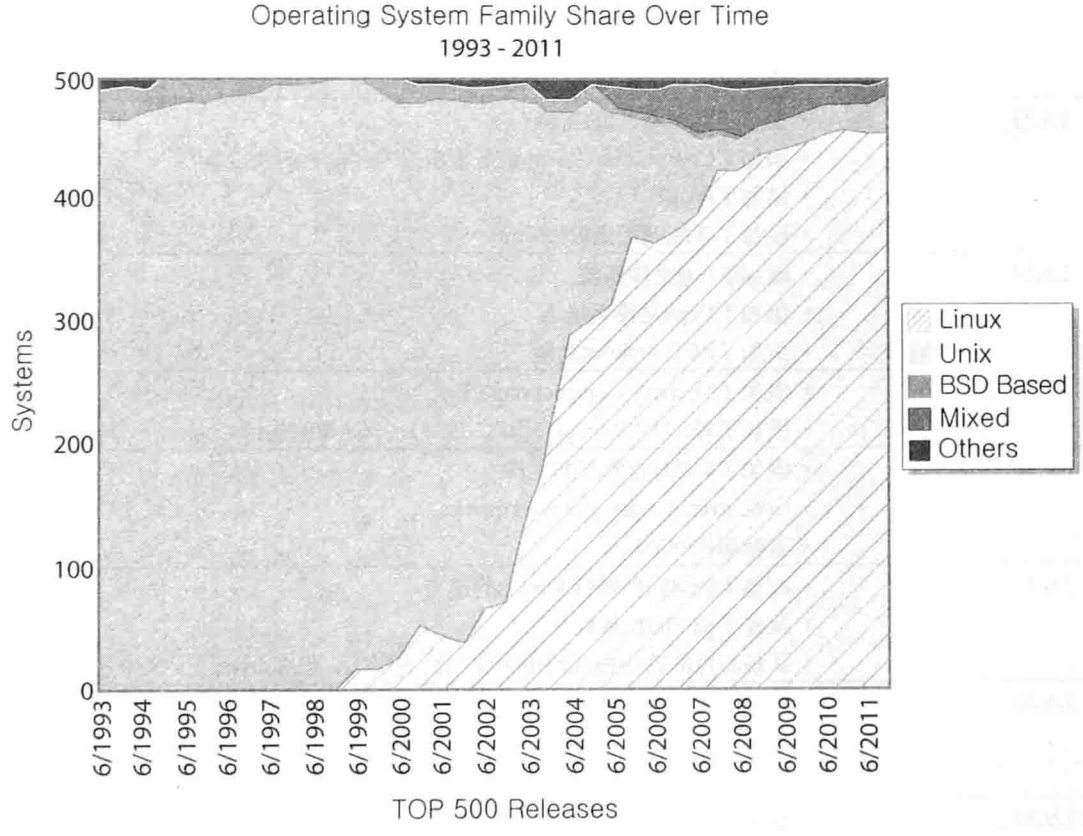


图1-3 TOP 500超级计算机的各OS占有率

1.2 内核 2.6 版和 3.2 版之间的差异

2011年7月，Linux内核3.0发布。2.6版是2003年首次发布的，从版本2过渡到版本3整整用了8年。但版本号的更改并不具有什么重大意义，只是用于简化版本号体系，以及纪念Linux内核发行20周年而已。

实际上内核结构并没有发生太大变化。但和每次发布时一样，很多地方还是有了一些修改。这些部分的修改程度和其他发布时期相差无几，所以也就没有什么特别大的意义。下表罗列了本书中将使用的2.6.30版至当前发布的3.2版中主要的变更事项。

表1-1　各Linux内核版本的主要变更事项

版　本	变更事项
2.6.31	• 添加了对USB 3的支持 • 添加了CUSE（Character device in Userspace） • 添加了Performance Counter子系统 • 添加了Kmemcheck、Kmemleak功能
2.6.32	• 添加了Per-backing-device-based writeback • 改善了btr文件系统 • 添加了Kernel Samepage Merging功能 • 添加了运行时电源管理功能
2.6.33	• 去除了对Android的支持 • 添加了Compcache（swap设备等虚拟内存盘上的块设备） • 添加了DRBD • 添加了系统调用函数recvmmsg()
2.6.34	• 添加了Log文件系统 • 添加了Ceph文件系统 • 添加了RCU lockdep功能
2.6.35	• 添加了Memory Compaction的实现 • 添加了对L2TP Version 3(RFC 3931）、CAIF的支持
2.6.36	• 添加了对Tile处理器的支持 • Concurrency-managed workqueues • 重新制作OOM
2.6.37	• 改善了Ext4的扩展性和mkfs的速度 • 改善了XFS的扩展性 • 从核心性能的内核代码中去除了所有BKL，但在driver、ioctl中还无法去除
2.6.38	• 添加了使用会话ID自动集成进程的功能 • 改善了VFS的扩展性（重新制作目录树缓存） • 添加了Transparent huge pages功能
2.6.39	• 默认根据2.6.37的Ext4的改善事项进行设置 • 去除所有BKL代码
3.0	• 防止自动整理Btr文件系统，添加了scrubbing功能的同时改善了性能 • 添加了系统调用函数sendmmsg() • 添加了对XEN dom0的支持 • 添加了用于改善页缓存性能的清理缓存 • 添加了Berkeley Packet Filter JIT filtering
3.1	• 添加了对OpenRISC处理器的支持 • 改善了writeback性能 • 改善了slab分配器的性能
3.2	• 支持增大ext4中较大的模块 • 增加Btr文件系统的scrub速度 • 去除CFS中的CPU带宽限制 • 添加了对Hexagon处理器的支持

从表中可以看出，随着版本的升级，变更的大部分事项只是添加或改善了新的设备驱动程序、对结构的支持、文件系统等，或删除了不必要的部分。由于我们把分析焦点放在Linux运行所需的最基本事项上，因此不会受到这些变更事项的影响。

Linux启动过程中执行的任务和Linux运行过程中所需的最基本要求是不会轻易变更的，所以我们推荐读者以本书内容为基础，先分析2.6版内核，再分析3.x版内核。

第2章 内核构建系统

操作系统是由许多逻辑构成要素组成的综合软件。Linux内核同样也是一种操作系统，由必要的调度程序、文件系统、内存管理器、网络系统等许多子系统组成。这种内核看起来既复杂又庞大，但其生成只需要一个二进制可启动文件，也就是由zImage（或bzImage）文件通过构建（building）过程生成。

图2-1表示可启动二进制映像文件zImage（又称内核）的生成过程。通过内核初始化、内核配置、内核构建、内核安装过程，形成Linux内核并处于可启动状态。

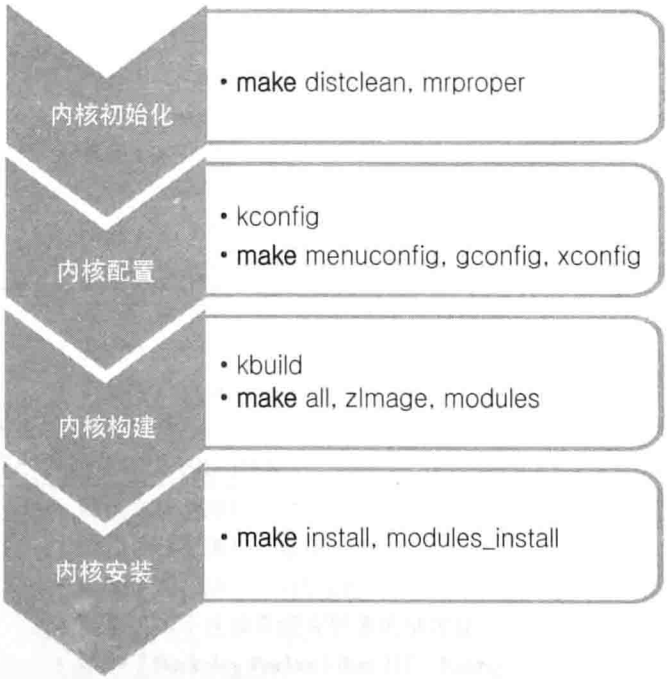


图2-1　内核构建过程

2.1　内核初始化

为了构建内核，用户必须先完成内核初始化过程。大部分用户可通过kernel.org等多个网站获

得已压缩成tar.gz格式的内核初始源代码。压缩成tar.gz的内核源代码解压后，会生成初始状态的内核源代码树，这种状态称为内核的初始状态。

此外，可通过make mrproper和make distclean等指令使之恢复到刚解压的状态。观察位于内核源代码树顶端的Makefile可以看出，所谓mrproper对象只清除包括“.config”文件在内的、为内核编译及链接而生成的诸多设置文件。[①]distclean对象执行mrproper命令，清除内核编译后生成的所有对象文件、备份文件等。

2.2 内核配置

虽然完成了内核的初始化过程，但并不意味着可以立即对内核进行编译。如果将初始化的内核直接进行编译，虽然能生成vmlinux，但大部分情况下会引起内核严重错误（kernel panic）。构建内核前需要执行的最重要、最需要慎重处理的部分是内核配置（kernel configuration）过程。内核配置过程也是适当选择与自身系统相吻合的各种内核要素的过程。

为了配置内核，在Makefile中使用具有代表性的xconfig、menuconfig、gconfig。这些都是图形用户界面（GUI）形式的内核配置前端。用户只需选择自己感到相对易于使用的前端即可。gconfig是基于GTK的前端，xconfig是基于QT的前端。

```
...
xconfig: $(obj)/qconf
    $< $(Kconfig)

gconfig: $(obj)/gconf
    $< $(Kconfig)

menuconfig: $(obj)/mconf
    $< $(Kconfig)
...
```

在script/kconfig/Makefile中可以查询xconfig、menuconfig、gconfig对象，各对象会执行与之对应的二进制文件。大部分用户使用menuconfig对象进行内核配置，就像从上面的Makefile中看到的那样，menuconfig执行mconf。但即使有基于GUI的前端，内核配置依然是一项会发生很多执行错误的庞大工作。幸运的是，每个系统均提供自定义配置文件。如果查看arch/$(ARCH)/configs目录，就会发现与各个SoC（System on Chip）相符的自定义配置文件。

```
code@kernel:~/src/linux-2.6.30.4/arch/arm/configs$ ls
...
iop33x_defconfig          rpc_defconfig
ixp2000_defconfig         rx51_defconfig
ixp23xx_defconfig         s3c2410_defconfig
ixp4xx_defconfig          s3c6400_defconfig
jornada720_defconfig      sam9_l9260_defconfig
```

① mrproper的意思是Mr.Proper。宝洁公司的洗涤剂品牌Mr.Clean在欧洲以Mr.Proper的名义得到广泛应用，所以李纳斯借用了其“清除干净”之意。

```
kafa_defconfig              shannon_defconfig
kb9202_defconfig            shark_defconfig
...
code@kernel:~/src/linux-2.6.30.4/arch/arm/configs$
```

例如，查看arch/arm/configs目录会发现如上所示，确实存在面向（基于ARM结构的）SoC的自定义配置文件。如果想使用三星S3C6400的系统配置文件，可通过简单的make s3c6400_defconfig命令缩短用make menuconfig配置时所需的大量时间。

无论是利用make menuconfig配置内核，还是简单地使用make xxx_defconfig进行配置，均会生成.config文件。

```
...
CONFIG_EXT2_FS=m
CONFIG_EXT2_FS_XATTR=y
CONFIG_EXT2_FS_POSIX_ACL=y
CONFIG_EXT2_FS_SECURITY=y
CONFIG_EXT2_FS_XIP=y
CONFIG_EXT3_FS=m
CONFIG_EXT3_DEFAULTS_TO_ORDERED=y
CONFIG_EXT3_FS_XATTR=y
CONFIG_EXT3_FS_POSIX_ACL=y
CONFIG_EXT3_FS_SECURITY=y
CONFIG_EXT4_FS=m
...
```

.config文件是只构建自身内核所需的内核配置目录。简单查看即知，它是在CONFIG_XXX变量值中用y、n、m这3个状态进行配置的目录。这种形态的内核配置系统叫做kconfig。根据kconfig提供的3个状态（y、n、m）决定是否构建内核。状态为“y”时，相应的二进制文件与vmlinux链接；状态为“m”时，虽然不会和vmlinux链接，但是作为模块执行编译。

前面介绍的mconf通过.config配置文件生成autoconf.h头文件。可通过scripts/kconfig/confdata.c的conf_write_autoconf()函数得知autoconf.h的生成过程。

```
...
#define CONFIG_EXT2_FS_MODULE 1
#define CONFIG_EXT2_FS_XATTR 1
#define CONFIG_EXT2_FS_POSIX_ACL 1
#define CONFIG_EXT2_FS_SECURITY 1
#define CONFIG_EXT2_FS_XIP 1
#define CONFIG_EXT3_FS_MODULE 1
#define CONFIG_EXT3_DEFAULTS_TO_ORDERED 1
#define CONFIG_EXT3_FS_XATTR 1
#define CONFIG_EXT3_FS_POSIX_ACL 1
#define CONFIG_EXT3_FS_SECURITY 1
#define CONFIG_EXT4_FS_MODULE 1
...
```

前面介绍的这些配置和include/linux/autoconf.h一样，通过#define语句生成头文件。.config文件起着非常重要的作用，它不仅参与文件单位的链接，还决定是否以源代码为单位执行包含。内

核利用以.config文件为背景生成的autoconf.h，在预处理阶段决定是否包含#ifdef语句。

2.3 内核构建

利用kconfig完成内核配置，并准备好具有自身内核配置的.config文件后，即可构建内核（kernel building）。构建内核是指，编译（compile）内核并链接二进制文件，由此生成一个二进制文件zImage的一系列过程。

内核编译通过一千多个Makefile完成，它们存在于各代码目录中。通过内核顶端目录中的Makefile开始编译内核，并通过因Kbuild系统而存在的script目录上的多个脚本文件生成zImage。

先看一下内核编译的开始阶段，即顶端目录中的Makefile。

:: Makefile

```
...
PHONY := _all
_all:
...
include $(srctree)/scripts/Kbuild.include                    ❶
...
ifeq ($(KBUILD_EXTMOD),)
_all: all
else
_all: modules
endif
...
all: vmlinux
...
include $(srctree)/arch/$(SRCARCH)/Makefile                  ❷
...
```

通过上面的代码找到_all对象的从属项目all对象，并再次搜寻vmlinux对象。

代码段❶包含的Makefile.include文件定义了用于构建内核的共同变量或各种脚本。分析Makefile时可能遇到未定义的符号，此时不必惊慌。其大多数可以在Kbuild.include内找到，也可见于script目录内的文件。

需要稍加留意的是，代码段❷包含了各结构的Makefile。这个Makefile在以后构建内核的过程中负责对从属结构部分执行编译。先观察一下。

从属结构的文件均位于arch目录下。对于这些从属结构的文件，执行编译时同样使用从属结构的Makefile。

首先观察arch/arm/Makefile。大部分嵌入式系统在闪存中通过主内存读取内核代码并启动，因此，可通过图2-2a部分确认KBULD_IMAGE变量名为zImage。根据图2-2b最终执行arch/arm/boot/Makefile的vmlinux对象。

arch/arm/Makefile

```
...
ifeq ($(CONFIG_XIP_KERNEL),y)
   KBUILD_IMAGE := xipImage
else
   KBUILD_IMAGE := zImage                                    -(a)
endif

all:  $(KBUILD_IMAGE)
boot := arch/arm/boot
...
zImage Image xipImage bootpImage uImage: vmlinux             -(b)
$(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
...
```

arch/arm/boot/Makefile

```
...
$(obj)/Image: vmlinux FORCE
   $(call if_changed,objcopy)
      @echo '  Kernel: $@ is ready'

$(obj)/compressed/vmlinux: $(obj)/Image FORCE
   $(Q)$(MAKE) $(build)=$(obj)/compressed $@

$(obj)/zImage: $(obj)/compressed/vmlinux FORCE               -(c)
   $(call if_changed,objcopy)'
      @echo '  Kernel: $@ is ready'
...
```

arch/arm/boot/compressed/Makefile

```
...
$(obj)/vmlinux: $(obj)/vmlinux.lds$(obj)/$(HEAD) $(obj)/piggy.o\      -(d)
            $(addprefix $(obj)/, $(OBJS)) FORCE
            $(call if_changed,ld)
            @:
$(obj)/piggy.gz: $(obj)/../Image FORCE
            $(call if_changed,gzip)
$(obj)/piggy.o:  $(obj)/piggy.gz FORCE
...
```

图2-2　从属结构的Makefile

下面观察arch/arm/boot/Makefile。

通过图2-2c执行存在于arch/arm/boot/compressed/Makefile内的$(obj)/vmlinux对象。$(call if_changed,objcopy)将执行script/Makefile.lib中声明的cmd_objcopy。

```
cmd_objcopy = $(OBJCOPY) $(OBJCOPYFLAGS) $(OBJCOPYFLAGS_$(@F)) $< $@
```

对内核编译生成的$(obj)/compressed/vmlinux执行objcopy命令，生成已清除多余符号的二进制映像。

从图2-2d中可以看到，在arch/arm/boot/compressed/Makefile中用gzip压缩已清除符号的vmlinux文件，从而生成piggy.gz，并利用各从属结构的链接器脚本（$(obj)/vmlinux.lds）执行链接。

以上大致介绍了从属结构的Makefile的执行。现在再次回到顶端目录中的Makefile，仔细观察vmlinux对象。

:: Makefile

```
...
init-y      := init/
drivers-y   := drivers/ sound/ firmware/
net-y       := net/
libs-y      := lib/
core-y      := usr/
...
core-y      += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
...
init-y      := $(patsubst %/, %/built-in.o, $(init-y))
core-y      := $(patsubst %/, %/built-in.o, $(core-y))
drivers-y   := $(patsubst %/, %/built-in.o, $(drivers-y))
net-y       := $(patsubst %/, %/built-in.o, $(net-y))
libs-y1     := $(patsubst %/, %/lib.a, $(libs-y))
libs-y2     := $(patsubst %/, %/built-in.o, $(libs-y))
libs-y      := $(libs-y1) $(libs-y2)
...
vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
vmlinux-lds  := arch/$(SRCARCH)/kernel/vmlinux.lds
export KBUILD_VMLINUX_OBJS := $(vmlinux-all)
...

vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o                    ❶
 $(kallsyms.o) FORCE
$(call if_changed_rule,vmlinux__)
...
$(sort $(vmlinux-init) $(vmlinux-main)) $(vmlinux-lds): $(vmlinux-dirs) ;            ❷
...
$(vmlinux-dirs): prepare scripts                                                     ❸
    $(Q)$(MAKE) $(build)=$@
...
quiet_cmd_vmlinux__?= LD      $@                                                     ❹
    cmd_vmlinux__?= $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
    -T $(vmlinux-lds) $(vmlinux-init)                          \
    --start-group $(vmlinux-main) --end-group                  \
    $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o FORCE ,$^)
...
```

仔细观察代码段❶会发现，vmlinux对象上有多个从属列表，各从属列表均对目录名加以标识。在代码段❷可以发现将前面标识的目录名作为对象的行，并具有从属列表vmlinux-dir。最终执行代码段❸。

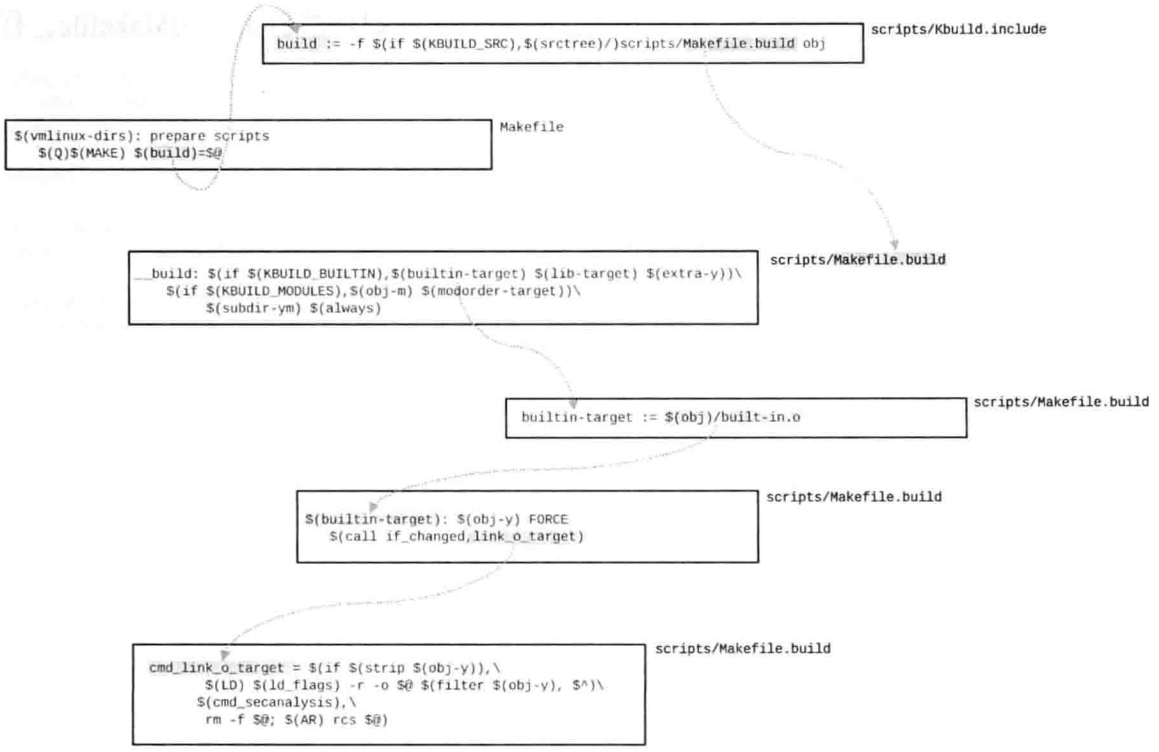


图2-3　built-in.o 生成脚本

$(build)定义于script/Kbuild.include，如图2-3所示。换言之，在代码段❸中，对vmlinux-dirs内包含的所有目录来说，如下这个obj变量中存储着各目录名，并执行Makefile.build。

```
@make -f Makefile.build obj=目录
```

Makefile.build中有个叫做__build的基本对象，并具有builtin-target从属项目。可以发现，builtin-target从属项目是一种以built-in.o对象文件作为对象的规则。built-in.o位于之前从顶端目录的Makefile传递来的、与obj相对应的目录下。

从图2-4中可以看出，obj变量中指定的各目录内有Makefile。在这些Makefile中，由Kconfig配置的各个obj-y用一个文件built_in.o执行链接。cmd_link_o_target使对应于obj-y的所有文件都与文件built-in.o链接。

换言之，各目录均通过Kconfig选择的组件生成综合对象文件built-in.o。生成的各个built-in.o通过代码段❹创建ELF格式的vmlinux文件。此时，通过vmlinux-lds变量对如何链接进行明确说明后，利用链接脚本执行链接。

这样链接的vmlinux是ELF格式的二进制映像，由文件系统、网络、设备驱动程序及调度程序、计时器管理等内核核心功能组成，指压缩成zImage前的内核二进制文件。构建内核过程中生成的各对象文件均通过链接器（linker）组合成一个名为vmlinux的较大的ELF格式对象文件。

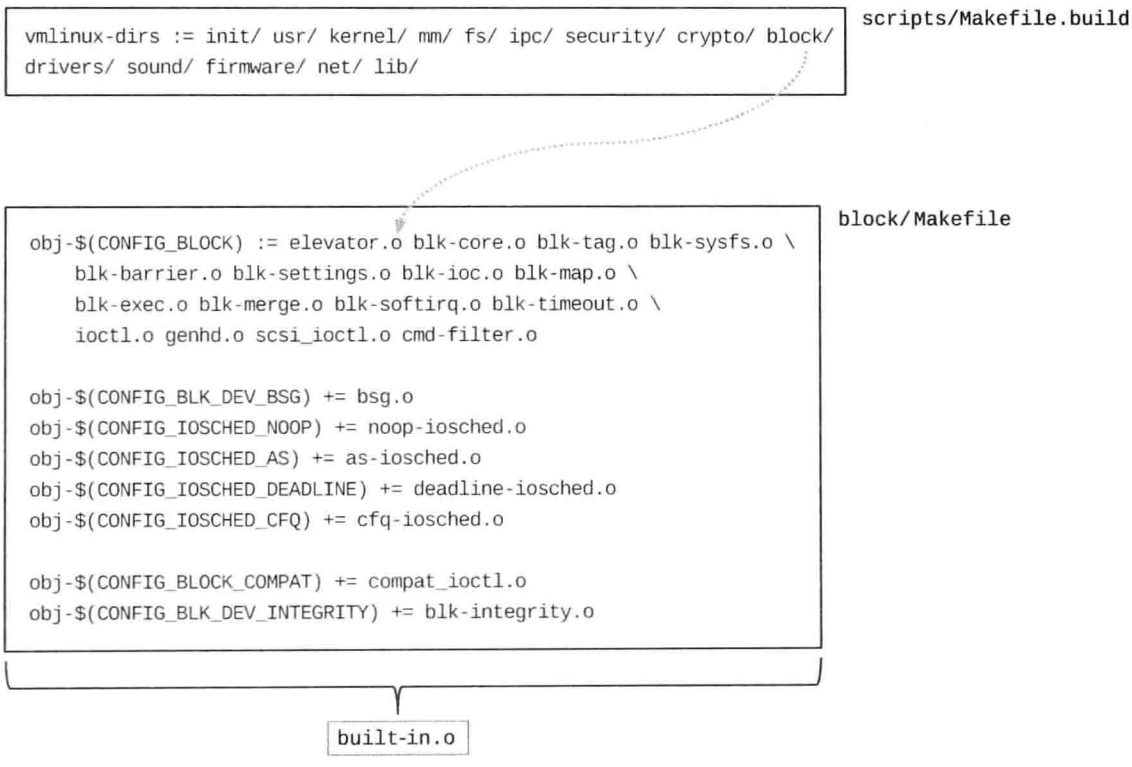


图2-4 built-in.o文件

:: Makefile Makefile的链接部分

```
...
vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all  := $(vmlinux-init) $(vmlinux-main)
vmlinux-lds  := arch/$(SRCARCH)/kernel/vmlinux.lds
export KBUILD_VMLINUX_OBJS := $(vmlinux-all)
...
quiet_cmd_vmlinux__?= LD      $@
    cmd_vmlinux__?= $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
    -T $(vmlinux-lds) $(vmlinux-init)                          \
    --start-group $(vmlinux-main) --end-group                  \
    $(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o FORCE ,$^)
...
head-y            :=      arch/arm/kernel/head.o arch/arm/kernel/init_task.o
init-y            :=      init/
core-y            :=      usr/
libs-y            :=      lib/
drivers-y         :=      drivers/ sound/  firmware/
net-y             :=      net/
...
core-y            +=      kernel/  mm/  fs/  ipc/  security/  crypto/  block/
...
```

上述代码只摘录了内核Makefile中与vmlinux链接相关的代码。即使无法理解Makefile语法也没关系，只要能够理解上面代码中的多种对象文件的链接顺序，并明白由此生成vmlinux即可。通过上述代码可知，按照如下顺序执行链接，生成的vmlinux映像结构如图2-5所示。

```
vmlinux := head.o init.o init/ kernel/  mm/  fs/  ipc/  security/  crypto/  block/ usr/
lib/ drivers/  sound/  firmware/  net/
```

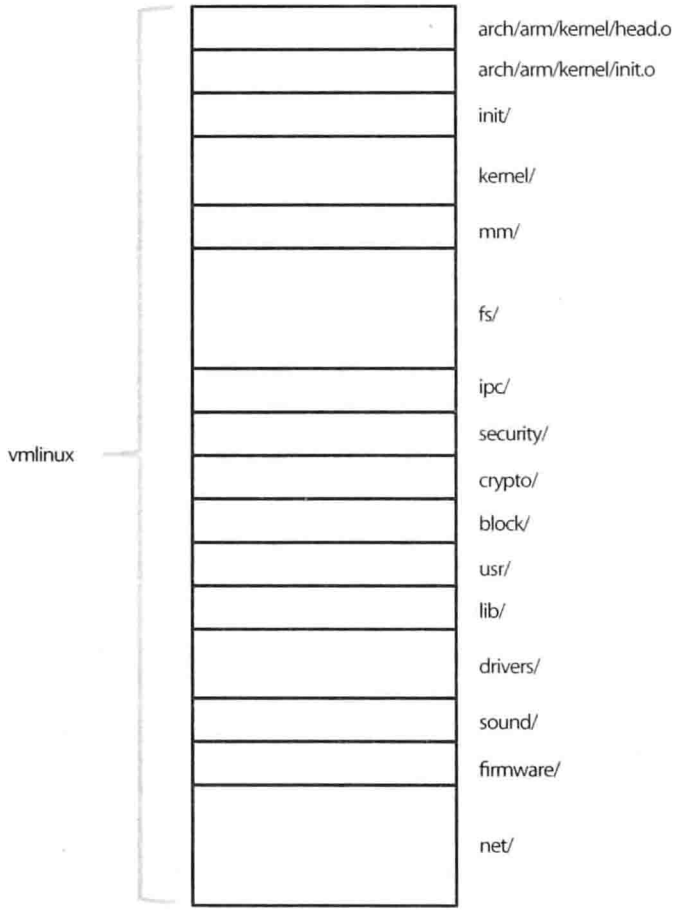


图2-5　vmlinux映像结构

生成的vmlinux是纯内核二进制文件，它具有用户定义的所有内核组件。但仅用这样生成的vmlinux二进制文件无法启动系统。为了将Linux内核映像加载到内存，使其处于可执行状态，需要引导程序加载项（bootstrap loader）的帮助。内核构建系统用objcopy命令清除不必要的节区，压缩ELF格式的vmlinux，然后通过引导程序加载项和链接生成可启动的最终二进制文件zImage。这些过程均由前面讲过的arch/arm/Makefile完成。那么，什么是zImage呢?

通过内核构建最终生成的结果是zImage二进制文件。zImage是已压缩的内核映像，由多个对象代码组成。由于嵌入式系统具有的资源十分有限，为了提高这种环境下内核内存的负载率和执行效率，使用压缩后的内核二进制文件zImage。

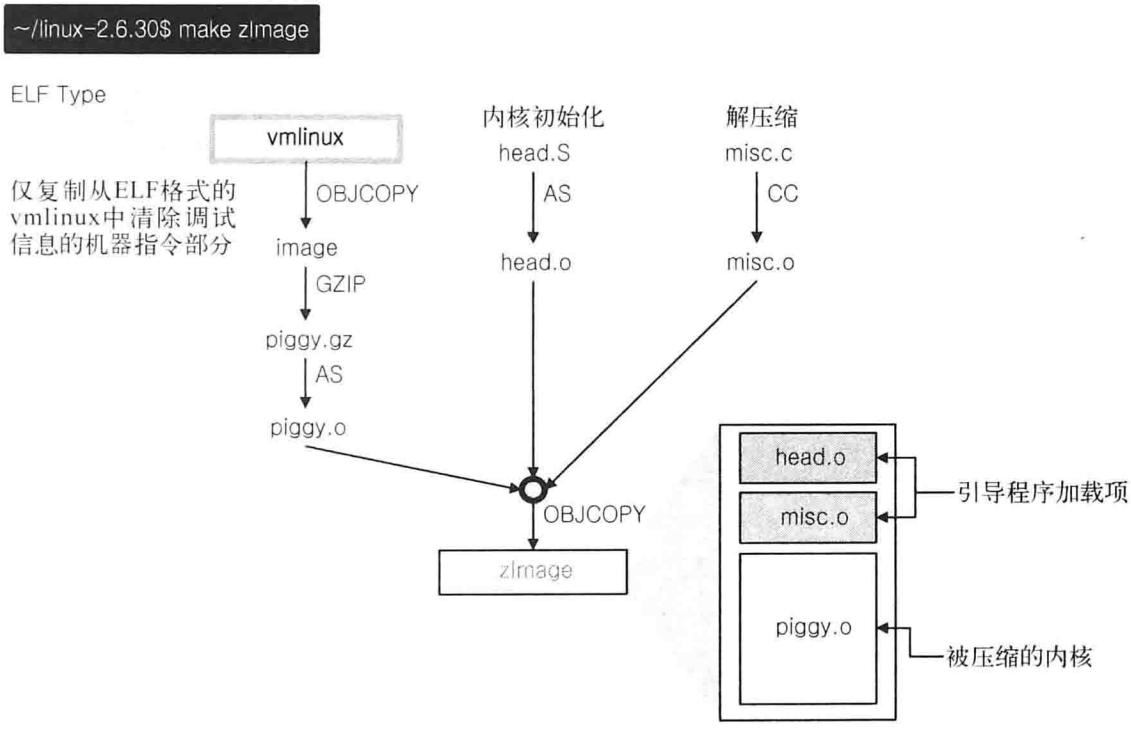


图2-6 生成zImage

为易于理解，图2-6将之前讲解的内核构建过程用图形进行了描述。如前所述，编译后的内核会生成ELF格式的二进制文件vmlinux。这是由Linux源代码树内的所有对象代码组成的纯内核二进制文件，并具有Linux内核的所有要素。vmlinux通过gzip压缩成piggy.o，和head.o.misc.o执行链接，最终生成zImage二进制文件。内核的内存加载及执行都能通过zImage提高速度，特别是head.o和misc.o相当于引导程序加载项，它们是将压缩后的vmlinux（piggy.o）加载到内存并执行解压缩的代码。

2.4 内核安装

我们在前面查看了内核映像zImage的生成过程，最后阶段是将生成的内核映像置于适当位置。

:: Makefile Makefile的链接部分

```
...
install: $(obj)/Image
    $(CONFIG_SHELL) $(srctree)/$(src)/install.sh $(KERNELRELEASE) \
    $(obj)/Image System.map "$(INSTALL_PATH)"

zinstall: $(obj)/zImage
    $(CONFIG_SHELL) $(srctree)/$(src)/install.sh $(KERNELRELEASE) \
    $(obj)/zImage System.map "$(INSTALL_PATH)"
```

```
zi:
    $(CONFIG_SHELL) $(srctree)/$(src)/install.sh $(KERNELRELEASE) \
    $(obj)/zImage System.map "$(INSTALL_PATH)"

i:
    $(CONFIG_SHELL) $(srctree)/$(src)/install.sh $(KERNELRELEASE) \
    $(obj)/Image System.map "$(INSTALL_PATH)"
...
```

内核配置阶段通过install.sh将生成的内核镜像放置在与$(INSTALL_PATH)对应的路径上。

```
export INSTALL_PATH ?= /boot
```

$(INSTALL_PATH)变量在顶端目录的Makefile内，定义如上。

第3章

了解ARM处理器

本书讲解ARM处理器上的Linux内核代码，因此，要理解代码相关说明，必须具备ARM处理器的一些背景知识。本章主要讲解学习本书所需的ARM处理器的基本内容，想对ARM处理器进行深入了解的读者请另行参考其他ARM专用书籍。

3.1 处理器概要和特征

1985年，Acorn公司首次对ARM处理器进行了商业化运作。当时采用公司名称，以Acorn RISC Machine之意将其称为ARM。现在使用的ARM的意思是Advanced RISC Machine。目前由ARM公司设计处理器，但ARM公司只是将处理器的设计提供给半导体公司，并不直接参与半导体芯片的制作及销售。

ARM处理器以Berkeley RISC架构为基础。RISC（Reduced Instruction Set Computer，精简指令系统计算机）比CISC（Complex Instruction Set Computer，复杂指令系统计算机）指令结构更简单，可在一小时内快速处理指令。ARM采用RISC方式降低指令复杂度，利用管道提高指令处理速度，由此提升性能。ARM中适用的RISC特征如下。

- **指令**：ARM使用的指令相对较少。这些指令将提供一个循环周期内能够执行的简单指令。通过具有一定长度的指令可实现管道。
- **管道**：指令在管道中是并列执行的。通过管道执行时，一边解码（decode）当前指令，一边获取（fetch）下一指令。
- **负载存储（load-store）结构**：ARM处理器会执行寄存器储存的指令。将内存中的数据读取到寄存器时使用load命令，将寄存器中的数据输入内存时使用store命令。

3.2 处理器架构与核心

下一节要介绍的ARM核心的基本结构指：ARM架构显示处理器中处理的数据大小、指令结构、寄存器等的结构和运行原理。ARM的主要架构如下。

- **ARMv4架构**：使用32位地址区域，可运行32位的ISA（Instruction Set Architecture，指令集架构）。ARMv4T架构还具有16位Thumb指令包。

- **ARMv5TE架构**：向ARM ISA添加了已改善的Thumb架构和Enhanced DSP指令包。它将改善ARM/Thumb彼此间的运行，包含了程序兼容，提高了性能。
- **ARMv5TEJ架构**：这是加载Java加速技术的架构，添加了Java硬件加速器Jazelle扩展指令。
- **ARMv6架构**：改善了对内存系统、异常处理、多进程环境的支持等内容，并包含了支持运行SIMD（Single Instruction Multiple Data，单指令多数据）的媒体指令。
- **ARMv7-A架构**：支持Linux、Linux第三方（Montavista、QNX、风河系统）、Symbian、Windows CE等大部分操作系统。具有ARM、Thumb、Thumb-2、Jazelle、DSP指令包，并包含Advanced SIMD扩展指令——NEON的软件多媒体处理功能。

提及ARM处理器时会经常使用“ARM核心”一词。ARM核心是利用ARM架构基本原理实现的处理器核心部分。图3-1表示ARM架构和ARM核心，以及该架构包含的与硬件特性间的关系。

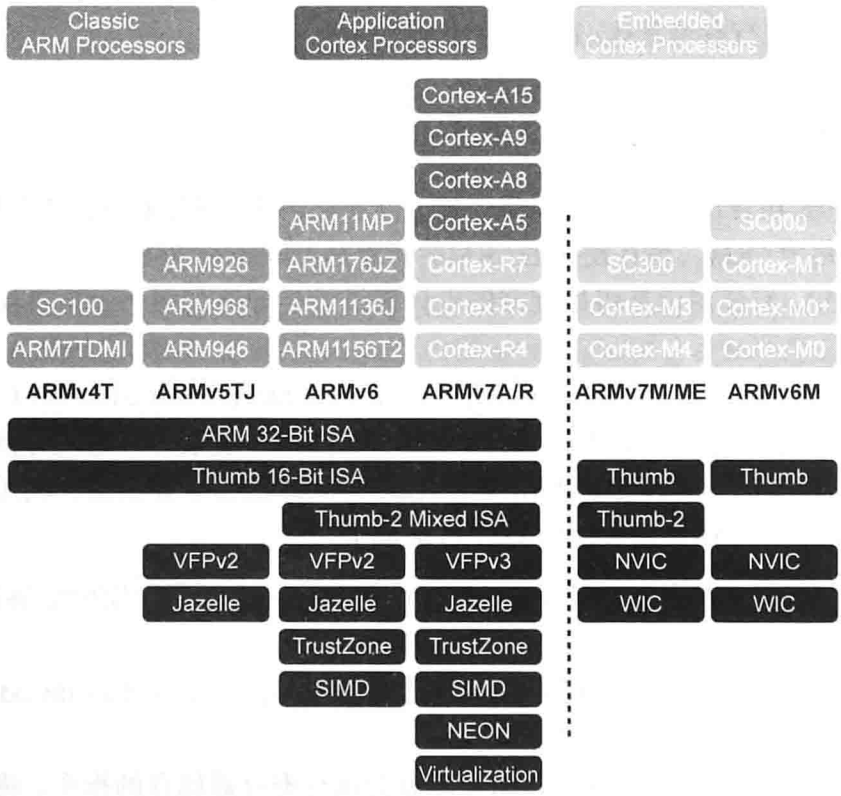


图3-1　ARM架构与ARM核心的关系

※ 参考文献：Processors-ARM（http://www.arm.com/products/processors/）

目前常用的ARM Cortex核心有几个系列，根据如下属性区分。

- **Cortex-A系列**：application专用配置文件
- **Cortex-R系列**：real-time专用配置文件
- **Cortex-M系列**：microcontroller专用配置文件

3.3 处理器命名规则

ARM处理器使用ARM7TDMI、ARM9TDMI、ARM10E、ARM11等名称，其中包含的各符号意义如表3-1所示。

表3-1 ARM处理器命名规则

ARM [X][Y][Z][T][D][M][I][E][J][F][-S]	
符 号	意 义
x	处理器族
y	MMU/MPU
z	缓存
T	Thumb 16位指令解码器
D	JTAG调试器
M	扩展高速乘子
I	嵌入式ICE微极化池（microcell）
E	DSP扩展指令
J	Jazelle Java 加速功能
F	浮点数处理装置
S	综合版本

ARM7TDMI以后推出的所有ARM处理器即使没有具备TDMI符号，也基本包括了TDMI功能。处理器族将具有相同硬件特性的多个处理器统称为一个族（family）。ARM926EJ-S、ARM946E-S、ARM968E-S等均属于ARM9处理器族。

3.4 处理器内部结构

ARM处理器内部结构如图3-2所示，用箭头符号将块和块之间进行连接。块表示能执行特定操作的装置或存储装置，箭头符号为数据总线，表示数据流向。

- **寄存器组（register bank）**：bank是将寄存器根据使用目的进行区分之后的统称。ARM具有12个用户寄存器和25个系统寄存器，共37个。
- **柱式位移器（barrel shifter）**：位移器负责移动进入ALU的一方的输入数据。ARM的柱式位移器移动时所需时间与位数无关，始终保持一致。
- **ALU（Arithmetic Logic Unit）**：执行与指令相应的算数运算及逻辑运算。
- **地址寄存器（address register）**：从内存读取或存入内存时储存数据地址。地址增加器每次增加4个地址值。
- **数据输入输出寄存器**：用于储存内存中读取的数据或将写入内存的数据。
- **指令解码器/控制装置**：负责解码指令并制作管理数据装置的控制信号。

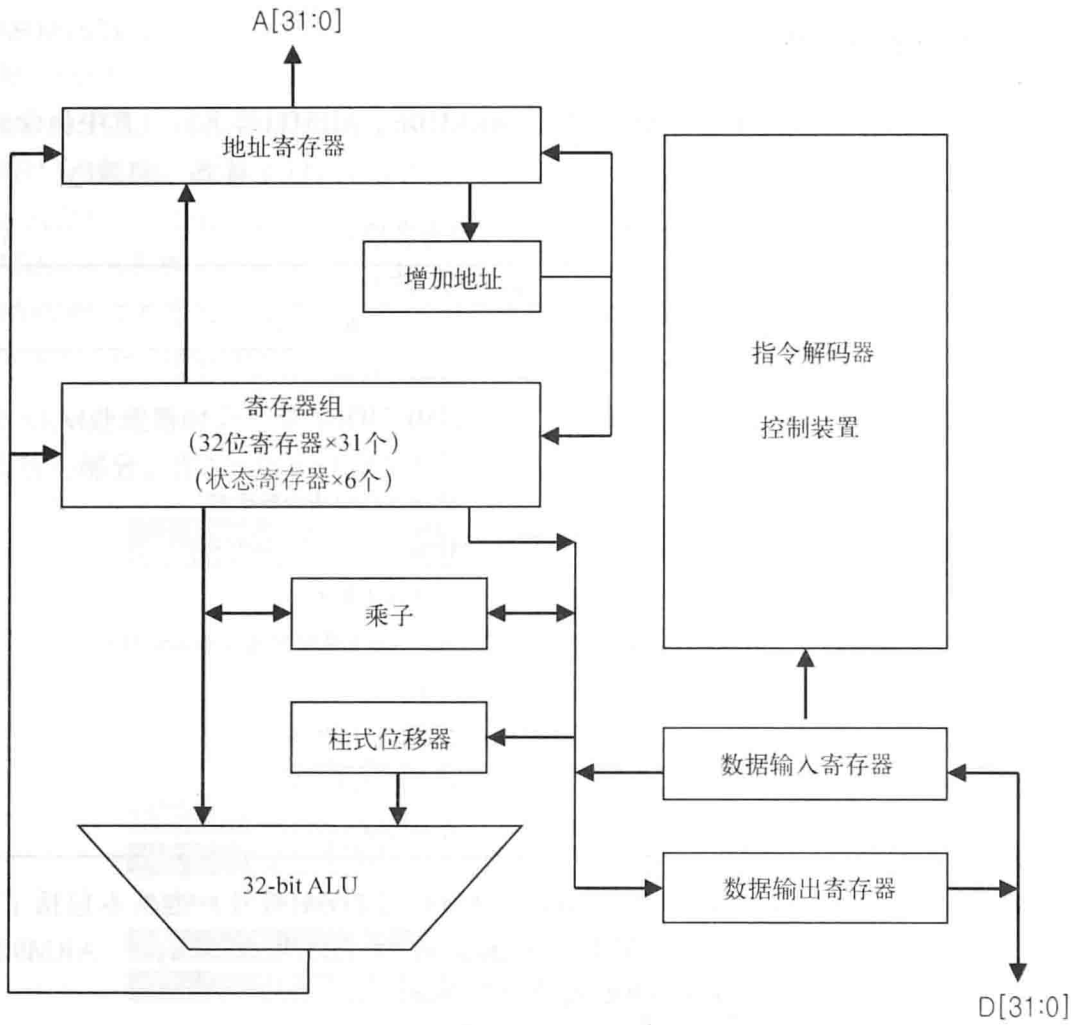


图3-2　ARM处理器内部结构

从图3-2可以看到，执行的指令和数据通过地址总线和数据总线输入核心。“将地址总线和数据总线并用”的形态称为冯·诺依曼体系结构，“将地址总线和数据总线分用”的形态称为哈佛体系结构。在ARM处理器中，ARM7采用冯·诺依曼体系结构，ARM9采用修正后的哈佛体系结构。

通过数据总线从外部存储器输入核心的数据被储存到内部存储空间——寄存器。为了从外部存储器读取数据，或将数据传送到外部存储器，我们使用负载存储指令：load指令用于从外部存储器将数据输入核心内部的寄存器，store指令用于从寄存器将数据传送到存储器。

ARM指令由2个源寄存器和1个结果寄存器组成。图3-2中，连接到ALU（Arithmetic Logic Unit，算术逻辑单元）的两侧总线相当于从2个源寄存器进行的输入。柱式位移器在输入ALU之前执行数据的预处理任务。ALU会读取输入的值，执行运算并输出结果。数据处理指令在寄存器中储存运算结果，负载存储指令读取地址寄存器中的地址，并将其送到地址总线。

3.5 处理器模式和寄存器

ARM处理器具有6个特权模式（System、FIQ、IRQ、Supervisor、Abort、Undefined）和1个普通模式（User），处理器模式显示当前处理器带着何种权限运行。根据模式的不同，被激活的寄存器和可访问寄存器的权限会有所不同。ARM处理器的模式具体如下。

- **User模式**：执行普通用户应用程序时的处理器运行模式。
- **System模式**：使用与User模式相同的寄存器，但可完全读写CPSR（Current Program Status Register，当前程序状态寄存器）的特殊模式。
- **FIQ（Fast Interrupt Request，快速中断请求）模式**：为处理快速中断而执行的处理器运行模式。
- **IRQ（Interrupt Request，中断请求）模式**：为处理通用中断而执行的处理器运行模式。
- **SVC（Supervisor）模式**：操作系统内核运行的通用处理器运行模式，发生Reset或软件中断（SWI）时执行。
- **Abort模式**：内存访问失败时执行的处理器运行模式。
- **Undefined模式**：处理器要执行未定义的指令时执行的处理器运行模式。

ARM处理器使用已定义的指令改变系统状态。处理器将执行指令时使用的数据储存在寄存器中，并通过寄存器执行运算。在特定模式下，1次可激活全部37个寄存器中的18个。这些寄存器由16个数据寄存器和2个程序状态寄存器组成。数据寄存器由r0到r15表示，其中，寄存器r13、r14、r15执行如下特殊功能。

- **寄存器r13**：作为栈指针（sp）使用，存储当前处理器模式的栈顶端地址值。
- **寄存器r14**：作为链接寄存器（lr）使用，储存调用子程序时的返回地址。
- **寄存器r15**：处理器读取指令后，通过程序计数器（pc）储存下一指令地址。

除了16个通用寄存器（r0～r15）之外，还有2个程序状态寄存器：CPSR和SPSR（Saved Program Status Register，程序状态保存寄存器）。如图3-3所示，通用寄存器和CPSR、SPSR根据处理器模式区分可使用的寄存器。根据模式隐藏的各寄存器会被激活，我们通常将其称为分组寄存器（banked register）。

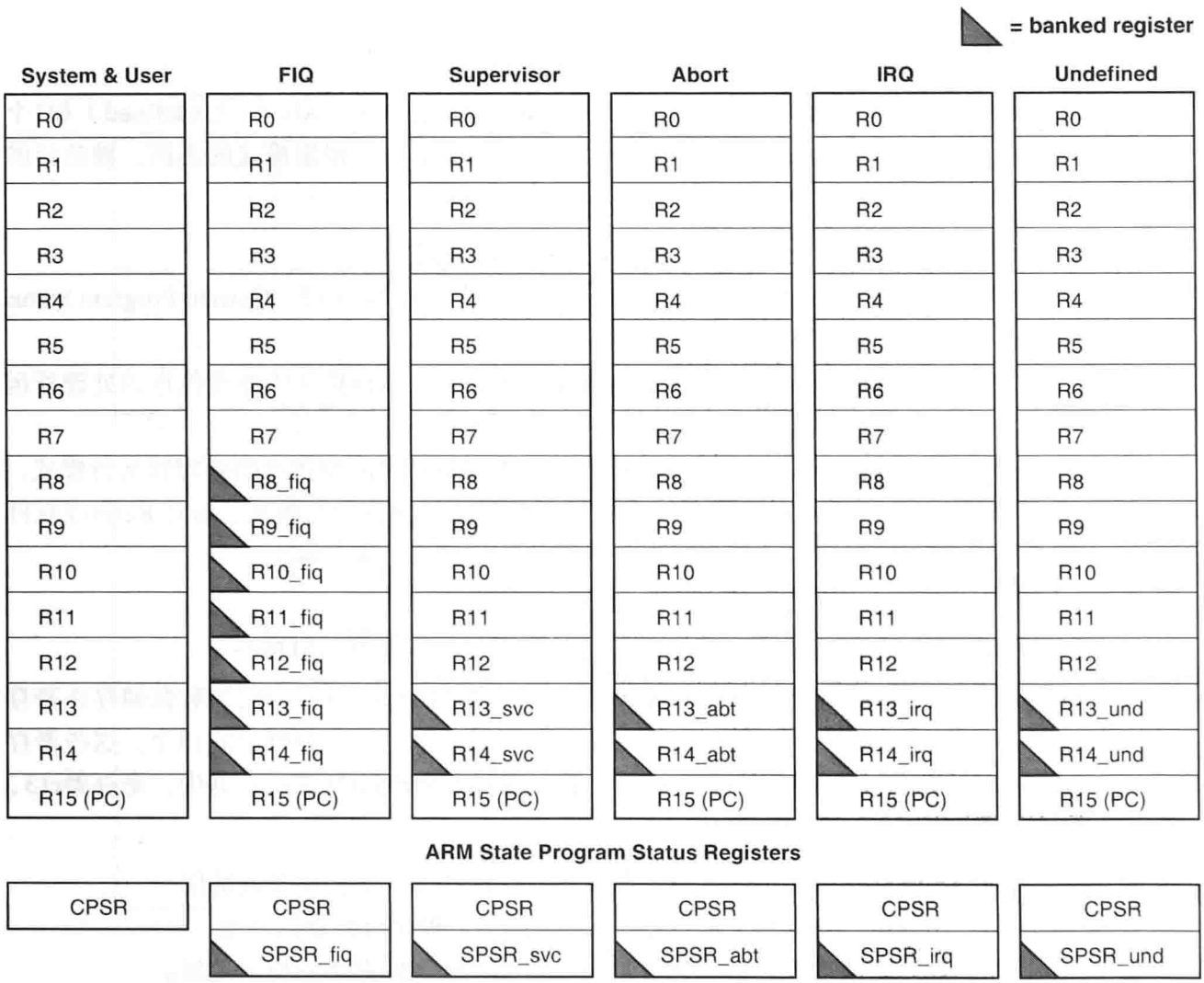
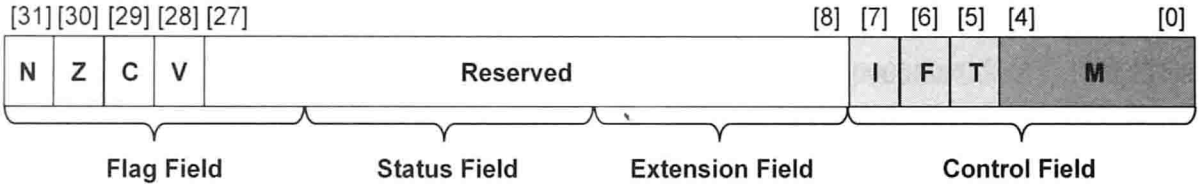


图3-3　各ARM处理器模式与相应寄存器

CPSR显示程序执行时的状态，用于保存条件位。各位将保存算术及逻辑运算结果，从而表示条件分支指令。如图3-4所示，CPSR由标志（Flag）、状态（Status）、扩展（Extension）、控制（Control）组成，各8位。状态域和扩展域是为以后使用而保留的部分。

SPSR用于处理器进入特权模式（System模式除外）时储存CPSR。处理器恢复到用户模式时，可从储存的SPSR恢复之前的CPSR状态。从特权模式再次进入特权模式时，将SPSR复制到通用寄存器并储存。



Condition Code Flags	
N	Negative result from ALU
Z	Zero result from ALU
C	ALU operation caused Carry
V	ALU operation oVerflowed

Control Bits	
I	1:disables IRQ
F	1:disables FIQ
T	1:Thumb, 0: ARM

Mode Bits	
M[4:0]	Mode
10000	User
11111	System
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined

图3-4 ARM CPSR结构

3.6 处理器异常

处理器发生异常或中断时，在程序计数器（pc）中输入特定存储器地址以处理异常/中断。表3-2的向量表中定义了该存储器地址。处理器将中断当前执行的程序，并读取向量表中相应位置的指令。由于各异常/中断表项包含了将其分散到特定程序的指令，故处理器将分散到相应位置。

表3-2 异常/中断向量表

异常/中断	模式	向量地址（Low/High）	优先顺序
Reset	SVC	0x00000000/0xFFFF0000	1
Undefined instruction	UND	0x00000004/0xFFFF0004	6
Software Interrupt(SWI)	SVC	0x00000008/0xFFFF0008	6
Prefetch Abort	Abort	0x0000000C/0xFFFF000C	5
Data Abort	Abort	0x00000010/0xFFFF0010	2
Reserved	–	0x00000014/0xFFFF0014	–
IRQ	IRQ	0x00000018/0xFFFF0018	4
FIQ	FIQ	0x0000001C/0xFFFF001C	3

在处理器中可将向量表设置为高位（high）地址使用，微软操作系统即具备该特征。设置为Low向量表时，分为0x00000000；设置为High向量表时，分为0xFFFF0000。

- Reset：接通电源时首次使用的位置。
- Undefined Instruction：执行未定义指令时使用。
- Software Interrupt：执行SWI指令时调用，并作为系统核心使用。
- Prefetch Abort：发生于无法读取内存指令时。
- Data Abort：发生于无法读写数据内存时。
- IRQ：为中断处理器当前执行流程，由外部硬件使用。
- FIQ：和IRQ类似，但由需要快速响应时间的硬件使用。[①]

3.7　硬件扩展功能

ARM处理器是向ARM核心添加硬件扩展功能的概念。为了提高ARM核心的性能、管理资源、提供附加功能，它添加了缓存、内存管理装置（MMU）、协处理器接口等硬件扩展功能。本节将对其进行介绍。

3.7.1　缓存

缓存是位于核心和主内存之间的高速内存，由核心使用时，能比外部存储器更快读取指令，从而提高性能。缓存由多种方式组成，通常可采取以下两种方式。

- **混合缓存**：将指令和数据存储在同一空间的冯·诺依曼体系结构。
- **单一缓存**：将储存指令的缓存和储存数据的缓存进行分离的哈佛体系结构。

两种结构如图3-5所示。ARM处理器具有的缓存形态是将二者结合后进行修正的哈佛体系结构。

3.7.2　内存管理装置

内存管理装置（MMU）负责管理内存，使各任务能够在各自独立的内存空间中执行。为此，它把虚拟地址变更为物理地址，向各任务提供虚拟内存地址空间，控制相应物理内存空间的访问权限。

3.7.3　协处理器

协处理器（coprocessor）采用与ARM处理器相互作用的方式处理任务，负责扩展指令包或提供可设置值的寄存器，以扩展ARM核心的功能。例如，协处理器15（CP15）是为了控制缓存或

① Linux中不使用FIQ。

MMU而分配的。

ARM具有与协处理器协同运行的指令，这些指令用于控制缓存和MMU等内存子系统。本书使用的MRC或MCR等指令从协处理器内部各寄存器向ARM寄存器执行读取或写入。

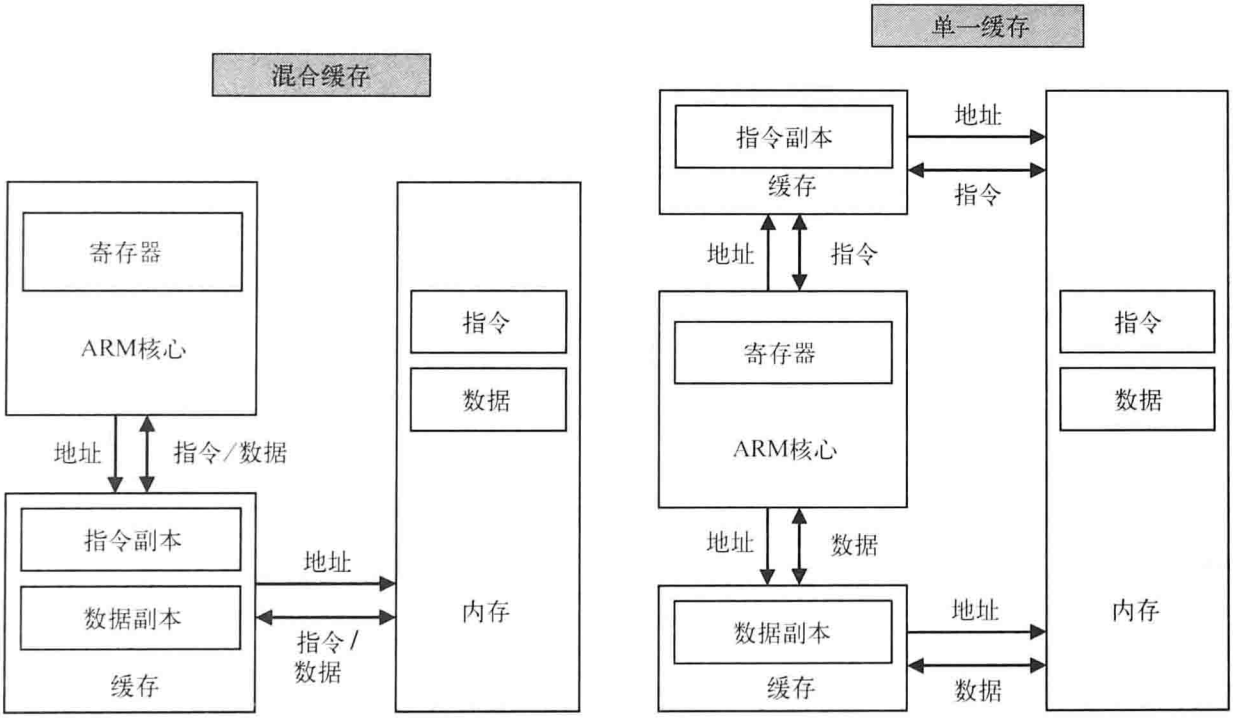


图3-5 混合缓存和单一缓存的结构

※ 参考文献：《ARM SoC体系结构》（Steve Furber著）

第4章

构建分析环境

分析工具是代码分析过程中的重要因素之一。Linux内核由压缩后仍超过70 MB的庞大源文件组成，因此，查找各函数的调用关系时需要打开或跟踪多个文件。如果要用纯编辑器分析这些任务，从个人能力上讲是有局限性的，需要耐力。打开文件并跟踪一段时间后突然迷失了方向，可想而知这有多糟糕。这等于是在熊熊燃烧的代码分析热情上泼一盆冷水！甚至可能当场宣布放弃。也许有人会想起“出色的木匠是不会埋怨工具的”这句俗话，把问题归结为自己没有耐性。但是我们必须明白，本应投入到代码分析中的大多数精力却用于寻找文件，这不得不说是一种巨大的浪费。内核分析是一条艰难又漫长的道路。因此，我们需要一种强大的分析工具让精力集中到源代码分析上。

我们在分析时可以使用Source Insight等商用工具。使用什么样的商用工具是个人的自由，但是Linux中有很难说清的“Hungry精神”。使用商用工具分析开放源码的Linux内核时，就像穿休闲服配皮鞋一样，看起来十分别扭。因此，本章构建一个完全只由开放源码构成的强大的Linux内核代码分析环境。学习如何仅用开放源码构建一个像Source Insight等工具一样出色的商用分析工具。

本章先学习如何下载需要分析的代码、如何构建vim+plugin、如何构建ctags+cscope数据库以分析内核。分析环境的构建以Ubuntu Linux发行版为准进行讲解，其标准编辑器则使用vim。[①]

4.1 下载并安装 Linux 源内核

4.1.1 下载源内核

首先下载要分析的对象——Linux内核。进入www.kernel.org下载代码，出现如图4-1所示画面。

本书分析时使用的内核版本为2.6.30.4。为了与书中讲解的内容保持一致，我们下载2.6.30.4版构建分析环境。[②]

① 其他Linux发行版用自带包管理器下载。

② 虽然可以下载最新版本的内核，但为了与书中内容保持一致，我们推荐下载2.6.30.4版构建分析环境。完成一次分析后，将2.6.30.4版和最新版进行比较，有助于提高学习效率。

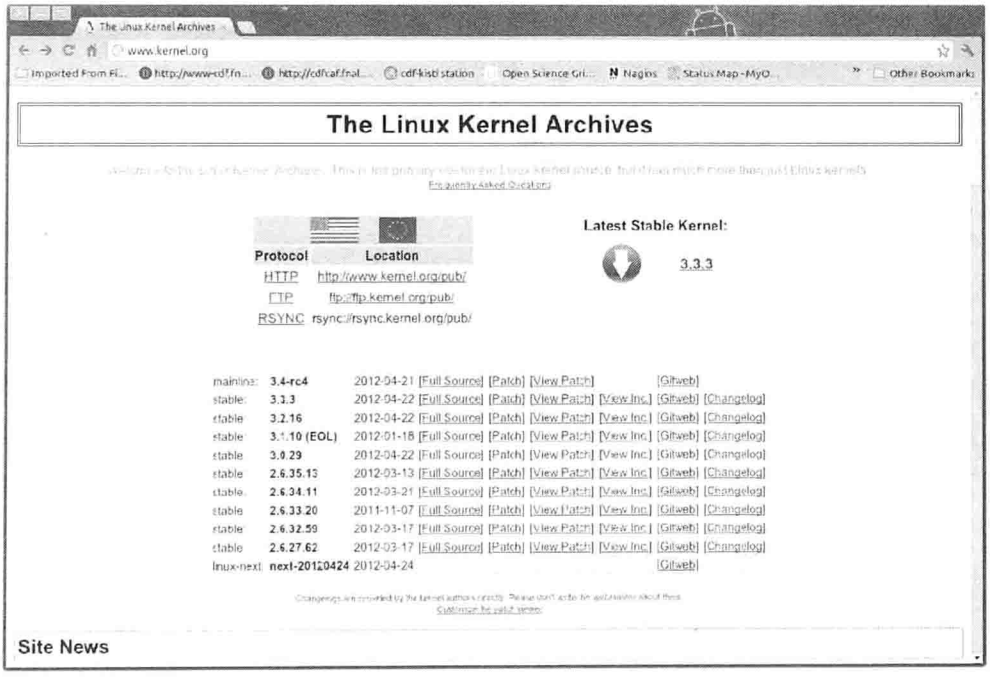


图4-1 www.kernel.org

在网页www.kernel.org中点击http://www.kernel.org/pub链接进入页面，将目录项按照linux→kernel→v2.6移动，出现如图4-2所示画面（也可直接跳转到http://www.kernel.org/pub/linux/kernel/v2.6/）。

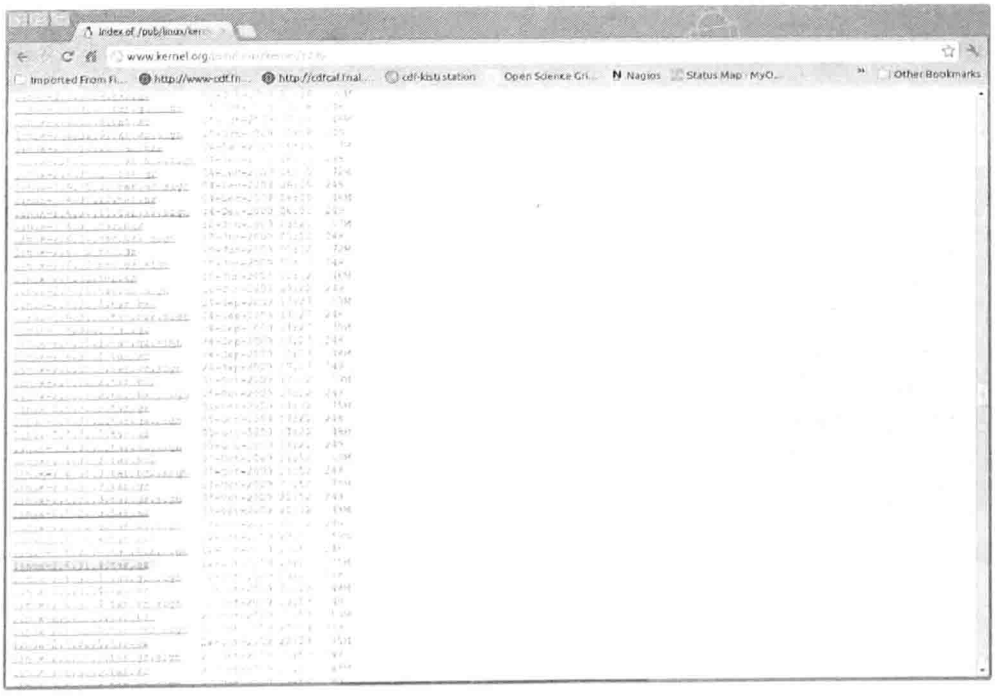


图4-2 下载源内核

点击代码文件linux-2.6.30.4.tar.gz链接进行下载。[①]

4.1.2　安装源内核

创建目录./src，将下载的Linux内核移动到此处进行分析。

```
code@kernel:~$ mkdir src
code@kernel:~$
code@kernel:~$ cp ~Download/linux-2.6.30.4.tar.gz ./src
code@kernel:~$ cd src/
code@kernel:~/src$
code@kernel:~/src$ ls
linux-2.6.30.4.tar.gz
code@kernel:~/src$
```

然后用tar命令解压。

```
code@kernel:~/src$ tar xvzf linux-2.6.30.4.tar.gz
linux-2.6.30.4/
linux-2.6.30.4/.gitignore
linux-2.6.30.4/.mailmap
linux-2.6.30.4/COPYING
linux-2.6.30.4/CREDITS
linux-2.6.30.4/Documentation/

        ...

linux-2.6.30.4/virt/kvm/irq_comm.c
linux-2.6.30.4/virt/kvm/kvm_main.c
linux-2.6.30.4/virt/kvm/kvm_trace.c
code@kernel:~/src$
```

查看内核目录结构，检查是否完成解压。

```
code@kernel:~/src$ ls
linux-2.6.30.4  linux-2.6.30.4.tar.gz
code@kernel:~/src$
code@kernel:~/src$ cd linux-2.6.30.4/
code@kernel:~/src/linux-2.6.30.4$
code@kernel:~/src/linux-2.6.30.4$ ls
arch       crypto         fs       Kbuild       Makefile  REPORTING-BUGS  sound
block      Documentation  include  kernel       mm        samples         usr
COPYING    drivers        init     lib          net       scripts         virt
CREDITS    firmware       ipc      MAINTAINERS  README    security
code@kernel:~/src/linux-2.6.30.4$
```

用tree命令查看生成的目录和文件数量。

① 如果要下载经bzip2压缩的文件，则下载linux-2.6.30.4.tar.bz2。

```
code@kernel:~/src/linux-2.6.30.4$ tree
.
├── arch
|   ├── alpha
|   |   ├── boot
|   |   |   ├── bootloader.lds
|   |   |   ├── bootp.c
                ...

└── virt
    └── kvm
        ├── coalesced_mmio.c
        ├── coalesced_mmio.h
        ├── ioapic.c
        ├── ioapic.h
        ├── iodev.h
        ├── iommu.c
        ├── irq_comm.c
        ├── kvm_main.c
        ├── kvm_trace.c

1738 directories, 27803 files
code@kernel:~/src/linux-2.6.30.4$
```


4.2 安装 ctags+cscope

4.2.1 用ctags制作源代码标签

ctags程序通过程序代码生成标签，可简单将其理解为索引。索引用于生成函数、变量、类成员、宏等的索引，以管理数据库。利用数据库信息在vim或emacs中查找源代码的函数、变量等数据时，索引还可让搜索变得更加容易。[①] ctags支持的程序语言包括C、C++、汇编语言等，共41个。

若想使用ctags，需要下载相关程序。通过Ubuntu包管理器下载，如下所示。

```
code@kernel:~/src/linux-2.6.30.4$
code@kernel:~/src/linux-2.6.30.4$ sudo apt-get install ctags
Reading package lists... Done
          ...
update-alternatives: using /usr/bin/ctags-exuberant to provide /usr/bin/etags
(etags) in auto mode.
```

通过ctags生成标签的命令是ctags -R。但下载后的Linux内核不仅带有我们要分析的ARM结构，还包括了x86在内的多种结构的代码。如果只用ctags -R命令生成标签，则生成内核中所有结构的源代码标签。这样就需要查找ARM文件，十分麻烦。因此，我们需要在生成标签时将结构限定在ARM中。幸运的是，内核源码提供用于生成各结构的ctags标签的脚本。

按如下操作确认tags.sh文件是否存在。

① 关于ctags的详细内容请参考http://ctags.sourceforge.net/。

```
code@kernel:~/src/linux-2.6.30.4$ ls -al ./scripts/tags.sh
-rwxr-xr-x 1 code code 4119 2009-07-31 07:34 ./scripts/tags.sh
code@kernel:~/src/linux-2.6.30.4$
```

如果tags.sh文件存在，用make命令如下图所示生成ARM标签。

```
code@kernel:~/src/linux-2.6.30.4$ make tags ARCH=arm
  GEN     tags
code@kernel:~/src/linux-2.6.30.4$
de@kernel:~/src/linux-2.6.30.4$ ls -alh tags
-rw-r--r-- 1 code code 89M 2011-11-30 18:31 tags
code@kernel:~/src/linux-2.6.30.4$
```

可以确认生成了89 MB大小的tags文件。为了确认tags文件是如何组成的，用vim编辑器打开文件，并查看具有start_kernel符号的第934行、856行（请参考图4-3）。



图4-3　tags文件结构：start_kernel的位置

tags文件的结构如下：

```
tags_name<TAB>file_name<TAB>ex_cmd;<TAB>extension_fields
```

各域之间由制表符（tab）分隔，各域的意义如表4-1所示。

表4-1　tags文件域的意义

域　名	说　明
tag_name	符号名
file_name	符号所在文件名
ex_cmd	在文件中查找符号时，使用vim的ex模式。在此模式中搜索范式的正则表达式
extension_fields	符号类型f=普通C函数，c=类，d=已定义的值

4.2.2 制作cscope标签数据库

ctags很强大，它能构建源代码符号的索引，并通过索引查找符号所在文件。但它也存在局限性，其中具有代表性的几条如下：

(1) 只能进行单方向搜索，无法搜索调用的函数；

(2) 无法输出调用该函数的函数；

(3) 无法输出该函数调用的函数。

cscope与ctags十分相似，但能补充ctags的不足。[①]因此，ctags+cscope组合能够兼具ctags的便利，又能够用cscope补充ctags的局限，算得上是Linux内核代码分析的强大工具。

通过Ubuntu包管理器将cscope包设置如下。

```
code@kernel:~$ sudo apt-get install cscope
Reading package lists... Done

       ...

Unpacking cscope (from .../cscope_15.7a-3.1_amd64.deb) ...
Processing triggers for man-db ...
Setting up cscope (15.7a-3.1) ...
code@kernel:~$
```

将分析文件的目录保存在cscope.files文件后，用cscope命令即可生成数据库。但Linux内核代码中的内置脚本可自动向./scripts/tags.sh文件构建数据库。因此，用make命令按如下操作就能轻松创建ARM的cscope数据库。

```
code@kernel:~/src/linux-2.6.30.4$ make cscope ARCH=arm
  GEN     cscope
code@kernel:~/src/linux-2.6.30.4$
code@kernel:~/src/linux-2.6.30.4$ ls -alh cscope.*
-rw-r--r-- 1 code code 449K 2011-12-01 20:05 cscope.files
-rw-r--r-- 1 code code 178M 2011-12-01 20:06 cscope.out
-rw-r--r-- 1 code code  34M 2011-12-01 20:06 cscope.out.in
-rw-r--r-- 1 code code 258M 2011-12-01 20:06 cscope.out.po
code@kernel:~/src/linux-2.6.30.4$
```

这样会生成4个文件，文件内容如表4-2所示。

表4-2 cscope的文件

文 件 名	说 明
Cscope.files	含有要分析的代码文件列表的文件
cscope.out	针对函数位置、函数调用、宏、变量、预处理符号的符号交叉引用文件，是实际的数据库
cscope.in	用-q选项生成数据库时生成的文件，用于提高符号搜索速度的反向索引
cscope.po.out	用-q选项生成数据库时生成的文件，用于提高符号搜索速度的反向索引。生成cscope.in、cscope.po.out这2个文件

① 关于cscope的详细内容请参考http://cscope.sourceforge.net/。

4.3　vim插件下载及环境设置

我们已经创建了分析内核源代码的符号数据库。向vim编辑器直接输入ctags命令和cscope命令，就能构建可分析代码的环境。但如果不熟悉这2条指令，那距完全掌握还差一步。为了弥补这些不足，我们需要构建更加直观的环境。如果将开放源码vim和plugin进行组合，与ctags和cscope联动，就可以构建更漂亮的个性化环境分析工具。

4.3.1　下载vim插件

构建分析环境需要Source Explorer、NERD Tree、Tag List插件。vim插件可进入www.vim.org下载（请参考图4-4）。

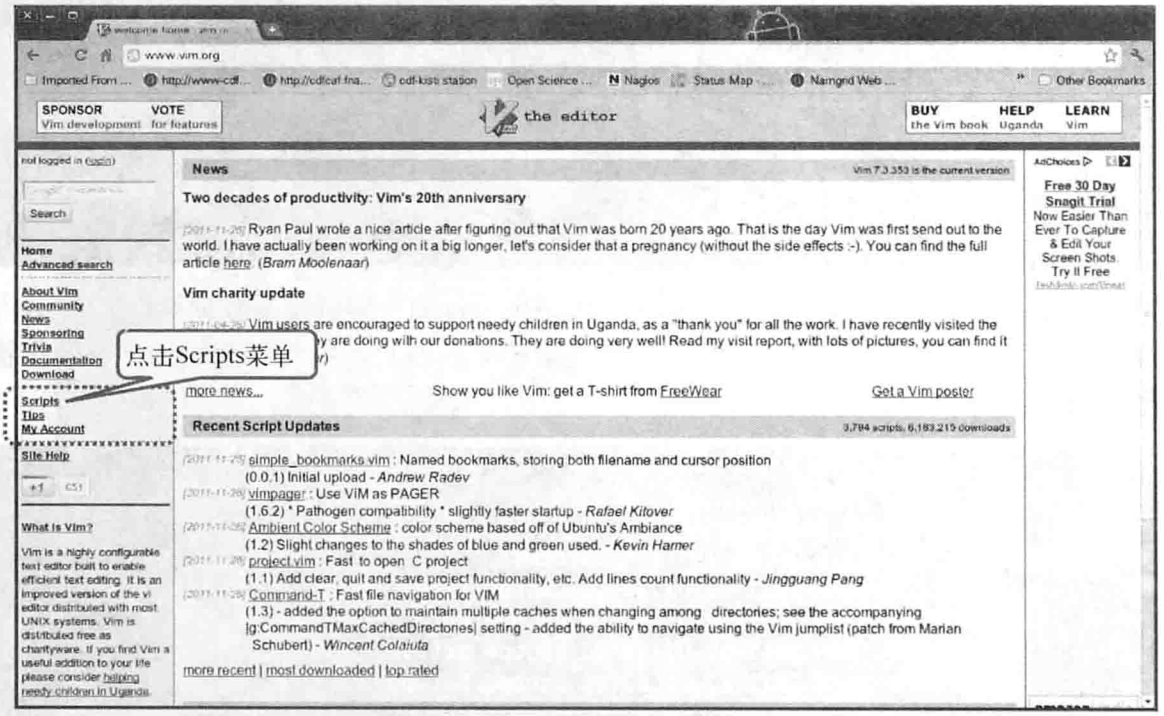


图4-4　www.vim.org主页面

在图4-4左侧链接菜单中点击Scripts，显示如图4-5所示的菜单列表，点击其中的Browse all。

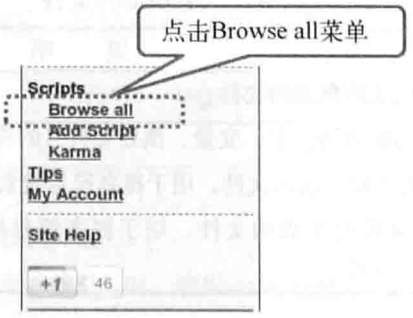


图4-5　选择Browse all链接

选择Bowse all后，在下方显示可搜索插件的窗口，如图4-6所示。

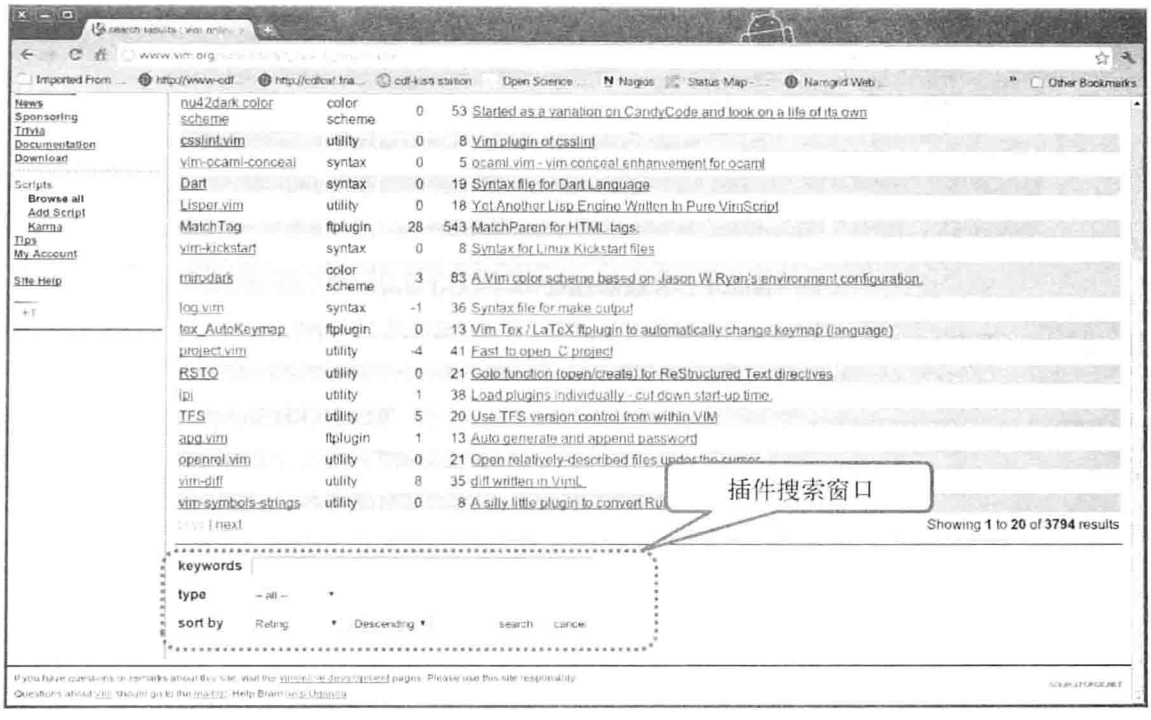


图4-6 插件搜索窗口

如图4-7所示，先输入3个中的第一个插件Source Explorer进行搜索。

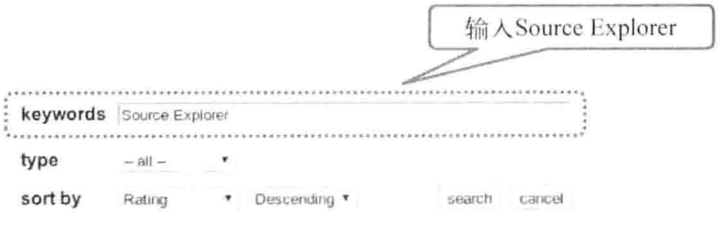


图4-7 搜索Source Explorer

显示的搜索结果如图4-8所示。在搜索结果中选择Source Explorer（srcexpl.vim）链接。

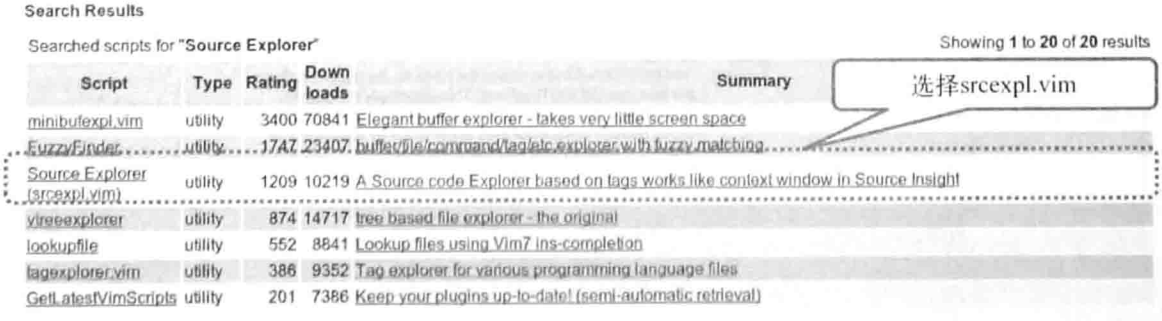


图4-8 选择Source Explorer（srcexpl.vim）链接

选择Source Explorer（srcexpl.vim）后显示可下载插件的页面，这里包含了对插件安装及使用方法的详细说明。

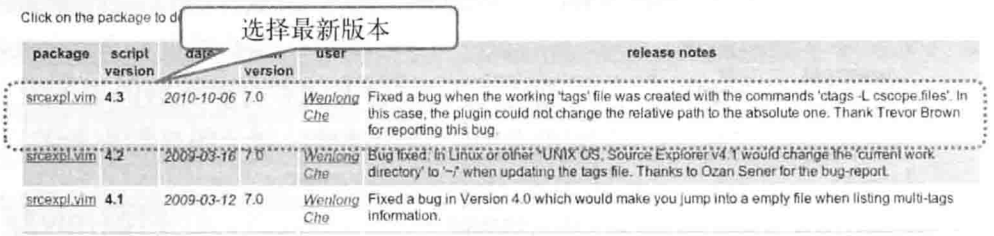


图4-9　下载最新版的srcexpl.vim

用相同方法下载NERD Tree插件。如图4-10所示，在搜索窗口中输入NERD Tree。

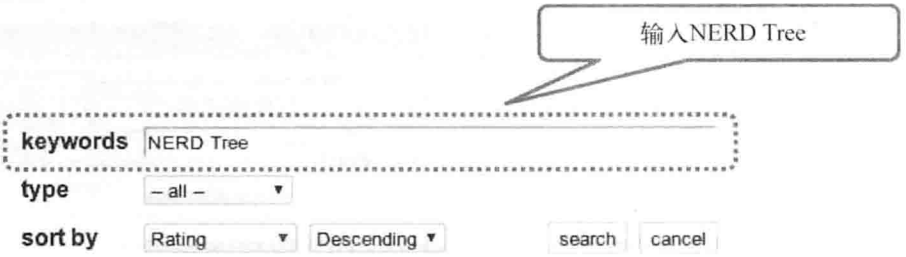


图4-10　搜索NERD Tree

显示的NERD Tree的搜索结果类似于图4-11，选择The NERD Tree。

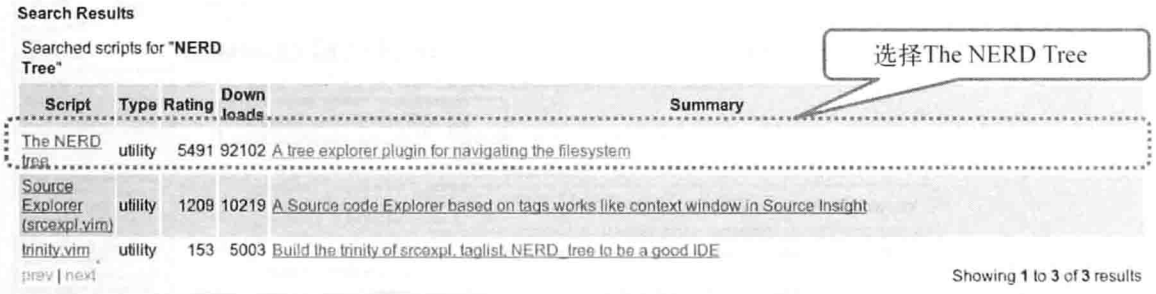


图4-11　NERD Tree的搜索结果

如图4-12所示，下载NERD Tree的最新版本。

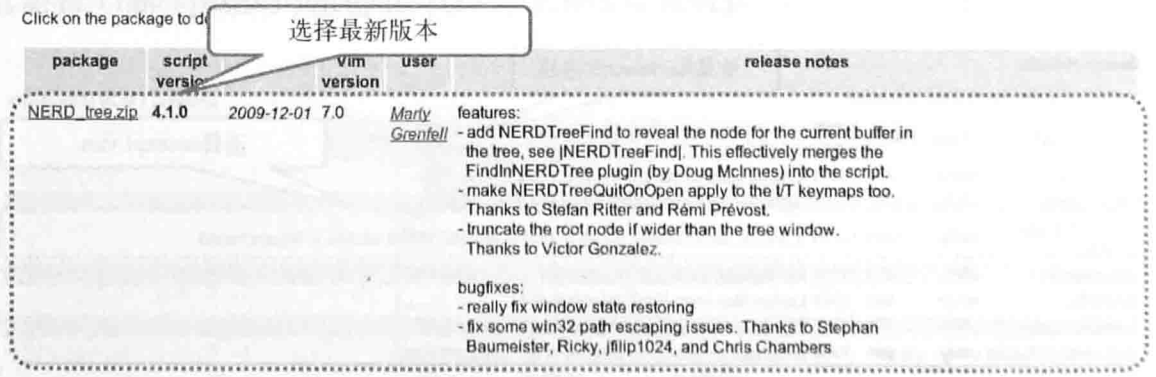


图4-12　下载最新版的NERD Tree

最后下载插件Tag List。用相同方法将Tag List作为关键字搜索插件，如图4-13所示。

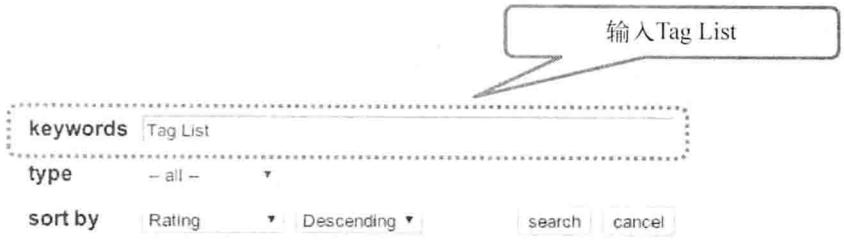


图4-13 搜索Tag List

图4-14显示Tag List的搜索结果，点击taglist.vim链接转到下载页面。

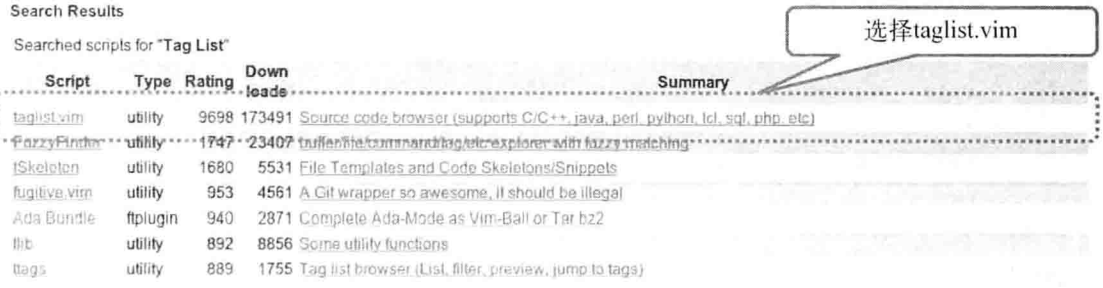


图4-14 Tag List的搜索结果

和别的插件一样，taglist.vim也下载最新版本（请参考图4-15）。

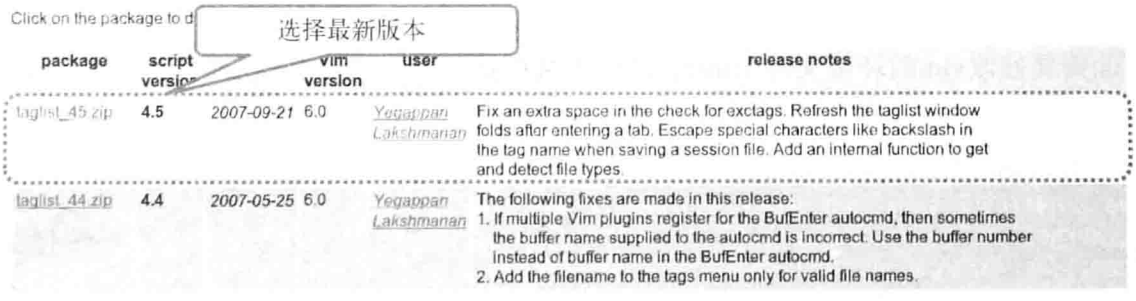


图4-15 下载最新版的Tag List

4.3.2 vim+plugin的环境结构

安装已下载的插件并设置vim环境。vim的插件要位于.vim/plugin目录。生成.vim目录后，将下载的各插件复制到.vim目录。

```
code@kernel:~$ mkdir .vim
code@kernel:~$ cp Download/srcexpl.vim .vim
code@kernel:~$ cp Download/NERD_tree.zip .vim
code@kernel:~$ cp Download/taglist_45.zip .vim
code@kernel:~$
```

```
code@kernel:~$ cd .vim
code@kernel:~/.vim$ mkdir plugin
code@kernel:~/.vim$ ls
NERD_tree.zip  plugin  srcexpl.vim  taglist_45.zip
code@kernel:~/.vim$
```

由于Source Explorer插件srcexpl.vim不是压缩状态，所以移动到./plugin目录，将剩余插件解压缩后移动到./plugin目录。

```
code@kernel:~/.vim$ mv srcexpl.vim ./plugin/
code@kernel:~/.vim$ unzip NERD_tree.zip
Archive:  NERD_tree.zip
  inflating: doc/NERD_tree.txt
  inflating: nerdtree_plugin/exec_menuitem.vim
  inflating: nerdtree_plugin/fs_menu.vim
  inflating: plugin/NERD_tree.vim
code@kernel:~/.vim$ unzip taglist_45.zip
Archive:  taglist_45.zip
  inflating: plugin/taglist.vim
  inflating: doc/taglist.txt
code@kernel:~/.vim$
```

如下图所示，./plugin目录中有3个插件，可视为插件已完成安装。

```
code@kernel:~/.vim$ ls ./plugin/
NERD_tree.vim  srcexpl.vim  taglist.vim
code@kernel:~/.vim$
```

4.3.3　vim环境设置

下面需要修改vim的环境文件.vimrc，使已安装的插件和ctags、cscope数据库能够联动。所需信息有ctags和cscope数据库文件的位置、ctags和cscope执行文件的位置。如下查看位置。

```
code@kernel:~$ whereis ctags
ctags: /usr/bin/ctags /usr/share/man/man1/ctags.1.gz
code@kernel:~$ whereis cscope
cscope: /usr/bin/cscope /usr/share/man/man1/cscope.1.gz
code@kernel:~$
```

如果ctags、cscope数据库文件和执行程序的位置已经得到确认，vim环境文件就对主目录的.vimrc文件进行如下修改。值得注意的是，需要将/home/code/src/linux-2.6.30.4目录变更为自己电脑中内核代码所在目录，将ctags、cscope执行文件的位置也变更为自己电脑的结果值。

:: vim环境设置.vimrc文件

```
"-------------------------------------------------------------------------"
" vim环境设置
"-------------------------------------------------------------------------"
    set nu                                          "line number
    set ai                                          "auto indent
    set ts=4                                        "tab size
```

```
     set bg=dark                                    "backgroud color

"-------------------------------------------------------------------------------"
" ctags database path设置
"-------------------------------------------------------------------------------"
     set tags=/home/code/src/linux-2.6.30.4/tags    "ctags DB位置<home/code>
                                                    "变为自己的路径

"-------------------------------------------------------------------------------"
" cscope database path设置
"-------------------------------------------------------------------------------"
     set csprg=/usr/bin/cscope                      "cscope位置
     set csto=0                                     "cscope DB search first
     set cst                                        "cscope DB tag DB search
     set nocsverb                                   "verbose off

     " cscope DB位置设置、<home/code>变更、使用绝对路径
     cs add /home/code/src/linux-2.6.30.4/cscope.out        /home/code/src/inux-
2.6.30.4
     set csverb                                     "verbose off

"-------------------------------------------------------------------------------"
" Tap List环境设置
"-------------------------------------------------------------------------------"
     filetype on                                    "vim filetype on
     nmap <F7> :TlistToggle<CR>                     "F7 Key = Tag List Toggling
     let Tlist_Ctags_Cmd = "/usr/bin/ctags"         "ctags程序位置
     let Tlist_Inc_Winwidth = 0                     "window width change off
     let Tlist_Exit_OnlyWindow = 0                  "tag/file完成选择时taglist
                                                    "window close = off
     let Tlist_Auto_Open = 0                        "vim开始时window open=off
     let Tlist_Use_Right_Window = 1                 "vim开始时window open=off

"-------------------------------------------------------------------------------"
" Source Explorer环境设置
"-------------------------------------------------------------------------------"
     nmap <F8> :SrcExplToggle<CR>                   "F8 Key = SrcExpl Toggling
     nmap <C-H> <C-W>h                              "向左侧窗口移动
     nmap <C-J> <C-W>j                              "向下端 (preview) 窗口移动
     nmap <C-K> <C-W>k                              "向上端窗口移动
     nmap <C-L> <C-W>l                              "向右侧窗口移动

     let g:SrcExpl_winHeight = 8                    "指定SrcExpl Windows高度
     let g:SrcExpl_refreshTime = 100                "refreshing time = 100ms
     let g:SrcExpl_jumpKey = "<ENTER>"              "跳转 (jump) 至相应定义definition
     let g:SrcExpl_gobackKey = "<SPACE>"            "back
     let g:SrcExpl_isUpdateTags = 0                 "tag file update = off

"-------------------------------------------------------------------------------"
" NERD Tree环境设置
"-------------------------------------------------------------------------------"
     let NERDTreeWinPos = "left"                    "NERD Tree位置=左侧
     nmap <F9> :NERDTreeToggle<CR>                  "F9 Key = NERD Tree Toggling
```

4.4　查看源码分析环境工具

完成所有操作后，在内核源代码所在目录中执行vim。

```
code@kernel:~$ cd src/linux-2.6.30.4/
code@kernel:~/src/linux-2.6.30.4$
code@kernel:~/src/linux-2.6.30.4$ vim
code@kernel:~/src/linux-2.6.30.4$
```

运行vim编辑器后，依次输入功能键F7、F8、F9。如图4-16所示，显示Tag List、Source Explorer、NERD Tree分割窗口。①

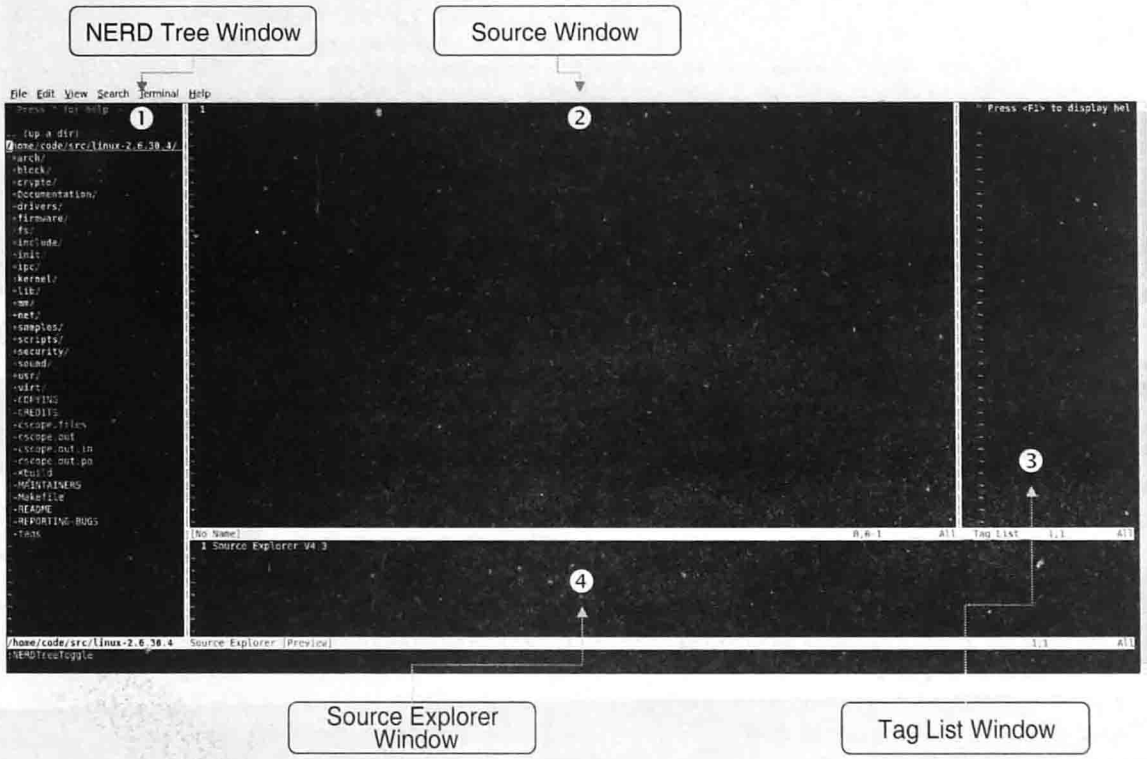


图4-16　运行vim编辑器后显示的分割窗口

在NERD Tree窗口中利用光标移动到./init目录后按Enter键，则显示./init目录所在文件列表。将光标移动到./init/main.c文件，按Enter键打开main.c文件。图4-17就是打开./init/main.c文件的画面。

① 对Tag List、Source Explorer、NERD Tree的键映射已经分别由F7、F8、F9指向.vimrc文件，但也可将其变为自己喜欢的键映射。

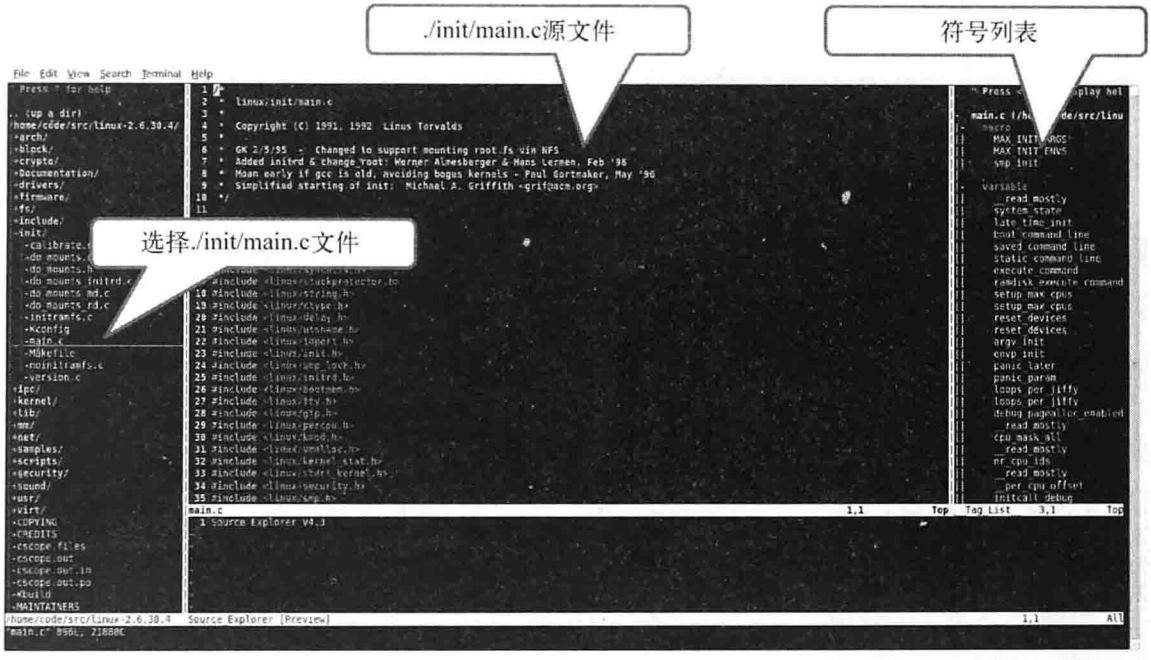


图4-17 打开./init/main.c文件

利用Ctrl+h（移动到左侧窗口）、Ctrl+j（移动到下端窗口）、Ctrl+k（移动到上端窗口）、Ctrl+l（移动到右侧窗口）键移动到各个窗口。如果各窗口的移动不是很顺利，那就需要确认.vimrc文件的Source Explorer节区的键映射是否正确。

将光标移动到Tag List窗口后，查找start_kernel符号。进入vim的ex模式（按Esc键）后，在下端窗口输入:/start_kernel，Tag List窗口的光标会移动到start_kernel符号。若按Enter键，则在代码文件窗口显示定义start_kernel的位置，如图4-18所示。

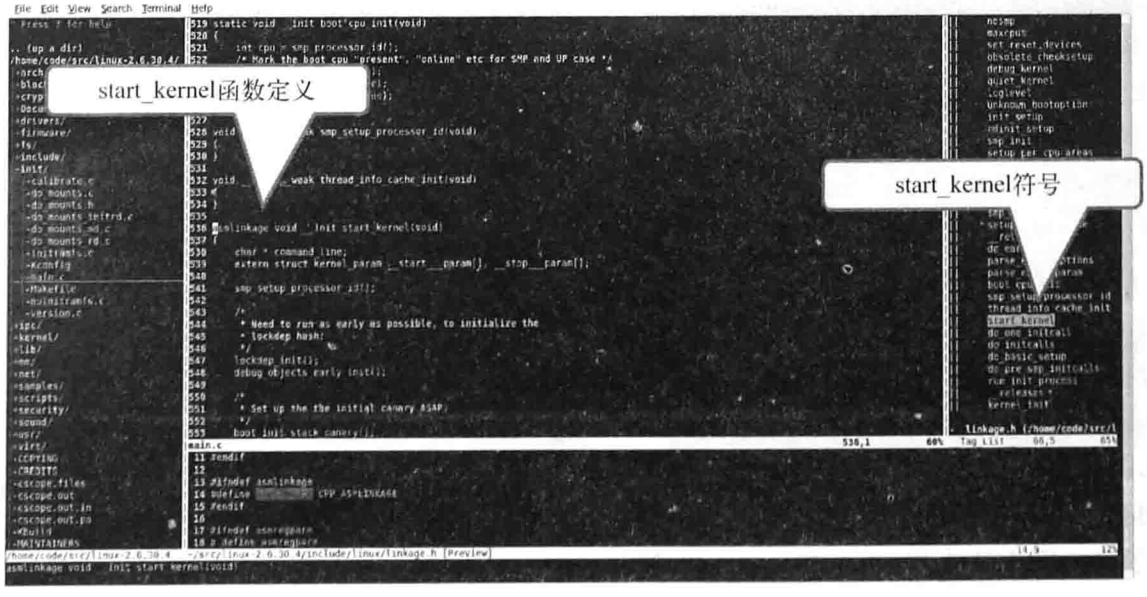


图4-18 移动到start_kernel函数

在代码中移动光标并放在函数或变量上时，下方的Source Explorer窗口显示定义相应符号的位置。例如：如图4-19所示，将光标移动到start_kernel后，Source Explorer窗口显示start_kernel符号定义的文件。



图4-19　定义start_kernel函数的文件列表

移动到下端窗口（Ctrl+j）后，选择列表中的1项（Enter键），在代码窗口显示该文件。例如，在下端的Source Explorer窗口选择第2个文件，可看到如图4-20所示画面。画面中的源代码表示./include/linux/start_kernel.h文件中声明的start_kernel函数。

像这样将光标移动到要找的符号，则Source Explorer窗口就显示与符号相关的文件列表；若移动到Source Explorer窗口并选择该文件，则代码窗口显示源代码。

我们一直用前进（forwarding）方式朝一个方向查找符号。但是有时需要回到当前状态之前的位置，此时按空格键（space bar）即可回到之前的状态。即在图4-20的状态下按空格键，就回到图4-19的状态。

以上就是使用（与ctags数据库联动的）vim插件构建分析环境的方法。这种分析环境对内核代码分析来讲已经足够了，但对内核进行实际分析时，会发现还有仅用ctags无法完成的部分。由于cscope数据库已完成环境设置，因此先学习cscope的使用方法。

如图4-21所示，在NERD Tree窗口中选择并打开./kernel/kthread.c文件，将光标从第233行的kthreadd()函数移动到第247行的kthread_create_list符号时，Source Explorer窗口显示Definition Not Found信息。即无法查找kthread_create_list符号。

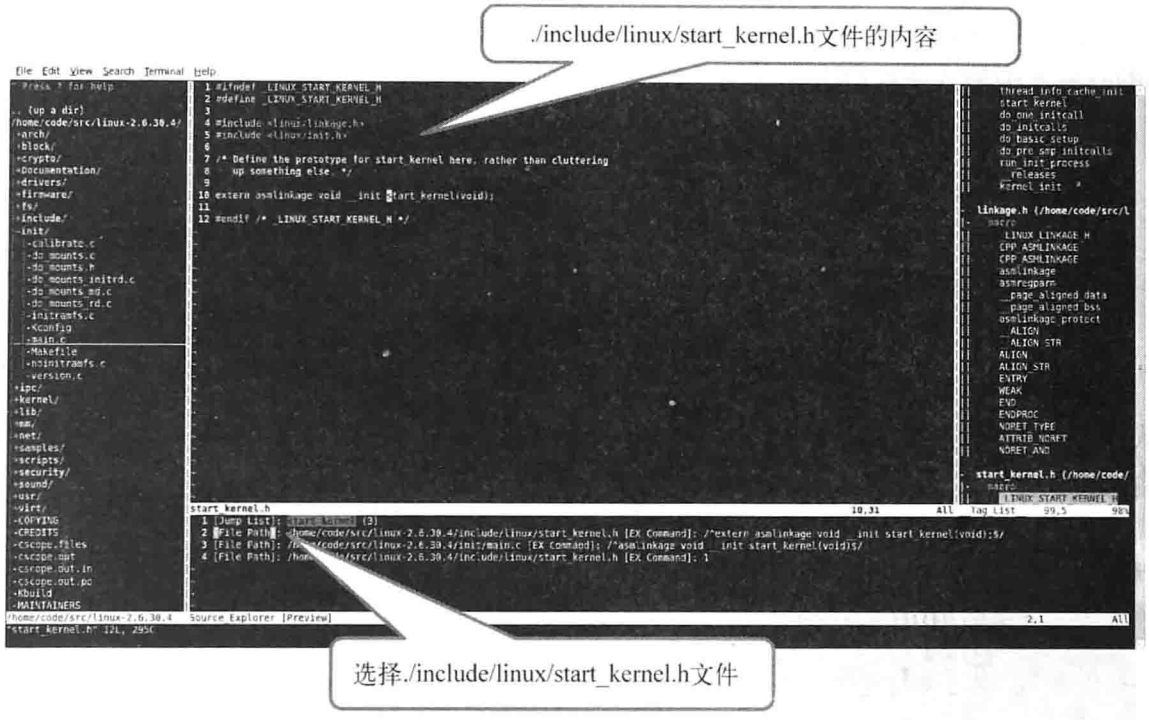


图4-20 在Source Explorer窗口中选择第2个文件（start_kernel.h）

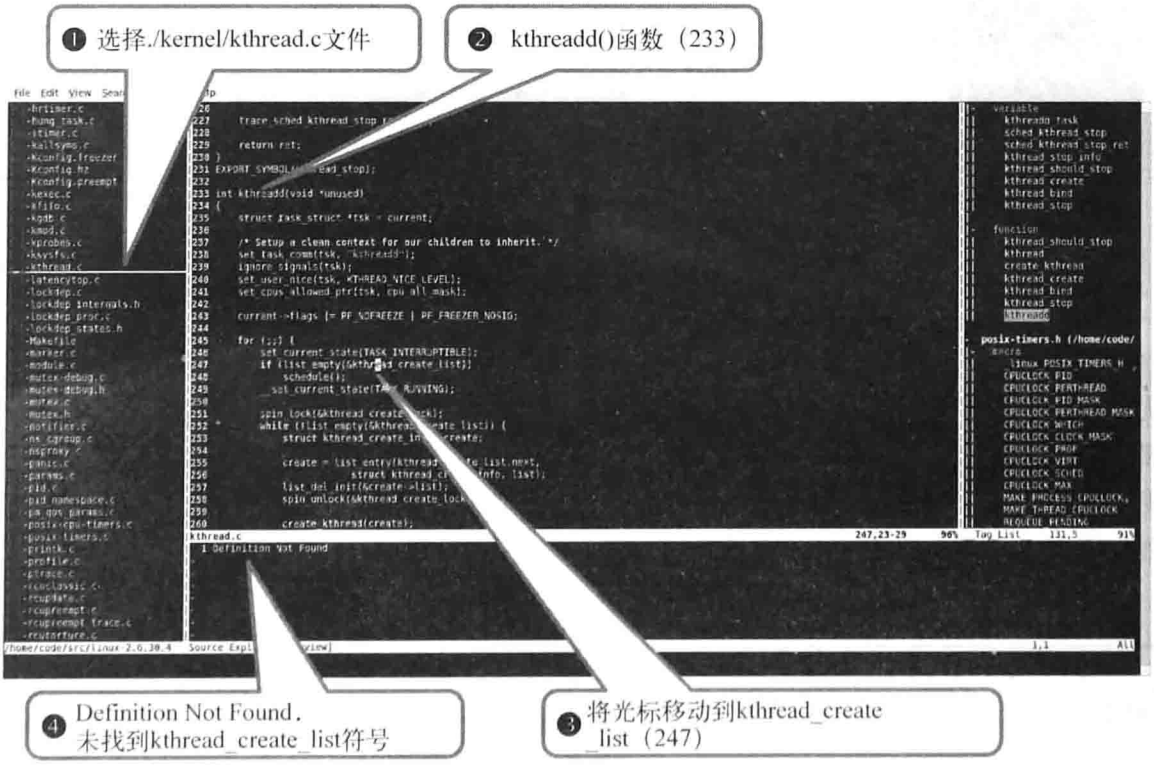


图4-21 无法从ctags数据库查找符号时：显示Definition Not Found消息

如果分析代码时发生这种情况，可通过cscope指令得到与kthread_create_list相关的文件列表。在vim的ex模式中输入图4-22的内容。

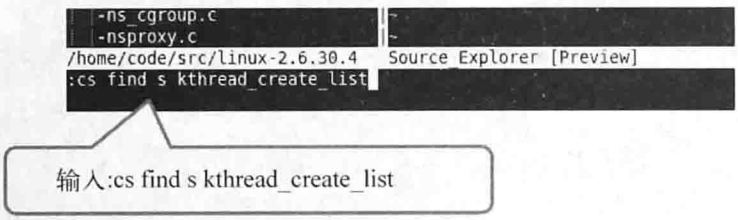


图4-22　通过cscope指令查找kthread_create_list符号

从图4-23的cscope数据库可知，Source Ex-plorer窗口显示搜索到的使用kthread_create_list的文件。输入搜索结果左侧显示的编号，可跳转到相应位置。

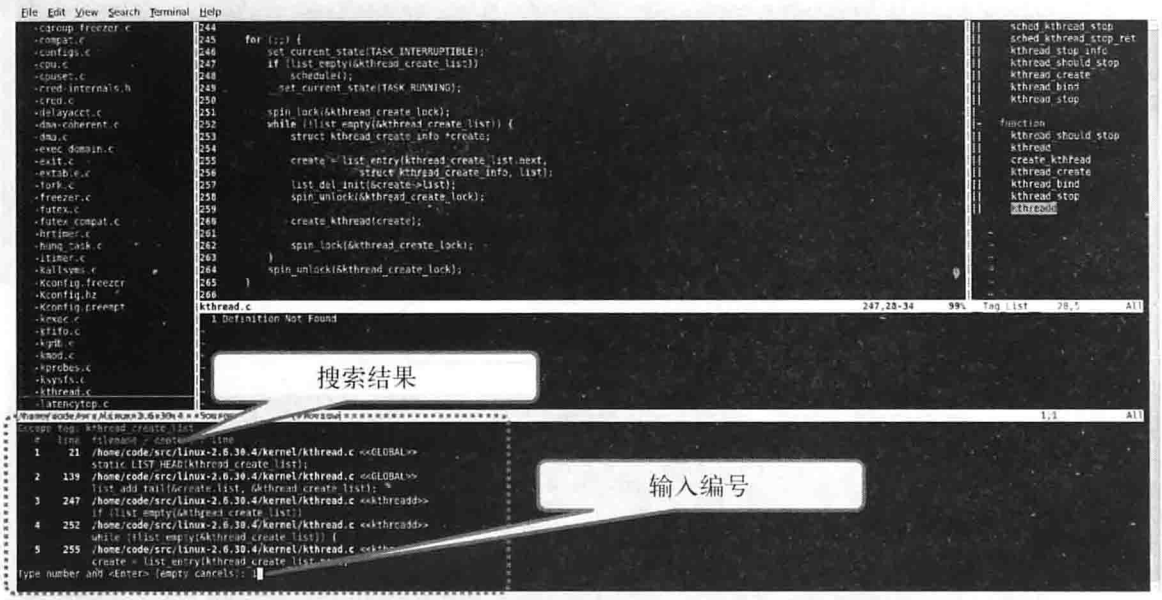


图4-23　用cscope搜索

从cscope数据库可找到kthread_create_list被定义的位置，因此可用cscope弥补ctags中未能搜索到的部分。

我们通过示例了解的ctags和cscope使用方法只利用了较少命令。其实，ctags和cscope提供的搜索命令很多，常用ctags指令如表4-3所示。

表4-3　常用ctags命令

命　令	说　明
Ctrl+]	移动到定义函数的位置
Ctrl+t	移动到移动前的阶段
:Tselect <function-name>	function-name目录列表

（续）

命　令	说　明
:tnext	出现多个搜索结果时移动到下一结果
:tprevious	移动到之前的结果
:tfirst	移动到第一个目录
:tlast	移动到最后的目录

用cscope搜索时，在vim的ex模式下使用如下格式提问。

```
: cs find <querytype> <name>
```

搜索我们曾用作示例的kthread_create_list符号时，在vim的ex模式下向<querytype>输入s、向<name>输入kthread_create_list，然后搜索。

```
: cs find s kthread_create_list
```

cscope提供的多种<querytype>如表4-4所示。

表4-4　cscope <querytype>

querytype	说　明
0 or s	查找C符号
1 or g	查找定义（definition）
2 or d	查找被该函数调用的（called）函数
3 or c	查找调用该函数的（calling）函数
4 or t	查找文本字符串（text string）
6 or e	查找egrep范式
7 or f	查找文件
8 or i	查找用#include包含该文件的文件

如想获得关于cscope的帮助，可在vim中转换为ex模式，然后通过:help cscope查看更详细的指令用法。

第二部分

内核的启动

——strt_kernel调用方法

内核的实际起始函数是 start_kernel() 函数，它再调用 100 多个函数以执行启动。但是，调用 start_kernel() 函数前，必须先将通过编译内核获得的 zImage 进行解压，完成页目录构建等基本任务。我们要通过第二部分了解 start_kernel() 函数调用前的一系列过程。

第二部分操作的所有代码均由 ARM 汇编语言编写而成。PC（Program Counter，程序计数器）从启动加载项移动到 start 标签后，就开始执行基本任务。

调用 start_kernel 的过程大体分为 3 个阶段。第一阶段是解压内核映像 zImage 前的准备阶段，通过与处理器版本相符的处理器类型列表，执行打开 / 关闭 / 清除缓存等任务，为 MMU 构建 16 KB 的页目录。第二阶段对 zImage 执行解压缩。最后的第三阶段检查处理器及机器信息、通过启动加载项获得的 atag 信息的有效性，然后激活 MMU 调用内核的起始函数——start_kernel() 函数。

附图 PART Ⅱ-1 是我们要在第二部分学习的主要的内核工作流（flow）。

第5章 准备解压内核

本章主要学习解压内核映像（zImage）需要经历的各个阶段。解压缩准备阶段将执行中断禁用、分配动态内存、初始化BBS区域、初始化页目录、打开缓存等任务。在该阶段，zImage解压位置的下级16 KB构建用于保存页目录的空间，在CP15的c2寄存器中保存页目录的位置。这是解压缩准备阶段的重点。ARM中，页目录将4 GB的内存以1 MB节区为单位进行管理。因此，为了管理4 GB的内存，需要有4096个以1 MB为单位的项。由于以32位的字符为单位管理各项，所以共需要16 KB（4字节×4096个项 = 16 KB）。之后，向相当于页目录位置的项设置cacheable和bufferable，使页目录得到缓冲并能快速访问。图5-1用图形描述了本章我们需要了解的内核代码的内部流程。

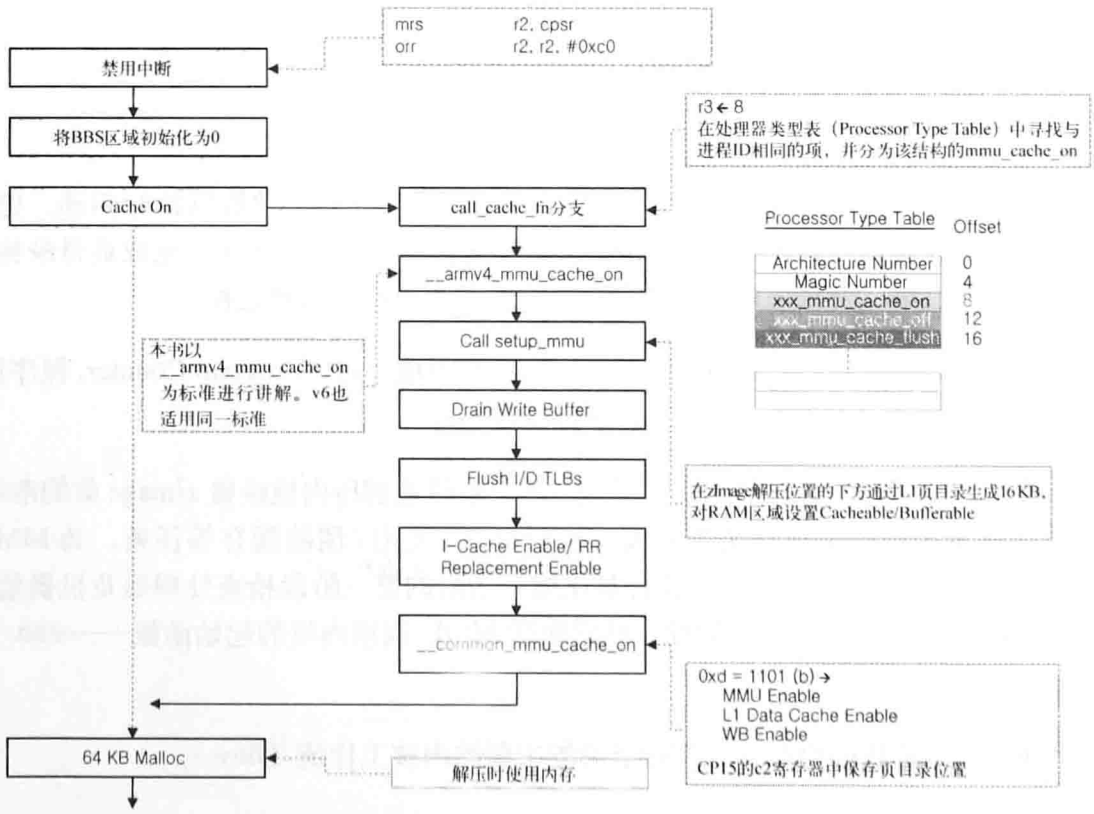


图5-1 从start标签到解压缩准备阶段的流程图

5.1 进入启动加载后结束首个启动——start 标签

通过启动加载项完成对软硬件的默认初始化任务后，最先执行的是arch/arm/boot/compressed/head.S的start标签中的代码。[①]在start标签中，从启动加载项接收结构ID和atags（参考7.4节）信息。

此外，还会禁用中断及初始化寄存器，并跳转到not_relocated标签。仔细观察代码5-1。

:: 代码5-1 arch/arm/boot/compressed/head.S的start标签

```
start:
    .type   start,#function
    ...
1:  mov r7, r1      @ 保存结构ID                                    ❶
    mov r8, r2      @ 保存atags指针
    ...
#ifndef __ARM_ARCH_2__                                              ❷
    ...



    mrs r2, cpsr                    @ 输入CPSR
    orr r2, r2, #0xc0               @ 在第6、7位中设置关闭IRQ、FIQ
    msr cpsr_c, r2                  @ 反映到CPSR并关闭中断
#else
    teqp    pc, #0x0c000003         @ 关闭中断                      ❸
#endif
    ...

.text                                                               ❹

    adr  r0, LC0
    ldmia    r0, {r1, r2, r3, r4, r5, r6, ip, sp}
    subs    r0, r0, r1

beq not_relocated
```

在start标签的代码段❶中，将启动加载项传递的结构ID和atags信息分别保存到寄存器r7、r8。

在代码段❷中禁用中断，同时禁用IRQ、FIQ。ARM处理器版本不同，禁用中断的方法也不一样。__ARM_ARCH_2__是通过GCC对ARM2、ARM3处理器进行声明的宏。

对于ARM版本2、3的核心，其禁用中断的方法如代码段❸所示。

代码段❹是字符段的开始区域，如下图所示设置寄存器r1~r6,ip,sp。

① 启动加载项必须提供5种功能：RAM初始化、串行端口初始化、查找机器类别、构建tagged list内核、将控制移交到内核镜像。

```
LC0                     ->      r1
__bss_start             ->      r2
_end                    ->      r3
zreladdr                ->      r4
_start                  ->      r5
_got_start              ->      r6
_got_end                ->      ip
user_stack+4096         ->      sp
```

表5-1中，ip、sp是GCC中使用的各ARM寄存器的命名规则（mnemonic）。GCC中使用的ARM寄存器名称和用法请参照表5-1。然后跳转到not_relocated标签。

表5-1　GCC中使用的ARM寄存器名称和使用方法

寄存器名称	寄存器命名规则	使用方法
r0	a1	• 传递到函数的第一个参数 • 结果值寄存器 • 临时寄存器
r1	a2	• 传递到函数的第二个参数 • 临时寄存器
r2	a3	• 传递到函数的第三个参数 • 临时寄存器
r3	a4	• 传递到函数的第四个参数 • 临时寄存器
r4	v1	• 寄存器变量
r5	v2	• 寄存器变量
r6	v3	• 寄存器变量
r7	v4	• 寄存器变量
r8	v5	• 寄存器变量
r9	v6 rfp	• 寄存器变量——实际的帧指针
r10	sl	• 栈界限（stack limit）
r11	fp	• 帧指针（frame pointer）
r12	ip	• 临时寄存器（scratch register）
r13	sp	• 栈指针（stack pointer）
r14	lr	• 连接寄存器（link register）
r15	pc	• 程序计数器（program counter）

5.2　BSS 系统域初始化——not_relocated 标签

解压zImage这一压缩内核的准备工作有：初始化BSS区域、激活缓存以及设置动态内存区域。这些设置是解压内核时必不可少的事项。通过代码5-2观察其原理。

:: 代码5-2 arch/arm/boot/compressed/head.S的not_relocated标签

```
not_relocated:
    mov r0, #0
1:                                                          ❶
    str r0, [r2], #4                    @ 初始化bss区域
    str r0, [r2], #4
    str r0, [r2], #4
    str r0, [r2], #4
    cmp r2, r3                          @ 比较r2、r3
    blo 1b                              @ 若r2比r3小，跳转到后续标签1

    bl  cache_on                                            ❷
    mov r1, sp                          @ malloc区域位于栈之上
    add r2, sp, #0x10000                @ 64k max, r2 = sp+ 0x10000
    ...
    bhs wont_overwrite                  @ 若r4为higher or same，则跳转      ❸
```

在代码段❶中，将BSS区域初始化为0。BSS区域是未初始化的用于全局数据的空间，它位于__bss_start和__end之间，通过链接脚本文件赋值。

通过代码段❷调用cache_on激活缓存，将用于解压内核的动态内存空间的起始地址（user_stack + 4 KB）和最终地址（user_stack + 4 KB + 64 KB）存入寄存器r1、r2。也就是说，动态内存空间为64 KB。至此完成解压内核的所有准备。该cache_on标签的代码分析将在下一节（5.3节）进行详细说明。

跳转到代码段❸的wont_overwrite标签，正式解压内核。关于wont_overwrite的详细内容请参考6.1节。



补充知识点 vmlinux.lds链接脚本文件

链接脚本文件被链接器浏览。它用某种方式与makefile生成的对象文件链接，从而生成内核映像信息。链接器一直使用链接脚本，如果不直接提供链接脚本，则利用默认脚本。默认脚本可通过ld-verbose查看。

```
code@kernel:~$ ld -verbose
GNU ld (GNU Binutils for Debian) 2.20.1.20100303
  Supported emulations:
   armelf_linux_eabi
   armelfb_linux_eabi
   elf_x86_64
   elf_i386
   i386linux
   elf_l1om
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
              "elf32-littlearm")
```

```
OUTPUT_ARCH(arm)
ENTRY(_start)
SEARCH_DIR("/usr/arm-linux-gnueabi/lib");
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x00008000)); . =
SEGMENT_START("text-segment", 0x00008000) + SIZEOF_HEADERS;
  .interp         : { *(.interp) }

  ...

  .text           :
  {
    *(.text.unlikely .text.*_unlikely)
    *(.text .stub .text.* .gnu.linkonce.t.*)
    /* .gnu.warning sections are handled specially by elf32.em.  */
    *(.gnu.warning)
    *(.glue_7t) *(.glue_7) *(.vfp11_veneer) *(.v4_bx)
  } =0

  ...

  .bss            :
  {
   *(.dynbss)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   /* Align here to ensure that the .bss section occupies space up to
      _end.  Align after .bss to ensure correct alignment even if the
      .bss section disappears because there are no input sections.
      FIXME: Why do we need it- When there is no .bss section, we don't
      pad the .data section.  */
    . = ALIGN(. != 0 - 32 / 8 : 1);
  }
  _bss_end__ = . ; __bss_end__ = . ;
  . = ALIGN(32 / 8);
  . = ALIGN(32 / 8);
  __end__ = . ;
  _end = .; PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);

  ...

}

==================================================
code@kernel:~$
```

通过-T选项向链接器提供自身的链接脚本。下列代码就是简单的链接脚本文件。

```
OUTPUT_FORMAT(  "elf32-littlearm",          /*需要执行的二进制文件，用elf格式生成*/
                "elf32-littlearm",
                "elf32-littlearm")
OUTPUT_ARCH(arm)                            /*用ARM标记结构*/
ENTRY(_start)                               /*将起始函数设置为_start*/
SECTIONS
```

```
{
    . = 0x00000000;                    /*映像的起始地址 - 不使用mmu时,
                                         可如0xA000000存入物理内存地址。*/

    . = ALIGN(4);
    .text : { *(.text) }               /*各对象(.o)文件的各text节区（代码区域）
                                         通过链接器重新分配为一个text节区。*/

    . = ALIGN(4);
    .rodata : { *(.rodata) }           /*各对象(.o)文件的各rodata节区（只读数据区域）
                                         通过链接器重新分配为一个rodata节区。*/
    . = ALIGN(4);                      /*各对象(.o)文件的data节区（数据区域）
    .data : { *(.data) }                 通过链接器重新分配为一个data节区。*/

    . = ALIGN(4);                      /*各对象(.o)文件的各got节区（global offset
    got : { *(.got) }                    table）通过链接器重新分配为一个got节区。*/

    . = ALIGN(4);                      /*各对象(.o)文件的各bss节区（未初始化的数据区域）
    .bss : { *(.bss) }                   通过链接器重新分配为一个bss节区。*/
}
```

5.3 激活缓存——cache_on 标签

如果已经完成了BSS区域的初始化和动态内存区域的设置，解压内核的最后准备就是执行打开缓存（cache_on）任务。Linux中“打开缓存”的实现方式根据ARM结构版本的不同而有所不同。在cache_on标签中查找并调用与当前系统ARM结构版本相符的“打开缓存“子程序。另外，调用__setup_mmu子程序对页目录进行初始化。请仔细观察代码5-3。

:: 代码5-3 arch/arm/boot/compressed/head.S的cache_on标签

```
cache_on:   mov r3, #8                 @ cache_on函数                          ❶
    b   call_cache_fn

    ...

call_cache_fn:
    adr   r12, proc_types              @ r12 = proc_types的标签起始地址        ❷

#ifdef CONFIG_CPU_CP15
    mrc p15, 0, r6, c0, c0             @ get processer ID
#else
    ldr r6, =CONFIG_PROCESSOR_ID       @ 从r6获得处理器ID
#endif
1:  ldr r1, [r12, #0]                  @ core ID                               ❸
    ldr r2, [r12, #4]                  @ Mask
    eor r1, r1, r6                     @ 执行(real ^ match) - > exclusive or 运算
                                       @ 2位相同时为0，不同时为1
    tst r1, r2                         @ &mask          ->适用掩码
                                       @ tst用于检查所选各“位”是否为all zero

    addeq   pc, r12, r3                @ 调用cache函数 ->如果结果值为0
```

```
                                        @ 在pc中输入r12+r13的值并跳转
    add r12, r12, #4*5                  @ 如果不一致，在r12加20
                                        @ (指向下一项。)
    b   1b                              @ 跳转到后续标签1
```

在代码段❶中，将常数8存入寄存器r3，并跳转到call_cache_fn标签。通过proc_types列表管理打开缓存（cache on）、清除缓存（cache flush）、关闭缓存（cache off）功能。这些功能根据ARM处理器版本的不同而有所不同。即，proc_types列表的各项都具有符合特定ARM处理器版本的（针对打开缓存、清除缓存、关闭缓存子程序的）信息。常数8用作偏移量，它指向proc_types列表各项的“打开缓存”子程序的起始地址。proc_types列表结构请参考下列“补充知识点”。

补充知识点　proc_types列表

每个ARM处理器版本都用proc_types列表显示为1项，其项结构如图5-2所示。

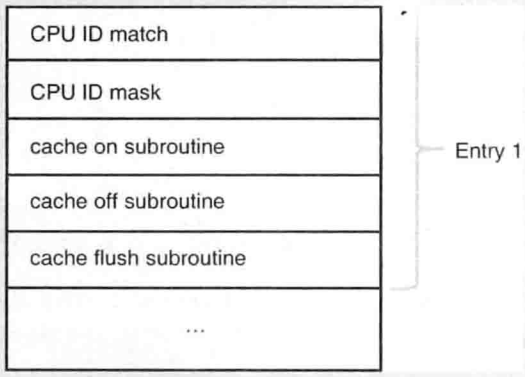


图5-2　proc_types列表

代码5-3的代码段❸中，((real_id ^ match) & mask) == 0时，项一致。而且，使用直写（write through，直写模式）策略的缓存具有打开/关闭缓存（cache on/off）的函数，但使用回写（write back，回写模式）策略的缓存需要另行添加清除缓存（cache flush）的函数，以与内存保持同步。

代码段❷中将之前讲解的proc_types列表起始地址存入寄存器r2。CONFIG_CPU_CP15是关系到协处理器（coprocessor1 CP15）[①]存在与否的内核设置宏。如果具有CP15的ARM结构，就通过CP15获取CPU ID。

代码段❸中，通过proc_types列表查找符合当前ARM结构的cache_on子程序，并跳转。

之前我们通过proc_types列表移动到了符合ARM处理器版本的、用于实现打开缓存任务的子程序。虽然每个ARM版本中“打开缓存”的实现方法不同，但执行的任务都一样。代码5-4是ARM

① 协处理器通过添加的指令集（instruction set）和配置寄存器（configuration register）支持ARM核心的附加功能。特别是CP15，它能让ARM处理器对缓存、TCM、识别寄存器、写缓冲、内存管理等进行控制。

v6的示例。ARM v6采用与ARM v4相同的方法执行“打开缓存”任务，因此运行__armv4_mmu_cache_on子程序。

:: 代码5-4 arch/arm/boot/compressed/head.S的__armv4_mmu_cache_on标签

```
__armv4_mmu_cache_on:
    mov r12, lr                                          ❶

    bl  __setup_mmu
    mov r0, #0

    mcr p15, 0, r0, c7, c10, 4                           ❷
    mcr p15, 0, r0, c8, c7, 0
    mrc p15, 0, r0, c1, c0, 0

    orr r0, r0, #0x5000                                  ❸
    rr r0, r0, #0x0030

    bl  __common_mmu_cache_on                            ❹

    mov r0, #0                                           ❺
    mcr p15, 0, r0, c8, c7, 0

    mov pc, r12
```

首先在代码段❶中将寄存器lr中的返回地址存入r12，之后调用__setup_mmu子程序。这是为了运行__setup_mmu子程序并返回后，还能继续执行剩余代码。

代码段❷中，利用CP15将所有写缓冲的内容更新到内存，并清除指令缓存（I-Cache）、数据缓存（D-Cache）、TLB等。

代码段❸中，读取CP15的控制寄存器（control register）的内容，设置指令缓存激活位、Round Robin缓存交替策略激活位。更详细的内容请参照下列“补充知识点”。已更改的设置适用于__common_mmu_cache_0n子程序。

补充知识点 CP15控制寄存器

ARM通过CP15的控制寄存器执行激活MMU和缓存、缓存交替策略等任务。CP15的控制寄存器如图5-3所示。

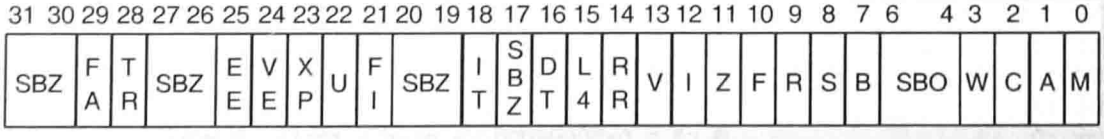


图5-3 CP15控制寄存器

0x5000通过设置第12位（I bit）和第14位（RR bit）以激活“指令缓存”，并设置缓存循环交替策略。

代码段❹的__common_mmu_cache_on子程序中，适用了代码段❸中变更的域（domain）设置和“指令缓存”激活、缓存循环交替策略。同时将页目录（page directory）的起始地址值存入CP15专用寄存器（Table Base Register）[①]。

最后，通过代码段❺清除指令缓存、数据缓存、TLB，并返回代码5-2中bl cache_on的下一个指令。

5.4　页目录项初始化——__setup_mmu 标签

__setup_mmu标签在cache_on标签内调用，用于初始化解压内核所需的页目录项。特别是对内存的256 MB区域设置cacheable、bufferable，这是因为解压内核时，使用缓存和写缓冲以提高解压性能。执行__setup_mmu后的页目录如图5-4所示。

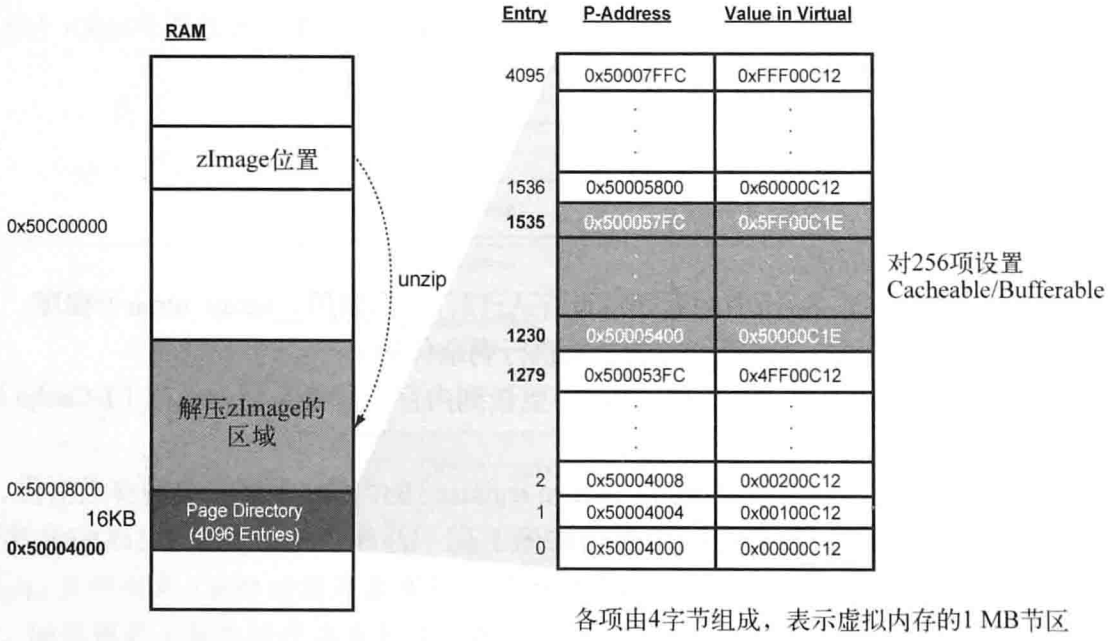


图5-4　执行__setup_mmu后的页表

下面通过代码5-5了解内核如何初始化页目录。

:: 代码5-5　arch/arm/boot/compressed/head.S的__setup_mmu标签

```
__setup_mmu:
    sub r3, r4, #16384                                          ❶
    bic r3, r3, #0xff           @ 指针排列
    bic r3, r3, #0x3f00
    mov r0, r3                  @ r3 = 0x50004000 -> page directory起始地址
```

① TTBR是特殊寄存器，在CP15内用于储存L1（Level 1）列表的物理地址。通过OS使用TTBR，目的在于快速访问L1列表。

```
    mov r9, r0, lsr #18          @ 向右移动18位right shift                    ❷
    mov r9, r9, lsl #18          @ 再向左移动18位left shift  ->r9 = start of RAM
    add r10, r9, #0x10000000     @ r10 = end of RAM

    mov r1, #0x12                @ 将r1和0xC00执行or运算  ->r1 = 0xC12
    orr r1, r1, #3 << 10         @ 3 * 2^10 -> 3k -> 0xC00

    /* r2与r4一致 (zreladdr) */
    add r2, r3, #16384
1:  cmp r1, r9
    orrhs   r1, r1, #0x0c        @ r1的值为RAM区域时， set cacheable、bufferable
    cmp r1, r10
    bichs   r1, r1, #0x0c        @ r1的值不是RAM区域时，clear cacheable、bufferable

    str r1, [r0], #4             @ 1:1映射

    add r1, r1, #1048576

    teq r0, r2                   @ 比较二者是否一致
    bne 1b                       @ 如果不一致，则跳转到后续标签1

    ...

    mov pc, lr                   @ lr在__armv4_mmu_cache_on标签中
                                 @ 指向bl__setup_mmu的下一地址
ENDPROC(__setup_mmu)
```

在代码段❶中将页目录的起始地址存入寄存器r0。该起始地址是通过ZRELADDR（zImage的物理地址，16 KB）获得的。它位于内核映像前，大小为16 KB。各结构具有的ZRELADDR值都不同，可在arch/arm/$(MACH)/Makefile.boot中找到。对于页目录请参考下列“补充知识点”。

通过代码段❷，页目录以1 MB节区（section）为单位管理内存。对于页目录内的4096个项，将访问权限（access permission）设置为可读/写。而且，对4096个项中的256个（256 MB区域）设置cacheable、bufferable。



补充知识点 MMU页表

MMU将虚拟地址变换为物理地址（virtual to physical translation）、控制内存访问权限（memory access permission control）、对内存各页设置缓存及写缓冲。如果MMU被禁用，虚拟地址和物理地址互相映射为1：1。

ARM MMU硬件具有多级页表（multilevel page table）结构。该结构是由L1和L2组成的2段结构。L1页表称为主页表（master page table）。每项具有L2页表的起始地址，并将页变换为1 MB。

因此又称区域页表（section page table）。由于主L1页表将4 GB以1 MB节区为单位进行分割，因此，共具有4096个项。

在前面介绍的代码5-5中，主L1页表起到区域页表的作用。该区域页表拥有1 MB虚拟页的起始地址。Linux中使用的区域表（section table）的各区域项（section entry）如图5-5所示。

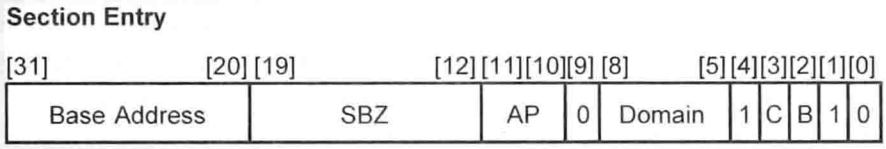


图5-5　MMU区域表项



补充知识点　缓存及写缓冲

在处理器中执行写入的速度和内存中处理写入的速度具有很大差距。在处理器中，写入命令结束前，处理器必须等待。由于内存写入在处理器中耗费大量时间（clock），通过写缓冲可防止这种延迟现象出现。写缓冲（write buffer）是很小且速度很快的FIFO内存缓冲，用于临时保存应写入主内存的数据，从而减少处理器执行内存写入时的时间消耗。

同样，处理器中执行读取的速度和内存中处理读取的速度差距也很大。在处理器中，读取命令结束前，处理器需要等待。缓存（cache）是速度极快的存储装置，用于临时保存处理器从内存读取的代码或数据。因此，处理器访问最近读过的数据时，可减少访问内存时的时间消耗。

在前面介绍的代码5-5中，我们看到对256个项设置了cacheable、bufferable。通过这些设置，对256个区域内的数据执行读取/写入时使用缓存和写缓冲。在区域项内设置的值如图5-6所示。

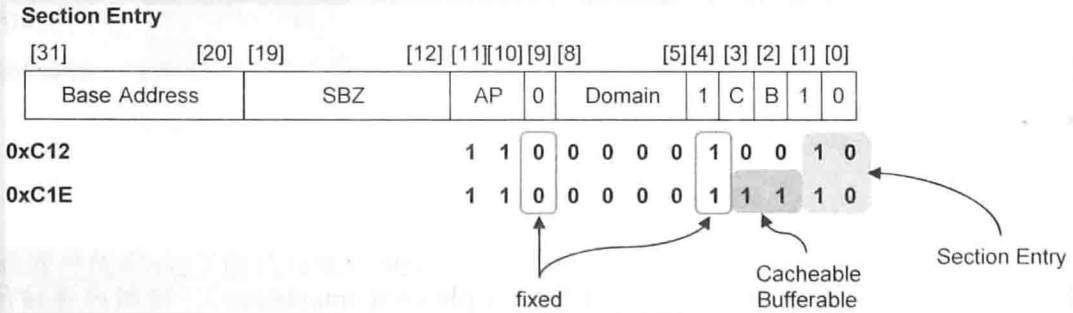


图5-6　MMU区域项中设置的值

5.5　指令缓存激活及缓存策略适用——__common_mmu_cache_on 标签

__common_mmu_cache_on在cache_on标签内被调用，只运行ARM版本7以前的处理器。对CP15适用激活指令缓存及缓存交替策略，并初始化TTBR。下面通过代码5-6了解内核如何激活指

令缓存及适用缓存策略。

:: 代码5-6 arch/arm/boot/compressed/head.S的__common_mmu_cache_on标签

```
__common_mmu_cache_on:

    ...

    mcr p15, 0, r3, c2, c0, 0          @ 保存页表地址                    ❶
    ...
1:  mcr p15, 0, r0, c1, c0, 0          @ 将r0存入控制寄存器               ❷
    mrc p15, 0, r0, c1, c0, 0
    sub pc, lr, r0, lsr #32
```

首先，代码段❶中，将寄存器r3内保存的页目录地址值存入CP15中的TTBR。

代码段❷中，对CP15的控制寄存器适用__armv4_mmu_cache_on标签中设置的缓存循环交替策略、指令缓存激活设置。

第6章 从压缩的内核zImage还原内核映像

我们之前通过not_relocated和cache_on做好了解压之前的准备工作：初始化BSS区域、设置动态内存区域、打开缓存等。现在正式对压缩后的内核映像zImage进行解压，并为内核初始化进行设置。

本章我们将通过gunzip对压缩后的内核zImage执行解压，并调用与当前处理器的ARM结构对应的缓存相关函数，使数据缓存、指令缓存、写缓冲等失效。[①]将程序计数器移动到zImage最终解压位置，使解压后的内核能够执行。图6-1用图形描述了本章要了解的内核代码的内部流程。

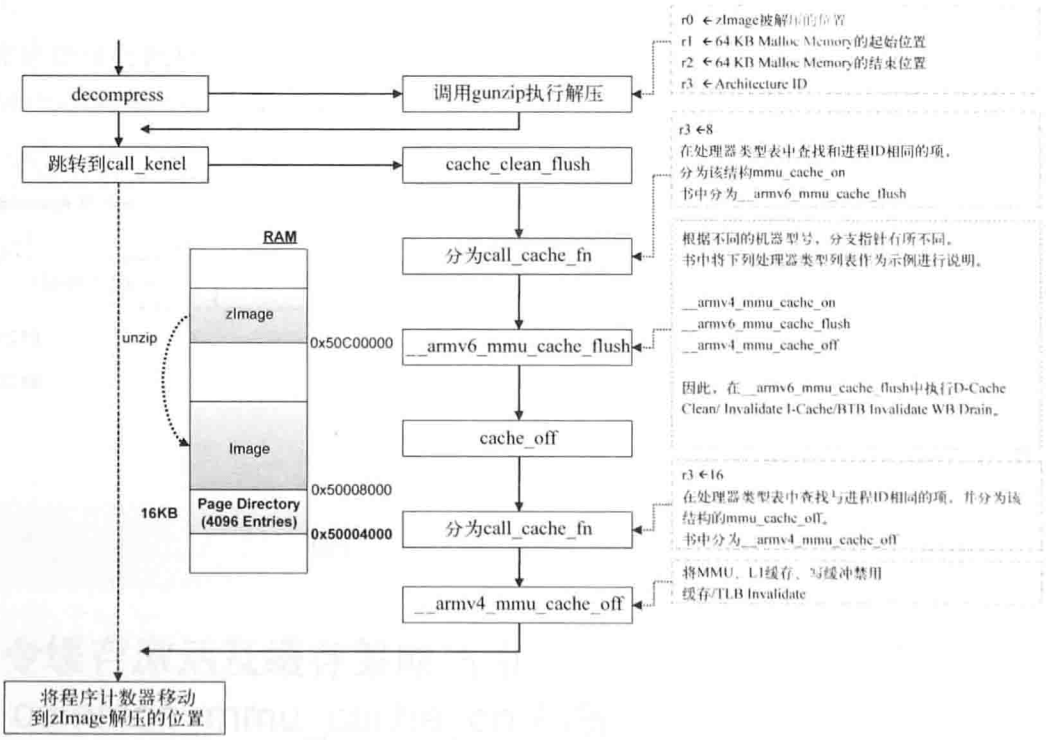


图6-1　解压zImage流程图

① gunzip位于内核源代码.arch/arm/boot/compressed/misc.c中。

6.1 解压内核并避免覆写——wont_overwrite、decompress_kernel 标签

wont_overwrite标签在cache_on标签后执行。我们已经完成对压缩内核zImage执行解压的所有准备工作。内核会解压到ZRELADDR地址，利用decompress_kernel函数完成解压工作。下面通过代码6-1了解内核执行的各项任务。

:: 代码6-1 arch/arm/boot/compressed/head.S的wont_overwrite标签

```
wont_overwrite:
    mov r0, r4                  @ r0 = 解压后的内核起始地址                    ❶
    mov r3, r7                  @ r3 = 结构ID

    bl  decompress_kernel       @ lr = bl decompress_kernel的下一指令

    b   call_kernel                                                          ❷
```


代码段❶中，将zImage解压的地址ZRELADDR存入寄存器r0，将结构ID存入寄存器r3后，调用decompress_kernel。如前所述，ARM中r0 ~ r3寄存器是调用函数时传递的参数（argument）。作为向decompress_kernel函数传递的信息，它已经具备了zImage解压的位置（r0）、动态内存起始位置（r1）、动态内存结束位置（r2）、结构ID（r3）等所有要素。decompress_kernel是解压zImage的子程序。这样，引导加载项（bootstrap loader）的所有操作基本结束，跳转到代码段❷的call_kernel标签执行内核初始化代码。

decompress_kernel函数是对压缩的内核（zImage）执行解压的实际子程序。从wont_overwrite标签接收解压所需参数进行解压。下面通过代码6-2了解内核对压缩内核执行解压的逻辑。

:: 代码6-2 arch/arm/boot/compressed/misc.c的decompress_kernel函数

```
ulg decompress_kernel(ulg output_start, ulg free_mem_ptr_p, ulg free_mem_ptr_end_p,
                      int arch_id)

    ...

    arch_decomp_setup();
    ...

    makecrc();
    putstr("Uncompressing Linux...");

    ...

    gunzip();                                                                ❶
    putstr(" done, booting the kernel.\n");

    /* r0 = output_ptr */
    return output_ptr;                                                       ❷
```

表6-1描述了代码段❶中向decompress_kernel函数传递的参数，以及与参数对应的寄存器的值。

表6-1　与参数对应的寄存器及内容

参　数	寄 存 器	内　容
ulg output_start	r0	zImage解压的位置
ulg free_mem_ptr_p	r1	动态内存起始位置
ulg free_mem_ptr_end	r2	动态内存结束位置
int arch_id	r3	结构ID

利用接收的动态内存空间（r1 ~ r2）执行解压，解压后的内核映像将位于zImage的执行地址（r0）。

通过代码段❷的output_ptr返回解压后的内核大小。就这样，zImage解压后，ZRELADDR（内核执行地址）中会出现vmlinux。

6.2　调用已解压内核——call_kernel 标签

解压zImage后，将控制权移交给已解压内核的起始位置ZRELADDR。对缓存和MMU执行禁用并使之失效后，控制权会移动到实际的内核初始化部分arch/arm/kernel/head.S的stext节区。将结构ID和atags指针作为参数传递。仔细观察代码6-3。

:: 代码6-3　arch/arm/boot/compressed/head.S的call_kernel标签

```
call_kernel:

    bl  cache_clean_flush          @ 清理缓存、清除处理          ❶
    bl  cache_off                  @ 关闭缓存                    ❷

    mov r0, #0                                                   ❸
    mov r1, r7                     @ 结构编号
    mov r2, r8                     @ 输入atags指针
    mov pc, r4                     @ 调用内核
```

在代码段❶中对内存适用数据缓存内容，使之与指令缓存一同失效，清除BTC（Branch Target Cache，分支目标缓存）。

通过代码段❷禁用数据缓存和MMU，并使数据/指令缓存和TLB失效，清除BTC。

代码段❸中，控制权会移动到arch/arm/kernel/head.S，正式执行内核初始化。将结构ID和atags指针作为参数传递。

6.3　缓存清理及清除——cache_clean_flush 标签

cache_clean_flush标签是在完成解压后通过call_kernel标签调用的。该标签从proc_types列表查找与结构相符的缓存清理子程序。通过代码6-4观察内核对缓存执行清理或清除的方法。

:: 代码6-4 arch/arm/boot/compressed/head.S的cache_clean_flush标签

```
cache_clean_flush:                                                    ❶
    mov r3, #16                         @ 在call_cache_fn中用作指向项内的函数
                                        @ 的索引
    b   call_cache_fn

    ...

call_cache_fn:                                                        ❷

    ...

    addeq   pc, r12, r3                 @ 如果结果值为0，则调用cache函数

    ...

__armv6_mmu_cache_flush:                                              ❸
    mov r1, #0
    mcr p15, 0, r1, c7, c14, 0          @ D-Cache 对内存适用数据缓存的内容后，使之失效
    mcr p15, 0, r1, c7, c5, 0           @ 使所有指令缓存和BTB失效

    mcr p15, 0, r1, c7, c15, 0
    mcr p15, 0, r1, c7, c10, 4          @ 对内存适用所有缓冲内容
    mov pc, lr                          @ 跳转到call_kernel标签的bl cache_off
```

清除缓存的实现方法和5.3节的“打开缓存”的实现方法相似。

代码段❶中，在与当前结构一致的项内求出的缓存清除函数地址的偏移量用值16表示。

通过代码段❷的call_cache_fn，利用进程ID和之前传递的项偏移量16搜索proc_types列表，移动到符合ARM处理器版本的mmu_cache_flush标签。清除缓存的实现方法在各版本ARM处理器中均不同，但都如代码段❸所示，对内存适用数据缓存的内容后使之失效，同时使指令缓存失效并清除BTC[①]。



补充知识点 ARM中的清理、清除、失效、清空

ARM中，缓存相关术语有清除（flush）、失效（invalidate）、清理（clean），写缓冲相关术语有清空（drain）。缓存中所谓的“清除”是指，将高速缓存线（cache line）的页面重写标志位（dirty bit）重置为0，使内存无反映。反之，“清理”是指，凡是设置为1的页面重写标志位全部反映到内存。此处“反映”是指，有写缓冲时，在其中执行写入操作。在高速缓存线中，既然是为了通过“清理”向内存执行反映而写入写缓冲，那么必须把写缓冲的内容全部反映到内存，这叫“清空”。这里需要注意的是，ARM缓存中使用的“清除”和我们常用的“清除”不是一个意思，反而和“清理”更相近。

① BTC或BTB（Branch Target Buffer，分支目标缓冲）是保存跳转对象地址的缓存。对需要跳转的对象地址进行预测，从而提高处理器效率。

还有一个需要注意的词是“失效”。观察内核代码可以发现，“失效”通常具有两种意思：第一种，与“清除缓存”同义，使页面重写标志位失效；第二种，在高速缓存线中使有效位（valid bit）失效。后者能使高速缓存线整体失效，从这一点上讲，它和“清除”完全不是一个概念。因此，这部分需要着重考虑，根据上下文准确判断“失效”的含义。

6.4　缓存禁用——cache_off 标签

cache_off标签是从call_kernel标签调用的。从proc_types列表搜索并执行符合结构的缓存禁用子程序。通过代码6-5观察内核如何禁用缓存。

:: 代码6-5　arch/arm/boot/compressed/head.S的cache_off标签

```
cache_off:                                                          ❶
    mov r3, #12                 @ 在call_cache_fn中指向项内的
                                @ 缓存禁用函数的偏移量
    b   call_cache_fn

    ...

call_cache_fn:                                                      ❷

    ...

    addeq   pc, r12, r3         @ 若结果值为0，则调用cache函数
    ...

__armv4_mmu_cache_off:                                              ❸

    mrc p15, 0, r0, c1, c0      @ 将control reg复制到r0
    bic r0, r0, #0x000d         @ 清空0x000D ->1101 0,2,3号位
    mcr p15, 0, r0, c1, c0      @ turn MMU and cache off，开启MMU并禁用缓存，适用变更事项

    /*使TLB和缓存全部失效*/
    mov r0, #0
    mcr p15, 0, r0, c7, c7
    mcr p15, 0, r0, c8, c7

    mov pc, lr
```

关闭缓存的实现方法和打开缓存相似。

代码段❶中，用值12表示在与当前结构一致的项内求出的关闭缓存函数地址的偏移量。

代码段❷中，如cache_clean_flush标签中执行的那样，使用处理器ID和项偏移量12搜索proc_types列表。之后，在列表中查找并移动到符合当前CPU的ARM版本的mmu_cache_off标签。

最后，在代码段❸中禁用数据缓存和MMU，使数据/指令缓存和TLB失效，并清除BTC。

第7章 调用start_kernel()

本章是调用start_kernel()函数的最后阶段。正式启动内核前，先查找保存自身处理器信息和机器信息的结构体位置，确认从启动加载项接收的atag信息是否有效。

如果自身处理器信息和机器信息有误，程序就立即结束。在所有信息有效的基础上，设置页表的MMU标识符并激活，最终调用C代码编成的start_kernel()内核起始函数。

图7-1使用图形描述了本章要学习的内核代码内部流程。

7.1 初始化指向——stext 标签

通过引导加载项[①]加载内核后，首先执行的部分就是stext。执行该标签时要求如下状态。

```
MMU          = off
D-Cache      = off
r0           = 0
r1           = machine number
r2           = atags pointer
```

在cache_off标签中关闭MMU和缓存，机器号和atags指针是从启动加载项接收的，所以满足上述所有要求。在stext标签中，首先转换为SVC模式（SVC_MODE）[②]，并禁用IRQ。然后调用多个检查程序，查找CPU和机器信息，并检查atag信息，追加设置页表后启动MMU。通过代码7-1仔细观察stext标签中执行的各项任务。

:: 代码 7-1 arch/arm/kernel/head.S的stext标签

```
ENTRY(stext)
    msr     cpsr_c, #PSR_F_BIT|PSR_I_BIT|SVC_MODE                    ❶
    mrc     p15, 0, r9, c0, c0              @ 获得处理器ID

    bl      __lookup_processor_type         @ r5=一致的procinfo地址    ❷
    movs    r10, r5
    beq     __error_p
```

① 该程序负责将压缩的内核加载到内存，并执行解压等内核启动所需操作。

② 其中罗列了include/as/ptrace.h中设置ARM模式的psr寄存器的各“位”定义、模式定义。

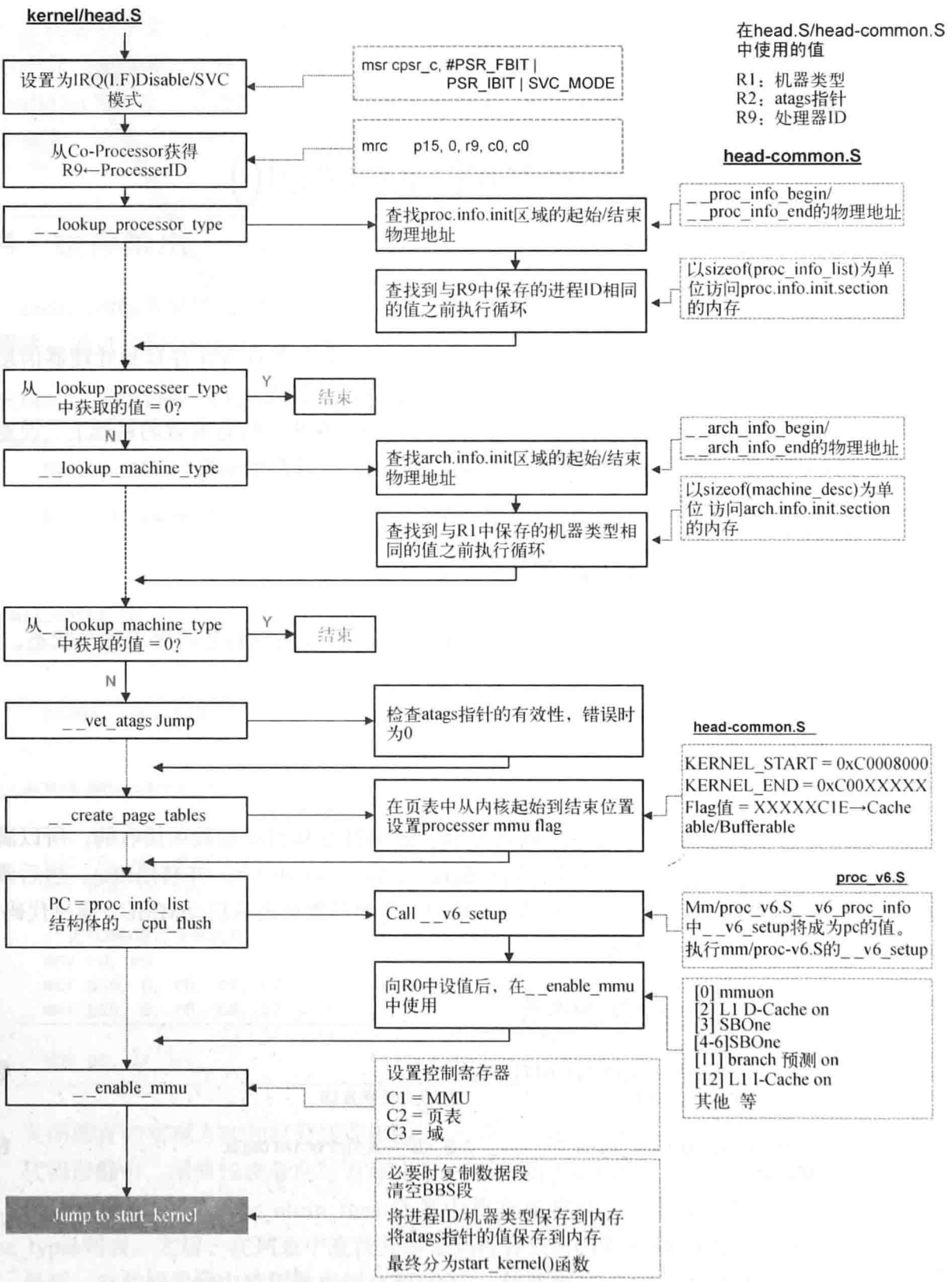


图7-1　解压缩后调用start_kernel的流程图

```
    bl      __lookup_machine_type           @ r5=一致的machinfo地址                ❸
    movs    r8, r5
    beq     __error_a

    bl      __vet_atags
    bl      __create_page_tables

    ...

    ldr     r13, __switch_data                                                    ❹
    adr     lr, __enable_mmu
    add     pc, r10, #PROCINFO_INITFUNC                                           ❺
ENDPROC(stext)
```

代码段❶中，使CPU进入SVC模式，禁用IRQ、FIQ。

然后在代码段❷中跳转至__lookup_processor_type，查找与自身处理器一致的处理器信息。若未找到，则跳转到__error_p并显示错误消息。此时r5中保存0，通过movs常数0保存在寄存器r10时，设置Z标识符并执行beq。

如果找到了与当前处理器一致的信息，则在代码段❸跳转到__lookup_machine_type，寻找与当前机器一致的信息。若未找到就参照代码段❷，跳转到__error_a并显示错误消息。7.2节和7.3节中将对__lookup_processor_type和__lookup_machine_type这2个标签进行具体说明。

代码段❹中，为了启动内核，将__switched_data保存在寄存器r13中。Lr寄存器中保存了__enable_mmu的地址，执行PROCINFO_INITFUNC后通过lr寄存器中的__enable_mmu执行返回，从而激活MMU。

在代码段❺执行进程初始化函数。r10中储存了通过__lookup_processor_type获得的proc_info_llist结构体的起始地址。该值加上PROCINFO_INITFUNC即可调用__v6_setup子程序。

通过arch/arm/kernel/asm-offset.c执行编译时生成的include/asm-offset.h[①]中，此时常数PROCINFO_INITFUNC的值定义如下。

:: arch/arm/kernel/asm-offsets.c

```
DEFINE(PROCINFO_INITFUNC, offsetof(struct proc_info_list, __cpu_flush));
```

这段代码的意思是，在struct proc_info_list结构体中，将__cpu_flush成员变量所在偏移量指定为PROCINFO_INITFUNC的值。定义该常数的详细过程请参考下列“补充知识点”。

① 所有结构都具有arch/$(ARCH)/asm-offset.c文件。ARM的asm-offset.c是将偏移量转换为汇编器之宏的前奏。该文件被转换为asm-offset.s，该汇编语言文件再次生成arch/$(ARCH)/asm-offset.h。
（参考文献：http://marc.info/?l=linux-kernel&m=100219616028442&w=2）



补充知识点　DEFINE()和offsetof的定义

为了理解DEFINE宏，我们需要了解DEFINE和offsetof是如何定义的。下列代码是对DEFINE的定义，这段代码从sym接收的变量中储存值val。

:: include/linux/kbuild.h

```
DEFINE(sym,val) asm volatile("\n->" #sym " %0 " #val :: "i"(val))
```

宏offsetof定义如下：

:: include/linux/stddef.h

```
#undef offsetof
#ifdef __compiler_offsetof
#define offsetof(TYPE,MEMBER) __compiler_offsetof(TYPE,MEMBER)
#else
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
#endif
```

如果编译器支持offsetof，就直接返回TYPE的MEMBER位置；若不支持，则以接收的TYPE为基准位置，求出MEMBER的地址。

通过下列代码查看offsetof运行情况。

:: test_offsetof.c

```
#include <stdio.h>

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE*)0)->MEMBER)

typedef struct data {
    int a;
    int b;
    char c;
    char d;
} Data;

int main()
{
    Data d;

    printf("offsetof (base 0): %lu\n", offsetof(Data, b));

    printf("direct (base 0): %lu\n", ((size_t)&((Data *)0)->b));
    printf("direct (base 5): %lu\n", ((size_t)&((Data *)5)->b));
    return 0;
}
```

运行结果

```
code@kernel: ~/code$ gcc -o test_offsetof test_offsetof.c
code@kernel: ~/code$ ./test_offsetof
offsetof (base 0): 4
direct (base 0): 4
direct (base 5): 9
code@kernel: ~/code$
```

从test_offsetof.c代码的执行结果可以看出，基址设置为0时，Data的成员变量b的偏移量变成4；基址设置为5时，Data的成员变量b的偏移量变成9。

利用DEFINE和offsetof定义内核中的MACHINFO_TYPE如下：

:: arch/arm/kernel/asm-offset.c

```
DEFINE(MACHINFO_TYPE, offset of(struct machine_desc, nr));
```

这段代码最终转换为如下代码。

```
#define MACHINFO_TYPE 0
```


7.2 处理器信息搜寻——__look_processor_type

为了让内核正常运行，必须把握内核要执行的处理器信息和机器信息。本章将学习编译内核时如何找出保存的处理器信息。

7.2.1 __lookup_processor_type标签

arch/arm/kernel/head-common.S文件的__lookup_processor_type标签记述了内核中求处理器类型的代码。跳转到__lookup_processor_type标签前，要查找的处理器的CPU ID保存在寄存器r9中。通过代码7-2了解内核如何找到保存自身处理器类型的位置。

:: 代码7-2 arch/arm/kernel/head-common.S__lookup_processor_type标签

```
__lookup_processor_type:
    adr r3, 3f                                              ❶
    ldmda   r3, {r5 - r7}

    sub     r3, r3, r7                                      ❷
    add     r5, r5, r3
    add     r6, r6, r3

1:  ldmia   r5, {r3, r4}                @ r3←cpu_val,       ❸
                                        @ r4←cpu_mask
    and r4, r4, r9                      @ cpu_mask & CPUID
    teq r3, r4                          @ 比较cpu_val的值
```

```
    beq 2f                                  @ 一致时,
                                            @ 找到处理器并脱离

    add r5, r5, #PROC_INFO_SZ               @ sizeof(proc_info_list)          ❹
    cmp r5, r6
    blo 1b
    mov r5, #0

2:  mov pc, lr

ENDPROC(__lookup_processor_type)                                              ❺

ENTRY(lookup_processor_type)                                                  ❻
    Stmfd sp!, {r4 - r7, r9, lr}
    mov r9, r0
    bl  __lookup_processor_type
    mov r0, r5
    ldmfd       sp!, {r4 - r7, r9, pc}
ENDPROC(lookup_processor_type)

    .long       __proc_info_begin                                             ❼
    .long       __proc_info_end
3:  .long       .
```

在代码段❶中，将位于当前代码之前的标签3的地址（3f）保存在寄存器r3后，通过ldmda指令按如下方式填充r5～r7的值。

```
r3      ←       label3                  /*物理地址*/
r5      ←       __proc_info_begin       /*虚拟地址*/
r6      ←       __proc_info_end         /*虚拟地址*/
r7      ←       . (location conter)     /*虚拟地址*/
```

标签3属于代码段❼。7.2.4节中将对代码的具体执行原理进行详细说明。

代码段❷将寄存器r5、r6中保存的__proc_info_begin、__proc_info_end的值变更为物理地址。因为当前MMU尚未激活，无法使用虚拟地址。7.2.3节中将对虚拟地址变更为物理地址的原理进行详细说明。

从__proc_info_begin到__proc_info_end保存着内核的所有处理器信息，代码段❸和代码段❹依次执行循环，输入处理器信息。输入时利用proc_info_list结构体保存，并比较proc_info_list的cpu_val和CPU ID的值。7.2.4节中将进一步说明执行原理。

代码段❺的ENDPROC(__lookup_processor_type)定义在include/linux/linkage.h中。使用ENDPROC()将__lookup_processor_type标签作为函数注册到符号列表，使其能够从外部调用。

代码段❻是查找处理器类型的汇编代码的C API版本。也可从C代码中调用lookup_processor_type()函数并使用。[①]需要确认的是，在C代码中调用lookup_processor_type()函数时，利用stmdf指令在栈中备份寄存器值，将作为参数传递的值r0（CPU ID）（和前面的代码一样）保存在r9中，并执

① 可在13.3节查看lookup_processor_type()函数实际调用。

行__lookup_processor_type子程序，将寄存器5的值（结果值）保存在（保存返回值的寄存器）r0中。最后，还原栈中备份的各寄存器并返回。



补充知识点 ENTRY()、ENDPROC宏

为了从外部调用汇编代码编写的子程序，链接器和外部模块必须知道子程序名，即符号（symbol）。Linux也为此提供了一些宏。特别是子程序被外部调用且有返回值时，为了显示静态分析工具的效果，推荐使用ENDPROC()宏。

ENTRY、ENDPROC宏定义如下：

:: include/linux/linkage.h

```
#define ENTRY(name)             \
    .globl name;                \
    ALIGN;                      \
name:

#define ENDPROC(name)           \
    .type name, @function;      \
    END(name)

#define END(name)               \
    .size name, .-name
```

上述代码中，.global关键字能使外部模块看到name符号。换言之，name为函数时，外部模块可以调用它。.type关键字指定符号类型，它在代码中是函数（function）。对象（object）属于其他类型，.size关键字指定该函数的大小。

7.2.2 __proc_info_begin和__proc_info_end中保存的信息

首先，再次观察我们上节见过的代码段❶和代码段❼。

```
        adr r3, 3f                                  ❶
        ldmda   r3, {r5 - r7}
        ...

        .long   __proc_info_begin                   ❼
        .long   __proc_info_end
3:      .long   .
```

将前面的第三号标签（3f）的地址保存在寄存器r3后，执行ldmda指令，以r3的地址为基准，将二级子地址中保存的值保存到r5，将一级子地址中保存的值保存到r6，将根地址中保存的值保存到r7。这里的标签3属于代码段❼。

标签3的地址中保存了位置计数器（location counter）的值，子代码中记述了指向.proc.info.init

节区[①]起始地址的__proc_info_begin，和指向节区结束地址的__proc_info_end。

至此，寄存器r5、r6中保存的值如下。

```
r5      ←      __proc_info_begin     /*虚拟地址*/
r6      ←      __proc_info_end       /*虚拟地址*/
```

那么，__proc_info_begin和__proc_info_end之间保存的信息是什么呢？如图7-2所示，__proc_info_begin和__proc_info_end的地址空间保存了各处理器信息，并具有所有结构的proc_info_list结构体。正是proc_info_list结构体保存着处理器信息。

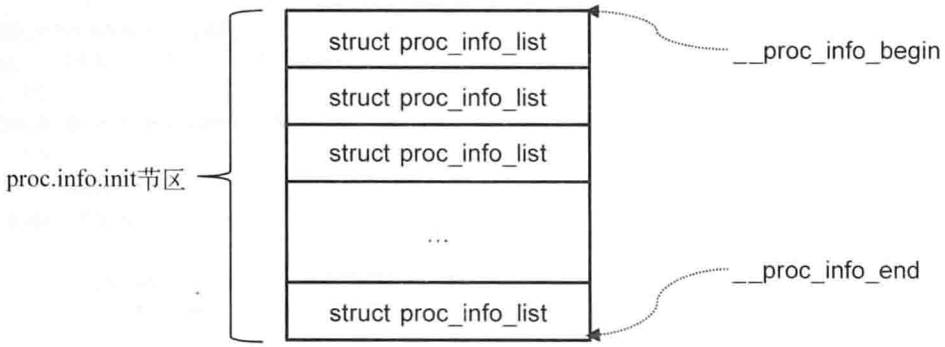


图7-2　.proc.info.init节区，与__proc_info_begin、__proc_info_end、proc_info_list的关系

最终，内核包含的所有处理器信息以proc_info_list结构体值的形式记录并保存在__proc_info_begin到__proc_info_end间。

∷ arch/arm/include/asm/procinfo.h

```
struct proc_info_list {
    unsigned int            cpu_val;
    unsigned int            cpu_mask;
    unsigned long           __cpu_mm_mmu_flags;    /* used by head.S */
    unsigned long           __cpu_io_mmu_flags;    /* used by head.S */
    unsigned long           __cpu_flush;           /* used by head.S */
    const char              *arch_name;
    const char              *elf_name;
    unsigned int            elf_hwcap;
    const char              *cpu_name;
    struct processor        *proc;
    struct cpu_tlb_fns      *tlb;
    struct cpu_user_fns     *user;
    struct cpu_cache_fns    *cache;
};
```

proc_info_list结构体中，正是cpu_val成员变量保存CPU ID的值。查找处理器类型时，与保存cpu_val变量值和CPU ID值的寄存器r9进行比较，更多说明参见7.2.4节。

① vmlinux.lds.S中记述了 .proc.info.init节区。在代码内，.proc.info.init定义的所有代码部分由链接器统合在一起。

7.2.3 在MMU禁用状态下将虚拟地址转换为物理地址

我们尚未激活MMU，因此无法通过虚拟地址访问内存。但是，保存处理器信息的区域地址__proc_info_begin和__proc_info_end是编译内核时指定的，且都是虚拟地址。因此，只有将这些虚拟地址变更为物理地址，才能访问具有处理器信息的proc_info_list结构体。

下面通过7.2.1节中观察过的代码段❷，了解MMU禁用状态下将虚拟地址变更为物理地址的原理。

```
sub     r3, r3, r7
add     r5, r5, r3
add     r6, r6, r3
```

要想理解上述代码，必须理解r3中保存的地址和r7中保存的地址位置计数器（.）、r5和r6中保存的地址之间的关系。为便于说明，下面再次记述r3、r5～r7中保存的值。

```
r3   ←   label 3                /*物理地址*/
r5   ←   __proc_info_begin      /*虚拟地址*/
r6   ←   __proc_info_end        /*虚拟地址*/
r7   ←   .(location counter)    /*虚拟地址*/
```

首先，r3保存的是物理地址，r7中保存的位置计数器（.）和r5、r6中保存的值都是虚拟地址。由于MMU尚未激活，故无法通过虚拟地址访问内存。

将虚拟地址变更为物理地址的方法十分简单。标签3是物理地址，r7的值（位置计数器）是虚拟地址。由于位置计数器（.）的虚拟地址是标签3的地址中保存的值，求出这2个地址的偏移量后，与虚拟地址__proc_info_begin和__proc_info_end的地址相加，即可获得对应的物理地址。图7-3表示求出偏移量后将虚拟地址变更为物理地址的原理。

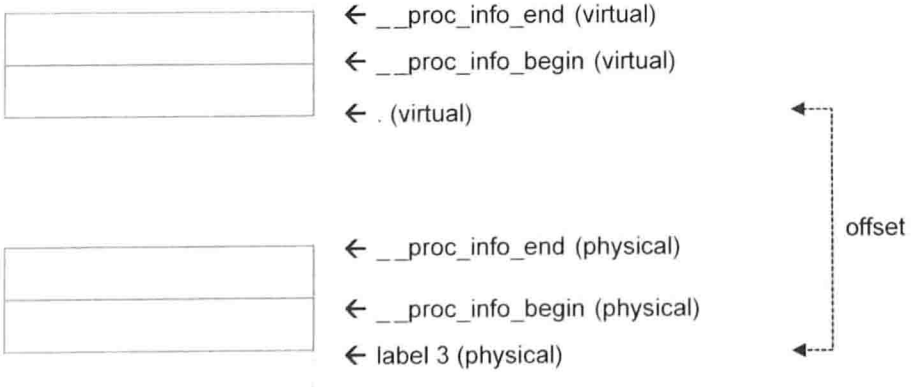


图7-3 虚拟地址和物理地址间的偏移量关系

```
physical_address (label 3) - virtual_address(.)    = r3 - r7
                                                   = offset
```

```
virtual_address(__proc_info_begin )+ offset        = r5 + offset
                                                   = physical_address(__proc_info_begin)

virtual_address(__proc_info_end )+ offset          = r6 + offset
                                                   = physical_address(__proc_info_end)
```

需要关注的是，偏移量的值有可能是负数。此处介绍的（将虚拟地址变更为物理地址的）方法同样适用于查找机器类型的__lookup_machine_type标签。

既然找到了保存各处理器类型信息的起始和结束物理地址，接下来需要和具有CPU ID值的r9进行比较，找出保存与当前处理器信息一致的位置（相应处理器的proc_info_list结构体）。

7.2.4　查找proc_info_list 结构体并比较处理器信息

既然已经找到了物理地址，那就在__proc_info_begin和__proc_info_end地址空间读取各proc_info_list结构体，查找对应于CPU ID的proc_info_list结构体。再次观察7.2.1节的代码段❸。

```
1:  ldmia   r5, {r3, r4}  @ r3←cpu_val,                                    ❸
                          @ r4←cpu_mask
    and r4, r4, r9        @ cpu_mask&CPU ID
    teq r3, r4            @ 比较cpu_val的值
    beq 2f                @ 一致时,
                          @ 找到处理器并脱离
```

为了查找当前处理器的信息，这段代码比较了proc_info_list结构体的cpu_val值和CPU ID值。寄存器r5中保存着__proc_info_begin的地址。__proc_info_begin不仅是第一个proc_info_list结构体的地址，而且和该结构体的成员变量cpu_val的地址相同。因此，“ldmia r5, {r3, r4}”使r3中保存cpu_val值，r4中保存cpu_mask值。对保存着CPU ID值的r9和当前cpu_info_list的cpu_mask执行AND运算，并将其结果和r3（保存cpu_val值）进行比较。

如果一致，表示找到了要找的处理器信息的结构体，因此移动到前方的第二个标签（2f）结束逻辑；如果不一致，查找下一个proc_info_list结构体，执行循环并比较。

之后，在proc_info_begin和proc_info_end中查找proc_info_list。下列代码即执行该任务。

```
    add r5, r5, #PROC_INFO_SZ   @ sizeof(proc_info_list)                   ❹
    cmp r5, r6
    blo 1b
    mov r5, #0

2:  mov pc, lr
```

寄存器r5具有之前proc_info_list结构体的起始地址，将该地址移动proc_info_list结构体大小之后，提取cpu_val值和cpu_mask值，与CPU ID比较。给寄存器r5加上#PROC_INFO_SZ。以下是#PROC_INFO_SZ的定义。

:: arch/arm/kernel/asm-offsets.c

```
DEFINE(PROC_INFO_SZ),      sizeof(struct proc_info_list));
```

PROC_INFO_SZ和sizeof(proc_info_list)相同，通过下列代码跳过proc_info_list大小的地址。

```
add r5,  r5 #PROC_INFO_SZ
```

如果（移动sizeof(proc_info_list)大小之后的）地址r5的值小于（保存__proc_info_end地址的）r6，则表示有可执行循环的proc_info_list结构体，故移动到后续标签1（1b）并执行循环；反之则表示不存在需要搜索的处理器信息，向r5中保存0并结束循环。

7.3 搜寻我的机型——__lookup_machine_type

若想正常运行内核，不仅要知道处理器信息，还需要了解系统信息，即机器信息。本节我们了解一下编译时生成的机器信息保存在何处，内核是如何找到它们的。这跟7.2节中提过的查找处理器信息的方法十分类似。

7.3.1 __lookup_machine_type标签

内核中，获取机器类型的代码记录在__lookup_machine_type标签，跳转到__lookup_machine_type标签前，寄存器r1中保存着机器结构号（machine architecture number）。通过代码7-3了解内核如何查找自身机器类型。

:: 代码7-3 arch/arm/kernel/head.S__lookup_machine_type标签

```
3:  .long   .                                                    ❶
    .long   __arch_info_begin
    .long   __arch_info_end

__lookup_machine_type:
    adr r3, 3b                          @ r3中保存label 3的地址   ❷
    ldmia   r3, {r4, r5, r6}            @ 增加地址值的同时执行多重加载

sub r3, r3, r4                                                   ❸
    add r5, r5, r3                      @ __arch_info_begin地址
    add r6, r6, r3                      @ __arch_info_end物理地址

1:  ldr r3, [r5, #MACHINFO_TYPE]        @ r3 = 机器类型           ❹
    teq r3, r1    @是否一致?
    beq 2f    @查找成功
    add r5, r5, #SIZEOF_MACHINE_DESC    @ 下一个machine_desc

    cmp r5, r6
    blo 1b
    mov r5, #0                          @ 查找失败

2:  mov pc, lr
ENDPROC(__lookup_machine_type)

ENTRY(lookup_machine_type)                                       ❺
    stmfd   sp!, {r4 - r6, lr}
```

```
    mov r1, r0
    bl  __lookup_machine_type
    mov r0, r5
    ldmfd   sp!, {r4 - r6, pc}
ENDPROC(lookup_machine_type)
```

代码段❶记录了寄存器中保存的所有地址。

代码段❷将下列值保存到寄存器r2、r4、r5、r6。

```
r3    ←    label 3                /*物理地址*/
r4    ←    .(location counter)    /*虚拟地址*/
r5    ←    __arch_info_begin      /*虚拟地址*/
r6    ←    __arch_info_end        /*虚拟地址*/
```

代码段❸中，为了访问__arch_info_begin到__arch_info_end地址空间保存的机器信息（machine_desc结构体）列表，查找与虚拟地址__arch_info_begin和__arch_info_end对应的物理地址。7.3.2节将对代码段❷和代码段❸做进一步说明。

在代码段❹中比较machine_desc结构体的第一个成员变量nr和寄存器r1的值，查找机器信息。7.3.3节将对获得结构体的相关内容做进一步说明。

代码段❺是我们已知的__lookup_machine_type汇编代码的C API版本，可在C代码中调用lookup_machine_type()函数。ENTRY()、ENDPROC()宏的相关内容请参考7.2.1节“补充知识点”。

7.3.2　保存在__arch_info_begin和__arch_info_end中的machine_desc信息及访问路径

内核将机器信息记录为machine_desc结构体，并保存在__arch_info_begin到__arch_info_end之间。请再次观察前面的代码。

```
3:    .long   .                                                    ❶
      .long   __arch_info_begin
      .long   __arch_info_end
__lookup_machine_type:
      adr r3, 3b                  @ r3中保存label 3的地址          ❷
      ldmia   r3, {r4, r5, r6}    @ 增加地址值的同时执行多重加载
```

将后续第3个标签（3b）的地址保存到寄存器r3后，以r3的地址为基准，每增加1个字符就执行 ldmia取值指令。因此，将r3地址的上一个字符地址中保存的值保存在r4中，将再上一个地址中保存的值保存在r5中，将根地址中保存的值保存在r6中。这种保存地址的方式和求出处理器信息的代码类似（请参考7.2节__lookup_processor_type）。标签3以位置计数器（.）为基准，记录了表示.arch.info.init节区起始地址的__arch_info_begin和结束地址的__arch_info_end。

需要确认的是，r3是物理地址，而r4、r5、r6都是虚拟地址。__arch_info_begin和__arch_info_end之间保存的信息是聚集在.arch.info.init节区各结构的machine_desc结构体列表，由machine_desc结构体保存机器信息，该结构体定义如下。

:: arch/arm/include/asm/mach/arch.h

```
struct machine_desc {
    unsigned int    nr;                 /* architecture number  */
    unsigned int    phys_io;            /* start of physical io */
    unsigned int    io_pg_offst;        /* byte offset for io * page tabe entry */
    const char      *name;              /* architecture name    */
    unsigned long   boot_params;        /* tagged list      */
    unsigned int    video_start;        /* start of video RAM   */
    unsigned int    video_end;          /* end of video RAM */
    unsigned int    reserve_lp0 :1;     /* never has lp0    */
    unsigned int    reserve_lp1 :1;     /* never has lp1    */
    unsigned int    reserve_lp2 :1;     /* never has lp2    */
    unsigned int    soft_reboot :1;     /* soft reboot      */
    void            (*fixup)(struct machine_desc *,
                    struct tag *, char **,
                    struct meminfo *);
    void            (*map_io)(void);    /* IO mapping function  */
    void            (*init_irq)(void);
    struct sys_timer    *timer;         /* system tick timer    */
    void            (*init_machine)(void);
};
```

machine_desc结构体中，第一个成员变量nr中保存着结构号。该值将会与保存机器结构号的寄存器r1值进行比较。

为获取机器信息，需要搜索虚拟地址__arch_info_begin和__arch_info_end之间的所有machine_desc结构体。但是，这2个值都是虚拟地址。由于MMU尚未激活，所以必须将__arch_info_begin和__arch_info_end的虚拟地址值变更为物理地址。和获取处理器信息的方法一样，通过下列代码求出虚拟地址和物理地址的偏移量后，将虚拟地址变更为相应的物理地址（请参考7.2.3节）。

```
sub r3, r3, r4                                          ❸
add r5, r5, r3          @__arch_info_begin地址
add r6, r6, r3          @__arch_info_end物理地址
```

寄存器r4中保存着位置计数器（.）的虚拟地址，r3中保存着标签3的物理地址。因此，两值之差将成为物理地址和虚拟地址之间的偏移量。最终，r5加偏移量即可获得与__arch_info_begin对应的物理地址，r6加偏移量即可获得与__arch_info_end对应的物理地址。这种方法和__lookup_processor_type中（将虚拟地址变更为物理地址时）使用的方法一致。

7.3.3 查找machine_desc结构体并比较机器信息

我们已经找到了与虚拟地址__arch_info_begin和__arch_info_end对应的物理地址，这样即可访问该地址空间中的machine_desc结构体，并进行比较。代码7-3的下列代码将执行该任务。

```
1:  ldr r3, [r5, #MACHINFO_TYPE]            @ r3 = 机器类型
    teq r3, r1                              @ 是否一致?
    beq 2f                                  @ 查找成功
    add r5, r5, #SIZEOF_MACHINE_DESC        @ 下一个machine_desc
```

```
    cmp r5, r6
    blo 1b
    mov r5, #0                                  @ 查找失败

2:  mov pc, lr
```

这段代码依次循环执行各machine_desc结构体，比较各machine_desc结构体的第一个成员变量nr和寄存器r1的值。

寄存器r1和比较对象r3的值是machine_desc结构体中移动#MACHINFO_TYPE大小后的位置值。#MACHINFO_TYPE定义如下。①

:: arch/arm/kernel/asm-offsets.c

```
DEFINE(MACHINFO_TYPE,   offsetof(struct machine_desc, nr));
```

这段代码在machine_desc结构体中求出对nr成员的偏移量。offsetof宏利用接收的参数值，返回该结构体中要查找的成员变量所在的偏移量（请参考7.1节“补充知识点”）。因此，寄存器r3中会保存machine_desc结构体的nr成员变量的值。

寄存器r3中有machine_desc结构体的nr的值，寄存器r1中有要查找的机器信息的值。如果两值一致，则表示找到了当前机器信息，并结束__lookup_machine_type逻辑；如果两值不一致，则访问并检查下一个machine_desc结构体。machine_desc结构体的地址会保存在寄存器r5中，若r5的地址值与储存__arch_info_end（.arch.info.init节的结束位置）的寄存器r6相比更大或一致，则表示查找机器类型失败，此时在寄存器r5中输入0，结束循环。

7.4　源自启动加载项的 atags——__vet_atags 标签

常量值ATAG_CORE的定义为0x54410001。该值是表示atag list启动的幻数（magic number）。对ATAG_CORE的说明请参考本节“补充知识点”。在_vet_atags中，确认启动加载项向第二个参数传递的标记列表（tagged list）的第一个要素是否为ATAG_CORE。若确认结果值不是ATAG_CORE，则将该参数值设置为0并返回。下面是详细代码。

:: 代码7-4　arch/arm/kernel/head-common.S的__vet_atags 标签

```
__vet_atags:
    tst r2, #0x3                        @ 检查是否以4字节为单位排序                    ❶
    bne 1f                              @ align未就绪时跳转到label 1

    ldr r5, [r2, #0]                    @ 读取atags指针的第一个值并存入r5              ❷
    subs    r5, r5, #ATAG_CORE_SIZE     @ 比较ATAG_CORE_SIZE和header的size值
    bne 1f                              @ 若不同则跳转到forward label 1
```

① 组成asm-offsets.c文件的代码生成汇编模块中使用的各类定义。

```
    ldr r5, [r2, #4]                @ 将atag的tag值（A_CORE）存入r5          ❸
    ldr r6, =ATAG_CORE
    cmp r5, r6                      @ 比较ATAG_CORE和tag
    bne 1f                          @ 若不同则跳转到forward label 1
    mov pc, lr                                                               ❹
1:  mov r2, #0                      @ atag信息与接收的值不同时               ❺
    mov pc, lr
ENDPROC(__vet_atags)
```

代码段❶的寄存器r2中保存着从启动加载项接收的atags地址值，检查该地址值是否以4字节为单位进行排序。若不是，则将atags的地址值设置为0并返回。

若atags的地址值正常排列，则在代码段❷中读取atags地址中保存的4字节值，存入寄存器r5。若该值与ATAG_CORE_SIZE值不同，则将r2(atags pointer)设置为0并返回。

之后，在代码段❸中读取atags的下一项（atags pointer+4）地址并存入寄存器r5。若该值与ATAG_CORE一致，则回到调用__vet_atags的位置；否则执行代码段❺，将atag地址设置为0并返回。

由于代码是用汇编语言编写的，看起来比较复杂。该标签执行的操作可简单理解为：对进入第三个参数的atag数据执行错误处理。

下面是将上述汇编代码用C语言重写之后的代码。仅掌握基础C语言的读者也可以看出，这段代码十分简单。

:: 代码7-5 arch/arm/kernel/head-common.S的__vet_atags-C语言版

```
void __vet_atags(struct tag* atag_list)
{
    if (atag_list & 0x3)
        goto clear_atags;

    if(!(atag_list->hdr.size == ATAG_CORE_SIZE && atag_list->hdr.tag == ATAG_CORE))
    {
        goto clear_atags;
    }

clear_atags:
    atag_list = 0;
    return ;
}
```



补充知识点 ATAGS

Linux内核从启动加载项接收3个参数。在ARM中遵循AAPCS（Procedure Call Standard for the ARM Architecture，ARM体系结构过程调用标准）标准时，少于4个的参数依次分配在r0、r1、r2、r3之中，对于5个以上的参数，其前4个参数依然从r0到r3依次进行分配，而之后的参

数则进入栈。因此，从启动加载项接收的参数在寄存器中设置如下。

```
r0 = 0
r1 =结构ID
r2 = tagged list 指针
```

tagged list由struct teg数组组成，包含内存、视频、serial、initrd、revision、cmdline等信息。如图7-4所示，tagged list从ATAG_CORE开始启动，用ATAG_NONE表示数组结束。

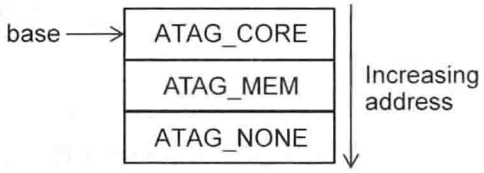


图7-4　tagged list结构

tagged list不得被内核解压器（decompressor）或bootp程序覆写（overwrite），因此，主要位于RAM的第一个16 KB。

在u-boot下可查看bootm.c的setup_start_tag函数，以及setup_end_tag中设置ATAG_CORE和ATAG_NONE的代码。

```
struct tag {
        struct tag_header hdr;
        union {
                struct tag_core             core;
                struct tag_mem32            mem;
                struct tag_videotext        videotext;
                struct tag_ramdisk          ramdisk;
                struct tag_initrd           initrd;
                struct tag_serialnr         serialnr;
                struct tag_revision         revision;
                struct tag_videolfb         videolfb;
                struct tag_cmdline          cmdline;

                /*
                * Acorn specific
                */
                struct tag_acorn            acorn;

                /*
                * DC21285 specific
                */
                struct tag_memclk           memclk;

                struct tag_mtdpart          mtdpart_info;
        } u;
};
```

补充知识点 使用AAPCS

AAPCS中定义了同时使用单独编译及汇编的多个模块时需要遵守的寄存器使用方法及栈规则，也有在默认标准基础上略加变形的各个版本。arm-gcc中也有多个变形版本，不过当前嵌入式系统中普遍遵循EABI（Embedded Application Binary Interface，嵌入式应用程序二进制接口）。

为ARM处理器编写代码时，要选择适当的AAPCS变形。

Linux内核中也有直接用汇编语言编写的部分，但定义了附加条件以支持多种API。

7.5 对虚拟内存进行基础创建——__create_page_tables 标签

在一般的32位操作系统中，处理器具有4 GB的虚拟内存。当然，PC市场上64位操作系统也十分活跃，但ARM的运行环境依然是32位。因此，我们以32位环境进行说明。尽管可以安装4 GB内存，只运行1个处理器来听音乐或浏览网页，但我们通常使用的是4 GB以内的物理内存。

换言之，4 GB以内的物理内存中，多个进程各自拥有4 GB的虚拟内存。这怎么可能？即使拥有4 GB的虚拟地址空间，也并非使用所有的4 GB内存。也许我们会认为4 GB区域都是自己的，但实际上，操作系统将需要的容量适当分配给物理内存。因此，使用页表（page table）链接虚拟地址和物理地址提起虚拟内存，会经常涉及页和框架的概念。二者都是将内存按一定大小分配后进行管理的单位，从物理内存的角度上称为框架（frame），从虚拟内存的角度上称为页（page）。

图7-5表示2个进程的虚拟内存和物理内存之间的关系。

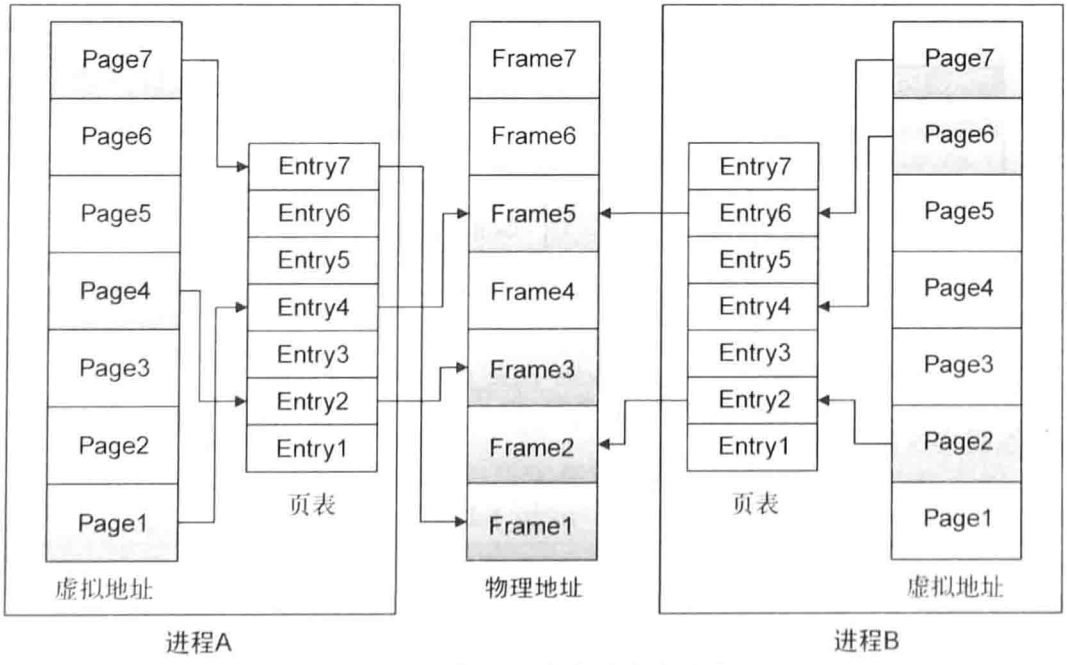


图7-5 连接虚拟内存和物理内存

A进程具有4GB的虚拟地址空间，同时具有只属于A进程的页表。Page 7利用页表信息，被分配到物理地址空间Frame 1；Page 4同样利用页表信息，被分配到Frame 3。B进程也具有只属于B进程的页表，利用自身的页表信息，Page 2被分配到Frame 2。

有趣的是，观察Frame 5可发现，A进程的Page 1和B进程的Page 7都处于连接状态。它们是Linux的共享对象（shared object）时，会在多个进程中同时使用。但与“复制到其他物理空间进行分配”相比，共享相同物理地址可减少内存使用频率，缩短分配时间。

现在观察内核中临时设置使用的页表代码。实际内核中使用的页表将在start_kernel后的第14章进行详细说明。

__create_page_tables中，在与KERNEL_RAM_PADDR相距0x4000的位置（KERNEL_RAM_PADDR - 0x4000）到KERNEL_RAM_PADDR的所有页表项执行循环，并初始化为0。对相当于内核区域的项设置节区基址和cacheable、buffereable值。执行__create_page_tables后的页表如图7-6所示。

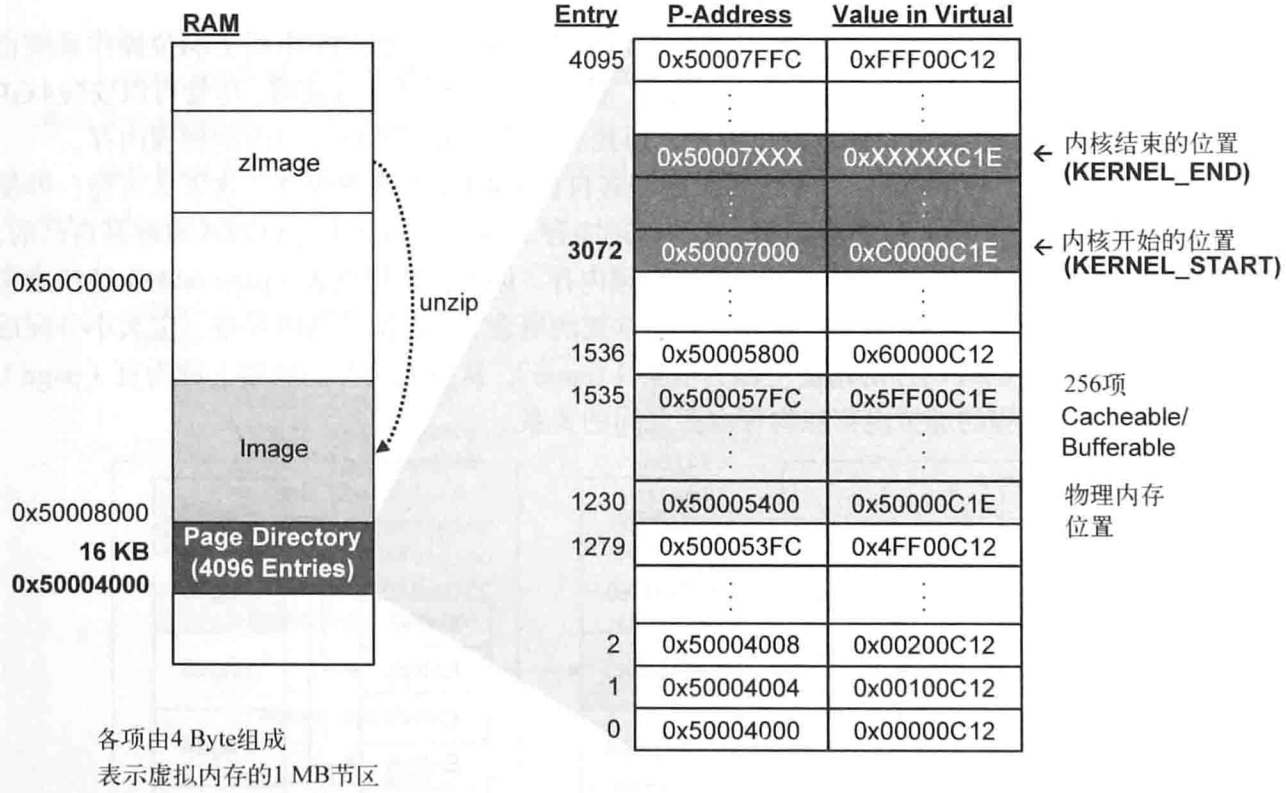


图7-6　执行__create_page_tables后的页目录

现在仔细观察这段代码。

:: 代码7-6　arch/arm/kernel/head.S的__create_page_tables标签

```
__create_page_tables:
    pgtbl    r4                                                    ❶

    mov r0, r4                                  @ r0 = 0x50004000
```

```
    mov r3, #0
    add r6, r0, #0x4000                             @ r6 = 0x50008000

1:  str r3, [r0], #4                                @ addr[0x50004000] = 0      ❷
    str r3, [r0], #4                                @ 地址增加4并继续执行
    str r3, [r0], #4                                @ 初始化为0
    str r3, [r0], #4

    teq r0, r6
    bne 1b

    ldr r7, [r10, #PROCINFO_MM_MMUFLAGS]            @ 输入mm_mmuflags值         ❸

    mov r6, pc, lsr #20
    orr r3, r7, r6, lsl #20
    str r3, [r4, r6, lsl #2]

    add r0, r4,  #(KERNEL_START & 0xff000000) >> 18   @ r0 = r4+ 0x3000        ❹
    str r3, [r0, #(KERNEL_START & 0x00f00000) >> 18]! @ [r0+0]
    ldr r6, =(KERNEL_END - 1)

    add r0, r0, #4
    add r6, r4, r6, lsr #18

1:  cmp r0, r6                                                                  ❺
    add r3, r3, #1 << 20
    strls   r3, [r0], #4

    bls 1b


    add r0, r4, #PAGE_OFFSET >> 18                  @ r0 = r4+0x3000
    orr r6, r7, #(PHYS_OFFSET & 0xff000000)         @ r6 = 0x50000C1E
    .if (PHYS_OFFSET & 0x00f00000)
    orr r6, r6, #(PHYS_OFFSET & 0x00f00000)
    .endif
    str r6, [r0]                                    @ addr[0x50007000] = r6

    mov pc, lr
ENDPROC(__create_page_tables)
```

代码段❶中pgtbl宏定义如下。

:: arch/arm/kernel/head.S

```
#define KERNEL_RAM_PADDR    (PHYS_OFFSET + TEXT_OFFSET)
.macro pgtbl, rd
ldr \rd, =(KERNEL_RAM_PADDR - 0x4000)
```

pgtbl是用于接收参数rd的宏，而rd中加载KERNEL_RAM_PADDR-0x4000的地址值。并且，物理内存的起始地址+内核代码起始地址的偏移量=KERNEL_RAM_PADDR。

换言之，KERNEL_RAM_PADDR就是内核起始地址的物理地址，rd中设置的是从该地址减去0x4000后的值。该区域将成为页表的起始地址。

利用页表的起始地址，在代码段❷中对从KERNEL_RAM_PADDR-0x4000到KERNEL_RAM_PADDR的所有页表项执行循环，并将其初始化为0。

代码段❸的寄存器r10中保存着proc_info_list结构体的地址，读取r10加上PROCINFO_MM_MMUFLAGS后的地址值，并将其保存到r7。PROCINFO_MM_MMUFLAGS和MACHINFO_TYPE（参考7.3.3节）用相同方法进行了定义，并成为proc_info_list结构体的成员变量__cpu_mm_mmu_flag的偏移量。

r7中设置了proc_info_list结构体的成员变量__cpu_mm_mmu_flags的值。__cpu_mm_mmu_flags值以proc-v7.S为基准，如下所示。

:: arch/arm/mm/proc-v7.S

```
PMD_TYPE_SECT ¦ PMD_SECT_BUFFERABLE ¦ PMD_SECT_CACHEABLE ¦
PMD_SECT_AP_WRITE ¦PMD_SECT_APE_READ
```

下列代码就是对该值的定义。

:: arch/arm/include/asm/pgtable-hwdef.h

```
#define PMD_TYPE_SECT                   (2 << 0)
#define PMD_SECT_BUFFERABLE             (1 << 2)
#define PMD_SECT_CACHEABLE              (1 << 3)
#define PMD_SECT_AP_WRITE               (1 << 10)
#define PMD_SECT_AP_READ                (1 << 11)
#define PMD_SECT_BUFFERED               (PMD_SECT_BUFFERABLE)
```

赋值后如下所示。

```
(2<<0)¦(1<<2)¦(1<<3)¦(1<<10)¦(1<<11)
```

最终，r7中设置了0xCIE值。将该值加上内核基址（顶端12位）后的值记录在页表项的第一个位置。

观察代码段❹可以发现KERNEL_START和KERNEL_END宏，在其内部还需要使用其他的宏。宏的定义如下图所示。

PAGE_OFFSET是RAM的起始地址（虚拟地址），TEXT_OFFSET是从RAM的起始地址进入实际内核的偏移量。假设各自的地址是0Xc0000000和0x8000，KERNEL_START将成为0xC0008000。KERNEL_END被定义为_end，该值在vmlinu.lds.S中指向内核代码段的最终地址。

使用已定义的宏KERNEL_START，获得相当于内核代码起始位置和最后位置的页目录项地址，分别存入寄存器r0、r6。在r3中临时保存需要保存到页目录项的值。

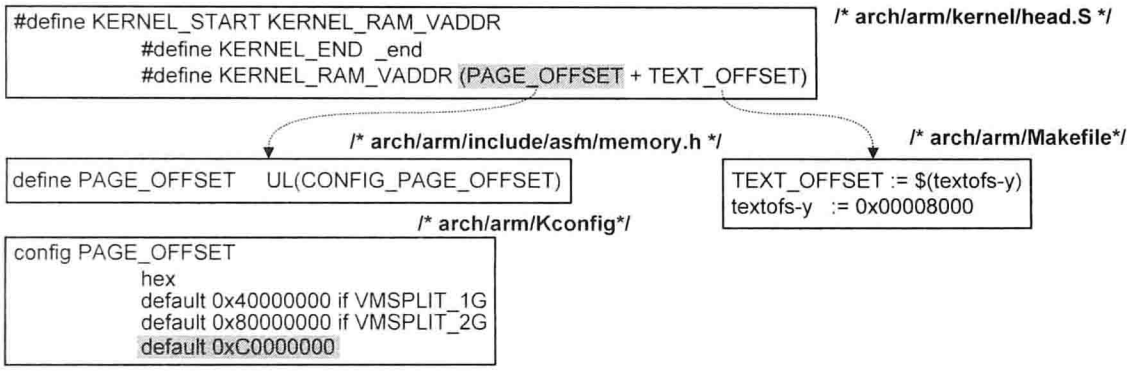


图7-7 KERNEL_START定义

在代码段❺中使用上面设置的值，在从KERNEL_START到KERNEL_END的页表项区域执行循环，用C0000C1E ~ XXXXXXXC1值进行设置。设置的值如图7-6所示，对项设置值的说明请参照5.4节。

7.6 设置核心（core）——__v6_setup 标签

完成7.5节的__create_page_tables设置后，运行__v6_setup程序设置当前处理器。参见代码7-1代码段❺中调用处理器初始化函数的部分。根据不同的ARM结构，处理器初始化函数执行过程也不一样。v6时调用__v6_setup、v7时调用__v7_setup，调用后即执行初始化任务。下列代码表示对应于ARM v6结构的__v6_setup代码。

:: 代码7-7 arch/arm/kernel/head.S的stext标签

```
__v6_setup:

#ifdef CONFIG_SMP                                                    ❶
    mrc     p15, 0, r0, c1, c0, 1         @ 设置为SMP或nAMP
    orr     r0, r0, #0x20
    mcr     p15, 0, r0, c1, c0, 1
#endif

    mov     r0, #0                                                   ❷
    mcr     p15, 0, r0, c7, c14, 0        @ D-Cache清理数据缓存并使之失效
    mcr     p15, 0, r0, c7, c5, 0         @ I-Cache使指令缓存失效
    mcr     p15, 0, r0, c7, c15, 0        @ 清理缓存并使之失效
    mcr     p15, 0, r0, c7, c10, 4        @ 清空写缓冲
#ifdef CONFIG_MMU                                                    ❸
    mcr     p15, 0, r0, c8, c7, 0         @ 使I、D TLB失效
    mcr     p15, 0, r0, c2, c0, 2
    orr     r4, r4, #TTB_FLAGS            @ r4 = 0x5000400A
    mcr     p15, 0, r4, c2, c0, 1         @ page table 传递页表起始地址

#endif /* CONFIG_MMU */

    adr     r5, v6_crval                                             ❹
```

```
ldmia  r5, {r5, r6}
mrc    p15, 0, r0, c1, c0, 0
bic    r0, r0, r5
orr    r0, r0, r6
mov    pc, lr
```

使用由2个或2个以上的相同处理器组成的多处理器时，代码段❶将激活SMP模式，并启动SCU（Snoop Control Unit，窥探控制单元）。SCU是内存和核心之间用于保持各核心数据一致性的装置。

代码段❷对（分为指令缓存和数据缓存的）哈佛体系结构和混合结构都可以执行初始化。清理所有缓存并使之失效后，清空写缓冲。清空操作中使用的协处理器命令是数据同步障碍（data synchronized barrier），上述命令均适用于协处理器后执行下一命令。

代码段❸中，使TLB失效后，在寄存器TTB1中保存页表起始地址。最后，代码段❹仅将MMU、缓存、分离预测、功能等的设置值保存到寄存器r0，并跳转到__enable_mmu。对寄存器r0的详细说明请看下一节。



补充知识点　什么是SMP?

SMP（Symmetric Multi-Processing，对称多处理系统）是多处理器计算机结构，由2个以上的相同处理器组成，所有处理器共享同一内存。多核心计算机结构与此相同。

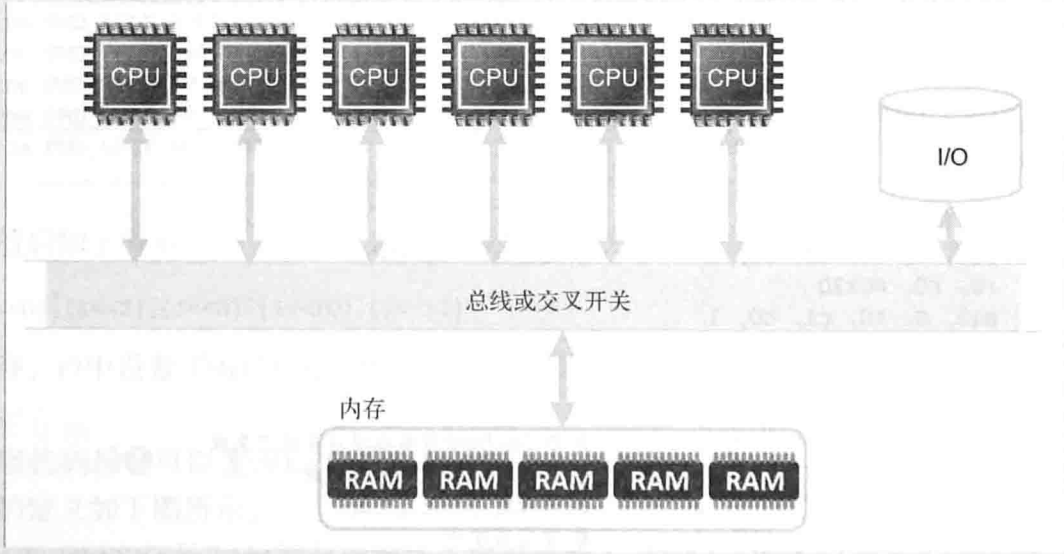


图7-8　SMP结构

7.7　打开 MMU 并使用虚拟地址——__enable_mmu/__turn_mmu_on 标签

在代码7-1讲解的stext程序中，调用__v6_setup子程序前，按如下方式将__enable_mmu地址设

置到寄存器lr。

:: arch/arm/kernel/head.S

```
ENTRY(stext)
...
adr     lr, __enable_mmu
```

运行__v6_setup后，移动到lr中保存的地址，依次执行__enable_mmu和__turn_mmu_on标签。

为了控制MMU的运行，使用协处理器15（CP15）的各个寄存器，控制寄存器CP15 c1寄存器将履行MMU系统控制中默认寄存器的作用。c1寄存器的部分位域如表7-1所示。这些值定义在源代码arch/arm/include/asm/system.h文件中，后续章节将对该部分进行详细说明。

表7-1 控制MMU运行的CP15 c1寄存器的位域

位	简 写	功 能	含 义
0	M	MMU	0 : disable / 1: enable
2	C	D-Cache	0 : disable / 1: enable
3	W	写缓冲	0 : disable / 1: enable
8	S	系统	与ROM位结合，具有页访问权限
9	R	ROM	与系统位结合，具有页访问权限
12	I	指令缓存	0 : disable / 1: enable
13	V	高向量表	0 : 在0x00000000放置向量表 1 : 在0xFFFF0000放置向量表

通过代码7-8仔细观察__enable_mmu运行方式。

:: 代码7-8 arch/arm/kernel/head.S的__enable_mmu标签

```
__enable_mmu:
#ifdef CONFIG_ALIGNMENT_TRAP                                          ❶
    orr     r0, r0, #CR_A
#else
    bic     r0, r0, #CR_A
#endif
#ifdef CONFIG_CPU_DCACHE_DISABLE
    bic     r0, r0, #CR_C
#endif

...
    mov     r5, #(domain_val(DOMAIN_USER, DOMAIN_MANAGER) |    \      ❷
                  domain_val(DOMAIN_KERNEL, DOMAIN_MANAGER) | \
                  domain_val(DOMAIN_TABLE, DOMAIN_MANAGER) |  \
                  domain_val(DOMAIN_IO, DOMAIN_CLIENT))
    mcr     p15, 0, r5, c3, c0, 0           @ 向域访问寄存器加载      ❸
    mcr     p15, 0, r4, c2, c0, 0           @ 加载页表指针            ❹
    b       __turn_mmu_on
ENDPROC(__enable_mmu)
```

在__enable_mmu程序中，设置所需要的位包以激活MMU硬件，将域访问寄存器和页表指针值加载到ARM处理器的寄存器。首先在代码段❶中为寄存器r0设置相当于CR的位，该设置必须符合内核设置标准。将这里设置的r0寄存器适用于下次执行的__turn_mmu_on标签。CR_位在arch/arm/include/asm/system.h中定义如下。

:: arch/arm/include/asm/system.h

```
/*
 * CR1 bits (CP#15 CR1)
 */
#define CR_M    (1 << 0)    /*激活MMU*/
#define CR_A    (1 << 1)    /*激活Alignment abort*/
#define CR_C    (1 << 2)    /*激活数据缓存*/
#define CR_W    (1 << 3)    /*激活写缓冲*/
#define CR_P    (1 << 4)    /*32位异常处理器*/
#define CR_D    (1 << 5)    /*32位数据地址范围*/
...
```

定义的值指向用于设置该属性的特定位地址。例如，假设代码段❶中定义了DCACHE_DISABLE属性。system.h中，相当于激活数据缓存的CR_C的位定义为(1<<2)。因此，bic r0,ro,#CR指令会清空（bic）CP15 c1寄存器的第二个位，也就是将1向左移动2次之后的值，从而禁用数据缓存。

在代码段❷中设置相当于域的位。ARM处理器使用“域”（domain）管理内存访问权限。域将虚拟内存区域分为特定个数的区域，为每个域区域赋予访问权限以执行控制。

如图7-9所示，位域使用L1页表项的第5号到第8号的4个位，并分为2^4 = 16个。

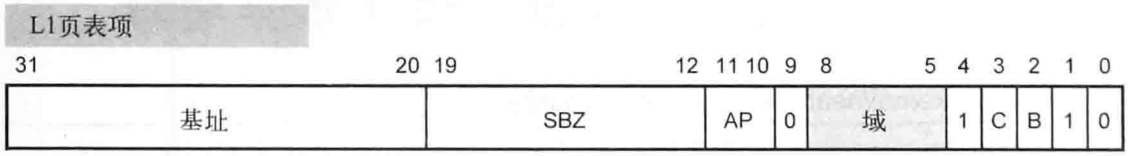


图7-9　域位

域的访问权限分配在图7-10所示的CP15的c3寄存器中，使用2个位对16个域中的每个域定义许可权限。

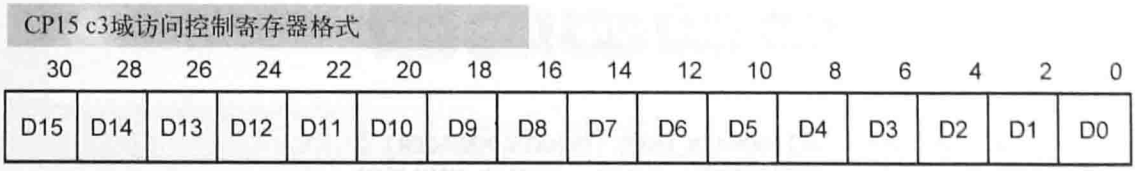


图7-10　域访问控制寄存器

定义该权限的2个位含义参见表7-2。

表7-2 域访问权限位

访 问	位 值	说 明
manager	11	无法进行访问控制，也不允许发生abort
client	01	根据页表项内设置的核准值进行访问控制

代码段❷的domain_val(A,B)宏用于将A向左移动2×B位，其格式如下。

:: arch/arm/include/asm/domain.h

```
#define domain_val(dom,type)       ((type) << (2*(dom)))
```

根据宏的定义，将domain_val部分具体展开后执行如下运算。

```
(domain_val(DOMAIN_USER, DOMAIN_MANAGER) | domain_val(DOMAIN_KERNEL, DOMAIN_MANAGER) |
domain_val(DOMAIN_TABLE, DOMAIN_MANAGER) | domain_val(DOMAIN_IO, DOMAIN_CLIENT))
  -> (3 << 2*1) | (3 << 2*0) | (3 << 2*0) | (1 << 2*2)
  -> (00001100) | (00000011) | (00000011) | (00010000)
  -> (00011111)
```

最后的0x1F值存入r5。在代码段❸中，将如上设置的寄存器r5的域访问权限位值读入域访问寄存器CP15的第3个寄存器c3。在代码段❹中，为TTB（Translation Table Base，转换表）寄存器加载临时页表的起始位置，并分为下一标签，也就是代码7-9的__turn_mmu_on。

:: 代码7-9 arch/arm/kernel/head.S的__turn_mmu_on标签

```
.align  5
__turn_mmu_on:
    mov       r0, r0
    mcr       p15, 0, r0, c1, c0, 0        @写入控制寄存器c1                    ❶
    mrc       p15, 0, r3, c0, c0, 0        @id
    mov       r3, r3                                                          ❷
    mov       r3, r3
    mov       pc, r13                                                         ❸
ENDPROC(__turn_mmu_on)
```

在__enable_mmu中，我们为寄存器r0设置了与MMU硬件相关的位。在__turn_mmu_on标签中，将已设置的寄存器r0值加载到CP15控制寄存器，从而激活实际MMU硬件，使虚拟地址可用。

代码7-8中，为寄存器r0设置了CP15寄存器c1需要的位值。代码段❶中，将r0寄存器值加载到控制寄存器c1，从而激活MMU。如前所述，c1是控制MMU系统的默认寄存器。

将值加载到c1寄存器后，在代码段❷中执行（负责同一寄存器内移动的）NOP运算，其作用是对这些值进行排序，使下面出现的指令在激活MMU后能够在同一缓存线上执行。最后在代码段❸中，跳转到__mmap_switched地址（相当于代码7-1的stext程序中加载的r13寄存器的值）。



补充知识点　ADR和MRC/MCR指令

```
ADR{L}<cond> Rd, <address>          MACRO
```

在Rd中保存<address>值的汇编器宏，计算pc减去相对地址的结果。

```
MRC ¦ MCR{<cond>}, cp, opcode1, Rd, Cn, Cm {, opcode2}
```

协处理器寄存器传送指令。MRC负责将数据从协处理器移动到ARM寄存器，MCR负责将数据从ARM寄存器移动到协处理器。cp是协处理器号，Cn和Cm是协处理器的默认寄存器和辅助寄存器，具有0~15的编号。

7.8　跳转至 start_kernel——__mmap_switched 标签

从__mmap_switched标签开始，MMU处于激活状态，代码通过非PIC的绝对地址执行。从start_kernel函数开始，代码由C语言编写而成，因此需要__switch_data标签中设置的data、bss、stack值，并将调用start_kernel后要使用的信息设置到__switch_data。下列代码表示__switch_data包含的数据。

:: arch/arm/kernel/head-common.S

```
__switch_data:
    .long    __mmap_switched
    .long    __data_loc                          @ r4
    .long    _data                               @ r5 (data值)
    .long    __bss_start                         @ r6 (bss值)
    .long    _end                                @ r7
    .long    processor_id                        @ r4
    .long    __machine_arch_type                 @ r5
    .long    __atags_pointer                     @ r6
    .long    cr_alignment                        @ r7, 保存cr_alignment的地址
    .long    init_thread_union + THREAD_START_SP  @ sp (栈指针值)
```

通过下列代码正式了解__mmap_switched标签中调用start_kernel的过程。

:: 代码7-10　arch/arm/kernel/head-common.S的__mmap_switched标签

```
__mmap_switched:
    adr  r3, __switch_data + 4
    ldmia r3!, {r4, r5, ,r6 , r7}                  ❶
    cmp r4, r5
    cmpne r5, r6                                   ❷
    ldrne fp, [r4], #4
    strne fp, [r5], #4
    bne 1b
```

```
    ldmia r3, {r4, r5, r6, r7, sp}                                    ❸
    str r9, [r4]

    str r1, [r5]                                                      ❹
    str r2, [r6]
    mov fp, #0
1:  cmp r6, r7
    strcc fp, [r6], #4
    bcc 1b
    bic r4, r0, #CR_A
    stmia r7, {r0, r4}
    b start_kernel
ENDPROC(__mmap_switched)
```

在代码段❶中，将__data_loc、_data、__bss_start、_end变量中保存的地址存入寄存器r4、r5、r6、r7。

在代码段❷中，将__data_loc、_data的值进行比较，如果__data_loc和_data的值不一致，则复制数据段。__data_loc指向二进制文件（binary file）中初始化数据（initialized data）区域的起始位置，_data指向内存中初始化数据区域的起始位置。由于内核在ROM中通过XIP（eXecute In Place，芯片内执行）执行时无法修改数据，因此将数据区域复制到RAM，使其能够修改。

在代码段❸中，从__switch_data的processor_id到THREAD_START_SP保存的地址按如下方式读入寄存器。若执行start_kernel，则运行静态声明的init线程，所以在sp寄存器中设置该线程的栈指针。

```
processor_id                        -> r4
__machine_arch_type                 -> r5
atags_porinter                      -> r6
cr_alignment                        -> r7
init_thread_union + THREAD_START_SP -> sp
```

在代码段❹中，将之前求出的处理器ID和机器信息、atags指针保存到__switch_data。由于BSS区域声明了全局变量，故会保存未初始化的数据，所以启动时必须将该区域初始化为0。

为了控制MMU运算，将A域以外的CP15 c1寄存器值保存到cr_no_alignment变量。至于已设置的A域的值，则另行保存到cr_alignment值。

cr_alignment和cr_no_alignment定义在arch/arm/kernel/entry-armv.S中。CP15 c1寄存器的A域决定是否允许数据地址对齐错误（data address alignment fault）。该域设置为“1”，若未对齐访问数据的地址，会发生Data Abort 陷阱（trap）。CP15 c1寄存器的格式如图7-11所示。

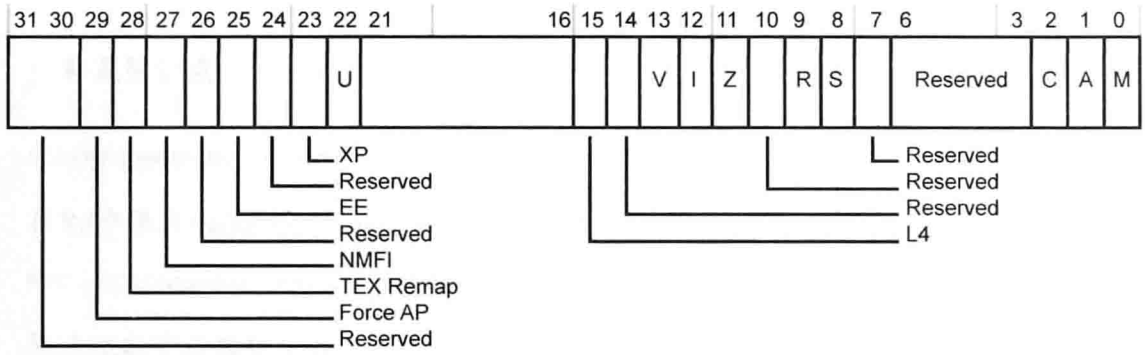


图7-11　CP15 c1寄存器格式

这样就完成了在ARM处理器中执行C代码所需的默认设置。通过代码段❺跳转到start_kernel函数即可执行init线程，并使用虚拟地址。

第三部分

内核的执行

——内核的起始与结束位置

这一部分着重分析内核的主函数——start_kernel() 函数，仅 start_kernel() 函数能够直接调用的函数就达 100 多个。start_kernel() 函数可以执行从属于平台和 CPU 的初始化、多种子系统的初始化、设备驱动器初始化等。另外，构建启动时使用的内存分配器后，初始化到一定程度时，它还可构建内核的内存分配器。

start_kernel() 函数生成 init 进程和内核线程之后，自己则成为 idle 进程。下列代码是 start_kernel() 函数的定义部分，通过 asmlinkage 可从汇编代码调用 start_kernel() 函数。如前一部分所讲，在 head-common.S 文件的 __mmap_switched 标签中调用 start_kernel() 函数。

:: init/main.c

```
asmlinkage void __init start_kernel(void)
{
    ...
}
```

图 PART Ⅲ-1 描述了 start_kernel() 函数调用的各函数在上层的调用关系。

第8章 start_setup_processor_id() ~lock_kernel()

本章将对在start_kernel()内部调用的函数（从smp_setup_processor_id()函数到lock_kernel()函数）逐一进行讲解。图8-1简要说明了各函数执行的任务。

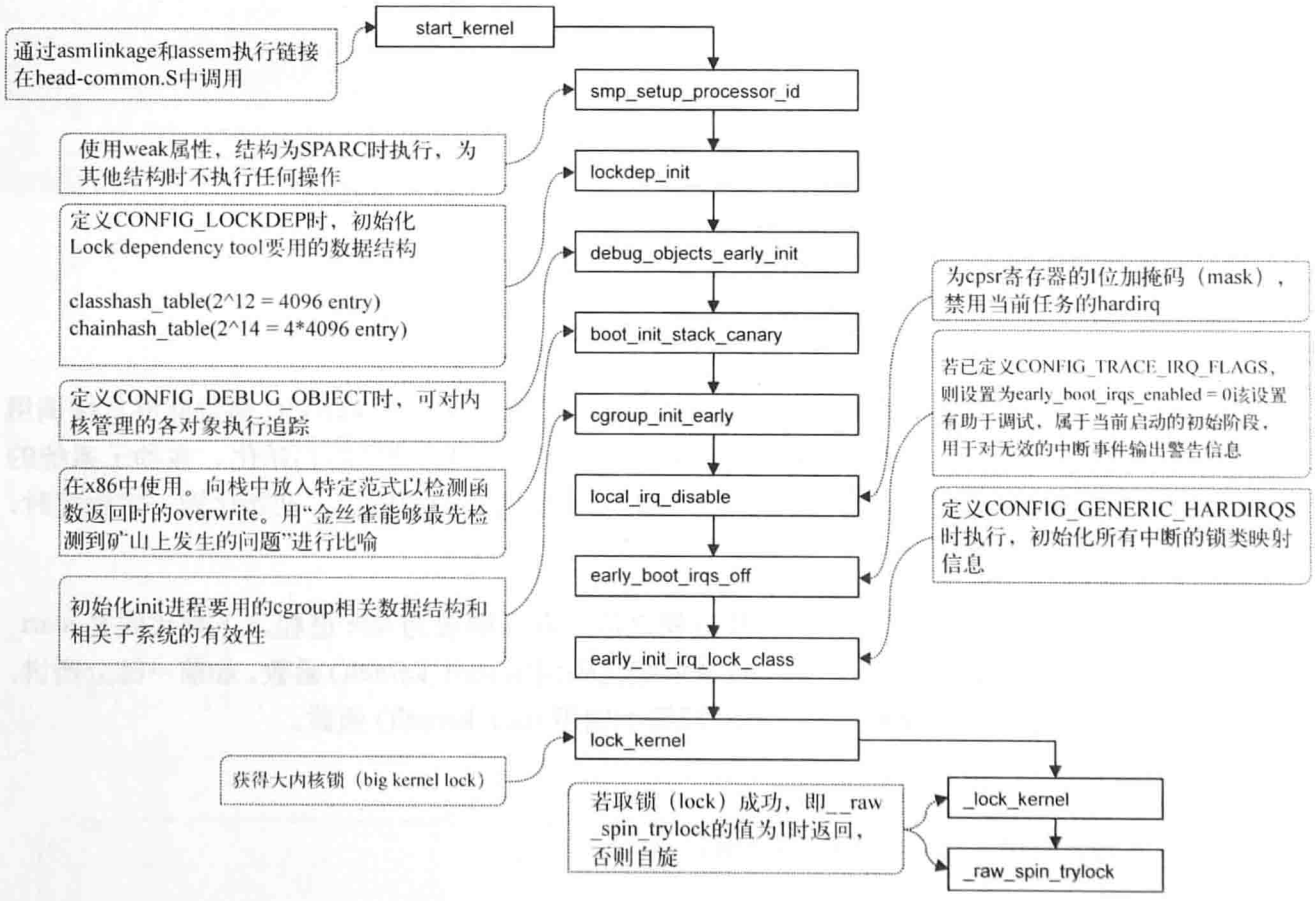


图8-1　strat_kernel()函数内调用的初始函数

作为内核初始化阶段调用的函数，它们主要用来进行调试相关设置。因此，未进行相关环境设置的情况下，大部分不执行任何操作。例如，未定义CONFIG_DEBUG_OBJECTS时，无法调

用debug_objects_early_init()函数。

8.1 smp_setup_processor_id()、lockdep_init()、debug_objects_early_init()

下面简单介绍一下这3个函数。

8.1.1 smp_setup_processor_id()

smp_setup_processor_id()的weak属性定义如下。

:: init/main.c

```
void __init __weak smp_setup_processor_id(void)
{
}
```

如上所示，如果有未定义weak属性的smp_setup_processor_id()函数，说明这是不执行的函数。在内核代码中结构为SPARC时，定义该函数。因此，ARM结构时不执行任何操作。

8.1.2 lockdep_init()

接下来了解lockdep_init函数。根据CONFIG_LOCKDEP定义与否，该函数执行的代码会有所不同。未定义CONFIG_LOCKDEP时，调用下列代码，不执行任何操作。

:: include/linux/lockdep.h

```
#define lockdep_init()                do { } while (0)
```

CONFIG_LOCKDEP用于调试，在Kconfig.debug文件中有如下记述。

:: lib/Kconfig.debug

```
config LOCKDEP
    bool
    depends on DEBUG_KERNEL && TRACE_IRQFLAGS_SUPPORT && STACKTRACE_SUPPORT && LOCKDEP_SUPPORT
    select STACKTRACE
    select FRAME_POINTER if !MIPS && !PPC && !ARM_UNWIND && !S390
    select KALLSYMS
    select KALLSYMS_ALL
```

为了进行调试，lockdep_init()函数对记录各锁之间依赖关系的结构体进行初始化，其定义如代码8-1所示。

:: 代码8-1　kernel/lockdep.c的lockdep_init()

```
void lockdep_init(void)
{
    int i;


    if (lockdep_initialized)                                    ❶
        return;

    for (i = 0; i < CLASSHASH_SIZE; i++)                        ❷
        INIT_LIST_HEAD(classhash_table + i);

    for (i = 0; i < CHAINHASH_SIZE; i++)                        ❸
        INIT_LIST_HEAD(chainhash_table + i);

    lockdep_initialized = 1;
}
```

代码段❶中，若设置lockdep_initiallized，则退出这段代码，不执行下列操作。这是因为，特定结构中由于具有自己的start_kernel函数，可能已经调用了lockdep_init()函数。

代码段❷和代码段❸中，将初始化struct list_head类型数组classhash_table和chainhash_table数组。这2个数组及其大小定义如下。

:: include/linux/lockdep.h

```
#define MAX_LOCKDEP_KEYS_BITS     13
```

:: kernel/lockdep_internals.h

```
#define MAX_LOCKDEP_CHAINS_BITS     15
```

:: kernel/lockdep.c

```
#define CLASSHASH_BITS      (MAX_LOCKDEP_KEYS_BITS - 1)
#define CLASSHASH_SIZE      (1UL << CLASSHASH_BITS)

static struct list_head classhash_table[CLASSHASH_SIZE];
#define CHAINHASH_BITS      (MAX_LOCKDEP_CHAINS_BITS-1)

static struct list_head chainhash_table[CHAINHASH_SIZE];
```

最终，初始化classhash_table[4096]、chainhash_table[16384]项。

8.1.3　debug_objects_early_init()

接下来了解debug_objects_early_init()函数。该函数与lockdep_init函数原理相同，未定义CONFIG_DEBUG_OBJECTS时，调用下列代码，不执行任何操作。

:: include/linux/debugobjects.h

```
static inline void debug_objects_early_init(void) { }
```

CONFIG_DEBUG_OBJECTS用于调试，在Kconfig.debug文件中有如下记述。

:: lib/Kconfig.debug

```
config DEBUG_OBJECTS
    bool "Debug object operations"
    depends on DEBUG_KERNEL
    help
        If you say Y here, additional code will be inserted into the
        kernel to track the life time of various objects and validate
        the operations on those objects.
```

debug_objects_early_init()函数对需要追踪各对象的结构体进行初始化，其定义如代码8-2所示。

:: 代码8-2 lib/debugobjects.c的debug_objects_early_init()

```
void __init debug_objects_early_init(void)
{
    int i;

    for (i = 0; i < ODEBUG_HASH_SIZE; i++)                    ❶
        spin_lock_init(&obj_hash[i].lock);

    for (i = 0; i < ODEBUG_POOL_SIZE; i++)                    ❷
        hlist_add_head(&obj_static_pool[i].node, &obj_pool);
}
```

代码段❶中，初始化struct debug_bucket类型的obj_hash[]数组所有元素的lock成员变量。

代码段❷中，将struct debug_obj类型的object_static_pool[]数组所有node成员连接到obj_pool散列表。

下列代码是对这2个结构体的声明。

:: lib/debugobjects.c

```
#define ODEBUG_HASH_BITS    14
#define ODEBUG_HASH_SIZE    (1 << ODEBUG_HASH_BITS)

#define ODEBUG_POOL_SIZE          512
#define ODEBUG_POOL_MIN_LEVEL     256

static struct debug_bucket  obj_hash[ODEBUG_HASH_SIZE];
static struct debug_obj     obj_static_pool[ODEBUG_POOL_SIZE] __initdata;
```

:: include/linux/debugobjects.h

```
struct debug_obj {
    struct hlist_node       node;           /*链接跟踪器列表中的对象*/
    enum debug_obj_state    state;          /*对象状态 */
    void                    *object;        /*对实际对象的指针 */
    struct debug_obj_descr  *descr;         /*用于调试的描述符结构体指针 */
};
```

:: lib/debugobjects.c

```
struct debug_bucket {
    struct hlist_head   list;
    spinlock_t      lock;
};
```

该函数将初始化用于调试的数据结构，在此我们不会对该函数进行详细阐述。但各位需要知道，通过该函数初始化的数据结构可追踪针对对象和各对象生存时间执行的运算。

8.2　栈溢出感应——__boot_init_stack_canary

用于发现栈缓冲溢出攻击，启动时设置栈中的canary值。[①]它在ARM结构中没有实现，在x86结构中实现了。由于没有在ARM结构中实现，所以不会对代码进行分析。但canary值与栈溢出攻击相关，因此我们先简单了解什么是canary值。

人们通常采用2种方法以防恶意的栈缓冲溢出攻击：第一种方法是，发生栈缓冲溢出时将其检测出来，从而阻止通过恶意代码更改指令指针；第二种方法是，未直接检测栈缓冲溢出的情况下，预防恶意代码攻击。

第一种方法中有种叫stack canaries的技术，在恶意代码执行前，用该技术发现栈缓冲溢出。由于和矿工为了检测矿山风险而使用金丝雀的方法类似，故该技术被命名为stack canaries。

程序启动时，stack canaries将随机选择的小整数值保存在返回地址保存位置之前。大部分缓冲溢出攻击是从内存地址的低位向高位覆写内存，所以，为了覆写返回地址，必须覆写canary值。

因此，可使用返回地址在返回前查看canary值，从而确认是否发生缓冲溢出攻击。

8.3　初始化提供进程集成方法的 cgroup——__cgroup_init_early()

cgroup集成各进程，为进程组合提供分配资源或进行控制的功能。确切地说，cgroup是只提供进程组合方法的一种机制，控制进程组或分配资源的作用由子系统负责。

---

① 美国的煤矿工人为了在开工初期检测有毒气体，在矿山上放置了金丝雀（canary）。这种鸟的呼吸系统结构独特，对空气中的有毒气体十分敏感。因此，矿工将它先放入矿山，以判断是否存在有毒气体。

表8-1 主要子系统的种类和作用

名　称	作　用
cpu	指定进程组能使用的CPU
memory	指定进程组能使用的内存量，并编写内存使用量报告
cpuset	指定进程组能使用的各CPU和节点集
freezer	可停止进程组状态或重新启动
cpuacct	对进程组使用的CPU资源编写报告
blkio	设置对模块设备的输入输出限制
devices	设置进程组能使用的设备
nel_cls	为数据包贴上（tag）classid，使之能够识别Linux的流量控制器是从哪一组获得的

对cgroup进行的初始化分为cgroup_init_early()和cgroup_init()这2次。这么做的原因是，需要对启动初始阶段使用的子系统率先进行初始化。分析函数之前，我们先观察cgroup中使用的几种数据结构。

1. cgroup_subsys_state结构体：具有特定cgroup中适用的子系统设置

子系统连接到cgroup后，会把对已连接子系统进行的设置另行适用到连接的每个cgroup上，为了保存这些设置而导入的数据结构就是cgroup_subsys_state结构体。适用的每个子系统都有1个cgroup_subsys_state结构体，并且与cgroup结构体的subsys连接成列表格式，如下图所示。cgroup_subsys_state可以说是基于cgroup的数据结构。

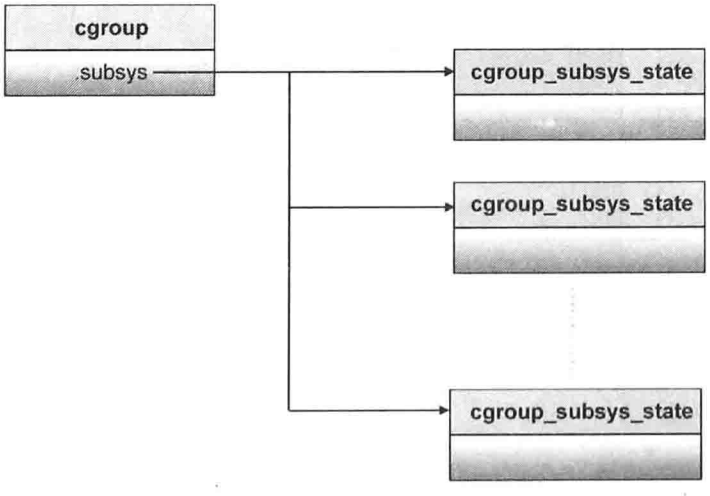


图8-2 cgroupfs_subsys_state和cgroup的关系

2. css_set：将进程所属的cgroup中设置的子系统设置信息集合在一起

进程可同属多个cgroup，若属于多个cgroup，则与任务所属cgroup相关的cgroup_subsys_state个数也会增加。此时，为了能够快速方便地浏览较多的cgroup_subsys_state，我们使用css_set数据结构。与cgroup_subsys_state不同，css_set是基于进程的数据结构。

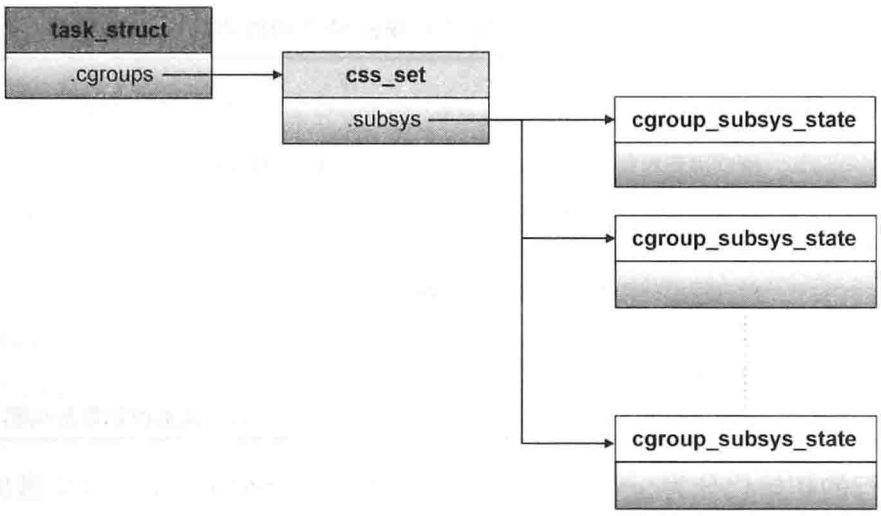


图8-3　task_struct和css_set、cgroup_subsys_state的关系

3. cgroupfs_root：显示cgroup文件系统层次图的顶端节点

cgroupfs_root结构体显示与自己相连接的cgroup层次图的顶端节点。而且，该结构体将cgroup_subsys结构体列表作为双向链表（double linked list）包含其中。值得注意的是，cgroup_subsys是记述子系统的结构体，与具有子系统设置信息的cgroup_subsys_state不同。层次图的顶端节点由成员变量top_cgroup记述。

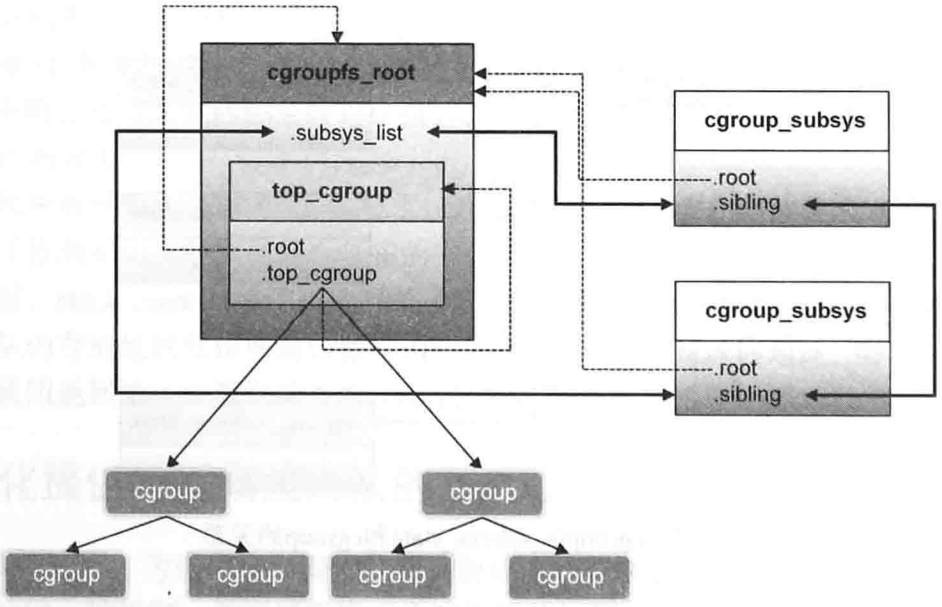


图8-4　cgroupfs_root和cgroup层、cgroup_subsys的关系

cgroup通过自己所属的cgroupfs_root结构体的subsys_list取得subsysid，并浏览自己的成员变量subsys[]，由此即可获得对自己适用的子系统设置信息cgroup_subsys_state。

在该函数的cgroup_init_early()中，对init任务使用的cgroup相关数据结构和优先子系统进行初

始化。现在正式了解cgroup_init_early()。

:: 代码8-3 kernel/cgroup.c的cgroup_init_early()

```
int __init cgroup_init_early(void)
{
    atomic_set(&init_css_set.refcount, 1);                              ❶
    INIT_LIST_HEAD(&init_css_set.cg_links);
    INIT_LIST_HEAD(&init_css_set.tasks);
    INIT_HLIST_NODE(&init_css_set.hlist);
    css_set_count = 1;

    init_cgroup_root(&rootnode);                                        ❷
    root_count = 1;

    init_task.cgroups = &init_css_set;                                  ❸

    init_css_set_link.cg = &init_css_set;                               ❹
    list_add(&init_css_set_link.cgrp_link_list,
             &rootnode.top_cgroup.css_sets);
    list_add(&init_css_set_link.cg_link_list,
             &init_css_set.cg_links);

    for (i = 0; i < CSS_SET_TABLE_SIZE; i++)                            ❺
        INIT_HLIST_HEAD(&css_set_table[i]);

    for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {                         ❻
        struct cgroup_subsys *ss = subsys[i];

        BUG_ON(!ss->name);                                              ❼
        BUG_ON(strlen(ss->name) > MAX_CGROUP_TYPE_NAMELEN);
        BUG_ON(!ss->create);
        BUG_ON(!ss->destroy);
        if (ss->subsys_id != i) {
            printk(KERN_ERR "cgroup: Subsys %s id == %d\n",
            ss->name, ss->subsys_id);
            BUG();
        }
        if (ss->early_init)
            cgroup_init_subsys(ss);                                     ❽
    }
    return 0;
}
```

在代码段❶中，对init任务使用的css_set结构体变量init_css_set进行初始化，并将refcount设置为1。此外，记录当前浏览的每个进程，并初始化与cg_cgroup_link、task_struct、css_set散列表相关的成员变量。最后，将全局变量css_set_count设置为1，由此记录系统所有css_set结构体变量的个数为1。

如前所述，使用init_css_set可以轻松查找init任务对自己所属cgroup适用的cgroup_subsys_state信息。

在代码段❷中，对cgroupfs_root结构体变量rootnode进行初始化。rootnode被声明为全局变量，是未适用于任何cgroup子系统而保留的虚拟层次图。关于此处调用的init_cgroup_root将在8.3.1节进行详细说明。初始化后，将表示系统中的cgroup层次图个数的全局变量root_count设置为1。

在代码段❸中，将init_css_set指定为init任务的css_set。如前所述，css_set结构体具有对任务所属cgroup适用的子系统设置信息（struct cgroup_subsys_state）列表，可快速浏览已设置的子系统。

cgroup可包括多个进程。将这些进程的css_set连接到cgroup的数据结构就是cg_cgroup_link结构体。可通过特定css_set查找拥有css_set的进程所属cgroup，或显示cgroup包含的进程个数。

在代码段❹中，将init任务使用的css_set与cgroup进行连接。

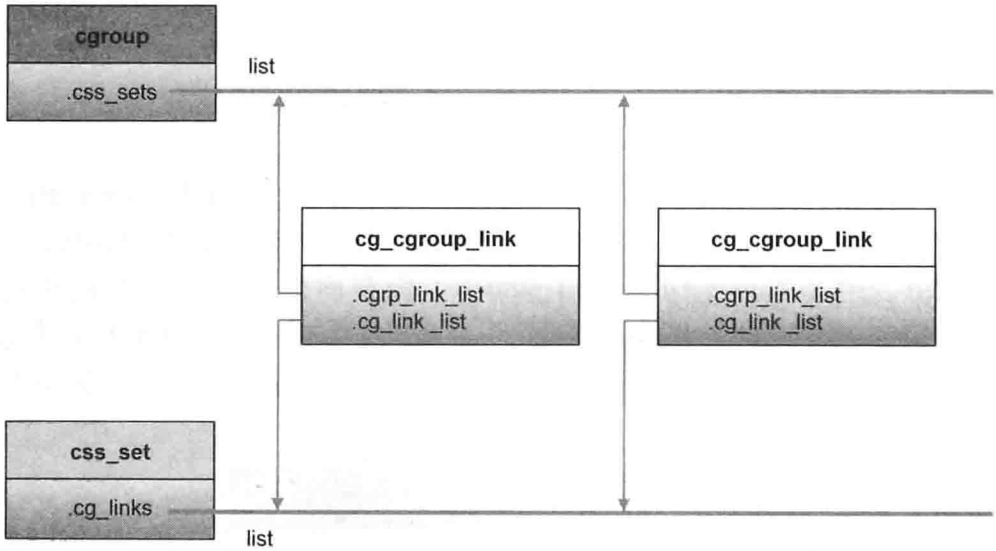


图8-5　将css_set连接到cgroup的cg_cgroup_link的连接关系

在代码段❺中初始化css_set_table[]数组，它将系统上的所有css_set构建为散列表。该散列表用双向链表将每个套接字连接到css_set结构体。使用散列可在诸多css_set中快速查找目标css_set。

在代码段❻中，对系统中需要初始init（early init）的cgroup子系统进行初始化。首先，在代码段❼中，检查子系统的默认信息是否有误，出现问题时停止初始化。检查之后，在代码段❽执行对该子系统的初始init。

这里调用的函数cgroup_init_subsys()将在8.3.2节进行详细说明。

8.3.1　cgroupfs_root和cgroup的关联初始化——init_cgroup_root()

该函数将作为参数传递的cgroupfs_root结构体变量root和自己所属cgroup层次图的top_cgroup进行连接，并初始化相关数据结构。

:: 代码8-4 kernel/cgroup.c的init_cgroup_root()

```
static void init_cgroup_root(struct cgroupfs_root *root)
{
    struct cgroup *cgrp = &root->top_cgroup;
    INIT_LIST_HEAD(&root->subsys_list);                    ❶
    INIT_LIST_HEAD(&root->root_list);
    root->number_of_cgroups = 1;
    cgrp->root = root;                                     ❷
    cgrp->top_cgroup = cgrp;
    init_cgroup_housekeeping(cgrp);                        ❸
}
```

参数root表示自己所属cgroup的层次图。在代码段❶中，初始化与root连接的子系统列表，将层次图中的cgroup个数设置为1。在代码段❷中，使层次图的顶端cgroup，即top_cgroup指向root，并将自身设置为层次图的顶端cgroup。

最后，在代码段❸调用init_cgroup_housekeeping()以初始化用于构建层次图的list变量、互斥变量。

8.3.2 初始化子系统——cgroup_init_subsys()

在cgroup_init_subsys()中对作为参数传递的子系统ss进行初始化。

:: 代码8-5 kernel/cgroup.c的cgroup_init_subsys()

```
static void __init cgroup_init_subsys(struct cgroup_subsys *ss)
{
    struct cgroup_subsys_state *css;
    printk(KERN_INFO "Initializing cgroup subsys %s\n", ss->name);

    list_add(&ss->sibling, &rootnode.subsys_list);         ❶
    ss->root = &rootnode;
    css = ss->create(ss, dummytop);                        ❷
    BUG_ON(IS_ERR(css));

    init_cgroup_css(css, ss, dummytop);                    ❸
    init_css_set.subsys[ss->subsys_id] = dummytop->subsys[ss->subsys_id];  ❹
    ...
    ss->active = 1;                                        ❺
}
```

表示cgroup层次图的rootnode具有系统中可使用的（尚未使用的）所有子系统列表。在代码段❶中，将作为参数传递的子系统与rootnode相连接后，子系统将rootnode设置为与自己相连的层次图。

在代码段❷中，为rootnode顶端cgroup，即dummytop[①]连接子系统ss。子系统连接到cgroup后，

① #define dummytop (&rootnode.top_cgroup)

生成具备该子系统设置信息的cgroup_subsys_state结构体变量。在代码段❸中，为数据结构设置与生成的css相连接的dummytop的相关关系。该函数执行的操作简单整理如下。

- 将拥有连接的子系统信息的css设置为已连接到dummytop
- 在成员变量refcss中设置1，表示css由之前设置的cgroup浏览
- 若设置css的cgroup为dummytop，则对flags设置CSS_ROOT位
- 为连接dummytop的子系统列表添加css

在代码段❹中，使之前生成的css能够指向init_css_set。dummytop是rootnode的顶端cgroup，rootnode显示init任务的cgroup层次图。由于init_css_set具有与当前初始化的init任务相连接的子系统的css列表，此处必须添加设置。

最后，在代码段❺中将ss->active设置为1，表示该子系统的初始化任务已结束，并可用。

8.4　禁用 IRQ

正式进行初始化前，先禁用当前启动的CPU的IRQ。由于尚未对中断代码和向量表、中断处理器进行初始化，所以必须关闭中断，该函数调用raw_local_irq_disable()以禁用IRQ。

:: 代码8-6　arch/arm/include/asm/irqflags.h的local_irq_disable()

```
#define local_irq_disable()①\
    do { raw_local_irq_disable(); trace_hardirqs_off()②; } while (0)

#define raw_local_irq_disable() \
    __asm__ ("cpsid i        @ __cli" : : : "memory", "cc")                    ❶
```

代码8-6的代码段❶是对raw_local_irq_disable宏的定义，设置CPSR的I位以关闭中断。

8.5　early_boot_irqs_off()、early_init_irq_lock_class()

本节简单介绍early_boot_irqs_off()函数和early_init_irq_lock_class()函数。首先，early_boot_irqs_off()函数是定义CONFIG_TRACE_IRQFLAGS时执行的函数，仅使early_boot_irqs_enabled=0。因此该函数与early_boot_irqs_on()函数成对，定义如下。

:: kernel/lockdep.c

```
static int early_boot_irqs_enabled;

void early_boot_irqs_off(void)
{
    early_boot_irqs_enabled = 0;
```

① 根据LINUX_ARM_ARCH的不同，调用的宏也不同。该常量由arch/arm/Makefile决定。

② 不使用CONFIG_TRACE_IRQFLAGS时不执行任何操作。除必要情况外，本书不会提及trace、debug的相关代码。

```
}

void early_boot_irqs_on(void)
{
    early_boot_irqs_enabled = 1;
}
```

这两个函数中使用的early_boot_irqs_enabled标识用来协助调试，执行当前初始启动代码时，对无效的中断激活事件输出警告。

下面了解一下early_init_irq_lock_class()函数。early_init_irq_lock_class()函数在定义CONFIG_GENERIC_HARDIRQS时执行。其定义如下所示，默认对所有中断的锁类映射信息进行初始化。

:: kernel/irq/handle.c

```
void early_init_irq_lock_class(void)
{
    struct irq_desc *desc;
    int i;

    for_each_irq_desc(i, desc) {
        lockdep_set_class(&desc->lock, &irq_desc_lock_class);
    }
}
```


该函数在执行循环的同时，通过for_erch_irq_desc宏取得中断描述符irq_desc结构体。这个宏访问的irq_desc[]数组和struct_irq_desc结构体声明如下。

:: include/linux/irq.h

```
extern struct irq_desc irq_desc[NR_IRQS];
```

:: include/linux/irq.h

```
struct irq_desc {
    unsigned int        irq;                    /*中断号*/
    irq_flow_handler_t  handle_irq;             /*irq中断处理器*/
    struct irqaction    *action;                /*IRQ动作列表*/
    unsigned int        status;                 /*IRQ状态*/

    ...

    spinlock_t      lock;                       /*锁*/
    const char      *name;                      /*在/proc/interrupts中输出的处理器名称*/
} ____cacheline_internodealigned_in_smp;
```

输入所有中断描述符并调用lockdep_set_class()函数，该函数在内部调用lockdep_init_map()，其作用是接收desc->lock结构体和irq_desc_lock_class结构体，填充lock（struct spilock_t类型）的成员变量dep_map结构体（struct lockdep_map类型）的值。struct lockdep_map结构体声明如下。

:: include/linux/lockdep.h

```
/*
 * Map the lock object (the lock instance) to the lock-class object.
 * This is embedded into specific lock instances:
 */
struct lockdep_map {
    struct lock_class_key          *key;
    struct lock_class              *class_cache;
    const char                     *name;
#ifdef CONFIG_LOCK_STAT
    int                            cpu;
    unsigned long                  ip;
#endif
};
```

该结构体将锁对象映射为锁类对象。

8.6　大内核锁——lock_kernel()

多处理器系统中，经常发生各进程同时访问相同内核资源的情况。最简单的解决方法是，每次只允许1个进程访问内核资源。

但是，这样会使系统使用效率大打折扣。Linux内核2.0版中，每次只能有1个进程访问内核资源。我们将这种情况称之为大内核锁（BKL，Big Kernel Lock），使用了只依赖于1个锁的锁机制（lock mechanism）。

换言之，不允许多个进程同时使用内核资源，只有获得BKL的进程才能使用内核资源。之后的Linux内核使用多个自旋锁以保护各内核数据结构。BKL用来确保与之前代码的兼容性。[①]

现在，人们使用自旋锁、读/写锁、RCU等同步机制控制对内核资源的同时访问。使用BKL的代码正在逐渐修正为新的同步机制。本节我们要了解的lock_kernel()函数执行的就是获得BKL的任务。

:: 代码8-7　lib/kernel_lock.c的lock_kernel()

```
void __lockfunc lock_kernel(void)                                    ❶
{
    int depth = current->lock_depth+1;                               ❷
    if (likely(!depth))                                              ❸
        __lock_kernel();                                             ❹
    current->lock_depth = depth;
}
```

具体了解lock_kernel()函数前，先了解 __lockfunc指示符。代码8-7的代码段❶中，函数名之前的__lockfunc是指向 __attribute__((section(".spinlock.text")))的宏，用于指定函数属性。

① 在Linux内核2.6.39中被全部删除。

在代码段❷中浏览当前进程的lock_depth。作为宏，current用于浏览正在执行代码的进程的描述符（task_struct）。每个进程描述符都有lock_depth域，表示BKL获得与否。lock_depth的初始值为-1，如为0以上则表示已获得BKL。

current宏是内核资源中经常见到的宏，所以需要仔细观察。

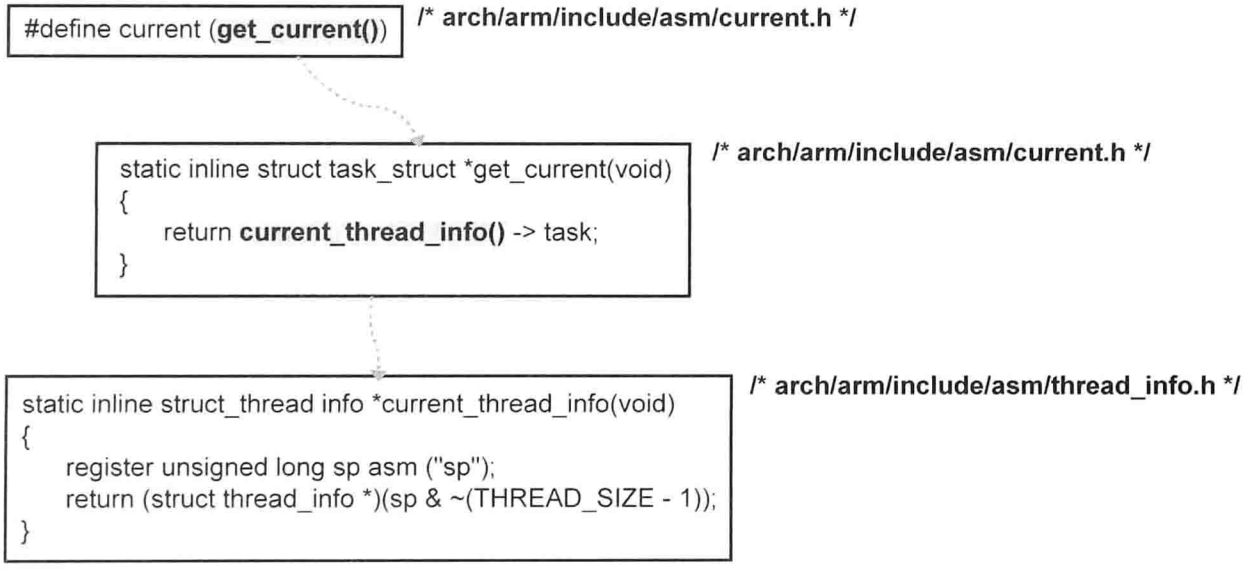


图8-6 current

仔细观察图8-6，可以了解current宏利用get_current()函数返回当前进程的描述符地址值。current_thread_infor()利用为每个进程分配的内核栈访问进程描述符[①]。对current宏的详细讲解参考本节“补充知识点”。

现在观察代码段❸的if条件语句。在条件语句内可以看到1个叫做likely()的陌生函数。这是内核宏，与GCC提供的内置函数（built – in function）相连接。

likely(a)将“a为真的概率较高”通知给编译器，编译器执行优化过程中重新分配代码时，给予提高性能的提示。除了likely()函数，也有unlikely()函数。unlikely()函数从GCC3.0开始添加的代码段❹中获得BKL，向描述符标明这些内容。

在代码8-7的代码段❸中可以看到__lock_kernel()，该函数就是获得内核锁的函数。__lock_kernel()函数按如下方式调用_raw_spin_lock()。

:: lib/kernel_lock.c

```
static inline void __lock_kernel(void)
{
    ...

    if (preempt_count() > 1) {
```

① 进程描述符（process discriptor）是操作系统显示系统内置进程的数据结构。Linux中使用task_struct结构体记述进程。进程描述符用于维持与进程相关的内核子系统（调度程序、内存管理器、文件系统等）的信息。

```
    _raw_spin_lock(&kernel_flag);
    return;
    }
    ...
}
```

_raw_spin_lock()定义如下。

:: include/linux/spinlock.h

```
#define _raw_spin_lock(lock)        __raw_spin_lock(&(lock)->raw_lock)
```

最终，__raw_spin_lock成为获得内核锁的核心函数。通过代码8-8分析一下。

:: 代码8-8　arch/arm/include/asm/spinlock.h的__raw_spin_lock()

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    unsigned long tmp;

    __asm____volatile__(                                                    ❶
                        "1: ldrex   %0, [%1]\n"                             ❷
                        "   teq %0, #0\n"
#ifdef CONFIG_CPU_32v6K
                        "   wfene\n"
#endif
                        "   strexeq %0, %2, [%1]\n"                         ❸
                        "   teqeq   %0, #0\n"
                        "   bne 1b"                                         ❹
                        : "=&r" (tmp)                                       ❺
                        : "r" (&lock->lock), "r" (1)                        ❻
                        : "cc");

    smp_mb();
}
```

首先，观察代码段❺和代码段❻。代码段❺通过输出变量指定tmp，代码段❻通过输入值指定lock->lock和常数1。这些相当于内联汇编中的%0、%1、%2。

```
%0=tmp
%1=&(lock->lock)
%2=1
```

在代码段❶中将lock->lock的值排他性地读入tmp。

在代码段❷中检查tmp的值是否为0，也就是确认当前是否挂锁。若为0则表示未挂锁。

在代码段❸中仅限于tmp值为0的情况下，通过strexeq命令排他性地将lock->lock值替换为1。需要注意的是，通过strexeq执行的结果再次变为%0，即保存到tmp变量。因此，查看tmp值，从而通过strexeq命令确认lock->lock值是否正常修正。

最后，通过代码段❹检查tmp值，以设置CPSR的Z标签。若tmp值为非0（strexeq未正常执行），则回到代码段❶重新取锁。



补充知识点 查找当前任务current的方法

current是显示当前任务的任务描述符，表现为stuct task_struct结构体。current的定义如下所示。

:: arch/arm/include/asm/current.h

```
static inline struct task_struct *get_current(void)
{
    return current_thread_info()->task;
}

#define current (get_current())
```

最终返回current_thread_info()的task成员，current_thread_info()函数传递struct thread_info结构体。

:: arch/arm/include/asm/thread_info.h

```
static inline struct thread_info *current_thread_info(void)
{
    register unsigned long sp asm ("sp");
    return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));
}
```

返回intread_info时使用的是栈指针（sp）和THREAD_SIZE宏。THREAD_SIZE按如下所示定义为8KB。

:: arch/arm/include/asm/thread_info.h

```
#define THREAD_SIZE                 8192
#define THREAD_START_SP             (THREAD_SIZE - 8)
```

因此，用THREAD_SIZE -1为sp加掩码，使sp的地址以8 KB为单位进行排列。这样即可取得内核栈的起始位置。thread_info结构体共享内核栈和内存，其定义如下所示。

:: include/linux/sched.h

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

内核栈最终成为以8KB（THREAD_SIZE）为1个字符进行保存的数组。thread_info结构体定义如下。

:: arch/arm/include/asm/thread_info.h

```
struct thread_info {
    unsigned long          flags;           /* low level flags */
    int                    preempt_count;   /* 0 => preemptable, <0 => bug */
    mm_segment_t           addr_limit;      /* address limit */
    struct task_struct     *task;           /* main task structure */
    struct exec_domain     *exec_domain;    /* execution domain */
    __u32                          cpu;              /* cpu */
    __u32                          cpu_domain;       /* cpu domain */
    struct cpu_context_save        cpu_context;      /* cpu context */
    __u32                          syscall;          /* syscall number */
    __u8                           used_cp[16];      /* thread used copro */
    unsigned long          tp_value;
    struct crunch_state    crunchstate;
    union fp_state         fpstate __attribute__((aligned(8)));
    union vfp_state        vfpstate;
#ifdef CONFIG_ARM_THUMBEE
    unsigned long          thumbee_state;  /* ThumbEE Handler Base register */
#endif
     restart_block    restart_block;
};
```

表示当前任务的current和内核栈、thread_info的关系如图8-7所示。当前任务current可通过内核的thread_info结构体进行访问，内核栈再次通过current->stack访问。而且，用THREAD_SIZE -1为栈指针加掩码（mask）即可获得thread_info的起始位置。

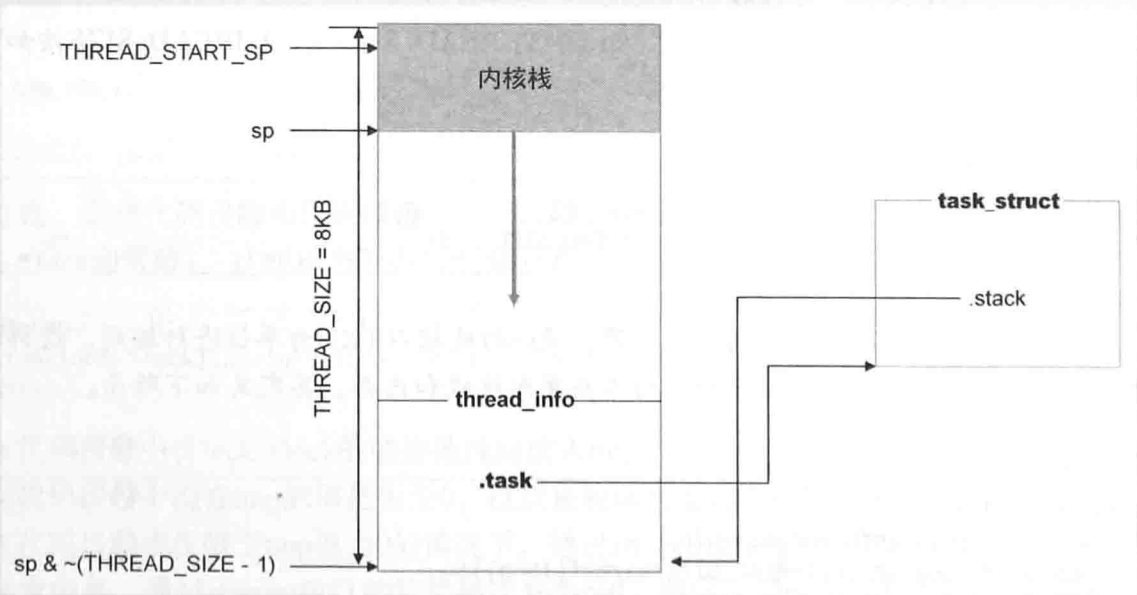


图8-7　thread_info、task_struct、内核栈的关系

第9章

注册针对时钟事件的处理器

本章主要了解内核为了处理从时钟设备传来的事件而注册处理器的过程。主要由tick_init()函数执行注册，其作用是，生成notifier_block型结构体tick_notifier后，将tick_notify函数注册到tick_notifier的函数指针。最终，对时钟事件的处理器就是tick_notify()函数。

生成的各notifier_block由链表构成，时钟事件框架clockevents_chain将成为链表的起始字符。从时钟设备发生事件时，通过clockevents_chain按顺序访问链表，从而调用处理器以处理事件。

9.1 函数的声明和定义——tick_init()

代码9-1定义了start_kernel中调用tick_init()函数时要对应的函数。如代码段❶，已定义CONFIG_GENERIC_CLOCKEVENTS时，会调用其他地方定义的tick_init()函数；未定义CONFIG_GENERIC_CLOCKEVENTS时，不执行任何操作。

:: 代码9-1 include/linux/tick.h的tick_init()的声明

```
#ifdef CONFIG_GENERIC_CLOCKEVENTS                                    ❶
extern void __init tick_init(void);

    ...

#else /*  CONFIG_GENERIC_CLOCKEVENTS*/
static inline void tick_init(void) { }                               ❷
```

那么CONFIG_GENERIC_CLOCKEVENTS是如何定义的？其默认值又是什么呢？CONFIG_GENERIC_CLOCKEVENTS是内核编译时设置的，其定义如下所示。

:: arch/arm/Kconfig

```
config GENERIC_CLOCKEVENTS
    bool
    default n
```

可以看出，未默认定义CONFIG_GENERIC_CLOCKEVENTS。但这段代码假设了内核编译

时已定义CONFIG_GENERIC_CLOCKEVENTS，由此分析tick_init()函数。

已定义CONFIG_GENERIC_CLOCKEVENTS时，如代码9-2所示，调用kernel/time/tick_common.c文件中定义的tick_init()函数。

:: 代码9-2　kernel/time/tick-common.c的tick_init()的声明

```
static struct notifier_block tick_notifier = {
    .notifier_call = tick_notify,
};

void __init tick_init(void)
{
    clockevents_register_notifier(&tick_notifier);                    ❶
}
```

代码段❶中，tick_init()函数只调用clockevents_register_notifier()函数。调用时传递tick_notifier参数，但tick_notifier定义为notifier_block的static变量。notifier_block结构体的定义如下所示。

:: include/linux/notifier.h

```
struct notifier_block {
    int (*notifier_call)(struct notifier_block *, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};
```

notifier_call的成员变量是处理事件的处理器的函数指针，next成员变量是下一notifier_block的指针。

最终，处理事件的结构是，tick_notifier将tick_notify()函数注册到notifier_call，使处理时钟事件时能够调用tick_notify()函数。tick_notify()函数作为static函数负责对时钟事件设备执行Notification，对事件进行如下处理。

:: kernel/time/tick-common.c

```
CLOCK_EVT_NOTIFY_ADD
CLOCK_EVT_NOTIFY_BROADCAST_ON
CLOCK_EVT_NOTIFY_BROADCAST_OFF
CLOCK_EVT_NOTIFY_BROADCAST_FORCE
CLOCK_EVT_NOTIFY_BROADCAST_ENTER
CLOCK_EVT_NOTIFY_BROADCAST_EXIT
CLOCK_EVT_NOTIFY_CPU_DYING
CLOCK_EVT_NOTIFY_CPU_DEAD
CLOCK_EVT_NOTIFY_SUSPEND
CLOCK_EVT_NOTIFY_RESUME
```

补充知识点 内核中使用的4种通知链（Notifier Chain）

1. 原子通知链（Atomic Notifier Chains）

在中断或原子上下文（atomic context）中运行chain回调函数，不允许阻塞回调函数。

2. 可阻塞通知链（Blocking Notifier Chains）

回调函数在进程上下文中执行，允许阻塞回调函数。

3. 原始通知链（Raw Notifier Chains）

对回调、注册、注销无任何限制，所有同步及保护机制由调用方（caller）运行。

4. SRCU通知链（Sleepable Read-Copy Update Notifier Chains）

可阻塞通知链之一，具有和可阻塞通知链相同的限制。虽然chain被频繁调用，但必须在notifier block中未全部清除的情况下使用SRCU通知链，为此需要执行运行时初始化。

9.2 注册处理事件的处理器——_clockevents_register_notifier()

我们已经知道tick_notifier是如何声明定义的。接下来查看clockevents_register_notifier()函数是如何定义的。代码9-3是clockevents_chain中注册notifier_block，即tick_notifier的部分。

:: 代码9-3 kernel/time/clockevents.c的clockevents_register_notifier()

```
int clockevents_register_notifier(struct notifier_block *nb)
{
    int ret;

    spin_lock(&clockevents_lock);                                    ❶
    ret = raw_notifier_chain_register(&clockevents_chain, nb);       ❷
    spin_unlock(&clockevents_lock);                                  ❸

    return ret;
}
```

我们先观察整体的代码流程。通过函数名可以把握该函数执行的操作。在代码段❶中为clockevents_lock挂上自旋锁。然后在代码段❷中将接收的tick_notifier参数注册到clockevents_chain。注册完成后，在代码段❸中对clockevents_lock解除自旋锁。我们需要通过此代码把握以下4点。

(1) 为clockevents_lock挂自旋锁的原理。

(2) clockevents_chain生成原理。

(3) 为clockevents_chain注册notifier_block结构体tick_notifier的方法。

(4) 对clockevents_lock解除自旋锁的原理。

9.2.1　为clockevents_lock添加自旋锁

在内核代码中，从clockevents_lock首次出现的部分开始，依次搜索各项定义可得到以下序列。

:: kernel/time/clockevents.c

```
static DEFINE_SPINLOCK(clockevents_lock);
```

:: include/linux/spinlock_types.h

```
#define DEFINE_SPINLOCK(x)  spinlock_t x = __SPIN_LOCK_UNLOCKED(x)
```

:: include/linux/spinlock_types.h

```
typedef struct {
    raw_spinlock_t raw_lock;
...
} spinlock_t;
```

:: include/linux/spinlock_types.h

```
typedef struct {
    volatile unsigned int lock;
} raw_spinlock_t;
```

:: include/linux/spinlock_types.h

```
#define __SPIN_LOCK_UNLOCKED(lockname) \
(spinlock_t)    {   .raw_lock = __RAW_SPIN_LOCK_UNLOCKED,   \
        SPIN_DEP_MAP_INIT(lockname) }
```

:: include/linux/spinlock_types.h

```
#define __RAW_SPIN_LOCK_UNLOCKED    { 0 }
```

综合以上代码，最终的DEFINE_SPINLOCK(clockevents_lock)如下所示。

```
spinlock_t clockevents_lock = (spinlock_t){ .raw_lock = {0},}①
```

现在观察取锁函数spin_lock()函数。找到spin_lock()函数的定义，发现spin_lock()函数按如下所示定义为_spin_lock()。

:: include/linux/spinlock.h

```
#define spin_lock(lock)         _spin_lock(lock)
```

① 在“以CONFIG_开始的设置均为off”的假定下，将代码进行了简化。

继续查看，_spin_lock()函数又定义如下，而且在spin_acquire()中指定当前任务的lock_depth和dep_map等。

:: kernel/spinlock.c

```
void __lockfunc _spin_lock(spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, _raw_spin_trylock, _raw_spin_lock);
}
```

在_spin_lock()函数中，为获取锁而修改值的代码就是LOCK_CONTENDED()宏，其定义如下所示。

:: include/linux/lockdep.h

```
#define LOCK_CONTENDED(_lock, try, lock) \
    lock(_lock)
```

这是第三个参数lock对_raw_spin_lock()函数的函数指针，将第一个参数_lock作为_raw_spin_lock()函数的参数传递。_raw_spin_lock()函数的定义如下所示。

:: include/linux/spinlock.h

```
#define _raw_spin_lock(lock)        __raw_spin_lock(&(lock)->raw_lock)
```

因此，代码9-3的spin_lock(&clockevents_lock)变更如下。

```
__raw_spin_lock(&(clockevents_lock)->raw_lock)
```

实际_raw_spin_lock()函数的说明请参考8.6节的大内核锁（BKL）。

9.2.2 clockevents_chain生成原理

下面了解一下clockevents_chain是如何定义的。从clockevents_chain在内核代码中首次出现的部分开始，依次查询定义可获得如下序列。

:: kernel/time/clockevents.c

```
static RAW_NOTIFIER_HEAD(clockevents_chain);
```

:: include/linux/notifier.h

```
#define RAW_NOTIFIER_HEAD(name)                 \
    struct raw_notifier_head name =             \
        RAW_NOTIFIER_INIT(name)
```

:: include/linux/notifier.h

```
struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block *head;
};
```

:: include/linux/notifier.h

```
#define RAW_NOTIFIER_INIT(name) {                \
    .head = NULL }
```

最终，将clockevents_chain声明为raw_notifier_head结构体变量，并将成员变量head初始化为NULL。这样生成的clockevents_chain具有头节点的作用，将后面的notifier_block结构体连接到链表。

9.2.3　在clockevents_chain中注册tick_notifier的方法

现在了解一下在clockevents_chain中注册notifier_block结构体tick_notifier的方法。将notifier_block注册到clockevents_chain的函数是通过调用raw_notifier_chain_register()形成的。代码9-4表示raw_notifier_chain_register()函数和实际调用该函数执行注册的notifier_chain_regiser()函数。

:: 代码9-4　kernel/notifier.c的raw_notifier_chain_register()、notifier_chain_register()

```
int raw_notifier_chain_register(struct raw_notifier_head *nh, struct notifier_block *n)
{
    return notifier_chain_register(&nh->head, n);                          ❶
}
EXPORT_SYMBOL_GPL(raw_notifier_chain_register);                             ❷

    ...

static int notifier_chain_register(struct notifier_block **nl,
                                   struct notifier_block *n)
{
    while ((*nl) != NULL) {                                                ❸
        if (n->priority > (*nl)->priority)                                 ❹
            break;

        nl = &((*nl)->next);
    }
    n->next = *nl;                                                         ❺

    rcu_assign_pointer(*nl, n);                                            ❻
    return 0;
}
```

在代码段❶中调用实际执行注册的notifier_chain_register函数，在代码段❷中注册符号，使之

能够调用raw_notifier_chain_register函数。

在代码段❸中，对与链表notifier_block连接的所有notifier_block依次执行循环，并搜索。在代码段❹中，以notifier_block的成员变量priority值为基准，决定接收的参数notifier_block的注册位置。按优先次序（priority）从高到低排列。

若找到要注册的位置，就在代码段❺中注册接收的notifier_block参数。

在代码段❻中，指定读取频繁的临界段中需要解引用（dereference）的新结构体指针时，使用rcu_assign_point函数。总之，代码段❻使链表的链接操作得到安全修改，以将接收的notifier_block插入链表。rcu_assign_pointer()函数定义如下。

:: include/linux/rcupdate.h

```
#define rcu_assign_pointer(p, v) \
    ({ \
        if (!__builtin_constant_p(v) ||  ((v) != NULL)) \
            smp_wmb(); \
        (p) = (v); \
    })
```

检查接收的参数p和v是否分别为常量和NULL。如果p为常量，可将v直接代入p；若非常量，则调用smp_mb()，插入内存屏障（memory barrier）。这样，指定指针值后，不允许对编译器初始化的代码执行重排序（reordering）。但这里使用rcu_assign_pointer()函数是为了提示：RCU的读取代码使各指针被解引用。

9.2.4 对clockevents_lock解除自旋锁的原理

最后，我们分析对clockevents_lock解除自旋锁的原理。下面是对spin_unlock函数的定义。

:: include/linux/spinlock.h

```
#define spin_unlock(lock) \
    do {__raw_spin_unlock(&(lock)->raw_lock); __release(lock); } while (0)
```

只调用了__raw_spin_unlock()函数。实际上，对clockevents_lock解除自旋锁的操作是通过这个函数执行的。在__raw_spin_unlock()函数之后执行的__release()函数相当于用于静态分析的内核代码，更详细的内容请参考下列“补充知识点”。代码9-5表示__raw_spin_unlock()函数。

:: 代码9-5 arch/arm/include/asm/spinlock.h的__raw_spin_unlock()

```
static inline void __raw_spin_unlock(raw_spinlock_t *lock)
{
    smp_mb();

    __asm__ __volatile__("   str %1, [%0]\n"                    ❶

    ...
```

```
    :
    : "r" (&lock->lock), "r" (0)                                    ❷
    : "cc");
}
```

代码段❶将第一个参数接收的值保存到第0个参数的地址，各参数值定义在代码段❷中。

代码段❷中，第0个参数是lock的地址，第一个参数常量值是0，对应于代码段❶。最终，将clockevents_lock->lock->lock成员变量的值设置为0，从而解除自旋锁。



补充知识点　__release中定义的 __context__

__release宏定义如下。

:: include/linux/compiler.h

```
#define __release(x)   __context__(x,-1)
```

为了理解__context__，需要掌握Sparse工具。Sparse是李纳斯·托沃茨从2003年起开发的静态分析工具。静态分析工具是指不运行软件的情况下，仅用源代码进行分析。它与其他静态分析工具的不同之处在于，Sparse为内核编程者提供了比较特殊的功能。最具代表性的是，能够检测到将用户空间和内核空间的指针混用时发生的错误。为此，Sparse必须在内核代码内可能出现问题的地方添加“特殊代码”，而__context__就相当于此。通过这些插入的代码，编译器在编译时向编程者提供“指针属于哪个地址空间？”或“函数取得什么样的lock并执行返回？”等相关内容信息。

更详细的内容请在wikipedia中搜索Sparse。

※ 参考文献：http://en.wikipedia.org/wiki/Sparse#cite_note-3

整理本章时，若CONFIG_GENERIC_CLOCKEVENTS定义在内核编译时期（kernel compile time），这就表示要处理从时钟设备生成的事件。为了处理时钟事件，需要注册处理器，而注册处理器的函数是通过tick_init()函数形成的。tick_init()函数生成notifier_block型结构体tick_notifier后，将tick_notify()静态函数注册到该结构体的成员变量函数指针。将这样生成的tick_notifier用链表进行管理，链表的头节点就是clockevents_chain。最终，从时钟设备发生事件时，通过clockevents_chain访问tick_notifier结构体以调用相应处理器处理事件。

第 10 章

在CPU位图中注册当前运行CPU/初始化HIGHMEM管理

本章主要了解boot_cpu_init()函数和page_address_init()函数，前者在显示CPU状态的位图中注册当前启动的CPU，后者对HIGHMEM管理进行初始化。

10.1 在包含热插拔信息的位图上添加执行 init_task 的 CPU——boot_cpu_init()

内核中有位图，用来维护系统内CPU的状态信息。具代表性的有cpu_possible_map、cpu_online_map、cpu_present_map，位与CPU1:1映射。

例如，第0位显示第一个CPU的状态信息。cpu_possible_map是对系统中可执行当前热插拔（hot plug）的CPU的位图。该位图针对启动时系统支持的CPU数量，一旦设置了位，则不可添加或删除。

cpu_online_map是将联机（online，“使用中”）的所有CPU的位设置为1的位图。若已完成从内核调度或设备接收中断的准备，则设置该位。包括中断在内的所有OS服务移动到其他CPU时，该位被删除。

cpu_pressent_map是用于显示系统内CPU的位图，但并非所有CPU均处于联机状态。例如，根据ACPI等子系统以物理方式处理热插拔时，设置该位。

cpu_possible_map、cpu_online_map、cpu_present_map位图定义如下。

:: kernel/cpu.c

```
static DECLARE_BITMAP(cpu_possible_bits, CONFIG_NR_CPUS) __read_mostly;
static DECLARE_BITMAP(cpu_online_bits, CONFIG_NR_CPUS) __read_mostly;
static DECLARE_BITMAP(cpu_present_bits, CONFIG_NR_CPUS) __read_mostly;
```

在boot_cpu_init()函数中，将对应于3种位图（cpu_possible_map、cpu_online_map、cpu_present_map）的CPU ID的位设置为1。如果CPU在逻辑上实际处于脱机（offline）状态，会发生什么呢?

发生任务的CPU_DAWN_PREPARE事件，对该事件的处理请求传送到内核中注册的模块

（notification机制）。以前的CPU中设置的所有中断和计时器/底半部（bottom half）/任务等所有进程将移动到新的CPU。先移送所有服务，内核将调用特定程序__cpu_disable()，CPU将处于脱机状态。若之前CPU的所有服务成功移送到新的CPU，将发送CPU_DEAD事件。

通过代码10-1分析boot_cpu_init()函数。

:: 代码10-1　init/main.c的boot_cpu_init()

```
static void __init boot_cpu_init(void)
{
    int cpu = smp_processor_id();                    ❶
    set_cpu_online(cpu, true);                       ❷
    set_cpu_present(cpu. true);
    set_cpu_possible(cpu, true);
}
```

在代码段❶中，通过smp_processor_id()函数获得执行当前任务的CPU号。在代码段❷中，将获得的与CPU对应的位设置到cpu_possible_map、cpu_online_map、cpu_present_map。

再观察代码段❶的smp_processor_id()函数。该函数在CONFIG_SMP未定义时，用raw_smp_processor_id()函数代替，而raw_smp_processor_id()则定义为current_thread_info()->cpu（参考图10-1）。最终，将正在执行init_task的CPU号从执行当前任务（init_task）的线程结构体保存到区域变量cpu。

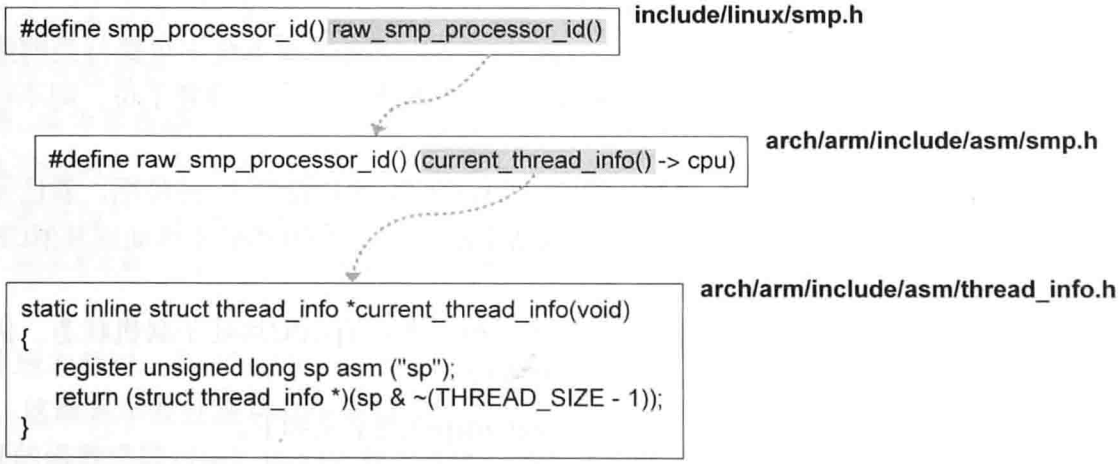


图10-1　smp_processor_id()

执行当前内核的任务init_task的定义如下。

:: arch/arm/kernel/init_task.c

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

struct task_struct init_task = INIT_TASK(init_task);
```

可通过thread_info结构体访问任务，请参考8.6节“补充知识点”。

10.2 管理高端内存——page_address_init()

内核通过page_address_pool访问HIGHMEM，通过page_address_maps[]进行管理。高端内存（high memory）区域是不能直接进行地址化的内存页，因此，使用kmap()映射高端内存（ZONE_HIGHMEM），用散列表page_address_htable另行管理HIGHMEM中分配的内存。[①]优先初始化的对象page_address_maps和page_address_htable的定义如下。

:: mm/highmem.c

```
static struct page_address_map page_address_maps[LAST_PKMAP];    /* LAST_PKMAP = 512 */

struct page_address_map {
    struct page *page;
    void *virtual;
    struct list_head list;
};

static struct page_address_slot {
    struct list_head lh;            /*page_address_maps列表*/
    spinlock_t lock;                /*保护锁*/
} ____cacheline_aligned_in_smp page_address_htable[1<<PA_HASH_ORDER];
```

通过代码10-2了解管理HIGHMEM的初始化函数page_address_pool()。

:: 代码10-2 mm/highmem.c的page_address_init()

```
void __init page_address_init(void)
{
    int i;

    INIT_LIST_HEAD(&page_address_pool);                                    ❶

    for (i = 0; i < ARRAY_SIZE(page_address_maps); i++)                    ❷
        list_add(&page_address_maps[i].list, &page_address_pool);

    for (i = 0; i < ARRAY_SIZE(page_address_htable); i++) {                ❸
        INIT_LIST_HEAD(&page_address_htable[i].lh);
        spin_lock_init(&page_address_htable[i].lock);
    }

    spin_lock_init(&pool_lock);                                            ❹
}
```

① 高端内存是RAM中896MB以上的内存空间，该区域用于用户空间程序或页缓存。高端内存不能直接执行映射，所以比低端内存（896MB以下）访问速度更慢。

在代码段❶中，初始化struct list_head类型page_address_pool。

代码段❷对page_address_maps[]的各数组执行循环，为page_address_pool添加page_address_maps[i].list列表。

代码段❸对page_address_htable执行循环，初始化其成员lh和lock变量。

在代码段❹中初始化pool_lock，它对page_address_pool执行访问控制。

图10-2表示page_address_pool、page_address_maps、page_address_htable的数据结构。

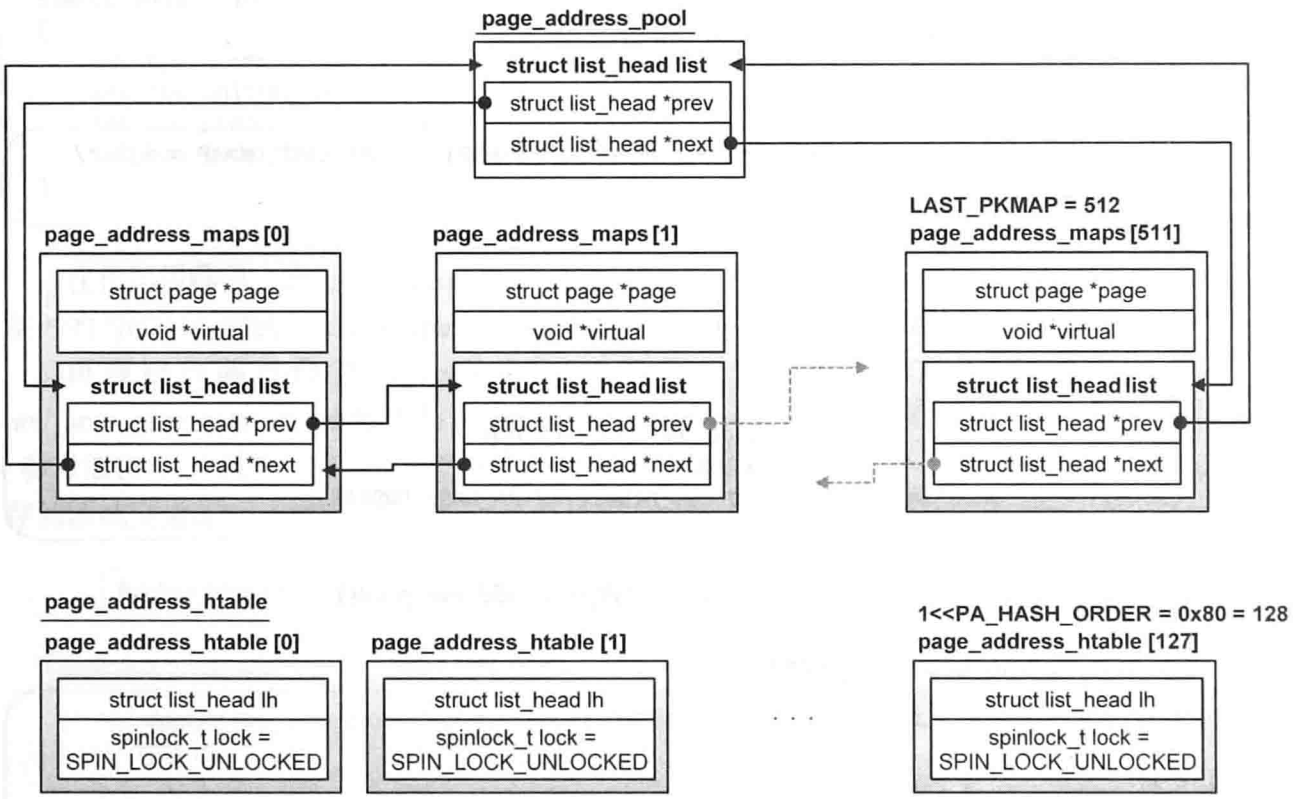


图10-2　page_address_pool、page_address_maps、page_address_htable的数据结构

第11章 整体指向——setup_arch

从本章开始，我们将分析setup_arch()函数。setup_arch()函数用于处理与架构相关的一系列事务，该函数也从内部调用许多函数。图11-1表示setup_arch()函数中调用函数的调用关系。

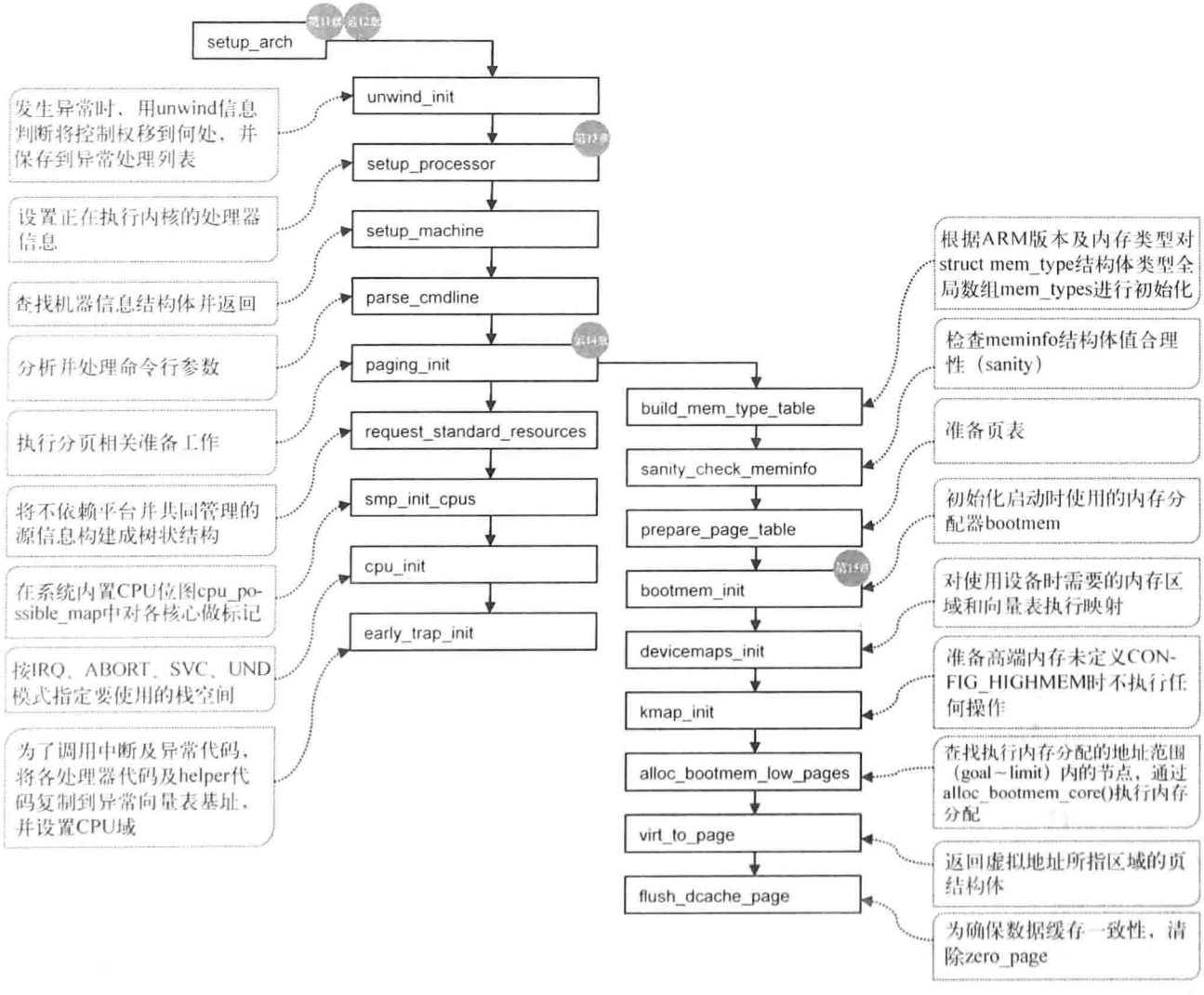


图11-1　setup_arch()函数的调用关系

在调用关系图中，setup_processor()函数和setup_machine()函数是7.2节和7.3节讲过的内容。但是setup_processor()函数具有更详细的处理器设置，对此将在第13章另行说明。

执行页相关初始化的paging_init()函数在内部调用许多函数。对于这些函数，我们将在第14章加以说明。此外，对paging_init()函数中用于激活启动时使用的内存分配器bootmem的bootmem_init()函数，将在第15章进行更详细的说明。

代码11-1表示setup_arch()函数的定义。对我们需要了解的各函数指定了标签，后续章节将进行详细分析。

:: 代码11-1　arch/arm/kernel/setup.c的setup_arch()

```
void __init setup_arch(char **cmdline_p)
{
    ...

    unwind_init();                                                    ❶
    setup_processor();                                                ❷

    mdesc = setup_machine(machine_arch_type);                         ❸

    ...

    parse_cmdline(cmdline_p, from);                                   ❹

    paging_init(mdesc);                                               ❺
    request_standard_resources(&meminfo, mdesc);                      ❻

#ifdef CONFIG_SMP
    smp_init_cpus();                                                  ❼
#endif

    cpu_init();                                                       ❽

    ...

    early_trap_init();                                                ❾
}
```

在代码段❶中初始化unwind信息。发生异常时，unwind信息判断将控制权移到何处。

在代码段❷中设置处理器，我专为该函数设置了章节，比第13章讲解更细。

代码段❸是查找机器信息结构体的函数，将在12.2节进行分析。

代码段❹是分析并处理命令行参数的过程。启动时需要多种信息，这些信息是从启动加载项传递的，或为编译时获得静态指定的值。根据各选项调用相关函数以设置环境，使内核能够正常启动（12.2节）。

代码段❺是分页相关的准备阶段，该函数在内部调用多个函数。具有代表性的是初始化启动时使用的内存分配器bootmem，使启动时以页为单位分配内存。对此将在第14章进行详解。

代码段❻中，将不依赖平台而共同管理的资源信息构建成树状结构（12.5节）。

代码段❼是SMP环境中调用的函数，初始化cpumaps中的cpu_possible_map。cpu_possible_map中以系统核心的数量设置对应于CPU号的位（12.6节）。

代码段❽中指定按ARM的IRQ、ABORT、SVC、UND（Undefined Instruction，未定义指令）模式使用的栈空间（12.7节）。

代码段❾中为了调用中断及异常代码，将各处理器代码及辅助器（helper）代码复制到异常向量表基址，并设置CPU域（12.8节）。

第 12 章 unwind_init()~early_trap_init()

12.1 栈回溯——unwind_init()

C++或Java等语言的内部工具可在调试时对栈执行追踪，并可确认发生程序错误时的栈内容。但C语言的环境十分恶劣，没有额外的调试器就很难调试。发生异常时，我们用unwind信息判断应当将控制权移到何处。它还能在重复调用函数而发生异常的情况下对栈执行回溯，从而获得调用自身的各函数信息。这种unwind信息由异常处理列表（exception handling table）保管。

下面通过代码12-1分析unwind_init()函数。

:: 代码12-1 arch/arm/kernel/unwind.c的unwind_init()

```
int __init unwind_init(void)
{
    struct unwind_idx *idx;                                          ❶
    for (idx = __start_unwind_idx; idx < __stop_unwind_idx; idx++)   ❷
        idx->addr = prel31_to_addr(&idx->addr);
    pr_debug("unwind: ARM stack unwinding initialised\n");
    return 0;
}
```

代码段❶中，内核中每个以.ARM.exidx开始的段都保存着具有栈展开（stack unwinding）信息的异常处理列表的项索引值。异常处理列表的各项大小不同，必须通过unwind_idx访问。

通过代码段❷对所有异常处理列表索引执行循环，求出异常处理列表各项所在绝对地址值。

12.2 求出包含机器信息的 machine_desc 结构体——setup_machine()

学习（查找机器信息的）setup_machine()函数前，为了在C语言中调用__lookup_machine_type（7.3.1节）中记述的汇编代码，我们将lookup_machine_type函数按如下所示注册到符号列表。

:: arch/arm/kernel/head-common.S

```
ENTRY(lookup_machine_type)
    stmfd   sp!, {r4 - r7, r9, lr}
    mov r9, r0
    bl  __lookup_machine_type
    mov r0, r5
    ldmfd   sp!, {r4 - r7, r9, pc}
ENDPROC(lookup_machine_type)
```

setup_machine()函数的定义如下所示。

:: arch/arm/kernel/setup.c

```
static struct machine_desc * __init setup_machine(unsigned int nr)
{
    struct machine_desc *list;

    list = lookup_machine_type(nr);
    if (!list) {
        printk("Machine configuration botched (nr %d), unable "
               "to continue.\n", nr);
        while (1);
    }

    printk("Machine: %s\n", list->name);
    return list;
}
```

最终，lookup_machine_type函数将执行__lookup_machine_type标签中的汇编代码，获得保存当前机器信息的结构体machine_desc的指针。

12.3 处理 ATAG 信息——setup_arch()

我们在上一节查找了机器描述符，现在处理ATAG，同时利用找到的机器描述符执行几项任务。

:: 代码12-2 arch/arm/kernel/setup.c的setup_arch()

```
void __init setup_arch(char **cmdline_p)
{
    struct tag *tags = (struct tag *)&init_tags;

    ...

    if (mdesc->soft_reboot)                                    ❶
        reboot_setup("s");              /*将重启模式设置为s*/

    if (__atags_pointer)                                       ❷
```

```
        tags = phys_to_virt(__atags_pointer);
    else if (mdesc->boot_params)
        tags = phys_to_virt(mdesc->boot_params);

    if (tags->hdr.tag != ATAG_CORE)                                    ❸
        convert_to_tag_list(tags);

    if (tags->hdr.tag != ATAG_CORE)                                    ❹
        tags = (struct tag *)&init_tags;

    if (mdesc->fixup)                    /*设置内存相关信息*/          ❺
        mdesc->fixup(mdesc, tags, &from, &meminfo);

    if (tags->hdr.tag == ATAG_CORE) {                                  ❻
        if (meminfo.nr_banks != 0)
            squash_mem_tags(tags);
        save_atags(tags);
        parse_tags(tags);
    }

    ...
}
```

代码12-2的代码段❶中，根据mdesc->soft_reboot的设置值，将reboot_mode变量的值初始化为“s”，或设为默认值“h”。发生reboot系统调用时，若传递了LINUX_REBOOT_CMD_RESTART2 (0xA1B2C3D4)参数，则调用下一函数。

(1) kernel_restart()

(2) machine_restart()

(3) arm_pm_restart()

arm_pm_restart是函数指针，设置为arm_machine_restart。先调用arm_machine_restart，再调用setup_mm_for_reboot、arch_reset等函数，它们按结构均有不同定义。这些函数中，之前设置的reboot_mode值不同，reboot代码也不同。但是，除了几个系统外，大部分系统中不使用这个值。

setup_arch()函数接收并处理启动加载项向内核传递的参数，它们又称标签（tag）。这些标签在启动加载项中成为列表，并完成初始化。将标签列表定义为atag，如下图所示从ATAG_CORE声明至ATAG_NONE，ATAG_CORE标签位置中是否存在值是不确定的。指向atags位置的值就是atags指针，相当于__atag_pointer变量。

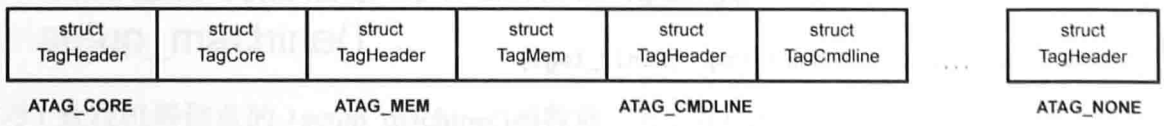


图12-1　atags列表的数据结构

接下来观察setup_arch()函数（代码12-2）中处理__atags_pointer值的部分。在代码段❷中查看有无从启动加载项传递来的参数，若有，则将其变更为虚拟地址之后保存到标签结构体；反之，

则使用包含机器信息的结构体的启动参数值。代码段❸中，如果之前传递参数的方式现在已无法使用（deprecated），则转换标签列表。如果此处也未设置标签结构体，则默认使用代码段❹中init_tags内定义的值。

代码段❺中，若机器信息结构体内存在fixup，那就按机器指定的信息设置到meminfo结构体。meminfo结构体具有BANK数和节点号、起始地址、大小等信息。代码段❻负责分析并处理传递来的ATAG信息。

squash_mem_tags(tags)中，若之前机器描述符的fixup函数已设置meminfo，则忽略ATAG的mem设置信息，并通过save_atags和parse_tags保存并分析atags信息。

在parse_tags()函数的内部，对tag执行循环的同时调用parse_tag()函数，而parse_tag()函数则查找保存在taglist.init段的标签信息，并执行要分析该标签的函数。通过__tagtable()宏静态指定需要按标签类型进行分析的函数。

补充知识点　在ARM中设置meminfo的方法

在ARM中设置meminfo的方法有3种。

(1) 在机器的fixup函数中设置。

(2) 接收启动加载项的ATAG_MEM标签并设置。

(3) 利用内核命令行的“mem=参数”设置。

部分制作公司的核心可能可以在内核设置中设置。

12.4 处理启动参数——parse_cmdline()

启动内核需要多种信息。这些信息是从启动加载项接收的，或是内核编译时静态获得的默认值。设置后的内核命令行参数根据各选项值调用相应函数，以设置内核启动环境。代码12-3表示命令行参数的分析及处理过程。

:: 代码12-3　arch/arm/kernel/setup.c的parse_cmdline()

```
static void __init parse_cmdline(char **cmdline_p, char *from)
{
    char c = ' ', *to = command_line;
    int len = 0;

    for (;;) {
        if (c == ' ') {
            extern struct early_params __early_begin, __early_end;
            struct early_params *p;

            for (p = &__early_begin; p < &__early_end; p++) {                    ❶
                int arglen = strlen(p->arg);
            if (memcmp(from, p->arg, arglen) == 0) {
```

```
                if (to != command_line)
                    to -= 1;

                    from += arglen;
                    p->fn(&from);                                    ❷

                    /*增加from直至出现字符*/
                    while (*from != ' ' && *from != '\0')
                        from++;
                    break;
            }
          }
        }
        c = *from++;
        if (!c)
            break;
        if (COMMAND_LINE_SIZE <= ++len)
            break;
        *to++ = c;
    }
    *to = '\0';
    /*指向包含未处理参数的空间*/
    *cmdline_p = command_line;                                       ❸
```

处理各参数的函数通过__early_param宏定义。我们以initrd参数为例进行说明。

initrd的声明和__early_param宏如下所示。

:: arch/arm/mm/init.c

```
__early_param("initrd=", early_initrd);
```

:: arch/arm/include/asm/setup.h

```
#define __early_param(name,fn)                          \
static struct early_params __early_##fn __used   \
__attribute__((__section__(".early_param.init"))) = { name, fn }
```

将"initrd="和具有early_initrd值的early_params结构体声明为__early_early_initrd，并分配到.early_param.init段。

.early_param.init由vmlinux.lds.S分配在__early_begin和__early_end之间。

:: arch/arm/kernel/vmlinux.lds.S

```
__setup_start = .;
*(.init.setup)
__setup_end = .;
__early_begin = .;
*(.early_param.init)
__early_end = .;
```

```
__initcall_start = .;
INITCALLS
__initcall_end = .;
```

最后，将通过__early_param定义的__early_params结构体分配在__early_begin和__early_end之间，之前声明的__early_early_initrd也分配在此。

在代码段❷中，对分配好的early_params结构体执行循环，并检查传递的参数名，调用相应函数。为initrd时，调用early_initrd。对所有early_params结构体执行循环后，在代码段❸中保存未处理的参数并结束。由于内核参数过多，具体事项请参考Documentation/kernel-parameters.txt。默认（default）使用的内核参数如表12-1所示。

表12-1 内核参数

名　称	说　明
initrd	指定初始虚拟内存盘（ramdisk）位置
init	根据init进程指定要执行的进程
console	提供控制台输出设备及选项
root	指定根文件系统
vmalloc	强制指定可用vmalloc区域的大小

12.5 构建源代码树——request_standard_resources()

该函数将不依赖于平台且被共同管理的源代码信息构建成树状结构。可通过/proc/iomem查看此处构成的信息。下面通过代码12-4进行分析。

:: 代码12-4 arch/arm/kernel/setup.c的request_standard_resources()

```
static void __init
request_standard_resources(struct meminfo *mi, struct machine_desc *mdesc)
{
    struct resource *res;
    int i;

    kernel_code.start   = virt_to_phys(_text);                          ❶
    kernel_code.end     = virt_to_phys(_etext - 1);
    kernel_data.start   = virt_to_phys(_data);
    kernel_data.end     = virt_to_phys(_end - 1);

    for (i = 0; i < mi->nr_banks; i++) {                                ❷
        if (mi->bank[i].size == 0)
            continue;

        res = alloc_bootmem_low(sizeof(*res));                          ❸
        res->name  = "System RAM";
        res->start = mi->bank[i].start;
        res->end   = mi->bank[i].start + mi->bank[i].size - 1;
```

```
        res->flags = IORESOURCE_MEM | IORESOURCE_BUSY;

        request_resource(&iomem_resource, res);                          ❹

        if (kernel_code.start >= res->start &&                           ❺
            kernel_code.end <= res->end)
            request_resource(res, &kernel_code);
        if (kernel_data.start >= res->start &&
            kernel_data.end <= res->end)
            request_resource(res, &kernel_data);
    }

    if (mdesc->video_start) {                                            ❻
        video_ram.start = mdesc->video_start;
        video_ram.end   = mdesc->video_end;
        request_resource(&iomem_resource, &video_ram);
    }

    /*
     * Some machines don't have the possibility of ever
     * possessing lp0, lp1 or lp2
     */
    if (mdesc->reserve_lp0)                                              ❼
        request_resource(&ioport_resource, &lp0);
    if (mdesc->reserve_lp1)
        request_resource(&ioport_resource, &lp1);
    if (mdesc->reserve_lp2)
        request_resource(&ioport_resource, &lp2);
}
```

在代码段❶中能看到kernel_code和kernel_data变量，这些变量的声明如下所示。

:: arch/arm/kernel/setup.c

```
static struct resource mem_res[] = {
    {
        .name = "Video RAM",
        .start = 0,
        .end = 0,
        .flags = IORESOURCE_MEM
    },
    {
        .name = "Kernel text",
        .start = 0,
        .end = 0,
        .flags = IORESOURCE_MEM
    },
    {
        .name = "Kernel data",
        .start = 0,
        .end = 0,
        .flags = IORESOURCE_MEM
    }
```

```
};

#define video_ram   mem_res[0]
#define kernel_code mem_res[1]
#define kernel_data mem_res[2]
```

kernel_code是resource结构体数组mem_res的第一个元素，kernel_data是第二个元素。

代码段❶将各内核代码段的起始/结束的虚拟地址变更为物理地址，并保存到kernel_code。数据段也将起始/结束地址变更为物理地址，保存到kernel_data。

在代码段❷中，利用包含（作为第一个参数传递的）内存信息的结构体（meminfo），对系统所有内存条执行循环，同时执行代码段❸、❹、❺。

在代码段❸中，调用alloc_bootmem_low()分配resource结构体，该函数被定义为alloc_bootmem_low宏，如图12-2所示。最终的结构是调用__alloc_bootmem_no_panic()函数分配内存的。内存分配过程请参考14.7.1节。

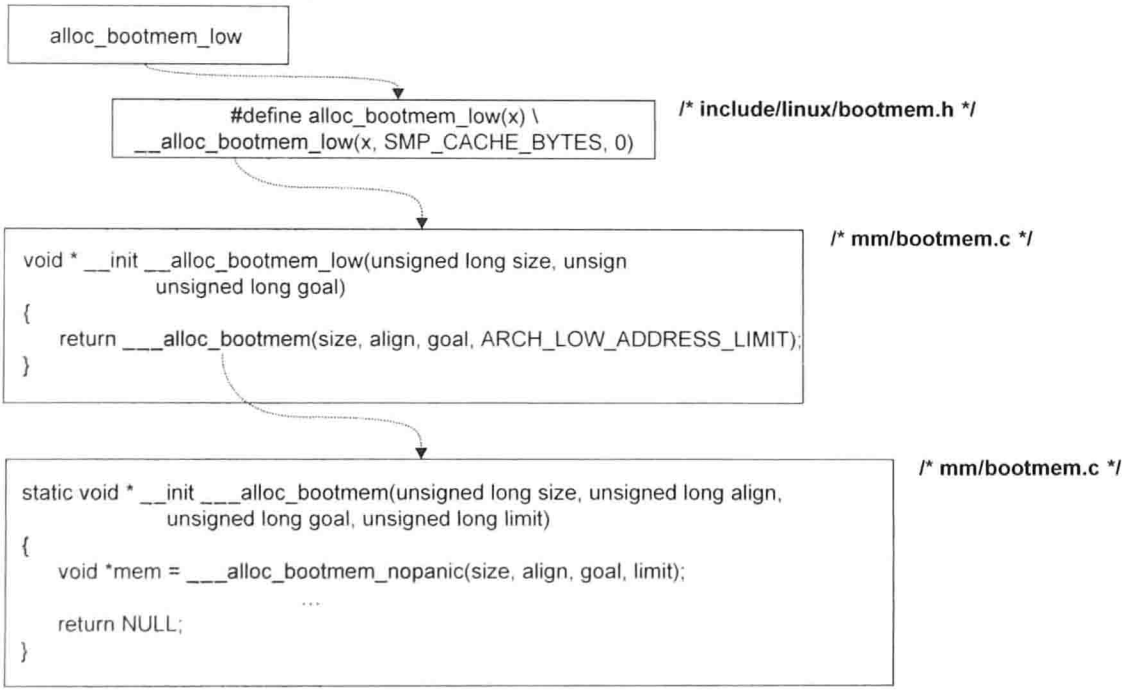


图12-2 alloc_bootmem_low函数调用关系图

在通过alloc_bootmem_low分配的resource结构体中，将name值设置为“System RAM”，并将系统内存条的起始/结束地址分别设置到start、end。利用设置好的值调用代码段❹的requeset_resource()函数，该函数如下所示。

:: kernel/resource.c

```
int request_resource(struct resource *root, struct resource *new)
{
```

```
    struct resource *conflict;

    write_lock(&resource_lock);
    conflict = __request_resource(root, new);
    write_unlock(&resource_lock);
    return conflict ? -EBUSY : 0;
}
```

调用__request_resource()函数前要挂锁（lock），调用后要解锁，以防源代码树在调用过程中变更。实际的源代码树是在__request_resource()函数内构成的。该函数负责构建具有父节点、子节点和同级节点的树。该函数虽然是request_standard_resources()的核心，但只是构建一般树的代码，因此将在本节的最后进行说明。

结束代码段❹后，相当于内存条的区域连接到iomem_resource。然后执行代码段❺，若该区域包含在相应内核代码区域中，则为当前源代码节点添加kernel_code节点；若相当于内核数据区域，则添加kernel_data节点。

现在结束对内存的循环，进入代码段❻。此处利用函数参数接收的机器描述符信息，如果已设置视频随机存取存储器（Video RAM），则调用requst_resource()函数，添加video_ram源代码节点。

代码段❼也依照相同方法，根据机器描述符的设置值向ioport_resource根节点添加lp0、lp1、lp2节点，由此结束request_standard_resources()函数。

如果有2个BANK，且第一个中有内核代码区域及数据区域，并已设置视频随机存取存储器区域，那就执行代码12-4，之后的源代码节点基本形态如图12-3所示。

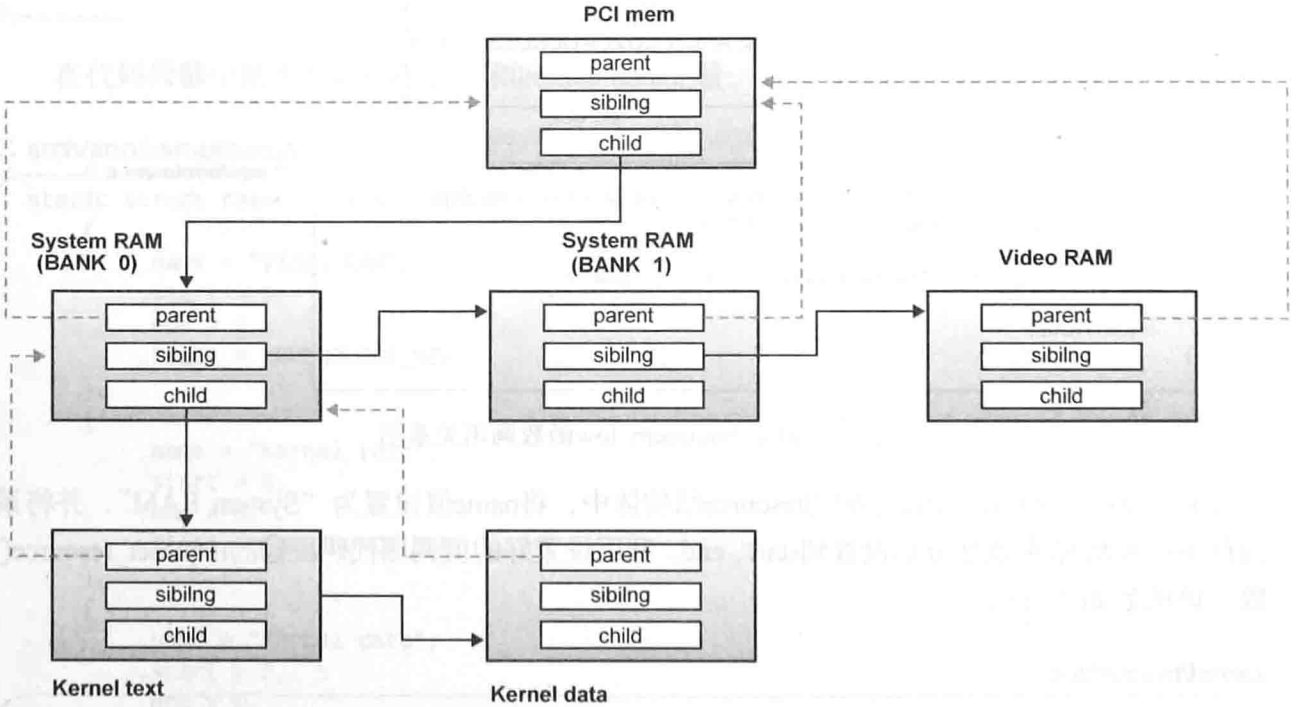


图12-3　执行request_standard_resources后的形态

以上就是__request_resource()函数的全部内容，该函数负责构建具有父节点、子节点和同级节点的树，如代码12-5所示。

:: 代码12-5 kernel/resource.c的__request_resource()

```
static struct resource * __request_resource(struct resource *root,
                                            struct resource *new)
{
    resource_size_t start = new->start;
    resource_size_t end = new->end;
    struct resource *tmp, **p;

    p = &root->child;
    for (;;) {

        tmp = *p;
        if (!tmp || tmp->start > end) {                                    ❶
            new->sibling = tmp;

            /* 下列代码为root->child = new */
            *p = new;
            new->parent = root;
            return NULL;
        }
        p = &tmp->sibling;

        if (tmp->end < start)                                              ❷
            continue;
        return tmp;
    }
}
```

在代码段❶中，将根节点的子节点设置到tmp变量。若tmp变量的值为0，或新节点地址比根节点靠前，就向同级节点设置tmp，并将根节点的子节点设置为新节点。最后，将新节点的父节点设置为根节点。在代码段❷中，若存在根节点的子节点，则对子节点的同级节点执行循环，并用上述方法插入新节点。希望上述代码能对数据结构学习者有所帮助。

补充知识点 查看iomem

可利用proc文件系统查看构建好的iomem，执行下列命令即可查看默认资源信息，还可查看各驱动中设置的已映射的I/O相关源代码。

```
code@kernel$ cat /proc/iomem
  480bd200-480bd2fb : omap3isp
480bd400-480bd4ff : omap-iommu.1
  480bd400-480bd4ff : omap-iommu.1
480bd800-480bd96f : omap3isp
  480bd970-480bd977 : omap3isp
48314000-4831404f : omap_wdt
```

```
   48314000-4831404f : omap_wdt
70000000-70ffffff : vrfb
71000000-71ffffff : vrfb
72000000-72ffffff : vrfb
73000000-73ffffff : vrfb
80000000-87ffffff : System RAM
  8002C000-80350fff : Kernel Text
  80352000-803b5407 : Kernel Data
88000000-8fffffff : System RAM
code@kernel$
```

12.6　初始化 cpu possible 位图——smp_init_cpus()

10.1节讲过的boot_cpu_init()函数通过位图管理系统中的CPU状态信息，其中，存在于系统的所有CPU位图就是cpu_possible_map。在cpu_possible_map中，smp_init_cpus()函数将相当于当前系统CPU核心编号的位设为1。

通过下列函数观察smp_init_cpus()函数。

:: arch/arm/mach-realview/platsmp.c

```
void __init smp_init_cpus(void)
{
    unsigned int i, ncores = get_core_count();

    for (i = 0; i < ncores; i++)
    cpu_set(i, cpu_possible_map);
}
```

ARM结构中的realview平台为SMP，拥有的代码可从该机器获得核心个数。[①]获得核心个数后，在cpu_possible_map中将该核心的位设为1。

获得核心个数的函数是get_core_count()，如代码12-6所示。

:: 代码12-6　arch/arm/mach-realview/platsmp.c的get_core_count()

```
static unsigned int __init get_core_count(void)
{
    unsigned int ncores;
    void __iomem *scu_base = scu_base_addr();                    ❶

    if (scu_base) {
        ncores = __raw_readl(scu_base + SCU_CONFIG);
        ncores = (ncores & 0x03) + 1;                            ❷
    } else
        ncores = 1;
```

① 3.0版以后，omap2、tegra、exynos等诸多平台开始支持SMP。

```
    return ncores;
}
```

在代码段❶中获得scu_base的地址。__iomem指示符通知编译器要访问I/O内存。[①]scu_base_addr()函数在ARM中使用，并获得SCU的基址。SCU作为ARM多核心技术的核心功能，其作用是维护多处理器间数据缓存的一致性，能够通过AXI（高级可扩展接口，Advanced eXtensible Interface）访问内存，并在多处理器试图访问L2缓存时执行仲裁。

代码段❷中，从scu_base地址获得核心的个数（ncores）后，对该值以0x03执行位屏蔽。因此，核心的个数最少1个，最多4个。

12.7 用栈指定各 ARM 异常模式——cpu_init()

在cpu_init()中指定将按照ARM的IRQ、ABORT、SVC、UND（undefined）模式使用的栈空间。通过代码12-7了解cpu_init()函数。

:: 代码12-7 arch/arm/kernel/setup.c的cpu_init()

```
void cpu_init(void)
{
    unsigned int cpu = smp_processor_id();                                    ❶
    struct stack *stk = &stacks[cpu];                                         ❷
    ...
    __asm__ (                                                                 ❸
        "msrcpsr_c, %1\n\t"
        "addsp, %0, %2\n\t"
        "msrcpsr_c, %3\n\t"
        "addsp, %0, %4\n\t"
        "msrcpsr_c, %5\n\t"
        "addsp, %0, %6\n\t"
        "msrcpsr_c, %7"
            :
            :"r" (stk),                                    /* %0 */
             "I" (PSR_F_BIT ¦ PSR_I_BIT ¦ IRQ_MODE),       /* %1 */
             "I" (offsetof(struct stack, irq[0])),         /* %2 */
             "I" (PSR_F_BIT ¦ PSR_I_BIT ¦ ABT_MODE),       /* %3 */
             "I" (offsetof(struct stack, abt[0])),         /* %4 */
             "I" (PSR_F_BIT ¦ PSR_I_BIT ¦ UND_MODE),       /* %5 */
             "I" (offsetof(struct stack, und[0])),         /* %6 */
             "I" (PSR_F_BIT ¦ PSR_I_BIT ¦ SVC_MODE)        /* %7 */
                                                           /* r14 ->lr */
    : "r14");
    ...
}
```

通过代码段❶查找当前处理器的编号。

① 实际编译器会忽略__iomem，因此其作用类似于注释。

代码段❷中出现的stacks是全局数组。stack结构体类型的stacks数组被声明为NR_CPUS（系统能够支持的最大CPU个数）大小，如下所示。

:: arch/arm/kernel/setup.c

```
struct stack {
    u32 irq[3];
    u32 abt[3];
    u32 und[3];
} ____cacheline_aligned;

static struct stack stacks[NR_CPUS];
```

需要注意stack结构体的____cacheline_aligned。分析内核源代码时会经常接触到____cacheline_aligned等指示符。下面是____cacheline_aligned宏的定义。

:: include/linux/cache.h

```
#define ____cacheline_aligned __attribute__((__aligned__(SMP_CACHE_BYTES)))
```

为了让编译器以L1缓存的大小为标准排列结构体成员，这个宏使用__attribute__为stack结构体赋予一些属性。

在代码12-7的代码段❸中需要访问结构的特定寄存器，因此使用了内联汇编（inline-assembly）。将IRQ模式时的栈指针设置为irq[0]，将ABORT模式时的栈指针设置为abt[0]，将UND（undefined）模式时的栈指针设置为und[0]，回到SVC模式后返回。

12.8　初始化以处理异常——early_trap_init()

为了调用异常代码，在early_trap_init()函数中将各处理器及辅助器代码复制到异常向量表基址，并设置CPU域。中断及异常是操作系统中非常重要的部分。详细内容请观察代码12-8。

:: 代码12-8　arch/arm/kernel/traps.c的early_trap_init()

```
void __init early_trap_init(void)
{
    /* exception vectors的基址 */
    unsigned long vectors = CONFIG_VECTORS_BASE;                          ❶

    extern char __stubs_start[], __stubs_end[];
    extern char __vectors_start[], __vectors_end[];
    extern char __kuser_helper_start[], __kuser_helper_end[];
    int kuser_sz = __kuser_helper_end - __kuser_helper_start;

    memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);    ❷
    memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);  ❸
    memcpy((void *)vectors + 0x1000 - kuser_sz, __kuser_helper_start, kuser_sz);  ❹
```

```
    memcpy((void *)KERN_SIGRETURN_CODE, sigreturn_codes,                    ❺
          sizeof(sigreturn_codes));

    flush_icache_range(vectors, vectors + PAGE_SIZE);                       ❻
    modify_domain(DOMAIN_USER, DOMAIN_CLIENT);                              ❼
}
```

在代码段❶中为vectors变量设置CONFIG_VECTORS_BASE值，CONFIG_VECTORS_BASE在内核编译时变更设置值。在ARM处理器中，根据硬件环境可获得0x00000000、0xFFFF0000这2个值。在当前代码中，我们将其假定为low vector（0x00000000）进行说明。

在代码段❷中复制发生异常时要执行的代码。复制的异常向量表中有跳转到相应处理器的代码，但现有状态下无效。执行代码段❸以后，在跳转对象地址中复制要执行的代码。

代码段❸只将__stubs_start到__stubs_end复制到vector + 0x200。__stubs_start和__stubs_end之间定义了按不同模式执行的异常处理器代码。

代码段❹也从__kuser_helper_start复制了kuser_sz大小到__vector_start + 0x1000 – kuser_sz。该区域中有__kuser_memory_barrier、kernel_cmpxchg、kernel_get_tls、kernel_helper_version等辅助器代码。

在代码段❺中，通过memcpy将sigreturn_codes复制到KERN_SIGRETURN_CODE地址。然后，KERN_SIGRETURN_CODE在arch/arm/kernel/signal.h中被定义为CONFIG_VECTORS_BASE + 0x00000500，sigreturn_codes如下所示。

:: arch/arm/kernel/signal.c

```
#define SWI_SYS_SIGRETURN    (0xef000000|(__NR_sigreturn)|
            (__NR_OABI_SYSCALL_BASE))
#define SWI_SYS_RT_SIGRETURN    (0xef000000|
            (__NR_rt_sigreturn)|(__NR_OABI_SYSCALL_BASE))

#define MOV_R7_NR_SIGRETURN (0xe3a07000 |
            (__NR_sigreturn - __NR_SYSCALL_BASE))
#define MOV_R7_NR_RT_SIGRETURN  (0xe3a07000 |
            (__NR_rt_sigreturn - __NR_SYSCALL_BASE))

#define SWI_THUMB_SIGRETURN
            (0xdf00 << 16 | 0x2700 | (__NR_sigreturn - __NR_SYSCALL_BASE))
#define SWI_THUMB_RT_SIGRETURN
            (0xdf00 << 16 | 0x2700 |
            (__NR_rt_sigreturn - __NR_SYSCALL_BASE))

const unsigned long sigreturn_codes[7] = {
    MOV_R7_NR_SIGRETURN,    SWI_SYS_SIGRETURN,    SWI_THUMB_SIGRETURN,
    MOV_R7_NR_RT_SIGRETURN, SWI_SYS_RT_SIGRETURN, SWI_THUMB_RT_SIGRETURN,
};
```

sigreturn_codes的实际指令为二进制形态，将各二进制代码转换为汇编指令，如表12-2所示。假设，__NR_SYSCALL_BASE为0，__NR_OABI_SYSCALL_BASE为0x900000。

表12-2　宏和二进制的关系

宏	汇编指令	二进制代码
MOV_R7_NR_SIGRETURN	mov r7,0x77	(0xe3a07077)
SWI_SYS_SIGRETURN	swi 0x77	(0xef900077)
SWI_THUMB_SIGRETURN	swi 0x77	(0xdf77)
MOV_R7_NR_RT_SIGRETURN	mov r7,0xad	(0xe3a070ad)
SWI_SYS_RT_SIGRETURN	swi 0xad	(0xef9000ad)
SWI_THUMB_RT_SIGRETURN	swi 0xad	(0xdfad)

执行代码12-4的代码段❺之后，图12-4如下所示。

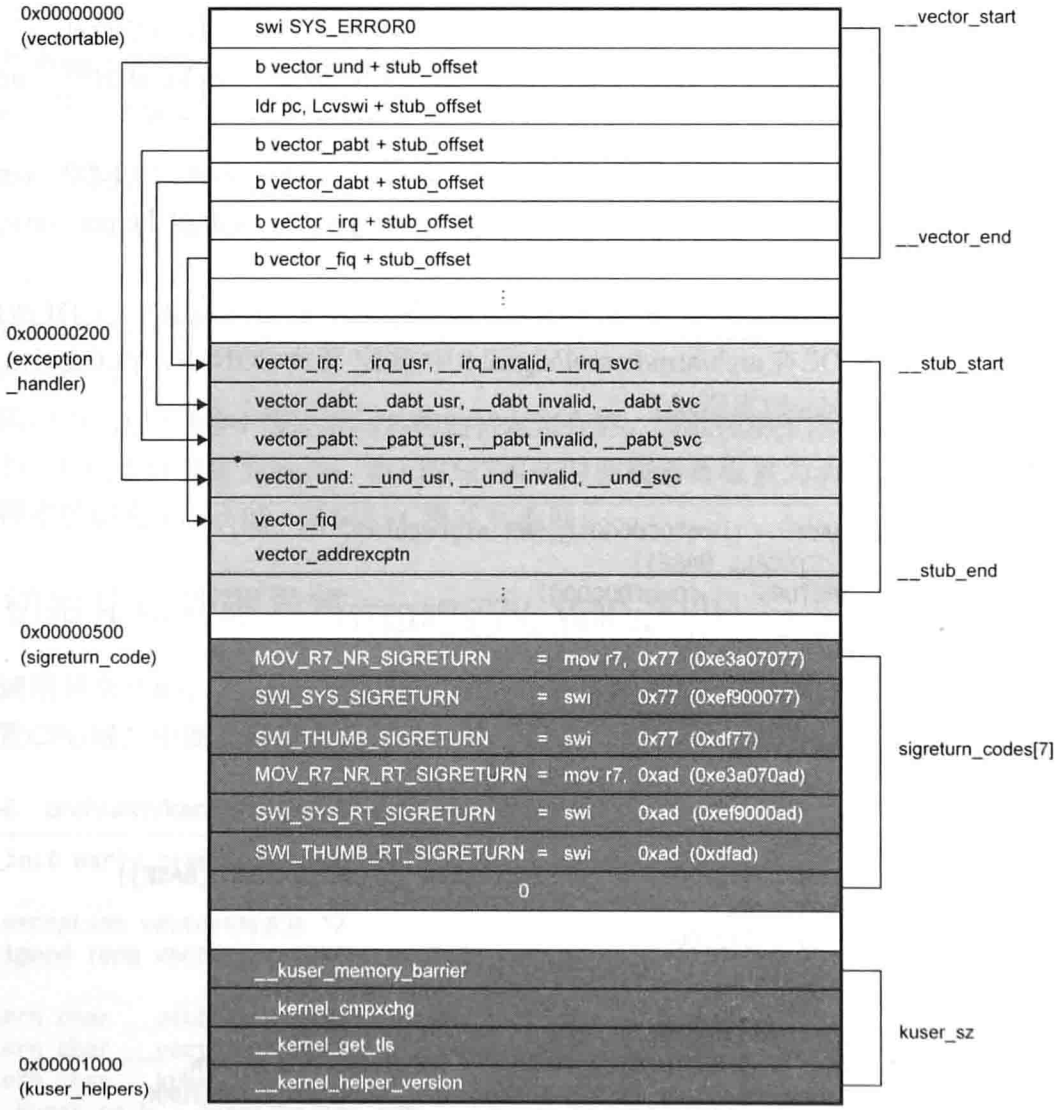


图12-4　exception_vector table

将发生异常之后要执行的代码复制到vector区域。vector区域是可执行区域，若未得到缓冲则不反映复制的代码。因此，执行前清除指令缓存（参考代码段❻）才能反映已变更的内存，并加

以适用。最后，调用代码段❼设置CPU域，这样就完成了early_trap_init()函数的所有任务。

虽然early_trap_init()函数已经结束，但更重要的是各异常处理器的作用。先观察实际的异常向量是如何定义的，下一节将逐一介绍各异常处理器。

异常向量表的1项只有32位。因此，不能在内存中进行所有处理，必须跳转到处理该异常的函数。接下来观察各异常处理器是如何定义及调用的。

下列代码描述了实际定义在Linux内核中的异常向量表。

:: arch/arm/kernel/entry-armv.S

```
.globl  __vectors_start
__vectors_start:
    swi SYS_ERROR0
    b   vector_und + stubs_offset
    ldr pc, .LCvswi + stubs_offset
    b   vector_pabt + stubs_offset
    b   vector_dabt + stubs_offset
    b   vector_addrexcptn + stubs_offset
    b   vector_irq + stubs_offset
    b   vector_fiq + stubs_offset

    .globl  __vectors_end
__vectors_end:
```

列表内有分支为各处理器的代码。除系统调用外，均由vector_stub按相同方式定义。先观察这段汇编宏。

:: 代码12-9 arch/arm/kernel/entry-armv.S的vector_stub宏函数

```
.macro  vector_stub, name, mode, correction=0                    ❶
    .align  5

vector_\name:                                                    ❷
    .if \correction
    sub lr, lr, #\correction
    .endif


    stmia   sp, {r0, lr}                    @ save r0, lr        ❸
    mrs lr, spsr
    str lr, [sp, #8]                        @ save spsr

    mrs r0, cpsr                                                 ❹
    eor r0, r0, #(\mode ^ SVC_MODE)
    msr spsr_cxsf, r0


    and lr, lr, #0x0f                                            ❺
    mov r0, sp
    ldr lr, [pc, lr, lsl #2]
    movs    pc, lr                                               ❻
ENDPROC(vector_\name)
    .endm
```

在代码段❶中定义vector_stub宏，这个宏接收要生成的标签名、要变更的模式、LR偏差校正等参数。通过这个参数，在代码段❷中用该名称定义函数标签，校正处理异常之后返回的寄存器lr。

为了处理异常后修复上下文，在代码段❸中将寄存器r0和发生异常模式的lr、spsr保存到栈。存入栈后在代码段❹中禁用中断，将spsr（以前模式的状态寄存器）设置为SVC32，准备进入SVC模式。

完成进入SVC模式的准备工作后，通过代码段❺转换为SVC模式，并准备跳转到在当前代码之后出现的异常处理器列表。该列表中有符合各模式的异常处理器地址。实际进入SVC的操作是通过代码段❻的movs pc,lr指令完成的。该指令将寄存器lr的值加载到pc，并将spsr值复制到cpsr。

下节将逐一讲解各模式的异常处理器。

补充知识点　ARM——异常发生时校正LR

发生异常（exception）时该如何处理呢？如前所述，从硬件角度上讲，PC要跳转到异常向量表并调用相应处理器，之后须再次返回某处。

之前讲过，寄存器LR保存着要返回的地址。那么，处理异常后要返回的地址不就是LR吗？如果真有这么简单那该多好！返回的地址根据异常的种类有所不同，可能是异常发生的地址，也可能是发生异常的代码之前或之后。

考虑返回何处前，需要先观察各异常是在何种管道阶段发生的。图12-5描述了5个阶段管道步骤。

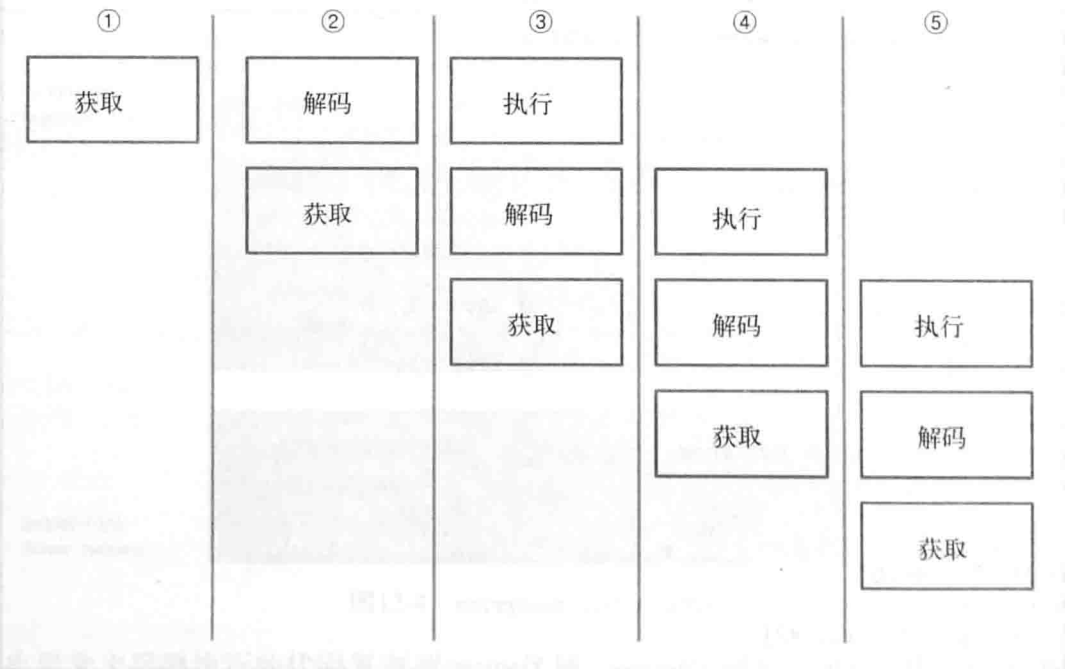


图12-5　管道步骤

一般PC指向的地址比当前执行的地址提前2个阶段，即具有的是“获取”（FETCH）阶段的地址，而非“执行”（EXECUTE）阶段的地址。例如，ARM模式时，如果当前执行的地址是0x1000，那PC值就指向2个周期（cycle）前的地址0x1008。希望大家记住这一点。异常的处理方法有很多，此处主要讲解Linux中采用的方法。

1. 重启处理器（Reset Handler）

由硬件信号导致，故无法知晓到底是在哪一阶段发生的，也无需设置返回地址。

2. 未定义处理器（Undefined Handler）

“解码”（Decode）阶段无法分析指令时发生。当前PC值具有“获取”阶段的地址。发生未定义异常时运行处理器，发生未定义异常后必须返回到下一地址。由于当前寄存器LR_<undefined>中有发生异常地址+4（ARM模式时）的值，故不校正寄存器LR。

3. 预取指令中止处理器（Prefetch Abort Handler）

在“获取”阶段无访问权限或试图读取错误地址时发生。当前PC值指向发生预取指令中止（Prefetch Abort）异常的地址。但是LR中会进入加1个周期的地址（ARM时+4），而不是发生预取指令中止的地址。

Linux允许预取指令中止处理器中执行发生预取指令中止异常的指令，运行处理器后，通过寄存器LR_<prefetch_abort>-4重新执行发生异常的指令。

4. 数据中止处理器（Data Abort Handler）

实际运行后才能了解数据有效性。因此，在EXECUTE阶段发生数据中止（Data Abort）异常。由于当前PC值指向Fetch阶段，所以具有加了2个周期的地址。

Linux允许数据中止处理器中执行发生数据中止异常的指令，运行处理器后，通过寄存器LR_<data_abort>-8重新执行发生异常的指令。

5. SWI（软件中断）处理器（Software Interrupt Handler）

发生在“解码”阶段。系统程序员为了转换为SVC模式而标明命令，由此引发SWI异常。和未定义异常一样，PC指向1个周期前的地址。在SWI处理器中执行需要的功能，并执行SWI异常的下一个指令，即可使返回地址变成寄存器LR。

6. IRQ处理器

执行某种指令的过程中由硬件引发。PC依然指向2个周期前的地址——“获取”阶段。中断处理器结束处理后，要执行中断发生地点的下一个指令，因此，通过LR_<irq>-4校正寄存器LR。

7. FIQ Handler

执行与IRQ相同的处理。但Linux中不使用FIQ。

12.9 查看中断处理器函数

中断是外部某种硬件信号引发的异步事件。我们每次按下键盘均会发生中断，周期性的硬件

计时器也会引发周期性中断。接下来看看Linux内核会执行何种操作以处理中断。图12-6表示发生中断时调用的函数流程。

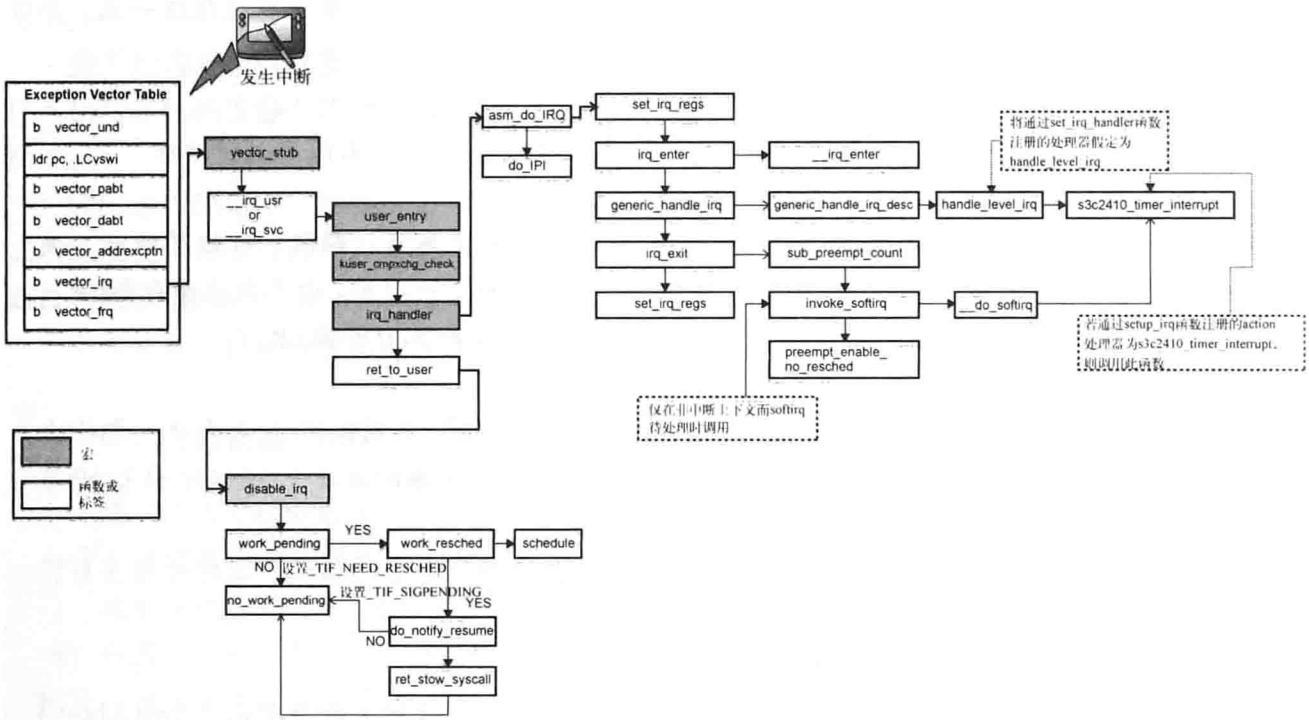


图12-6　中断处理过程

下面了解一下中断处理器。中断处理器在异常向量表中定义如下。

:: arch/arm/kernel/entry-armv.S

```
b   vector_irq + stubs_offset
```

我们先观察通过vector_stub定义的vector_irq。vector_irq的定义如下所示，代入之前讲解的vector_stub即可知晓。

:: arch/arm/kernel/entry-armv.S

```
vector_stub irq, IRQ_MODE, 4
```

定义了vector_irq标签，进入SVC模式并跳转到模式处理器函数。模式处理器函数如下所示。

```
.long   __irq_usr
.long   __irq_invalid
.long   __irq_invalid
.long   __irq_svc
.long   __irq_invalid
.long   __irq_invalid
```

所有模式中都会发生中断，因此，每个模式均具有自己的处理器。Linux内核中只使用用户模式和SVC模式，因此，只有在这2个模式中处理的处理器才有效。所以要仔细观察用户模式下发生中断时执行处理的__irq_usr函数，以及SVC模式下发生中断时执行处理的__irq_svc函数。这2个函数没有什么太大区别，只要理解下面出现的__irq_usr代码，那么分析__irq_svc并不难。

:: 代码12-10 arch/arm/kernel/entry-armv.S的__irq_usr标签

```
__irq_usr:
    usr_entry                                                    ❶
    kuser_cmpxchg_check                                          ❷

get_thread_info tsk                                              ❸
#ifdef CONFIG_PREEMPT
    ldr r8, [tsk, #TI_PREEMPT]      @ get preempt count
    add r7, r8, #1                  @ increment it
    str r7, [tsk, #TI_PREEMPT]
#endif

    irq_handler                                                  ❹
#ifdef CONFIG_PREEMPT
    ldr r0, [tsk, #TI_PREEMPT]                                   ❺
    str r8, [tsk, #TI_PREEMPT]
    teq r0, r7
    strne   r0, [r0, -r0]
#endif
    mov why, #0                                                  ❻
    b   ret_to_user

 UNWIND(.fnend      )
ENDPROC(__irq_usr)
```

通过代码段❶的usr_entry宏，将用户模式的上下文保存到SVC模式的栈。图12-7描述的是将当前寄存器值存入SVC模式的栈。寄存器内的值只是为了讲解而填充的无意义的值。重要的是，我们临时使用寄存器r0以复制上下文，因此，发生异常时具有的寄存器r0的值也一并保存。

在ARM中，用户模式下不具备系统调用或任何异常捕获（trap）时，也可以进行原子操作（atomic operation）。若执行这种操作的过程中发生中断，可能引发问题。因此，在代码段❷中处理该异常。

代码段❸为tsk获得当前任务的thread_info结构体地址。下面是tsk的定义。

:: arch/arm/kernel/entry-header.S

```
tsk .req    r9      @ current thread_info
```

接着将之前获得的thread_info结构体的成员变量preempt_count加1，使得无法执行抢占。

R0	0
R1	0x112
R2	0x114
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11 (fp)	0x5555
R12 (ip)	0x6666
R13 (lr)	0xv7ff6230
R14 (sp)	0xbffff4fc
R15 (pc)	0x8048318

用户模式上下文



R0	0x83728
R1	0x112
R2	0x114
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11 (fp)	0x5555
R12 (ip)	0x6666
R13 (lr)	0xb7ff6230
R14 (sp)	0xbffff4fc
R15 (pc)	0x8048318
cpsr	USR_MODE
Original_r0	0

SVC模式栈

图12-7　复制上下文

在代码段❹中通过irq_handler宏查看中断状态，无论发生何种中断均获取该中断号，并调用asm_do_IRQ()函数。一般的中断处理是在这里执行的，结束该函数后会调用（只能在SMP中特别处理的）inter-processor中断处理函数——asm_do_IPI()函数。

代码段❸中增加了preempt_count，在代码段❺中再次恢复原值。最后，在代码段❻中完成所有操作，调用ret_to_user返回发生过中断的用户模式。

这就是__irq_usr的大致流程。与__irq_svc的不同之处在于，第一段代码通过svc_entry（而非use_entry）保存上下文，而最后并未调用ret_to_user，而是执行下列指令。

```
ldmia  sp,{r0 - pc}^
```

通过ldmia指令将SVC模式栈中的上下文复制到寄存器。最后的^后缀将SPSR自动复制到CPSR。此外还有一些其他代码，但中断处理器的核心是通过irq_handler宏调用asm_do_IRQ()函数。想进一步了解的读者请直接分析__irq_svc。

我们需要进一步了解代码段❹中调用的asm_do_IRQ()和代码段❻中调用的ret_to_user，下面逐一进行讲解。

12.9.1 调用IRQ处理器——asm_do_IRQ()

前面讲过发生中断时从最初进入点执行的全部内容。接下来仔细观察之前注册的中断处理器函数是如何调用的。

:: 代码12-11 arch/arm/kernel/irq.c的asm_do_IRQ()

```
asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);                    ❶

    irq_enter();                                                      ❷

    if (irq >= NR_IRQS)
        handle_bad_irq(irq, &bad_irq_desc);
    else
        generic_handle_irq(irq);                                      ❸

    /* AT91 specific workaround */
    irq_finish(irq);

    irq_exit();                                                       ❹
    set_irq_regs(old_regs);                                           ❺
}
```

代码段❶将发生中断时保存的上下文保存到per-cpu变量__irq_regs，并返回之前的值。在代码段❷的irq_enter()函数内，将当前任务的preempt_count增加HARDIRQ_OFFSET大小。

init_IRQ()函数将在18.8节详细讲解，该函数通过set_irq_handler()注册符合各结构的中断处理器。注册好的中断处理器在代码段❸的generic_handle_irq()中调用。

如果通过set_irq_handler注册的处理器函数为handle_level_irq()，则该函数会被调用。在handle_level_irq()函数内部则调用（通过setup_irq()注册的）irqaction结构体的处理器函数。

19.5节中对硬件计时器执行初始化的time_init()用setup_irq()注册s3c2410_timer_irq结构体。因此，会调用通过该结构体的处理器注册的s3c2410_timer_interrupt()。

将（代码段❷中增加的）当前任务的preempt_count在代码段❹中再次减去HARDIRQ_OFFSET大小。若非中断上下文，而是softirq处于待处理（pending）状态，那就调用invoke_softirq()，从而调用通过setup_irq()注册的irqaction结构体的处理器。之后再次调用preempt_enable_no_resched()函数，将preempt_count减去1。

最后，重新恢复代码段❶中返回的原上下文值，结束该函数。这就是注册的中断处理器函数调用过程。现在只需要回到中断发生的时间点。

12.9.2 返回中断之前——ret_to_user标签

在用户模式下发生中断时，需要返回之前“正在执行”的状态。通过下列代码了解一下利用ret_to_user返回中断发生前要执行的操作和返回方法。

:: 代码12-12　arch/arm/kernel/entry-common.S的ret_to_user标签

```
ENTRY(ret_to_user)
ret_slow_syscall:
    disable_irq                                                    ❶
    ldr r1, [tsk, #TI_FLAGS]                                       ❷
    tst r1, #_TIF_WORK_MASK
    bne work_pending                                               ❸
no_work_pending:
arch_ret_to_user r1, lr                                            ❹

    ldr r1, [sp, #S_PSR]              @ 获得调用模式的cpsr          ❺
    ldr lr, [sp, #S_PC]!              @ 获得pc值
    msr spsr_cxsf, r1                 @ 保存spsr_svc值
    ldmdb   sp, {r0 - lr}^            @ 获得调用模式的r0~ lr值
    mov r0, r0
    add sp, sp, #S_FRAME_SIZE - S_PC                               ❻
    movs    pc, lr                                                 ❼
ENDPROC(ret_to_user)
```

代码段❶以禁止中断开始，在代码段❷中查看当前thread_info结构体的flags值。若下级8位中已设置了值，就跳转到代码段❹的work_pending，否则继续执行代码段❹的arch_ret_to_user后的代码。我们最后再讲work_pending。

在arch_ret_to_user内，返回用户模式前执行从属于结构的操作，但大部分结构中不执行任何操作。在代码段❺中，将发生中断时保存到栈的上下文恢复到用户模式寄存器。现在不需要栈中保存的上下文，因此在代码段❻中将栈指针缩小到保存上下文之前的大小。执行代码段❼，从而再次返回用户模式，继续执行之前的代码。

下面分析代码12-2代码段❸中调用的work_pending，如代码12-13所示。

:: 代码12-13　arch/arm/kernel/irq.c的work_pending标签

```
work_pending:
    tst r1, #_TIF_NEED_RESCHED                                     ❶
    bne work_resched
    tst r1, #_TIF_SIGPENDING                                       ❷
    beq no_work_pending
    mov r0, sp                  @ 'regs'
    mov r2, why                 @ 'syscall'
    bl  do_notify_resume                                           ❸
    b   ret_slow_syscall        @ Check work again                 ❹

work_resched:
    bl  schedule
```

代码段❶中，如果当前任务的thread_info结构体的flags值中已设置_TIF_NEED_RESCHED，就跳转到work_resched，并调用schedule()函数转换为新任务。该flags值需要重新调度时进行设置。例如，利用nice命令变更优先顺序值时，或耗费所有进程分配时间时，由scheduler_tick()函数设置。

在代码段❷中再次检查flags值，查看是否设置_TIF_SIGPEDING。还有未处理信号的状态下，需要再次发送信号时设置该flags值。

若未设置该值，就回到代码12-12的代码段❹，返回用户模式。反之，若已设置_TIF_SIGPENDING值，则调用代码段❸的do_notify_resume()，在其内部调用do_signal()函数。

顾名思义，do_signal()函数负责处理信号。结束处理后，执行代码段❹并再次执行ret_to_user，以便反复执行本节流程。

第 13 章

13 设置处理器——setup_processor()

本章我们主要了解setup_arch()函数内调用的setup_processor()函数。从setup_processor()函数的名称可以看出，该函数用于设置正在执行当前内核的处理器信息。

13.1 查看 setup_processor()结构

setup_processor()函数默认查找保存相应处理器信息的结构体，从结构体对设置好的处理器相关变量值进行全局设置。代码13-1是setup_processor()函数。本节我们将setup_processor()函数分为8个代码段进行讲解。

:: 代码13-1 arch/arm/kernel/setup.c的setup_processor()

```
extern struct proc_info_list *lookup_processor_type(unsigned int);

static void __init setup_processor(void)
{
    struct proc_info_list *list;                                          ❶

    list = lookup_processor_type(read_cpuid_id());                        ❷
    if (!list) {
        printk("CPU configuration botched (ID %08x), unable "             ❸
               "to continue.\n", read_cpuid_id());
        while (1);
    }

    cpu_name = list->cpu_name;                                            ❹

#ifdef MULTI_CPU
    processor = *list->proc;
#endif
#ifdef MULTI_TLB
    cpu_tlb = *list->tlb;
#endif
#ifdef MULTI_USER
    cpu_user = *list->user;
```

```
#endif
#ifdef MULTI_CACHE
    cpu_cache = *list->cache;
#endif

    printk("CPU: %s [%08x] revision %d (ARMv%s), cr=%08lx\n",                    ❺
           cpu_name, read_cpuid_id(), read_cpuid_id() & 15,
           proc_arch[cpu_architecture()], cr_alignment);

    sprintf(init_utsname()->machine, "%s%c", list->arch_name, ENDIANNESS);      ❻
    sprintf(elf_platform, "%s%c", list->elf_name, ENDIANNESS);

    ...

    cacheid_init();                                                             ❼

    cpu_proc_init();                                                            ❽
}
```

代码段❶的proc_info_list结构体已在7.2.2节讲过，该结构体是包含处理器信息的结构体。

代码段❷中利用（通过read_cpuid_id()函数得出的）CPU ID在lookup_processor_type()函数内查找与该CPU对应的proc_info_list结构体的值。read_cpuid_id()函数和lookup_processor_type()函数将在13.2节和13.3节进一步说明。

代码段❸中，未找到相应处理器信息时，通过prink输出错误消息，并进入无限循环状态。

代码段❹中，未找到相应处理器信息时，从proc_info_list结构体读取值，并将该值保存到被全局声明的cpu_name、processor、cpu_tlb、cpu_user、cpu_cache。

在代码段❺中通过prink()函数输出处理器名、CPU ID、结构信息、控制寄存器值cr_alignment。[①]此处使用cpu_architecture()函数获得处理器结构信息（参考13.4节）。

在代码段❻中将proc_info_list结构体的arch_name成员变量和elf_name变量的值分别复制到具有内核名称和版本信息的utsname结构体的machine成员以及全局变量elf_platform。

在代码段❼中调用cacheid_init()函数，该函数能够获得处理器的缓存类型，并将其保存到全局变量cacheid（参考13.5节）。

代码段❽的cpu_proc_init()函数只调用包含相应处理器信息的proc_info_list结构体的_proc_init()函数（参考13.6节）。

13.2 查找 CPU ID——read_cpuid_id()

read_cpuid_id()函数只将CPU ID用作参数调用read_cpuid()。

① cpu_alignment的值是通过arch/arm/include/asm/system.h的get_cr()函数从控制寄存器读取并设置的。

:: arch/arm/include/asm/cputype.h

```
#define CPUID_ID                    0
#define CPUID_CACHETYPE             1
#define CPUID_TCM                   2
#define CPUID_TLBTYPE               3

static inline unsigned int __attribute_const__ read_cpuid_id(void)
{
    return read_cpuid(CPUID_ID);
}
```

read_cpuid_id()函数只调用read_cpuid()函数，该函数定义如代码13-2所示。

:: 代码13-2　arch/arm/include/asm/cputype.h的read_cpuid()

```
#ifdef CONFIG_CPU_CP15
#define read_cpuid(reg)                                         \
    ({                                                          \
        unsigned int __val;                                     \
        asm("mrc    p15, 0, %0, c0, c0, " __stringify(reg)      \                    ❶
            : "=r" (__val)                                      \
            :                                                   \
            : "cc");                                            \
        __val;                                                  \
    })
#else
extern unsigned int processor_id;                                                    ❷
#define read_cpuid(reg) (processor_id)
#endif
```

这段代码可分为2个部分。代码段❶中，处理器具有协处理器时，将CP15的c0寄存器值保存到局部变量__val。[①]该值将成为read_cpuid()函数的结果值。

代码段❷中，如果没有协处理器，则外部声明的processor_id值将成为read_cpuid()函数的结果值。代码段❶中使用的__stringify(reg)的定义如下所示。

:: include/linux/stringify.h

```
#define __stringify_1(x...) #x
#define __stringify(x...)   __stringify_1(x)
```

为了理解__stringify(x)，需要仔细观察CP15指令，从CP15读取c0值的指令结构如下所示。

```
mrc p15, <opcode_1>, <Rd>, c0, c0, <opecode_2>
```

为获得主ID寄存器的值，需要将opcode_1和opcode_2均指定为0。

① CP15的c0寄存器是主ID寄存器，具有保存处理器信息的设备ID代码。

```
mrc p15, 0, <Rd>, c0, c0, 0
```

最终，__springify(x)将opcode_2的值作成字符，以编写mrc命令字符串。

13.3 查找处理器信息——lookup_processor_type()

重新回顾lookup_processor_type()函数。我们之前通过read_cpuid_id()函数获得了当前运行的处理器的CPU ID。lookup_processor_type()函数再次使用记述在__lookup_processor_type标签里的汇编代码（7.2.1节）。

为了能在C代码内调用__lookup_processor_type中记述的汇编代码，按如下所示将lookup_processor_ttype作为函数注册到符号列表。

:: arch/arm/kernel/head-common.S

```
ENTRY(lookup_processor_type)
    stmfd   sp!, {r4 - r7, r9, lr}
    mov r9, r0
    bl  __lookup_processor_type
    mov r0, r5
    ldmfd   sp!, {r4 - r7, r9, pc}
ENDPROC(lookup_processor_type)
```

最终，lookup_processor_type()函数会一同执行__lookup_processor_type标签里的汇编代码，从而获得包含当前处理器信息的结构体proc_info_list的指针。

13.4 查找处理器结构信息——cpu_architecture()

下列代码描述了cpu_architecture()函数，看起来有些复杂。该函数通过read_cpuid_id()函数从CP15的c0寄存器获得CPU ID之后，与能够区分CPU结构的值执行AND运算，以寻找CPU的结构。

:: arch/arm/kernel/setup.c

```
int cpu_architecture(void)
{
    int cpu_arch;

    if ((read_cpuid_id() & 0x0008f000) == 0) {
        cpu_arch = CPU_ARCH_UNKNOWN;
    } else if ((read_cpuid_id() & 0x0008f000) == 0x00007000) {
        cpu_arch = (read_cpuid_id() & (1 << 23)) ? CPU_ARCH_ARMv4T : CPU_ARCH_ARMv3;
    } else if ((read_cpuid_id() & 0x00080000) == 0x00000000) {
        cpu_arch = (read_cpuid_id() >> 16) & 7;
        if (cpu_arch)
            cpu_arch += CPU_ARCH_ARMv3;
    /* arm11 */
    } else if ((read_cpuid_id() & 0x000f0000) == 0x000f0000) {
        unsigned int mmfr0;
```

```
        /* armv7 */

        asm("mrc    p15, 0, %0, c0, c1, 4"
            : "=r" (mmfr0));
        if ((mmfr0 & 0x0000000f) == 0x00000003 ||
            (mmfr0 & 0x000000f0) == 0x00000030)
            cpu_arch = CPU_ARCH_ARMv7;
        else if ((mmfr0 & 0x0000000f) == 0x00000002 ||
             (mmfr0 & 0x000000f0) == 0x00000020)
            cpu_arch = CPU_ARCH_ARMv6;
        else
            cpu_arch = CPU_ARCH_UNKNOWN;
    } else
        cpu_arch = CPU_ARCH_UNKNOWN;

    return cpu_arch;
}
```

查找到的CPU结构信息定义如下。

:: arch/arm/include/asm/system.h

```
#define CPU_ARCH_UNKNOWN    0
#define CPU_ARCH_ARMv3      1
#define CPU_ARCH_ARMv4      2
#define CPU_ARCH_ARMv4T     3
#define CPU_ARCH_ARMv5      4
#define CPU_ARCH_ARMv5T     5
#define CPU_ARCH_ARMv5TE    6
#define CPU_ARCH_ARMv5TEJ   7
#define CPU_ARCH_ARMv6      8
#define CPU_ARCH_ARMv7      9
```

将这些值作为索引，输出proc_arch[]数组中保存的对应于当前处理器结构的字符。proc_arch[]数组的定义如下所示。

:: arch/arm/kernel/setup.c

```
static const char *proc_arch[  ] = {
    "undefined/unknown",
    "3",    "4",
    "4T",
    "5",
    "5T",
    "5TE",
    "5TEJ",
    "6TEJ",
    "7",
    "?(11)",
    "?(12)",
```

```
    "?(13)",
    "?(14)",
    "?(15)",
    "?(16)",
    "?(17)",
};
```

实际输出的示例如下所示。

```
CPU: ARMv7 Processor [411fc083] revision 3 (ARMv7), cr=10c5387f
```

补充知识点 utsname结构体

内核中utsname结构体的定义如下所示，它提供内核版本和发布信息等。

:: include/linux/utsname.h

```
struct utsname
{
  char sysname[65];
  char nodename[65];
  char release[65];
  char version[65];
  char machine[65];
  char domainname[65];
};
```

在用户空间访问utsname结构体信息的函数是uname()，其定义如下所示。

```
#include <sys/utsname.h>

int uname(struct utsname *buf)
```

下面通过示例查看其使用方法。

:: utsname.c

```
#include <sys/utsname.h>

#include <stdio.h>

int main()
{
    struct utsname buf;
    uname(&buf);

    printf("%s : %s : %s\n", buf.sysname, buf.release, buf.machine);
    return 0;
}
```

运行结果

```
code@kernel$ gcc -o utsname utsname.c
code@kernel$
code@kernel$ ./utsname
Linux : 2.6.35-28-generic : armv7l
code@kernel$
```

※ 参考文献：http://www.joinc.co.kr/modules/moniwiki/wiki.php/man/2/uname



补充知识点　ENDIANNESS——在内核中了解字节顺序

内核中查看字节顺序的宏就是ENDIANNESS，它在arch/arm/kernel/setup.c文件中定义如下。

```
#define ENDIANNESS ((char)endian_test.l)
```

endian_test是union型的变量，定义为具有4字节大小的char型变量和unsigned long型变量（参考图13-1）。

```
static union {
        char c[4];
        unsigned long l;
} endian_test __initdata = { { 'l', '?', '?', 'b' } };
```

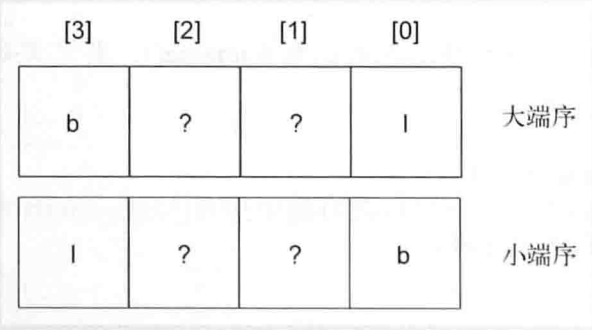


图13-1　endian_test数组

将unsigned long型转换为char型后，会选择下级1字节。大端续（Big Endian）时下级保存b，小端序（Little Endian）时下级保存l，从而执行排序。

13.5　查找处理器缓存类型_cacheid_init()

cacheid_init()函数的定义如代码13-3所示。cacheid_init()函数看起来有些复杂，但其作用非常简单：负责查找处理器的缓存类型并保存到全局变量cacheid。

:: 代码13-3 arch/arm/kernel/setup.c的cacheid_init()

```
static void __init cacheid_init(void)
{
    unsigned int cachetype = read_cpuid_cachetype();                    ❶
    unsigned int arch = cpu_architecture();

    if (arch >= CPU_ARCH_ARMv6) {                                       ❷
        if ((cachetype & (7 << 29)) == 4 << 29) {
            /* ARMv7 register format */
            cacheid = CACHEID_VIPT_NONALIASING;
            if ((cachetype & (3 << 14)) == 1 << 14)
                cacheid |= CACHEID_ASID_TAGGED;
        }
        else if (cachetype & (1 << 23))
            cacheid = CACHEID_VIPT_ALIASING;
        else
            cacheid = CACHEID_VIPT_NONALIASING;
    } else {
        cacheid = CACHEID_VIVT;
    }
   printk("CPU: %s data cache, %s instruction cache\n",                  ❸
        cache_is_vivt() ? "VIVT" :
        cache_is_vipt_aliasing() ? "VIPT aliasing" :
        cache_is_vipt_nonaliasing() ? "VIPT nonaliasing" : "unknown",
        cache_is_vivt() ? "VIVT" :
        icache_is_vivt_asid_tagged() ? "VIVT ASID tagged" :
        cache_is_vipt_aliasing() ? "VIPT aliasing" :
        cache_is_vipt_nonaliasing() ? "VIPT nonaliasing" : "unknown");
}
```

在代码段❶中，调用read_cpuid_cachetype()函数获得缓存类型，而read_cpuid_cachetype()函数和read_cpuid_id()函数都在内部调用read_cpuid()。只是参数从CPUID_ID变更为CPUID_CACHETYPE。下面是read_cpuid_cachetype()函数的定义。

:: arch/arm/include/asm/cputype.h

```
#define CPUID_ID                        0
#define CPUID_CACHETYPE                 1
#define CPUID_TCM                       2
#define CPUID_TLBTYPE                   3

static inline unsigned int __attribute_const__ read_cpuid_cachetype(void)
{
    return read_cpuid(CPUID_CACHETYPE);
}
```

代码段❷中，对v6以上的ARM结构读取求出的缓存类型值的特定位，并设置缓存ID值；对于v6以下的ARM结构，则将全部缓存ID设置为VIVT。

代码段❸中，通过printk()函数输出处理器具有的类型。输出示例如下所示。

```
CPU: VIPT nonaliasing data cache, VIPT nonaliasing instruction cache
```

补充知识点　缓存类型

缓冲具有索引（index）和标签（tag）（参考图13-2）。

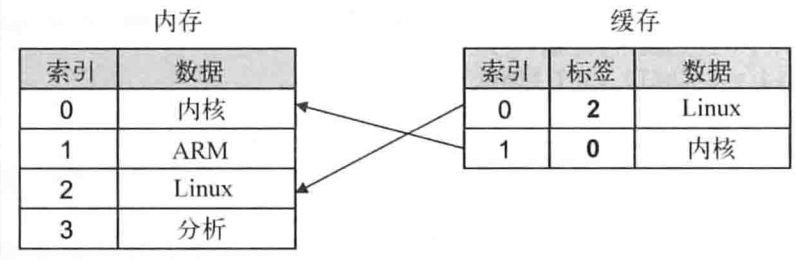


图13-2　主内存与缓存的索引及标签关系

根据索引和标签表现为物理地址或虚拟地址，缓存类型共分为4种。在缓存中，索引是处理器浏览的地址，该地址又分为虚拟地址和物理地址，即分为Virtually Indexed和Physically Indexed。

若索引为物理地址，则需要MMU介入，因为处理器中需要通过虚拟地址进行访问。缓存中有标签，该标签具有指向主内存位置的值。根据该值是虚拟地址或物理地址，分为Virtually Tagged和Physically Tagged。

根据缓存的索引和标签使用的是物理地址或虚拟地址，可表现出不同地址。因此，缓存共有4种类型。

(1) PIPT（Physically Indexed，Physically Tagged）

(2) VIVT（Virtually Indexed，Virtually Tagged）

(3) VIPT（Virtually Indexed，Physically Tagged）

(4) PIVT（Physically Indexed，Virtually Tagged）

现在我们逐一进行分析。

1. PIPT（Physically Indexed，Physically Tagged）

该结构对索引和标签均使用物理地址，是最为简单的方法。由于物理地址是唯一的，因此没有地址假名（Address Aliasing）问题，但需要变更物理地址（参考图13-3）。此外需要介入TLB（Translation Lookaside Buffer，页表缓冲），发生TLB失败（miss）时会降低速度。

即使不发生TLB失败，可TLB lookup后需要连续执行缓存lookup，也会降低整体速度。

2. VIVT（Virtually Indexed，Virtually Tagged）

该结构对索引和标签均使用虚拟地址。由于使用虚拟地址，故不需要为查找物理地址而介入MMU。从图13-4中可以看到，处理器使用虚拟地址直接访问缓存。但是这有可能发生“不同虚拟地址指向相同物理地址”的假名问题，该问题发生于不同虚拟地址向相同物理地址缓冲时，这会引发一致性问题（Coherency Problem）。

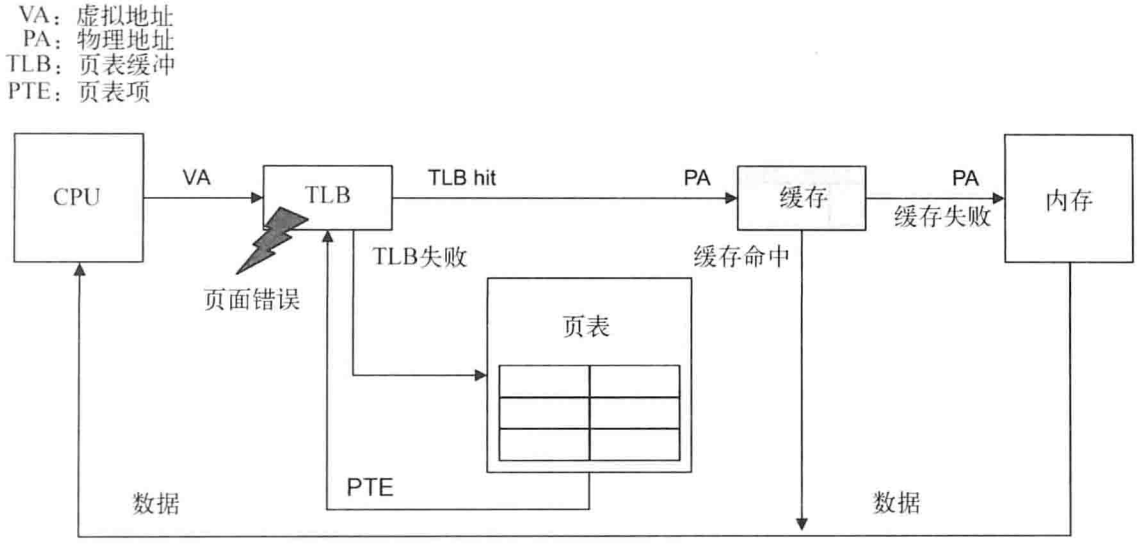


图13-3 PIPT缓存类型

例如，不同用户应用程序对相同文件执行mmap时，用户应用程序的虚拟地址虽然不同，但实际具有相同的物理地址。另一个问题在于，这有可能更换V->P映射（virtual to physical）。此时要注意TLB项的变更，要在项变更前清除相应缓存线，因为虚拟地址已经失效。

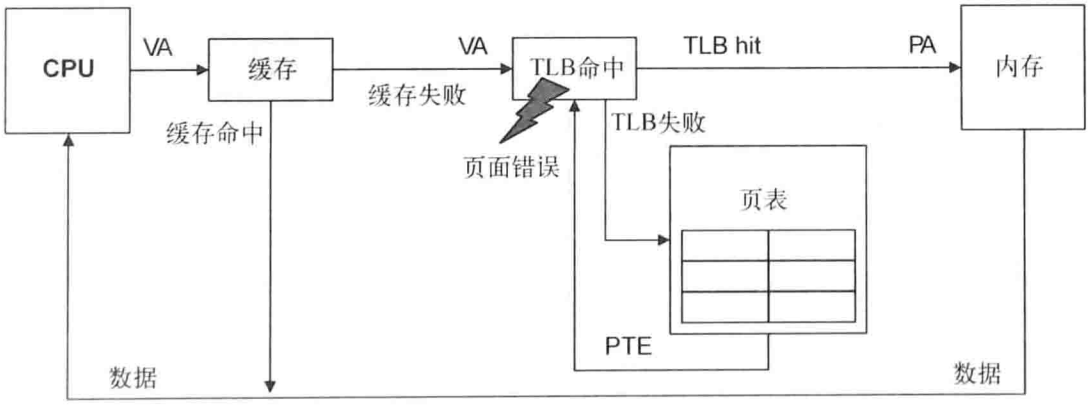


图13-4 VIVT缓存类型

3. VIPT（Virtually Indexed，Physically Tagged）

该缓存类型对索引使用虚拟地址，对标签使用物理地址。与PIPT缓存类型相比，它的优点在于可缩短查找物理地址耗费的时间——潜伏期（latency），还可通过TLB并排查找缓存线。但由于标签使用物理地址，因此，找到物理地址前无法比较标签。

处理器浏览的索引是虚拟地址，因此，必须将其变更为物理地址。与VIVT相比，VIPT的优点就在于标签是物理地址，所以缓存可检测出假名。VIPT中，索引位不再表现相同地址，因此更需要标签位（参考图13-5）。

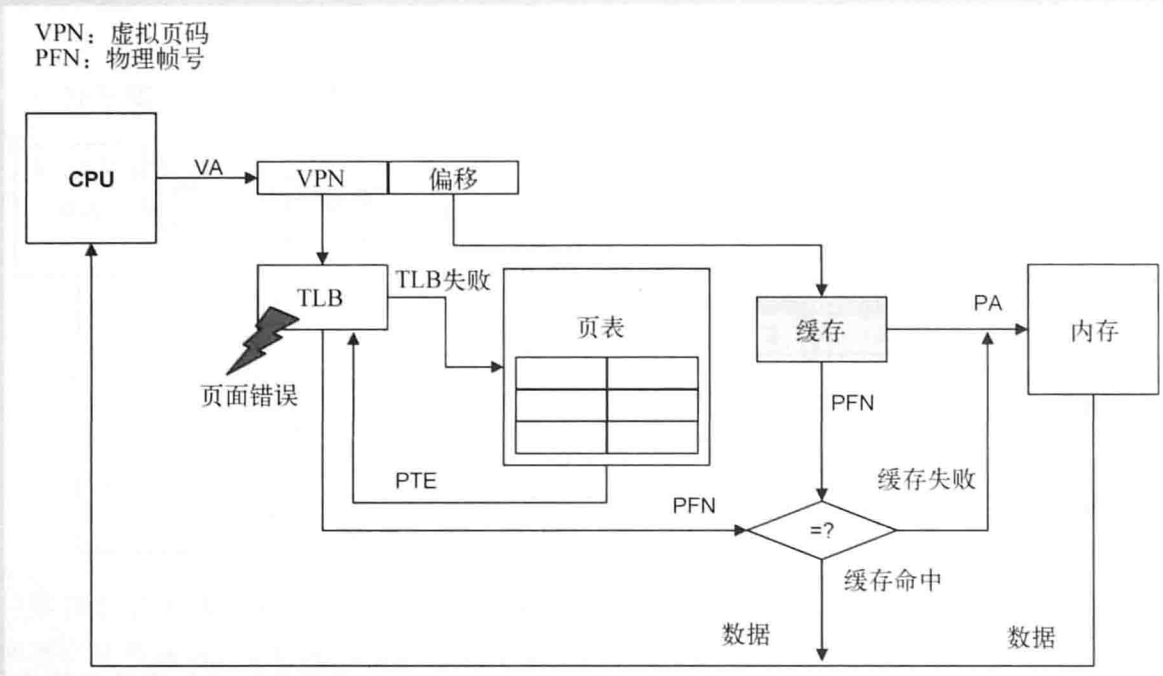


图13-5　VIPT缓存类型

4. PIVT（Physically Indexed，Virtually Tagged）

该缓存类型理论上存在，但实际并不使用。

13.6　调用处理器初始化函数——cpu_proc_init()

cpu_proc_init()函数的定义如下所示。

:: arch/arm/include/asm/cpu-multi32.h

```
#define cpu_proc_init()    processor._proc_init()
```

该函数只调用processor结构体的函数指针_proc_init()函数。processor结构体是从代码13-1的proc_info_list结构体获得的，其定义如下所示。

:: arch/arm/include/asm/cpu-multi32.h

```
extern struct processor {
    void (*_data_abort)(unsigned long pc);      /*data abort获得数据中止地址或标签*/
    unsigned long (*_prefetch_abort)(unsigned long lr);
                                                /*prefetch fault取得预取指令中止地址*/
    void (*_proc_init)(void);                   /*按处理器设置执行特殊化的内容*/
    void (*_proc_fin)(void);                    /*按处理器解除执行特殊化的设置*/
    void (*reset)(unsigned long addr) __attribute__((noreturn));
                                                /*重设时处理的操作*/
```

```
    int (*_do_idle)(void);                          /*将处理器设置为Idle状态*/
    void (*dcache_clean_area)(void *addr, int size);
                                                    /* D-Cache从数据缓存清理虚拟地址区域*/
    void (*switch_mm)(unsigned long pgd_phys, struct mm_struct *mm);
                                                    /*设置页表*/
    void (*set_pte_ext)(pte_t *ptep, pte_t pte, unsigned int ext);
                                                    /*设置扩展PTE */
} processor;
```

从代码的函数指针名可以看出，它定义的是该处理器要对特定事件执行的函数。此处成员变量_proc_init是由cpu_proc_init()函数调用的函数指针。

为各处理器相应proc_info_list结构体定义与processor结构体对应的函数，ARM中，与arch/arm/mm/proc-v*S中定义的v*_processor_functions相对应。[①]这一段我们以ARM v6为例进行讲解。代码13-4是（与processor结构体对应的）ARM v6的v6_processor_functions的定义。

:: 代码13-4 arch/arm/mm/proc-v6.S的v6_processor_functions标签

```
.type   v6_processor_functions, #object
ENTRY(v6_processor_functions)
    .word   v6_early_abort
    .word   pabort_noifar
    .word   cpu_v6_proc_init                                    ❶
    .word   cpu_v6_proc_fin
    .word   cpu_v6_reset
    .word   cpu_v6_do_idle
    .word   cpu_v6_dcache_clean_area
    .word   cpu_v6_switch_mm
    .word   cpu_v6_set_pte_ext
    .size   v6_processor_functions, . - v6_processor_functions

    ...

ENTRY(cpu_v6_proc_init)                                         ❷
    mov pc, lr
```

代码段❶与processor结构体的_proc_init函数指针对应。实际调用_proc_init()后，ARM v6中将PC移动到代码段❷定义的cpu_v6_proc_init中执行。

代码段❷中，调用_proc_init()函数时不执行任何操作就结束。ARM v6、v7中虽然不执行任何操作，但如果有需要在该处理器中进行特别处理的内容，那此处必须执行特定任务。

我们通过本章了解了setup_processor()函数。setup_processor()函数从C代码调用__lookup_processor_type的汇编代码（7.2节）以获得相应处理器的结构体信息。

① ARM v6中，arch/arm/mm/proc-v6.S的v6_processor_functions与processor结构体对应；ARM v7中则为arch/arm/mm/proc-v7.S的v7_processor_functions。v6和v7均具有相同结构。

setup_processor()函数的执行步骤是：从协处理器获得CPU ID，在proc_info_list结构体获得执行当前内核的处理器信息，并设置到全局变量，注册对该处理器进行初始化（proc_init）等特定时段需要调用的处理器函数。

还可看到各函数按结构另行定义在其他文件中。而且，setup_processor()函数中设置了当前处理器的缓存类型等。

第 14 章

准备内存分页——paging_init()

14.1 查看 paging_init()的整体结构

本章主要分析setup_arch()函数内部调用的paging_init()函数。如图14-1所示，paging_init()函数在setup_arch()函数调用关系的中间部分调用。

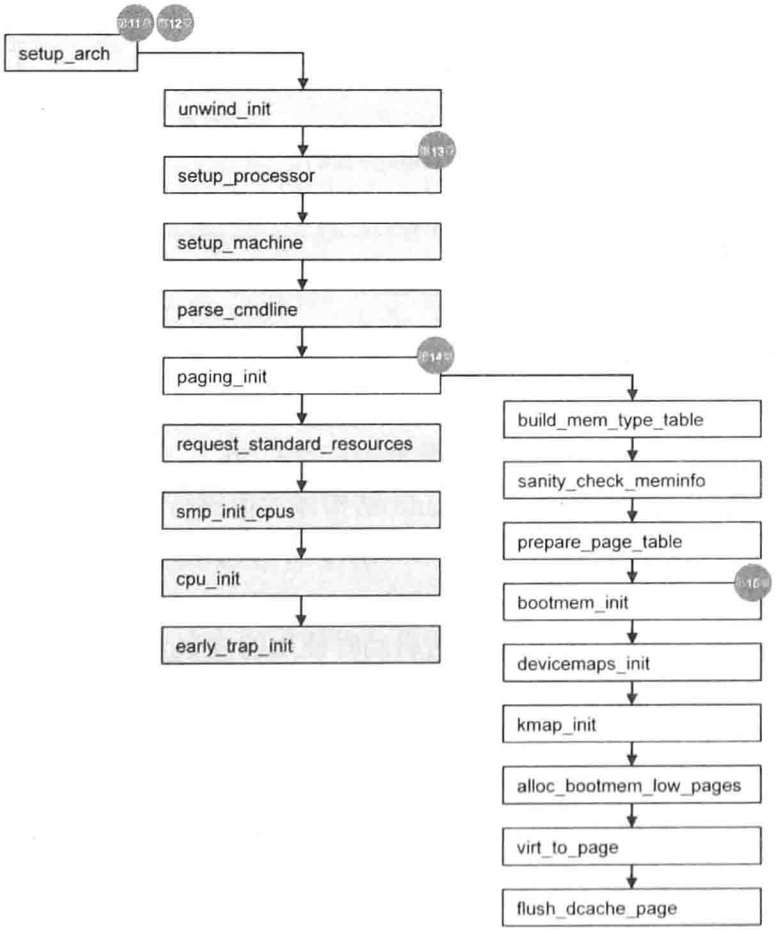


图14-1 paging_init()函数的调用关系

除paging_init()函数外，其他函数都已在第12章和第13章讲过。paging_init()函数在内部又调用许多函数，其中，bootmem_init()函数激活启动时使用的内存分配器，我们将在第15章分析它。

通过代码14-1了解paging_init()函数的整体结构。该函数执行分页相关的准备工作，其作用是设置页表、激活启动时的内存分配器并最终生成零页。

:: 代码14-1　arch/arm/mm/mmu.c的paging _init()

```
void __init paging_init(struct machine_desc *mdesc)
{
    void *zero_page;

    build_mem_type_table();                                          ❶

    sanity_check_meminfo();                                          ❷

    prepare_page_table();                                            ❸

    bootmem_init();                                                  ❹

    devicemaps_init(mdesc);                                          ❺

    kmap_init();                                                     ❻

    top_pmd = pmd_off_k(0xffff0000);

    zero_page = alloc_bootmem_low_pages(PAGE_SIZE);                  ❼

    empty_zero_page = virt_to_page(zero_page);                       ❽

    flush_dcache_page(empty_zero_page);                              ❾
}
```

在代码段❶中构建mem_types列表。该列表由内存类型项构成，各项检查结构，根据相应结构要求将不需要的位清空为0，将需要的位设置为1（14.2节）。

在代码段❷中，对保存内存信息的meminfo结构体的值进行有效性检查（14.3节）。

代码段❸是准备页表阶段，将对应于内核映像下方及内核空间的页目录项的pmd段均清空为0（14.4节）。

代码段❹的bootmem_init()函数用于激活启动时使用的内存分配器，直至激活slab分配器。该函数在内部调用多种函数，我们将在第15章专门分析。

在代码段❺中对向量表进行映射，还执行使用设备时必需的操作，如对页表中指向VMALLOC_END上方的所有项进行初始化等（14.5节）。

代码段❻只在已设置CONFIG_HIGHMEM时执行，ARM中对高端内存的支持设置为Experimental。此处定义的高端内存支持是为了在系统具有多于1GB内存时提供支持，kmap_init()函数将在14.6节进行讲解。

在代码段❼生成初始化为0的零页（14.7节）。

在代码段❽中，从生成的零页的虚拟地址获得page结构体（14.7.3节）。

在代码段❾中清除零页，以确保数据缓存一致性（14.8节）。

需要确认的一点是，paging_init()函数中设置的页表由内核使用，无法在用户空间访问。

14.2 设置内存类型表——build_mem_type_table()

在build_mem_type_table()函数中，根据ARM版本及内存类型，对struct mem_type结构体类型的全局数组mem_types进行初始化。mem_types结构体数组是具有（为了按内存类型使用虚拟内存的）L1、L2列表和缓存策略、域属性信息的内核数据结构。

什么是内存类型？Linux中对内核使用内存的所有用例进行分类，从而定义了内存类型。内核定义的内存类型出现在arch/arm/mm/mmu.c中，表14-1进行了简单介绍。如表14-1所示，根据内存使用目的不同，内存类型对是否使用缓存、是否使用写缓冲、是否共享、域等信息定义了不同设置，通过mem_types全局结构体数组进行管理。

表14-1 内存类型

内存类型	使用目的
MT_DEVICE	ARMv6的共享设备
MT_DEVICE_NONSHARED	ARMv6的非共享设备
MT_MEMORY_NONCACHED	未缓存的内存区域
MT_DEVICE_CACHED	使用写缓冲及缓存的设备
MT_DEVICE_WC	未使用写缓冲及缓存的设备

接下来通过代码14-2分析构成内存类型列表的build_mem_type_table()函数。

:: 代码14-2 arch/arm/mm/mmu.c的build_mem_type_table()

```
static void __initbuild_mem_type_table(void)
{
    structcachepolicy *cp;
    unsigned int cr = get_cr();                                    ❶
    unsigned int user_pgprot, kern_pgprot, vecs_pgprot;
    int cpu_arch = cpu_architecture();                             ❷
    int i;

    if (cpu_arch< CPU_ARCH_ARMv6) {                                ❸
        ...
        cachepolicy = CPOLICY_BUFFERED;
        ...
    }
...
#ifdef CONFIG_SMP
    cachepolicy = CPOLICY_WRITEALLOC;
#endif

    if (cpu_arch< CPU_ARCH_ARMv5)                                  ❹
```

```
        for (i = 0; i < ARRAY_SIZE(mem_types); i++)
    mem_types[i].prot_sect&= ~PMD_SECT_TEX(7);

    ...
    printk("Memory policy: ECC %sabled, Data cache %s\n",
    ecc_mask ? "en" : "dis", cp->policy);

    ...
}
```

在代码段❶中通过get_cr()函数读取当前控制寄存器的值。通过mrc命令读取CP15中保存的控制寄存器的值，并保存到变量cr。

在代码段❷中调用cpu_architecture()函数，求出CP15的c0中保存的CPU ID值。利用求出的CPU ID值查出ARM版本，返回到cpu_arch变量。cpu_architecture()函数的详细内容请参考13.4节。

代码段❸中出现cachepolicy全局变量，根据ARM版本设置缓存策略。对缓存策略的说明请参考下列“补充知识点”。该变量在arch/arm/mm/mmu.c中声明，默认值为CPOLICY_WRITEBACK。

从代码段❹开始，根据ARM版本正式填充mem_types列表内容。代码段❹看起来十分冗杂，但主要根据内存类型对段（section或LI或PMD）、列表项（L1或PTE）以及域（domain）进行保护设置。



补充知识点　缓存策略的种类

内核使用的缓存策略有UNCACHED、BUFFERED、WRITE THROUGH、WRITEBACK、WRITEALLOC（参考表14-2）。

表14-2　缓存策略

缓存策略	意　义
UNCACHED	不使用缓存
BUFFERED	向缓存和内存之间使用写缓冲
WRITE THROUGH	缓存中发生写事件时，直接反映到内存和内存下层。主要用于USB等经常插拔的存储设备
WRITE BACK	优先适用于缓存，在缓存替换（cache replacement）策略作用下，当数据需要从缓存区域脱离时反映到内存
WRITE ALLOC	发生缓存失败时，缓存控制器分配缓存线的方法有两种 1. read-allocate方式：从内存读取数据时分配缓存线 2. write-allocate方式：向内存写入数据时分配缓存线

14.3　检验内存信息——sanity_check_meminfo()

我们在12.3节的setup_arch()中，通过不同机器定义的fixup函数完成了为meminfo结构体填充

BANK数和节点号、起始地址、大小等内存信息的任务。本节主要执行合理性检查，确认进入meminfo结构体的值是否正确。

∷代码14-3 arch/arm/mm/mmu.c的sanity_check_meminfo()

```
static void __init sanity_check_meminfo(void)
{
    ...

    for (i = 0, j = 0; i < meminfo.nr_banks; i++) {                    ❶
        struct membank *bank = &meminfo.bank[j];
        *bank = meminfo.bank[i];

#ifdef CONFIG_HIGHMEM
    if (__va(bank->start) > VMALLOC_MIN ||
        __va(bank->start) < (void *)PAGE_OFFSET)                       ❷
        highmem = 1;

    bank->highmem = highmem;

    if (__va(bank->start) < VMALLOC_MIN &&
        bank->size > VMALLOC_MIN - __va(bank->start)) {                ❸
        if (meminfo.nr_banks >= NR_BANKS) {
            ...
        } else {
            memmove(bank + 1, bank,
                (meminfo.nr_banks - i) * sizeof(*bank));               ❹
            meminfo.nr_banks++;
        ...
        }
        bank->size = VMALLOC_MIN - __va(bank->start);                  ❺
    }
#else
    if (__va(bank->start) >= VMALLOC_MIN ||
        __va(bank->start) < (void *)PAGE_OFFSET) {                     ❻
            ...
        continue;
    }

    if (__va(bank->start + bank->size) > VMALLOC_MIN ||
        __va(bank->start + bank->size) < __va(bank->start)) {          ❼
        unsigned long newsize = VMALLOC_MIN - __va(bank->start);       ❽
            ...
        bank->size = newsize;
    }
#endif
       j++;
    }
}
```

在代码段❶中对所有内存条执行循环，同时逐一获得内存条信息。使用高端内存（high

memory）时执行代码段❷~❺。和下列步骤的整体流程基本相同。

在代码段❷中检查内存条区域是否对应于高端内存区域，并仅向membank结构体显示“对应于高端内存”。在代码段❸中检查内存条区域和VMALLOC区域有无重叠，若有，则在代码段❹中将内存信息移动到其他BANK（现有内存条+1）并记录。在代码段❺中修正现有BANK信息，使其符合剩余不重叠的部分。

不使用高端内存的情况对应于代码段❻和代码段❼。在代码段❻中检查内存条区域与VMALLOC是否全部重叠，在代码段❼中检查是否部分重叠。其操作大体上与使用高端内存时相似，但如果有与VMALLOC区域重叠的内存条区域，就会从BANK信息中排除重叠部分（代码段❽）。因为，不使用高端内存的话，脱离VMALLOC的区域无实际意义。



补充知识点　高端内存区域

由于硬件限制，Linux将物理内存空间分为3个区域进行管理。为了使用DMA，基于ISA的旧设备只能使用内存开始部分的16 MB。而内核区域和用户区域分为1 GB和3 GB，此时只能直接访问1 GB的空间，不能直接访问其以上空间。因此，我们将128 MB的空间放在896 MB~1024 MB之间，并映射1 GB上方的区域。896 MB~4 GB对应的区域称为高端内存。



图14-2　内存区域（zone）

14.4　准备页表——prepare_page_table()

本节通过prepare_page_table()函数讲解准备页表的过程。

14.4.1　prepare_page_table()

prepare_page_table()函数大致分为2个阶段执行任务。

(1) 将对应于内核映像下方的页目录各项的pmd（与pgd相同）段全部清空为0。

(2) 将对应于内核空间的页目录各项的pmd段全部清空为0。

不同阶段需要考虑不同事项。下面通过代码14-4了解准备页表的过程。

:: 代码14-4 arch/arm/mm/mmu.c的prepare_page_table()

```
static inline void prepare_page_table(void)
{
    unsigned long addr;

    for (addr = 0; addr < MODULES_VADDR; addr += PGDIR_SIZE)                    ❶
        pmd_clear(pmd_off_k(addr));

#ifdef CONFIG_XIP_KERNEL                                                        ❷
    /*XIP时省略初始化过程*/
    addr = ((unsigned long)_etext + PGDIR_SIZE - 1) & PGDIR_MASK;
#endif
   for ( ; addr < PAGE_OFFSET; addr += PGDIR_SIZE)                              ❸
        pmd_clear(pmd_off_k(addr));

    for (addr = __phys_to_virt(bank_phys_end(&meminfo.bank[0]));                ❹
        addr < VMALLOC_END; addr += PGDIR_SIZE)
        pmd_clear(pmd_off_k(addr));
}
```

在代码段❶中，从虚拟地址0x00000000到MODULES_VADDR，以PGDIR_SIZE（2 MB）为单位增加地址值，并调用pmd_clear()函数以初始化页目录项的pmd段。pmd_clear()函数和pmd_off_k()函数将在14.4.3节及14.4.4节详细讲解。

在代码段❷中，由于内核是在模块区域以XIP模式运行的，因此省略页目录项的初始化过程。省略的过程中以2 MB为单位进行初始化，因此以表示内核代码结束的_etext为基准，利用PGDIR_MASK进行调整，确保省略区域为2 MB。①

在代码段❸中对PAGE_OFFSET之前的页目录项执行初始化。PAGE_OFFSET表示内核空间的起始地址，在32位系统中，地址空间最多为4 GB。内核编译时通过Kconfig分为内核空间和用户空间。但是，内核空间和用户空间的比例一般为3：1，内核空间的起始地址是0xC0000000。

:: arch/arm/Kconfig

```
config PAGE_OFFSET
    hex
    default 0x40000000 if VMSPLIT_1G
    default 0x80000000 if VMSPLIT_2G
    default 0xC0000000
```

这样就可以对0xC0000000之前的虚拟地址空间的页目录项进行初始化。

最后，在代码段❹中，对从meminfo.bank[0]的最终地址到VMALLOC_END为止的页目录项进行初始化。meminfo通过机器描述符中定义的fixup()函数填充，并具有当前系统具有的内存信息（12.2节）。VMALLOC_END是内核中可通过内核分配函数vmalloc()函数使用的最大地址值。

① PGDIR_MASK在arch/arm/include/asm/pgtable.h文件中定义如下。

```
#define PGDIR_MASK    (~(PGDIR_SIZE-1))
```

14.4.2　Linux的分页结构

32位Linux具有3个阶段的分页结构，但ARM Linux中的实际实现方式与pgd（page global directory，全局页目录）和pmd（page middle directory，中间页目录）相同（参考图14-3）。

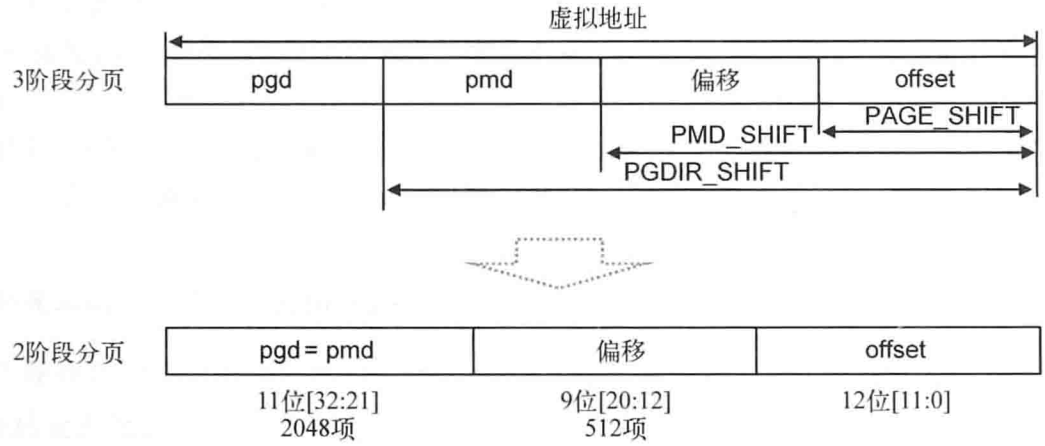


图14-3　映射到ARM内核中使用的3阶段分页的2阶段分页结构

也就是说，使用2阶段分页初始化页目录各项的pmd段，就意味着初始化pgd段。MOUDLE_VADDR和PGDIR_SIZE定义如下。

:: arch/arm/include/asm/pgtable.h

```
#define PGDIR_SHIFT       21
#define PGDIR_SIZE        (1UL << PGDIR_SHIFT)
```

:: arch/arm/include/asm/memory.h

```
#define MODULES_VADDR     (PAGE_OFFSET - 16*1024*1024)
```

PAGE_OFFSET是表示内核空间起始地址的符号，一般具有0xC0000000的值。因此，MODULES_VADDR具有（位于PGDIR_SIZE以下16 MB的）0xBF000000的值。

14.4.3　求出页目录项

再次观察代码14-4的代码段❶中调用的pmd_clear()函数，以了解求出页目录项的方法。

```
pmd_clear(pmd_off_k(addr));
```

为了初始化页目录项，需要获得要初始化的项地址。从上述代码可以看出，pmd_clear()函数将pmd_off_k()函数用作输入值。正是通过pmd_off_k()函数获得项地址。pmd_off_k()函数按如下所示，将pgd_offset_k()函数的结果值用作参数以调用pmd_off()函数。

:: arch/arm/mm/mm.h

```
static inline pmd_t *pmd_off_k(unsigned long virt)
{
    return pmd_off(pgd_offset_k(virt), virt);
}

static inline pmd_t *pmd_off(pgd_t *pgd, unsigned long virt)
{
    return pmd_offset(pgd, virt);
}
```

pgd_offset_k()函数的定义如下所示。

:: arch/arm/include/asm/pgtable.h

```
#define pgd_offset_k(addr)        pgd_offset(&init_mm, addr)
#define pgd_offset(mm, addr)      ((mm)->pgd+pgd_index(addr))
#define pgd_index(addr)           ((addr) >> PGDIR_SHIFT)

#define pmd_offset(dir, addr)     ((pmd_t *)(dir))
```

pgd_offset_k()调用pgd_offset()函数，传递的参数将成为init_mm的地址。pgd_index()将输入地址（addr）以PGDIR_SHIFT的大小（21位）向右移动，因此会求出对应于输入地址（addr）的页目录项编号，并通过pgd_offset()获得用于管理ADDR所属内存块的页目录的相应项地址。

通过pgd_offset_k()求出的相应地址页目录内的项地址会传递到pmd_offset()，而pmd_offset()只将输入值的类型转换为pmd_t。最终，在页目录项中共同处理1阶段目录和2阶段目录。

init_mm根据INIT_MM进行初始化，其定义如下所示。

:: arch/arm/kernel/init_task.c

```
struct mm_struct init_mm = INIT_MM(init_mm);
```

:: include/linux/init_task.h

```
#define INIT_MM(name)                                                    \
{    \
    .mm_rb        = RB_ROOT,                                             \
    .pgd          = swapper_pg_dir,                                      \
    .mm_users     = ATOMIC_INIT(2),                                      \
    .mm_count     = ATOMIC_INIT(1),                                      \
    .mmap_sem     = __RWSEM_INITIALIZER(name.mmap_sem),                  \
    .page_table_lock =  __SPIN_LOCK_UNLOCKED(name.page_table_lock),      \
    .mmlist       = LIST_HEAD_INIT(name.mmlist),                         \
    .cpu_vm_mask     = CPU_MASK_ALL,                                     \
}
```

init_mm是mm_struct结构体变量，因此，由INIT_MM填充成员变量。

:: include/linux/mm_types.h

```
struct mm_struct {
    ...

    struct rb_root mm_rb;
    pgd_t * pgd;                        /*页目录位置*/
    atomic_t mm_users;                  /*用户空间的用户数量*/
    atomic_t mm_count;                  /*浏览mm_struct的数量*/
    spinlock_t page_table_lock;         /*用于保护页表的自旋锁*/
    struct list_head mmlist;
    cpumask_t cpu_vm_mask;
    ...
};
```

表14-3表示通过INIT_MM()宏进行初始化的mm_struct各成员变量。

表14-3　通过INIT_MM()进行初始化的mm_struct成员变量

成　员	说　明
mm_rb	虚拟内存各区域用vm_area_struct进行说明，而进程区域用2种方式排列：单向链表（single linked list）方式和红黑树（red-black tree）方式，mm_rb指向红黑树的root节点
pgd	全局页目录，指向页目录
mm_users	表示使用该内存管理结构体（mm_struct）的处理器数量
mm_count	用于mm_struct的浏览账户，所有mm_users=mm_count+1
mmap_sem	读写信号量变量
page_table_lock	用于保护页表或计数器值的自旋锁变量
mmlist	系统内所有mm_struct连接到双向链表，其第一个节点通过INIT_MM()进行初始化
cpu_vm_mask	以系统内CPU数量（NR_CPUS）为cpumask_t类型的cpu_vm_mask加掩码

14.4.4　pmd_clear()

下面通过代码14-5了解pmd_clear()函数。

:: 代码14-5　arch/arm/include/asm/pgtable.h的pmd_clear定义

```
#define pmd_clear(pmdp)                 \
    do {                                \
        pmdp[0] = __pmd(0);             \                    ❶
        pmdp[1] = __pmd(0);             \
        clean_pmd_entry(pmdp);          \                    ❷
    } while (0)
```

在代码段❶中，传递给pmd_clear()的参数pmdp为2 MB单位（PGDIR_SIZE），页目录各项为1MB单位。因此，将pmdp分为2个，并初始化为0。

最终，pmdp所指地址的0号和1号均初始化为0。实际页目录各项对应1 MB，但ARM Linux内核中以2 MB为单位管理各项。

对__pmd的定义只将接收的参数值转换为pmd_t类型，而pmd_t是指向虚拟地址的unsigned long，其定义如下所示。

:: arch/arm/include/asm/page.h

```
typedef struct { unsigned long pmd; } pmd_t;
#define __pmd(x)        ((pmd_t) { (x) } )
```

代码段❷中变更了页目录值，因此调用clean_pmd_entry()函数向内存反映。clean_pmd_entry()函数的定义如代码14-6所示，在代码段❶中清空（clean）对应于虚拟地址pmd的MMU的数据缓存线。

:: 代码14-6 arch/arm/include/asm/tlbflush.h的clean_pmd_entry()

```
static inline void clean_pmd_entry(pmd_t *pmd)
{
    const unsigned int __tlb_flag = __cpu_tlb_flags;

    if (tlb_flag(TLB_DCLEAN))
        asm("mcr    p15, 0, %0, c7, c10, 1  @ flush_pmd"                    ❶
            : : "r" (pmd) : "cc");

    if (tlb_flag(TLB_L2CLEAN_FR))
        asm("mcr    p15, 1, %0, c15, c9, 1  @ L2 flush_pmd"
            : : "r" (pmd) : "cc");
}
```

总而言之，prepare_page_table()函数的作用是将页目录各项的pmd段初始化为0。图14-4通过prepare_page_table()函数展示了初始化页目录时需要考虑的内存区域对应关系。

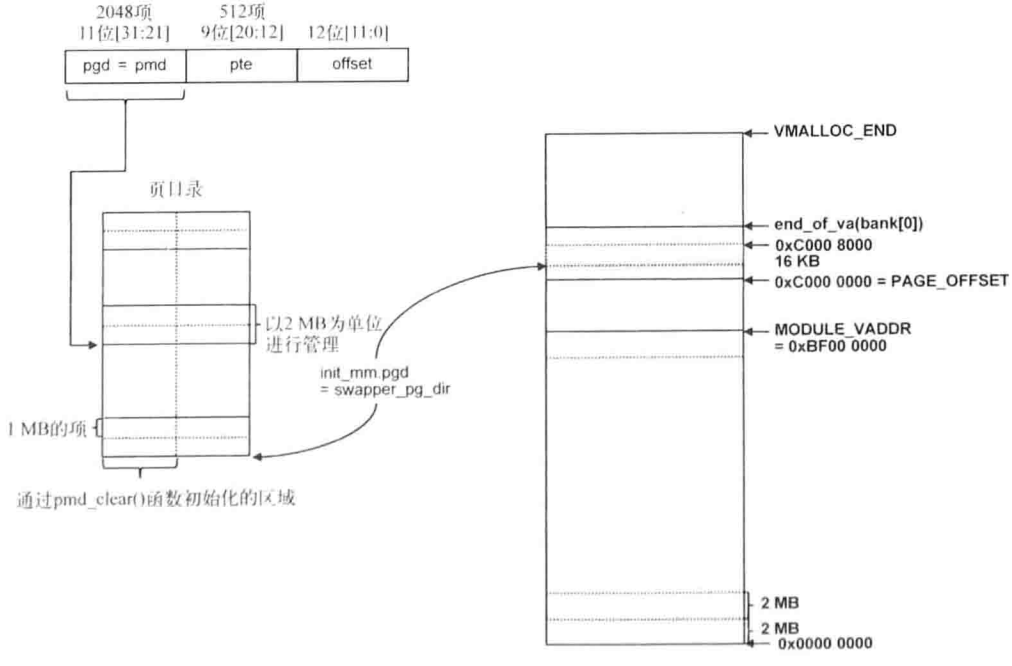


图14-4 通过prepare_page_table执行的内存和页目录的关系

pmd在页目录各项中表示2阶段页表地址，与pgd的值是一样的。虽然其目的是反映3阶段分页结构，但实际代码中使用的是2阶段的分页结构。

14.5　设备区域映射准备——devicemaps_init()

在该函数中分配并映射用于向量表的内存后，初始化特定区域的pgd项。然后映射IO设备，清除（flush）TLB和缓存。详细内容请看下列代码。

:: 代码14-7　arch/arm/mm/mmu.c的devicemaps_init()

```
static void __init devicemaps_init(struct machine_desc *mdesc)
{
    struct map_desc map;
    unsigned long addr;
    void *vectors;

    vectors = alloc_bootmem_low_pages(PAGE_SIZE);                    ❶

    for (addr = VMALLOC_END; addr; addr += PGDIR_SIZE)               ❷
        pmd_clear(pmd_off_k(addr));
    ...

    map.pfn = __phys_to_pfn(virt_to_phys(vectors));                  ❸
    map.virtual = 0xffff0000;
    map.length = PAGE_SIZE;
    map.type = MT_HIGH_VECTORS;
    create_mapping(&map);

    if (!vectors_high()) {                                           ❹
        map.virtual = 0;
        map.type = MT_LOW_VECTORS;
        create_mapping(&map);
    }

    if (mdesc->map_io)                                               ❺
        mdesc->map_io();

    local_flush_tlb_all();                                           ❻
    flush_cache_all();
```

在代码段❶中，以PAGE_SIZE的量分配用于向量表的内存后，在代码段❷中将指向VMALLOC_END以上地址的pgd项均进行初始化。这些pgd是内核启动初始阶段编写的，且只对内核代码区域进行了映射。

如前所述，ARM中使用2阶段页表，Linux中使用3阶段页表。[①]ARM中使用的页表代码中，为了兼容ARM的2阶段页表，PMD被设置为可直接使用PGD。

① 为了支持64位系统，现在也添加了4阶段。

在代码段❸中使用map_desc结构体和create_mapping()函数执行映射，以通过0xFFFF0000也能访问向量表的虚拟地址。create_mapping()将在代码14-8中详细讲解。

pfn是页帧号（page frame number），virtual是映射的虚拟地址，length是映射长度，type决定要链接的类型。代码段❹中，向量表不是上级向量时，映射为下级向量。

在代码段❺中，如果记述当前机器信息的mdesc的成员变量map_io有效，就调用map_io()。该函数指针依据不同机器均有不同设置，主要用于将当前机器的IO设备映射到内核地址空间。映射后，通过映射地址访问设备。S3C6410时，调用arch/arm/mach-s3c6410/mach-smdk6410.c的smdk6410_map_io()。

执行映射并变更PGD项后，为了正确反映变更事项，在代码段❻中清除TLB和缓存。下面通过代码14-8讲解最重要的create_mapping()函数。

:: 代码14-8 arch/arm/mm/mmu.c的create_mapping()

```
void __init create_mapping(struct map_desc *md)
{
    unsigned long phys, addr, length, end;
    const struct mem_type *type;
    pgd_t *pgd;

    ...

    if (md->pfn >= 0x100000) {
        create_36bit_mapping(md, type);
        return;
    }

    addr = md->virtual & PAGE_MASK;                                    ❶
    phys = (unsigned long)__pfn_to_phys(md->pfn);

    length = PAGE_ALIGN(md->length + (md->virtual & ~PAGE_MASK));

    if (type->prot_l1 == 0 && ((addr | phys | length) & ~SECTION_MASK)) {
        printk(KERN_WARNING "BUG: map for 0x%08lx at 0x%08lx can not "
               "be mapped using pages, ignoring.\n",
               __pfn_to_phys(md->pfn), addr);
        return;
    }

    pgd = pgd_offset_k(addr);                                          ❷
    end = addr + length;

    do {
        unsigned long next = pgd_addr_end(addr, end);

        alloc_init_section(pgd, addr, next, phys, type);

        phys += next - addr;
        addr = next;
    } while (pgd++, addr != end);
}
```

在代码段❶之前试图执行区域检查和分配4GB以上区域时，可以看到利用超级节区进行的36位映射，但并不重要，故省略。

在代码段❷中，利用接收的参数将虚拟地址和物理地址按页大小进行排列，如果prot_l1中没有任何值，就检查地址是否以节区地址进行排列。prot_l1不使用超级节区或节区，在要使用2级页表时进行设置。

在代码段❷中，利用addr和length求出pgd偏移量的起始与末尾。然后执行do {} while语句，按大小开始映射。下面观察实际执行映射的alloc_init_section()函数。

:: 代码14-9　arch/arm/mm/mmu.c的alloc_init_section()

```
static void __init alloc_init_section(pgd_t *pgd, unsigned long addr,
                                       unsigned long end, unsigned long phys,
                                       const struct mem_type *type)
{
    pmd_t *pmd = pmd_offset(pgd, addr);                                    ❶

    if (((addr | end | phys) & ~SECTION_MASK) == 0) {                      ❷
    pmd_t *p = pmd;

        if (addr & SECTION_SIZE)
            pmd++;

        do {
            *pmd = __pmd(phys | type->prot_sect);
            phys += SECTION_SIZE;
        } while (pmd++, addr += SECTION_SIZE, addr != end);
        flush_pmd_entry(p);
    } else {                                                               ❸
        alloc_init_pte(pmd, addr, end, __phys_to_pfn(phys), type);
    }
}
```

在代码段❶中求出对应于addr位置的pmd指针。在代码段❷中检查addr、end、phys是否以节区单位（1 MB）进行排列，如果是，则向pmd记录其内容（参考图14-5）。

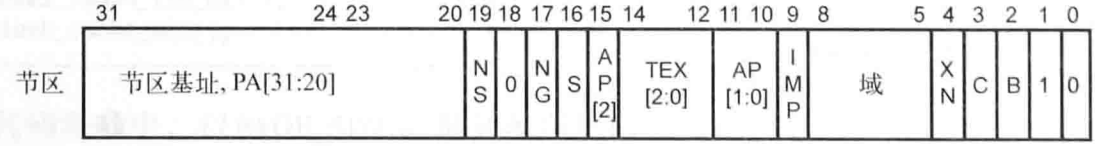


图14-5　1级页表项

```
*pmd=__pmd(phys | type->prot_sect);
```

上述代码将phys | type->prot_sect值直接写入pmd位置。phys是（以节区单位排列的）31位中前20位的物理内存值，由prot_sect填充图14-5所示的属性。分配节区时，默认prot_sect中必须有PMD_TYPE_SECT值，且定义为(2 << 0)。

换言之，在（区分利用2级列表还是节区的）底端2位记录为节区，之后在MMU中变换地址时使用。按大小填充pmd之后，将当前修正的内容反映到内存。下面观察未按节区单位排列的情况。

未按节区单位排列时设置2级列表PTE，它是代码段❶的alloc_init_pte()。下面观察create_mapping()的末尾函数。

:: 代码14-10 arch/arm/mm/mmu.c的alloc_init_pte()

```
static void __init alloc_init_pte(pmd_t *pmd, unsigned long addr,
                                   unsigned long end, unsigned long pfn,
                                   const struct mem_type *type)
{
    pte_t *pte;

    if (pmd_none(*pmd)) {                                                    ❶
        pte = alloc_bootmem_low_pages(2 * PTRS_PER_PTE * sizeof(pte_t));
        __pmd_populate(pmd, __pa(pte) | type->prot_l1);
    }

    pte = pte_offset_kernel(pmd, addr);                                      ❷
    do {                                                                     ❸
        set_pte_ext(pte, pfn_pte(pfn, __pgprot(type->prot_pte)), 0);
        pfn++;
    } while (pte++, addr += PAGE_SIZE, addr != end);
}
```

在代码段❶中检查相应pmd是否进行初始化，若未进行，则为pte分配新空间，设置pmd项（entry）。1个pmd具有PTRS_PER_PTE×2的pte。

下面简单介绍一下pte的个数。ARM的硬件页表结构为1级4096项2级256项，在Linux中管理页表时，为了补充硬件页表中的缺失信息，将2个硬件1级项用作1个pmd，用1个pmd管理512个2级项和512个Linux pte，共1,024个pte项。

代码段❷的pte_offset_kernel()函数中返回pte项中addr所属项地址。代码段❸中，在addr变成end前设置pte项值。set_pte_ext通过宏根据不同CPU执行不同函数，此处最终调用cpu_v6_set_pte_ext的汇编函数。

先设置作为参数传递的Linux pte信息后，初始化与此相符的硬件pte，然后清除pte。

14.6 准备使用高端内存——kmap_init()

正如14.4.1节所述，根据PAGE_OFFSET设置可指定内核使用的空间。一般指定为3 GB，内核使用这3GB以后的固定区域。如果该部分被物理内存或IO映射等耗尽，使内核需要更多可映射区域，那就通过调整PAGE_OFFSET直接扩展映射空间。

高端内存区域是不固定的、可变的管理区域。使用最后1个PMD管理高端内存，该函数就是为其准备的。该函数只在使用CONFIG_HIGHMEM设置时执行。当前版本的内核中设置为Experimental。[①]

① 3.2版本中正式配置了CONFIG_HIGHMEM。

:: 代码14-11　arch/arm/mm/mmu.c的kmap_init()

```
static void __init kmap_init(void)
{
#ifdef CONFIG_HIGHMEM

    pmd_t *pmd = pmd_off_k(PKMAP_BASE);                                    ❶
    pte_t *pte =
            alloc_bootmem_low_pages(2 * PTRS_PER_PTE * sizeof(pte_t));     ❷
    BUG_ON(!pmd_none(*pmd) || !pte);

    __pmd_populate(pmd, __pa(pte) | _PAGE_KERNEL_TABLE);                   ❸
    pkmap_page_table = pte + PTRS_PER_PTE;                                 ❹
#endif
}
```

在代码段❶中，从PKMAP_BASE获得pmd表。pkmap是persistent kernel map（持久内核映射）的缩写。在代码段❷中为pte分配空间，在代码段❸中设置pmd。从下列定义可知，它被设置为可使用非节区pte的表。

```
#define _PAGE_KERNEL_TABLE (PMD_TYPE_TABLE|PMD_BIT4|PMD_DOMAIN(DOMAIN_KERNEL))
```

最后，在代码段❹中指示pkmap_page_table变量能够直接访问pte，然后结束。此处也为硬件pte加上pte个数，使其指向Linux pte项。

14.7　初始化零页

零页是用0填充的、全局共享的、具有特殊用途的页。该页因初始化为0的数据（BSS分段）或访问读取而发生分页请求，用于向页表项映射。映射后设置COW（Copy On Write，写时复制），对该页执行写入时替换为可写入的页。分配用于零页的内存后，将零页的page结构体地址保存到empty_zero_page。

:: 代码14-12　arch/arm/mm/mmu.c的paging_init()

```
void __init paging_init(struct machine_desc *mdesc)
{
    void *zero_page;

    ...

    zero_page = alloc_bootmem_low_pages(PAGE_SIZE);                        ❶
    empty_zero_page = virt_to_page(zero_page);

    flush_dcache_page(empty_zero_page);                                    ❷

}
```

在代码段❶中，利用bootmem分配器分配用于零页的内存，求出零页的page结构体并保存到empty_zero_page。下面逐一介绍被调用的函数。

14.7.1 分配内存——__alloc_bootmem_nopanic()

alloc_bootmem_low_pages()被定义为宏，查询其调用路径可发现，调用的是__alloc_bootmem_nopanic()函数。该函数查找要执行内存分配的地址范围（goal ~ limit）内的节点，以分配内存。分配失败时，放弃调用该函数时指定的地址范围，以整体内存地址为范围重新分配。通过代码14-13分析__alloc_bootmem_nopanic()函数。

:: 代码14-13 mm/bootmem.c的___alloc_bootmem_nopanic()

```
static void * __init ___alloc_bootmem_nopanic(unsigned long size, unsigned long align,
                                               unsigned long goal, unsigned long limit)
{
    bootmem_data_t *bdata;
    void *region;

restart:
    region = alloc_arch_preferred_bootmem(NULL,size,align,goal,limit);                    ❶
    if (region) return region;

    list_for_each_entry(bdata, &bdata_list, list) {                                       ❷
        if (goal && bdata->node_low_pfn <= PFN_DOWN(goal))
            continue;
        if (limit && bdata->node_min_pfn >= PFN_DOWN(limit))
            break;
        region = alloc_bootmem_core(bdata, size, align, goal, limit);                     ❸
        if (region)
            return region;
        }
    if (goal) {                                                                           ❹
        goal = 0; goto restart;
    }
    return NULL;
}
```

在代码段❶中，调用的alloc_arch_preferred_bootmem()函数在ARM中不执行任何操作，返回NULL并结束。

在代码段❷中，对系统的全部节点表执行循环，并查找符合条件的节点。此处的“条件”指goal、limit这2个参数，只要节点范围包含在它们之间即可。图14-6中的节点2、3就符合条件。

limit为0时，从goal到最后的内存地址就是使用范围。找到符合地址范围的节点后，在代码段❸中从该节点执行内存分配。即使对所有节点执行循环仍未找到符合条件的节点时，就将goal初始化为0并重新搜索。一旦goal变为0，可请求的地址范围随之变大，分配成功率会变高。下面观察实际执行内存分配的alloc_bootmem_core()。

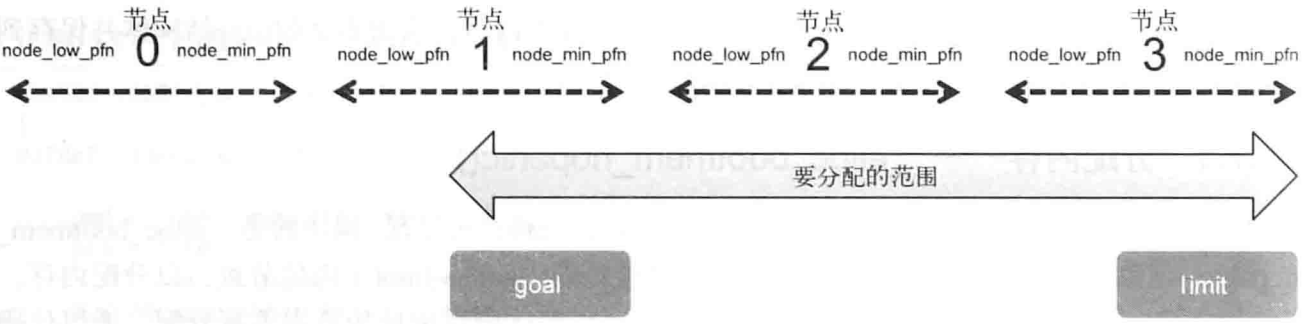


图14-6　包含于分配范围（goal ~ limit）内的节点

14.7.2　在指定节点使用fallback分配内存——alloc_bootmem_core

alloc_bootmem_core()函数在指点节点使用备份（fallback）分配内存。下面通过代码14-14进行分析。

:: 代码14-14　mm/bootmem.c的alloc_bootmem_core()

```
static void * __init alloc_bootmem_core(struct bootmem_data *bdata,
                                        unsigned long size, unsigned long align,
                                        unsigned long goal, unsigned long limit)
{
    unsigned long fallback = 0;

    sidx = start - bdata->node_min_pfn;                                    ❶
    midx = max - bdata->node_min_pfn;

    if (bdata->hint_idx > sidx) {                                          ❷
        fallback = sidx + 1;
        sidx = align_idx(bdata, bdata->hint_idx, step);
    }

    while (1) {                                                            ❸
find_block:
        sidx = find_next_zero_bit(bdata->node_bootmem_map, midx, sidx);
        sidx = align_idx(bdata, sidx, step);
        eidx = sidx + PFN_UP(size);

        if (sidx >= midx || eidx > midx) break;                            ❹

        for (i = sidx; i < eidx; i++)                                      ❺
            if (test_bit(i, bdata->node_bootmem_map)) {
                sidx = align_idx(bdata, i, step);
                if (sidx == i)
                    sidx += step;
                goto find_block;
            }

        if (bdata->last_end_off & (PAGE_SIZE - 1) &&
                    PFN_DOWN(bdata->last_end_off) + 1 == sidx)
```

```
            start_off = align_off(bdata, bdata->last_end_off, align);
        else
            start_off = PFN_PHYS(sidx);

        merge = PFN_DOWN(start_off) < sidx;
        end_off = start_off + size;

        bdata->last_end_off = end_off;                                    ❻
        bdata->hint_idx = PFN_UP(end_off);


        if (__reserve(bdata, PFN_DOWN(start_off) + merge,                 ❼
                      PFN_UP(end_off), BOOTMEM_EXCLUSIVE))
            BUG();
        region = phys_to_virt(PFN_PHYS(bdata->node_min_pfn) + start_off);
        memset(region, 0, size);
        return region;
    }
    if (fallback) {                                                       ❽
        sidx = align_idx(bdata, fallback - 1, step);
        fallback = 0;
        goto find_block;
    }
    return NULL;
}
```

该函数使用索引代替内存地址执行分配。[①]在代码段❶中，将传递来的地址参数转换为索引。

代码段❷的hint_idx具有该节点最近分配的区域的最后索引。由于hint_idx之前索引的页可能已被分配，因此，hint_idx比sidx更大时，将sidx设置为hint_idx。若已决定要执行分配的地址范围sidx、hint_idx，则进入代码段❸的循环。

在代码段❸中，调用find_next_zero_bit()查找sidx之后空出的第一页。查找的页的索引保存到sidx。在代码段❹中，若要分配的地址范围（sidx ~ edix）超出节点的地址范围，则无法执行分配，因此结束循环并执行代码段❽。

使用hint_idx执行分配时，原先要执行分配的索引保存到代码段❷的fallback。在代码段❽中恢复保存的索引后，跳转到find_block重新分配。

在代码段❺中，检查sidx ~ edix之间有无正在使用的页。若有，则将该页的下一个索引保存到sidx，并跳转到代码段❸。否则，因为已决定要分配的区域，因此在代码段❻中将已决定的最终地址保存到last_end_off，再次转换通过PFN_UP()转换的值，并保存到hint_idx（参考图14-7）。

① 节点的页分配与否记录在位图中，索引在位图中指向页。

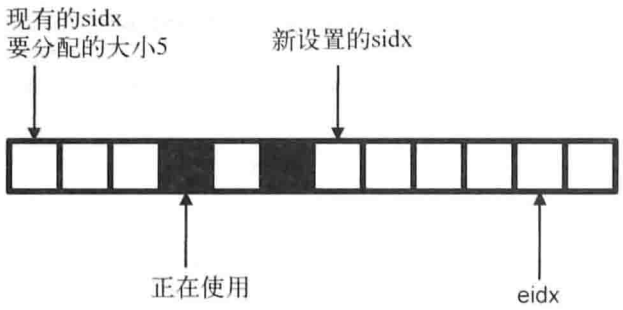


图14-7　重置sidx

在代码段❼中标记位图，表示要通过__reserve()函数使用该区域。若分配成功，则将内存初始化为0，并返回起始地址。

14.7.3　将虚拟地址变换为page结构体——virt_to_page

分配用于零页的页后，调用virt_to_page()求出零页的page结构体。

arch/arm/include/asm/memory.h

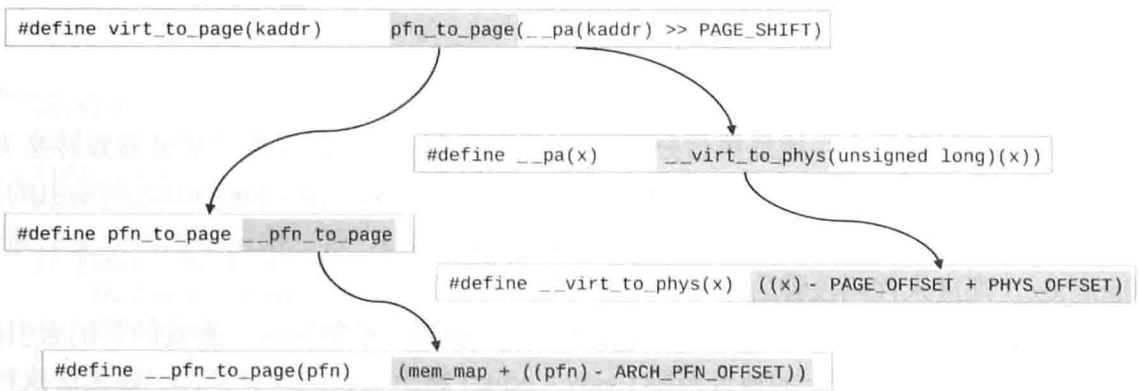


图14-8　virt_to_page()的内部函数流程

通过调用几种函数，virt_to_page()返回虚拟地址指向的页的page结构体。首先，调用__pa()将虚拟地址变更为物理地址，将该值向右移动（shift）PAGE_SHIFT值的大小可获得页帧号。将这个值作为参数调用pfn_to_page()，则返回page结构体。

mem_map指向相应节点的page结构体列表的起始地址，pfn – ARCH_PFN_OFFSET值是偏移量，指向page结构体。

14.8　保持数据缓存一致性——flush_dcache_page()

在分页初始化过程的最后阶段，为了确保数据缓存一致性，需要清除zero_page。清除缓存是指，将缓存中保存的数据向内存执行回写（writeback），从而将缓存内容反映到内存，以此确保缓存和内存间的一致性。下面观察代码14-15。

:: 代码14-15 arch/arm/mm/flush.c的__flush_dcache_page()

```
void __flush_dcache_page(struct address_space *mapping, struct page *page)
{
    if (!PageHighMem(page)) {                                          ❶
        __cpuc_flush_dcache_area(page_address(page), PAGE_SIZE);       ❷
    } else {
        void *addr = kmap_high_get(page);
        if (addr) {
            __cpuc_flush_dcache_area(addr,PAGE_SIZE);                  ❸
    kunmap_high(page);
        } else if (cache_is_vipt()) {                                  ❹
            addr = kmap_atomic(page);
            __cpuc_flush_dcache_area(addr, PAGE_SIZE);                 ❺
            kunmap_atomic(addr);
        }
    }

    if (mapping && cache_is_vipt_aliasing())                           ❻
        flush_pfn_alias(page_to_pfn(page),
            page->index << PAGE_CACHE_SHIFT);
}
```

在代码段❶中，执行清除缓存需要确认内存页是否属于高端内存，因为内核无法直接访问高端内存空间的页帧。因此，属于高端内存时，需要另外添加转换步骤。

代码段❷、❸、❺的__cpuc_flush_dcache_area()是在分散的各程序中最终执行的函数。如果映射到内核的内存页地址作为参数传递，就将相关缓存中的数据向内存执行回写。这样就能使物理页的数据与内核映射始终保持一致。

代码段❸表示，VIPT时，未映射的内存页可能仍然留在缓存中，将其写入内存。VIPT的缓存方式请参考13.5节“补充知识点”。

在VIPT方式中，具有缓存别名时，需要对别名的页帧另外添加清除操作。在代码段❻中，传递相应内存页的页帧号，清除别名页帧。

第 15 章 在启动时初始化内存分配器

各位用C语言编程时，可能经常使用malloc()函数在用户空间分配内存。内核也是一种程序，因此，进行内核编程时必须有函数用于分配内核使用的内存，因为我们无法使用用户空间中提供的函数。[①]

内核中用于分配内存的函数是kmalloc()和vmalloc()。需要考虑的是，若想通过kmalloc()和vmalloc()函数在内核中分配内存，需要有发出请求时执行的某种东西。Linux内核中管理内存的子系统就是slab内存分配器（slab memory allocator）。[②]

问题在于，内核处于启动阶段时，slab内存分配器尚未准备好。由于启动时也需要使用内存，所以要有（请求分配内存时能够对此进行处理的）启动时的内存分配器。启动时执行内存分配器作用的就是bootmem。



补充知识点 kmalloc vs. vmalloc

kmalloc()函数用于对物理上连续的内存进行分配，不需要在物理上连续时则使用vmalloc()。vmalloc()对物理上不连续的页修正页表，使其能够虚拟连续。也就是说，即使vmalloc()在物理上是不连续的，但在虚拟上也是连续的。查找物理上连续的内存相对比较困难，所以，需要分配比kmalloc()更多的内存时就使用vmalloc()。

那么，能用kmalloc()分配的最大内存是多少呢？其答案由slab.h文件中定义的KMALLOC_MAX_SIZE定义。

:: include/linux/slab.h

```
#define KMALLOC_SHIFT_HIGH  ((MAX_ORDER + PAGE_SHIFT - 1) <= 25 ? \
    (MAX_ORDER + PAGE_SHIFT - 1) : 25)

#define KMALLOC_MAX_SIZE    (1UL << KMALLOC_SHIFT_HIGH)
#define KMALLOC_MAX_ORDER   (KMALLOC_SHIFT_HIGH - PAGE_SHIFT)
```

由于KMALLOC_SHIFT_HIGH的最大值为25，因此2^25，即通过kmalloc()函数最多可分配32 MB。换言之，用kmalloc()函数最多只能分配32 MB。[③]

---

① 这和内核中使用printk替代printf的原理是一样的。

② 除slab内存分配器外，还有slub、slob分配器。关于这些分配器请参考第22章“补充知识点 其他内存分配器slub、slob”。

③ 根据MAX_ORDER、PAGE_SHIFT值的不同，通过kmalloc()进行分配的值可能小于32 MB。

15.1 bootmem 函数流和数据结构

bootmem_init()函数对启动时的内存分配器bootmem进行初始化，执行bootmem_init()函数后，通过alloc_bootmem_low_pages()函数分配内存。启动时执行的内存分配不必精确到字节单位，因为启动临近结束时会由精确管理内存的slab分配器替代，所以不需要辛苦地对启动时使用的内存分配器执行精确操作。即使调用alloc_bootmem_low_pages()函数要求内存，也可以页单位进行分配。

从结构复杂度上讲，本章的bootmem_init()函数在内核代码内属于上级组合。结构如此复杂的代码在完成启动后让位给slab分配器，自己则消亡。因此，对其进行深入分析显得十分困难。

本章不会精确了解到底层级别，而是先把握整体流程，再分析各函数的意义。就算消亡也不能完全忽略bootmem，因为替换为slab分配器前，它会初始化slab分配器使用的所有节点描述符，所以可通过bootmem把握内存管理结构。

图15-1是bootmem_init()函数的整体流程（flow），简单了解其作用后再分析代码。

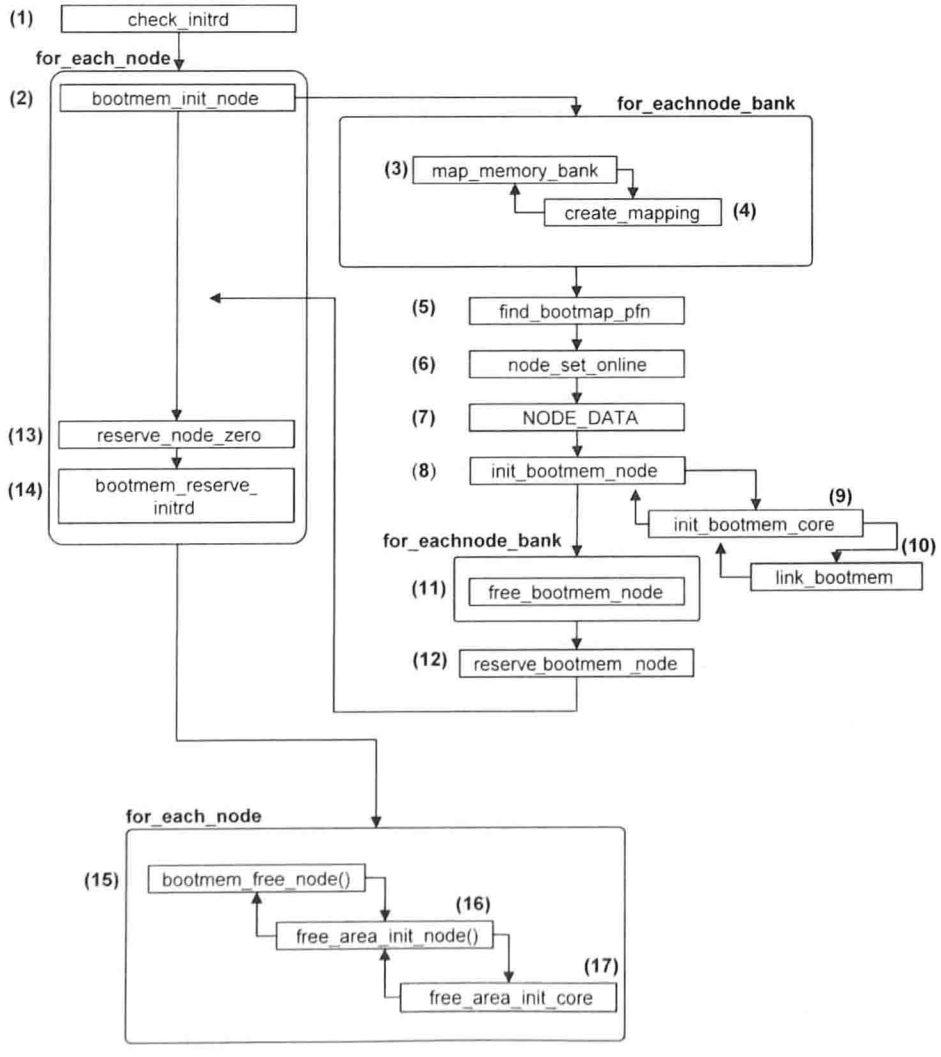


图15-1 bootmem_init()函数的整理流程

函数流程中的标签简介如下。

(1) check_initrd()：查找具有虚拟内存盘映像（ramdisk image）的节点（node）。[①]查找的节点通过(14)bootmem_reserve_initrd()函数预约（reserve）对应于虚拟内存盘映像的页帧（page frame），使请求内存时不被选为可用页。

(2) bootmem_init_node：对构成系统内存的各节点执行，节点均由BANK构成。对各BANK从(3)map_memory_bank()函数执行到(12)reserve_bootmem_node()函数。利用属于节点的BANK信息修正页目录（init_mm.pgd指向的地方）项。

通过位图管理节点的所有页，将这些页的大小和位置设置到pgdat->bdata。可以认为pgdat是表示节点的节点描述符（node descriptor）。此外，管理各节点的页的管理位图中显示，所有页均可用。最后，对各位图页显示“正在使用”。[②]

(13) reserve_node_zero()：在管理节点页的位图中，将内核的text、data、bss区域和页目录所在0号节点的页均设置为“正在使用”。

(14) bootmem_reserve_initrd()：在位图中，将（通过(1)check_initrd()函数找到的）对应于虚拟内存盘映像的页帧设置为正在使用的页。[③]

(15) bootmem_free_node()：系统的物理内存条的集合叫做节点，NUMA结构中有多个节点。Linux中将属于节点的各页按属性分类，并将各集合称为ZONE，用struct zone进行管理。

bootmem_free_node()函数最终初始化zone结构体的free_area，并设置struct page的各标签值，这些值与ZONE中的物理页帧1:1映射。各结构体由bootmem_free_node()进行初始化，为以后slab分配器的再次管理做准备。[④]



补充知识点　Linux中的内存条、节点、NUMA、UMA、ZONE、页帧

Linux中使用的内存条、节点、NUMA、ZONE、页帧的关系如图15-2所示。

- 内存条（Memory Bank）
  - 由内存插槽（memory slot）组成。
  - 各插槽中安装相同类型的内存称之为内存条。
- 节点（Node）
  - 从1个CPU访问速度相同的内存集合。
  - 多个BANK可构成1个节点。
  - 在内核中记述为pg_data_t类型，表示节点的结构体称为节点描述符（node descriptor）。

① 内核启动时使用的文件系统所在内存，该文件系统在完成启动后不再使用。

② 因为有分页请求时不能分配位图所属各页。

③ 所谓“设置”是指，将对应于位图页帧号的位从0变更为1。

④ 在bootmem管理的各页目录（catalog）位图中，free_bootmem_node()函数将可用页设置为可用。与此相反，bootmem_free_node()函数将ZONE管理的各页中的可用页设置为0。这2个函数名称使用的单词虽然一样，但意义不同。

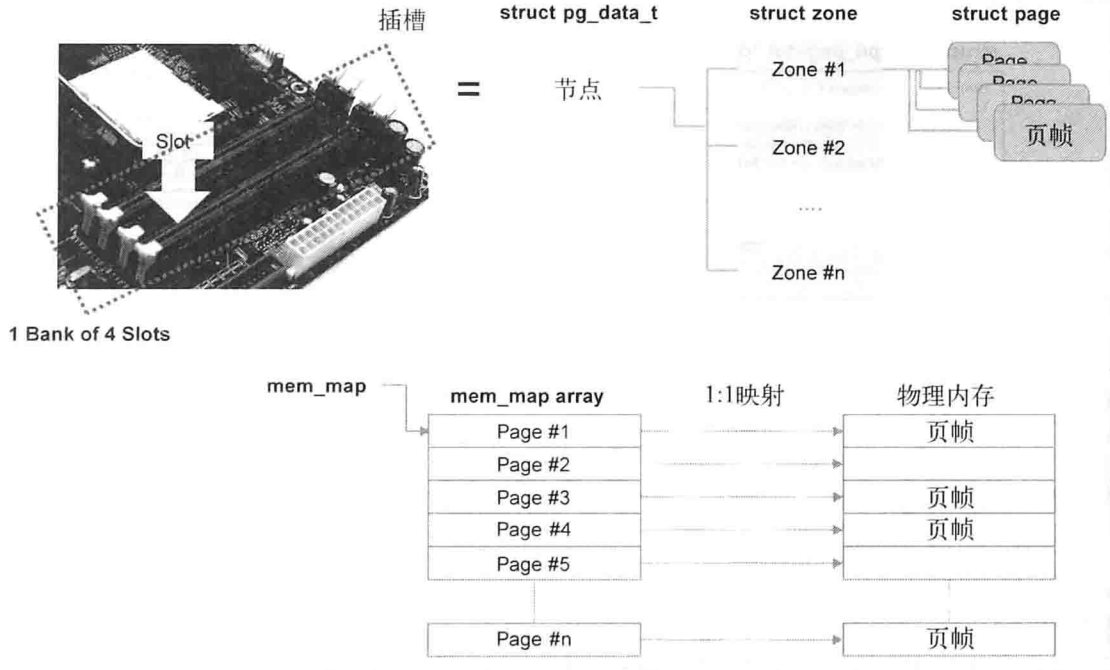


图15-2 Linux的内存条、节点、ZONE、页帧的关系

- NUMA（Non-Uniform Memory Access，非统一内存访问）vs. UMA（Uniform Memory Access，统一内存访问）[①]
  - Linux中显示内存条的结构。
  - NUMA是从处理器对内存的访问速度不同的结构。
  - UMA是处理器与所有内存的访问速度相同的结构。
- ZONE
  - 节点中具有相同属性的区域。
  - 有DMA、NORMAL、HIGHMEM、ZONE_MOVABLE等，但不使用CONFIG_HIGHMEM时，只存在1个NORMAL_ZONE。
  - 内核中由struct zone结构体管理。
- 页帧（Page Frame）
  - ZONE中管理物理内存的最小单位称为页帧。
  - 页帧在Linux中由page结构体管理，通过mem_map全局数组访问。

本章讲解的内容看起来多少有些复杂，图15-3用图形描述了Linux中管理内存的各结构体间的关系。从1到8是本章要了解的各结构体间的关系，希望各位记住这些结构体之间的关系并分析代码。

① 为保证NUMA和UMA平台间的代码一致性，各UMA平台均具有静态分配的pg_data_t。

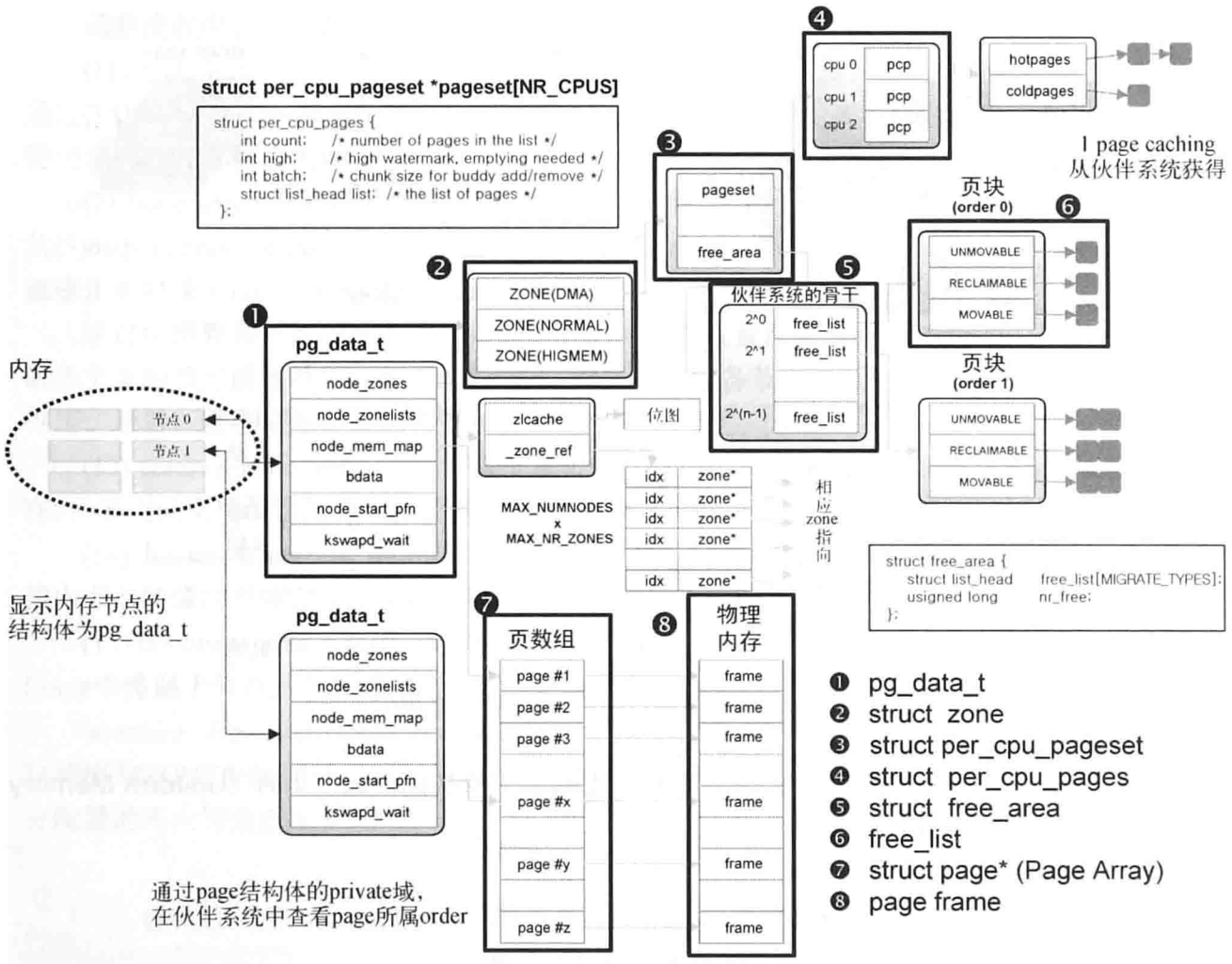


图15-3　Linux中管理内存的结构体之间的关系图

15.2　查看 bootmem_init()结构

代码15-1是bootmem_init()函数。

∷ 代码15-1　arch/arm/mm/init.c的bootmem_init()

```
void __init bootmem_init(void)
{
    struct meminfo *mi = &meminfo;
    int node, initrd_node;

    initrd_node = check_initrd(mi);                                    ❶

    for_each_node(node) {
        unsigned long end_pfn = bootmem_init_node(node, mi);           ❷
        if (node == 0)
```

```
            reserve_node_zero(NODE_DATA(node));                    ❸
        if (node == initrd_node)
            bootmem_reserve_initrd(node);                          ❹
    ...

    for_each_node(node)
        bootmem_free_node(node, mi);                               ❺

    ...
}
```

在代码段❶中获得虚拟内存盘所在节点号（15.3节）。

在代码段❷中，将组成内存的所有节点的BANK信息反映到页目录。在管理所有页的位图中显示可用，然后对保存位图的页显示正在使用，使之不能分配内存（15.4节）。

在代码段❸中预约0号节点，使之不能分配内存。0号节点是内核的text、data、bss、页目录所在位置（15.5节）。

在代码段❹中预约代码段❶中获得的虚拟内存盘节点，也使之不能分配内存（15.6节）。

在代码段❺中调用bootmem_free_node()函数，将ZONE管理的所有页的nr_free成员变量设置为0，显示无可用页（15.7节）。

15.3 查找虚拟内存盘位置——check_initrd()

通过代码15-2分析check_initrd()函数。

:: 代码15-2 arch/arm/mm/init.c的check_initrd()

```
static int __init check_initrd(struct meminfo *mi)
{
    int initrd_node = -2;

#ifdef CONFIG_BLK_DEV_INITRD                                       ❶
    unsigned long end = phys_initrd_start + phys_initrd_size;      ❷

    if (phys_initrd_size) {
        unsigned int i;
        initrd_node = -1;

        for (i = 0; i < mi->nr_banks; i++) {                       ❸
            struct membank *bank = &mi->bank[i];
            if (bank_phys_start(bank) <= phys_initrd_start &&
                end <= bank_phys_end(bank))
                initrd_node = bank->node;
        }
    }

    if (initrd_node == -1) {                                       ❹
        printk(KERN_ERR "INITRD: 0x%08lx+0x%08lx extends beyond "
               "physical memory - disabling initrd\n",
```

```
                phys_initrd_start, phys_initrd_size);
        phys_initrd_start = phys_initrd_size = 0;
    }
#endif

    return initrd_node;
}
```

在check_initrd()函数之前记述的__init让check_initrd()函数位于.init.text节区，完成启动后，该区域内存可重新使用。

在代码段❶中，代码根据是否定义CONFIG_BLK_DEV_INITRD分别执行，要编译内核时指定如下。

:: init/Kconfig

```
config BLK_DEV_INITRD
    bool "Initial RAM filesystem and RAM disk (initramfs/initrd) support"
    depends on BROKEN || !FRV
    ...
    If unsure say Y.
```

CONFIG_BLK_DEV_INITRD表示是否支持虚拟内存盘，不支持时只返回-2。先假定支持虚拟内存盘，然后分析代码。

根据代码段❷的phys_initrd_size值决定是否执行代码段❸，phys_initrd_start和phys_initrd_size变量的定义如下所示。

:: arch/arm/mm/init.c

```
static unsigned long phys_initrd_start __initdata = 0;
static unsigned long phys_initrd_size __initdata = 0;
```

这两个变量通过early_initrd(char **p)函数分析从启动加载项接收的参数，并设置值。

:: arch/arm/mm/init.c

```
static void __init early_initrd(char **p)
```

假设从启动加载项接收了以下信息。

```
console=ttySAC0,115200 root=/dev/ram init=/bin/bash initrd=0x51000000,4M
```

通过early_initrd()函数设置虚拟内存盘的起始地址和大小。

```
phys_initrd_start = 0x51000000
phys_initrd_size = 0x400000
```

在代码段❸中查找具有虚拟内存盘的系统内存条，将该内存条所属节点号保存到initrd_node

变量，并返回initrd_node，从而查找具有虚拟内存盘的节点。

在代码段❹中，若未找到虚拟内存盘所在BANK，则输出错误信息。

15.4 将节点的BANK信息反映到页目录——bootmem_init_node()

bootmem_init_node()函数对各节点的BANK执行循环，生成管理系统内存的位图。通过代码15-3对bootmem_init_node()函数进行整体指向。

:: 代码15-3 arch/arm/mm/init.c的bootmem_init_node()

```
static unsigned long __init bootmem_init_node(int node, struct meminfo *mi)
{
    ...

    for_each_nodebank(i, mi, node) {                                    ❶
        struct membank *bank = &mi->bank[i];

        ...

        map_memory_bank(bank);
    }

        ...

    boot_pages = bootmem_bootmap_pages(end_pfn - start_pfn);            ❷

    boot_pfn = find_bootmap_pfn(node, mi, boot_pages);                  ❸

    node_set_online(node);                                              ❹

    pgdat = NODE_DATA(node);                                            ❺

    init_bootmem_node(pgdat, boot_pfn, start_pfn, end_pfn);             ❻

    for_each_nodebank(i, mi, node) {                                    ❼
        struct membank *bank = &mi->bank[i];
        free_bootmem_node(pgdat, bank_phys_start(bank), bank_phys_size(bank));
        ...
    }

    reserve_bootmem_node(pgdat, boot_pfn << PAGE_SHIFT,                 ❽
                boot_pages << PAGE_SHIFT, BOOTMEM_DEFAULT);

    return end_pfn;
}
```

代码段❶对接收的节点循环属于节点的BANK，同时调用map_memory_bank()函数。map_memory_bank()函数利用BANK信息填充map_desc结构体的值，再利用map_desc结构体中的信息修正对应于BANK的页目录项值。

最后，利用接收的节点号修正对应于该节点的所有内存条的页目录项（15.4.1节）。

在代码段❷中调用bootmem_bootmap_pages()函数，利用节点的页帧数（end_pfn – start_pfn）求出保存（管理该节点中的页的）位图bootmap的页数（15.4.2节）。

在代码段❸中调用find_bootmap_pfn()函数，查找放置（通过bootmem_bootmap_pages()函数求出的）bootmap页的起始页帧号（15.4.3节）。

在代码段❹中调用node_set_online()函数，将当前节点的状态设置为ONLINE（15.4.4节）。

在代码段❺中调用NODE_DATA()宏，从discontig_node_data数组获得pg_data_t类型的相应节点描述符。在内核中访问节点时经常使用NODE_DATA()宏（15.4.5节）。

在代码段❻中调用init_bootmem_node()函数，对节点描述符pgdat（pg_data_t类型）的成员变量bootmem_data结构体类型的bdata执行初始化，将bdata连接到链表后，再将显示节点页可用与否的所有node_bootmem_map位图设置为正在使用（15.4.6节）。

在代码段❼中调用free_bootmem_node()函数，将bootmem可分配的页初始化为0（15.4.7节）。

在代码段❽中请求内存分配时，将包含用于管理的位图的页从分配对象中排除。在位图中，reserve_bootmem_node()函数将用于管理的位图页对应的所有位均设置为1，表示不可用（15.4.8节）。

15.4.1　map_memory_bank()

通过代码15-4分析map_memory_bank()函数。

:: 代码15-4　arch/arm/mm/init.c的map_memory_bank()

```
static inline void map_memory_bank(struct membank *bank)
{
#ifdef CONFIG_MMU
    struct map_desc map;
    map.pfn = bank_pfn_start(bank);
    map.virtual = __phys_to_virt(bank_phys_start(bank));
    map.length = bank_phys_size(bank);
    map.type = MT_MEMORY;

    create_mapping(&map);
#endif
}
```

map_memory_bank()函数利用BANK信息填充map_desc结构体。map_desc结构体定义如下。

:: arch/arm/include/asm/mach/map.h

```
struct map_desc {
    unsigned long virtual;      /*起始虚拟地址*/
    unsigned long pfn;          /*起始页帧号*/
    unsigned long length;       /*内存大小*/
    unsigned int type;          /*内存类型*/
};
```

向BANK信息map_desc结构体填充值后，调用create_mapping()函数，并利用map_desc结构体的值修正与BANK对应的页目录项值。修正页目录项值时，可设置多种内存类型。

通过代码15-5了解create_mapping()函数执行的内容。

:: 代码15-5 arch/arm/mm/mmu.c的create_mapping()

```
void __init create_mapping(struct map_desc *md)
{
    const struct mem_type *type;

    ...

    type = &mem_types[md->type];                                    ❶

    ...

    addr = md->virtual & PAGE_MASK;
    pgd = pgd_offset_k(addr);
    end = addr + length;

    do {                                                            ❷
        /*以2MB为单位执行循环*/
        unsigned long next = pgd_addr_end(addr, end);
        alloc_init_section(pgd, addr, next, phys, type);

        phys += next - addr;
        addr = next;
    } while (pgd++, addr != end);
}
```

在代码段❶中，利用接收的参数map_desc的内存类型，获得mem_types数组中静态保存的内存类型信息。struct mem_type和静态声明的mem_types数组的值根据内存类型按如下方式进行初始化。

:: arch/arm/mm/mm.h

```
struct mem_type {
    unsigned int prot_pte;
    unsigned int prot_l1;
    unsigned int prot_sect;
    unsigned int domain;
};
```

:: arch/arm/mm/mmu.c

```
static struct mem_type mem_types[ ] = {

    ...

    [MT_HIGH_VECTORS] = {
        .prot_pte  = L_PTE_PRESENT ¦ L_PTE_YOUNG ¦ L_PTE_DIRTY ¦
```

```
                L_PTE_USER | L_PTE_EXEC,
        .prot_l1   = PMD_TYPE_TABLE,
        .domain    = DOMAIN_USER,
    },
    [MT_MEMORY] = {
        .prot_sect = PMD_TYPE_SECT | PMD_SECT_AP_WRITE,
        .domain    = DOMAIN_KERNEL,
    },
    [MT_ROM] = {
        .prot_sect = PMD_TYPE_SECT,
        .domain    = DOMAIN_KERNEL,

    ...

};
```

假定md->type＝MT_MEMORY时，代码段❶的type变量指向具有内存类型列表的静态内存类型结构体mem_types[MT_MEMORY]的值。

在代码段❷中，以2MB为单位执行循环，同时调用alloc_init_section()函数，根据type变量中保存的内存类型，更改对应于被指定BANK的各页目录项。[①]循环的单位next为pgd_addr_end的结果值，是以2 MB为单位排列的地址。pgd_addr_end的定义如下所示。

∷ include/asm-generic/pgtable.h

```
#define pgd_addr_end(addr, end)                         \
({  unsigned long __boundary = ((addr) + PGDIR_SIZE) & PGDIR_MASK;  \
    (__boundary - 1 < (end) - 1)? __boundary: (end);        \
})
```

PGDIR_SIZE和PGDIR_MASK的定义如下所示。

∷ arch/arm/include/asm/pgtable.h

```
#define PGDIR_SHIFT     21
#define PGDIR_SIZE      (1UL << PGDIR_SHIFT)
#define PGDIR_MASK      (~(PGDIR_SIZE-1))
```

此处不讲解alloc_init_section()函数的内容，但有一点需要确认，通过alloc_init_section()修正页目录项值，因此必须反映到内存。向内存反映的函数是flush_pmd_entry()函数，通过alloc_init_section()函数调用。flush_pmd_entry()函数如下所示。[②]

---

① level 1的pgd列表以2MB为单位执行循环，因为level 2中的1个pmd页与level 1的2个项对应，从而相当于2MB区域。

② 需要注意，函数名以flush开头并在内存中引起反映。ARM中对正在使用的缓存的命令有清除（flush，无内存反映）和清理（clean，有内存反映），二者经常混用，所以在上下文中需要适当进行解释。

:: arch/arm/include/asm/tlbflush.h

```
static inline void flush_pmd_entry(pmd_t *pmd)
{
    const unsigned int __tlb_flag = __cpu_tlb_flags;

    if (tlb_flag(TLB_DCLEAN))
        asm("mcr    p15, 0, %0, c7, c10, 1  @ flush_pmd"
            : : "r" (pmd) : "cc");

    if (tlb_flag(TLB_L2CLEAN_FR))
        asm("mcr    p15, 1, %0, c15, c9, 1  @ L2 flush_pmd"
            : : "r" (pmd) : "cc");

    if (tlb_flag(TLB_WB))
        dsb();
}
```

15.4.2 bootmem_bootmap_pages()

下面了解bootmem_bootmap_pages()函数。bootmem_bootmap_pages()函数将相应节点的所有页帧数读入参数。代码15-6表示bootmem_bootmap_pages()函数及其调用的bootmap_bytes()函数。

:: 代码15-6 mm/bootmem.c的bootmem_bootmap_pages()
bootmap_bytes()

```
unsigned long __init bootmem_bootmap_pages(unsigned long pages)
{
    unsigned long bytes = bootmap_bytes(pages);                    ❶

    return PAGE_ALIGN(bytes) >> PAGE_SHIFT;                        ❷
}

static unsigned long __init bootmap_bytes(unsigned long pages)
{
    unsigned long bytes = (pages + 7) / 8;

    return ALIGN(bytes, sizeof(long));
}
```

代码段❶中，bootmap_bytes()函数用于计算用位表示（作为参数接收的）相应节点页数所需字节数。此处通过ALIGN宏将字节数以long型（4字节）排列。最终，求出用位表示相应节点页数所需字节数后，以4字节为单位（假定long为4字节）排列。宏ALIGN()的定义如下所示。

:: include/linux/kernel.h

```
#define __ALIGN_MASK(x,mask)    (((x)+(mask))&~(mask))
#define ALIGN(x,a)      __ALIGN_MASK(x,(typeof(x))(a)-1)
```

求出的字节数再次通过代码段❷的PAGE_ALIGN()宏求出所需页数。

:: include/linux/mm.h

```
#define PAGE_ALIGN(addr) ALIGN(addr, PAGE_SIZE)
```

:: arch/arm/include/asm/page.h

```
#define PAGE_SHIFT      12
#define PAGE_SIZE       (1UL << PAGE_SHIFT)
```

最后，通过bootmem_bootmap_pages()函数求出保存（用于管理相应节点中）各页的位图bootmap的页数。

15.4.3　find_bootmap_pfn()

前面已经通过bootmem_bootmap_pages()函数求出bootmap页数，现在需要决定求出的页该放在何处。代码15-7的find_bootmap_pfn()函数执行该任务。

:: 代码15-7　arch/arm/mm/init.c的find_bootmap_pfn()

```
static unsigned int __init
find_bootmap_pfn(int node, struct meminfo *mi, unsigned int bootmap_pages)
{
    unsigned int start_pfn, i, bootmap_pfn;

    start_pfn   = PAGE_ALIGN(__pa(_end)) >> PAGE_SHIFT;                    ❶
    bootmap_pfn = 0;

    for_each_nodebank(i, mi, node) {
        struct membank *bank = &mi->bank[i];
        unsigned int start, end;

        start = bank_pfn_start(bank);                                      ❷
        end   = bank_pfn_end(bank);

        if (end < start_pfn)                                               ❸
            continue;

        if (start < start_pfn)                                             ❹
            start = start_pfn;

        if (end <= start)                                                  ❺
            continue;

        if (end - start >= bootmap_pages) {                                ❻
            bootmap_pfn = start;
            break;
        }
    }
```

```
    ...

    return bootmap_pfn;
}
```

在代码段❶中，决定管理节点页的位图bootmap的保存位置时，为了不使用text、data、bss区域，求出指向bss区域末尾的虚拟地址_end的物理地址后，向右移动PAGE_SHIFT（12位）大小。这样就可求出保存bss区域的页帧的最后帧号（2^12 = 4 KB单位）。

代码段❷到代码段❻中，求出节点的各BANK起始和末尾页帧号后，就能获得BANK页帧号，其起始位置比内核bss区域的页帧号大，且最后帧号能够包含bootmap页。该页帧号就是要放置bootmap的页。在该代码中查找位置的条件可简单表述如下。

```
start_pfn < start
bootmap_pages < end - start
```

这样就通过find_bootmap_pfn()函数求出了放置bootmap页的页帧号。

15.4.4 node_set_online()

node_set_online()函数将当前节点状态设置为ONLINE。节点状态和node_set_online()函数的定义如下所示。

:: include/linux/nodemask.h

```
enum node_states {
    N_POSSIBLE,             /*将要联机（online）的节点*/
    N_ONLINE,               /*当前联机的节点*/
    N_NORMAL_MEMORY,        /*普通内存类型的节点*/
#ifdef CONFIG_HIGHMEM
    N_HIGH_MEMORY,          /*普通内存或高端内存类型的节点*/
#else
    N_HIGH_MEMORY = N_NORMAL_MEMORY,
#endif
    N_CPU,                  /*节点具有1个或多个CPU*/
    NR_NODE_STATES          /*表示node_states的末尾*/
};

#define node_set_online(node)       node_set_state((node), N_ONLINE)
```

继续观察node_set_online()函数，确认ONLINE状态记录位置。由于Linux内核中利用DECLARE_BITMAP()宏制作并使用位图，因此，先分析node_set_online()函数，一并了解DECLARE_BITMAP()宏。node_set_online()函数的调用顺序分析如下。

```
node_set_state() → __node_set() → set_bit()
```

:: include/linux/nodemask.h

```
static inline void node_set_state(int node, enum node_states state)
{
    __node_set(node, &node_states[state]);
}

static inline void __node_set(int node, volatile nodemask_t *dstp)
{
    set_bit(node, dstp->bits);
}
```

set_bit()函数在位图中设置对应于节点的位，此处位图定义为nodemask_t结构体。nodemask_t结构体的定义如下所示。

:: include/linux/nodemask.h

```
typedef struct { DECLARE_BITMAP(bits, MAX_NUMNODES); } nodemask_t;
```

:: include/linux/types.h

```
#define DECLARE_BITMAP(name,bits) \
    unsigned long name[BITS_TO_LONGS(bits)]
```

用DECLARE_BITMAP宏制作最小的unsigned long数组，该数组可表示MAX_NUMNODES数大小的位。

DECLARE_BITMAP宏在Linux内核中常用于生成显示状态的位图。例如，用于显示CPU状态的cpumask结构体利用DECLARE_BITMAP定义如下。

:: include/linux/cpumask.h

```
typedef struct cpumask { DECLARE_BITMAP(bits, NR_CPUS); } cpumask_t;
```

15.4.5　NODE_DATA宏

NODE_DATA()是内核中经常使用的宏，其定义如下所示。

:: include/linux/nodemask.h

```
#define NODE_DATA(nid)      (&discontig_node_data[nid])
```

只在discontig_node_data数组中返回对应于节点号nid的项地址，discontig_node_data具有节点描述符数组。discontig_node_data是pg_data_t类型数组，共具有16个（MAX_NUMNODES）项，其定义如下所示。

:: arch/arm/mm/discontig.c

```
pg_data_t discontig_node_data[MAX_NUMNODES] = {
    { .bdata = &bootmem_node_data[0] },
    { .bdata = &bootmem_node_data[1] },
    { .bdata = &bootmem_node_data[2] },
    { .bdata = &bootmem_node_data[3] },
#if MAX_NUMNODES == 16
    { .bdata = &bootmem_node_data[4] },
    { .bdata = &bootmem_node_data[5] },
    { .bdata = &bootmem_node_data[6] },
    { .bdata = &bootmem_node_data[7] },
    { .bdata = &bootmem_node_data[8] },
    { .bdata = &bootmem_node_data[9] },
    { .bdata = &bootmem_node_data[10] },
    { .bdata = &bootmem_node_data[11] },
    { .bdata = &bootmem_node_data[12] },
    { .bdata = &bootmem_node_data[13] },
    { .bdata = &bootmem_node_data[14] },
    { .bdata = &bootmem_node_data[15] },
#endif
};
EXPORT_SYMBOL(discontig_node_data);
```

最终，表示内核能够支持的最大节点数为16个。

pg_data_t类型的定义如下所示，所有节点在内核中以pg_data_t类型记述并管理。

:: include/linux/mmzone.h

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];   /*节点的zone数组*/

    /*对zone的备份列表，zone中无可用页时，寻找节点的类型相同的其他zone以分配内存*/

    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;           /*节点内zone的个数*/

    /*页数组指针指向属于相应节点的页的起始地址*/
    struct page *node_mem_map;

    ...

    /*未激活slab分配器时，即启动时的内存管理器bootmem为管理内存所需结构体，作为bootmem_data成员，它具
      有能够判别相应节点页是否可用的位图*/
    struct bootmem_data *bdata;

    ...

    /*节点的起始页帧号，系统内的所有节点的页赋予连续编号，也是整个系统的唯一编号*/
    unsigned long node_start_pfn;
    unsigned long node_present_pages; /*物理页总数*/
    unsigned long node_spanned_pages; /*包含空洞（hole）的总页数*/
    int node_id;                      /*节点号*/
```

```
    /*用于交换守护进程 (swap daemon) 的等待队列 (wait queue)，用于对zone中的帧进行交换*/
    wait_queue_head_t kswapd_wait;
    /*对交换守护进程的task struct指针*/
    struct task_struct *kswapd;
    /*使用交换系统 (swapping system)，用于决定free区域的大小*/
    int kswapd_max_order;
} pg_data_t;
```

NODE_DATA()宏从discontig_node_data数组返回pg_data_t类型的相应节点描述符。

15.4.6　init_bootmem_node()

init_bootmem_node()函数对该节点描述符pgdat(pg_data_t类型)的成员变量bdata执行初始化。具有用于管理的位图信息的bdata是bootmem_data结构体变量，将bdata连接到链表后，将所有位图设置为正在使用。

bootmem_data结构体的定义如下所示。

:: include/linux/bootmem.h

```
typedef struct bootmem_data {
    unsigned long node_min_pfn;     /*节点的起始pfn*/
    unsigned long node_low_pfn;     /*节点的最后pfn*/
    void *node_bootmem_map;         /*用于管理的位图*/
    unsigned long last_end_off;     /*最后分配的位置*/
    unsigned long hint_idx;         /*下一次使用的页 (提示) */
    struct list_head list;          /*连接其他节点的bootmem_data的指针*/
} bootmem_data_t;
```

由init_bootmem_node()函数设置bootmem_data结构体，如代码15-8所示，调用并执行init_bootmem_core()函数和link_bootmem()函数。

:: 代码15-8　mm/bootmem.c的init_bootmem_node()
init_bootmem_core()
link_bootmem()

```
unsigned long __init init_bootmem_node(pg_data_t *pgdat, unsigned long freepfn,
            unsigned long startpfn, unsigned long endpfn)
{
    return init_bootmem_core(pgdat->bdata, freepfn, startpfn, endpfn);
}


static unsigned long __init init_bootmem_core(bootmem_data_t *bdata,
    unsigned long mapstart, unsigned long start, unsigned long end)
{
    unsigned long mapsize;

    ...
```

```
    bdata->node_bootmem_map = phys_to_virt(PFN_PHYS(mapstart));                ❶
    bdata->node_min_pfn = start;
    bdata->node_low_pfn = end;

    link_bootmem(bdata);                                                       ❷

    mapsize = bootmap_bytes(end - start);
    memset(bdata->node_bootmem_map, 0xff, mapsize);                            ❸

    return mapsize;
}

static void __init link_bootmem(bootmem_data_t *bdata)
{
    struct list_head *iter;

    /*bdata_list是指向自己的list_head头*/
    list_for_each(iter, &bdata_list) {
        bootmem_data_t *ent;

        ent = list_entry(iter, bootmem_data_t, list);
        if (bdata->node_min_pfn < ent->node_min_pfn)
            break;
    }
    list_add_tail(&bdata->list, iter);
}
```

在代码段❶中设置bootmem_node成员变量的值。

通过代码段❷将bootmem_node插入链表。link_bootmem函数的执行方法请参考图15-4。

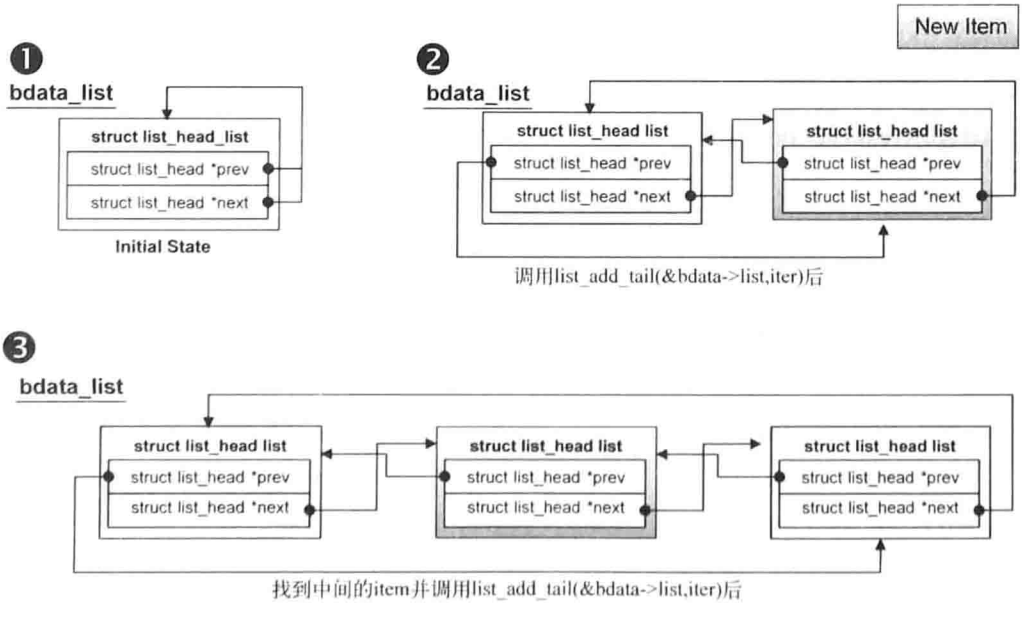


图15-4 bdata->list链表

通过代码段❸将用于管理的位图bootmem_data结构体的node_bootmem_map设置为0xff，也就是将所有位设置为1，表示无可用页。

15.4.7　free_bootmem_node()

管理节点页的管理位图node_bootmem_map在init_bootmem_node()函数中均已设置为1。代码15-9表示free_bootmem_node()函数的定义，调用mark_bootmem_node()函数将所有页重置为0。

:: 代码15-9　mm/bootmem.c的free_bootmem_node()

```
void __init free_bootmem_node(pg_data_t *pgdat, unsigned long physaddr,
                              unsigned long size)
{
    unsigned long start, end;
    start = PFN_UP(physaddr);
    end = PFN_DOWN(physaddr + size);

    mark_bootmem_node(pgdat->bdata, start, end, 0, 0);
}
```

在管理属于pgdat->bdata的成员节点的页的位图（pgdt->bdata->node_bootmem_map）中，将第strat位到第end位均设置为0，表示相应页可用。此处的start和end是属于对应节点的BANK的起始和末尾帧地址。[①]

15.4.8　reserve_bootmem_node()

虽然已在mark_bootmem_node()中设置相应节点的所有页可用，但请求分配内存时，必须将实际包含用于管理的位图的页从分配对象中排除。在用于管理的位图（pgdt->bdata->node_bootmem_map）中，正是reserve_bootmem_node()函数将相应页（boot_pfn开始boot_pages大小）的位均设置为1，表示不可用。

代码15-10表示reserve_bootmem_node()函数调用mark_bootmem_node()函数，并将相应位设置为1。

:: 代码15-10　mm/bootmem.c的reserve_bootmem_node()

```
int __init reserve_bootmem_node(pg_data_t *pgdat, unsigned long physaddr,
                unsigned long size, int flags)
{
    unsigned long start, end;
    start = PFN_DOWN(physaddr);
    end = PFN_UP(physaddr + size);

    return mark_bootmem_node(pgdat->bdata, start, end, 1, flags);
}
```

① 请确认代码15-3的代码段❼通过for_each_nodebank执行for循环。

这就是bootmem_init_node()函数，其作用概括如下。

- 为节点的各内存条设置相应页目录项值。
- 求出用于管理的节点页的页管理器位图起始位置和页数。
- 将节点设置为ONLINE。
- 填充节点描述符（pg_data_t类型）的值。

将对应于节点的页管理器位图的页标记为正在使用，使内存分配器分配内存时无法分配包含位图的页。

15.5 排除 0 号节点——reserve_node_zero()

在节点用于管理页的位图中，reserve_node_zero()函数将内核的text、data、bss及页目录区域所在0号节点的相应页默认设置为正在使用。图15-5是表示内核区域的起始和末尾虚拟地址_stext和_end。

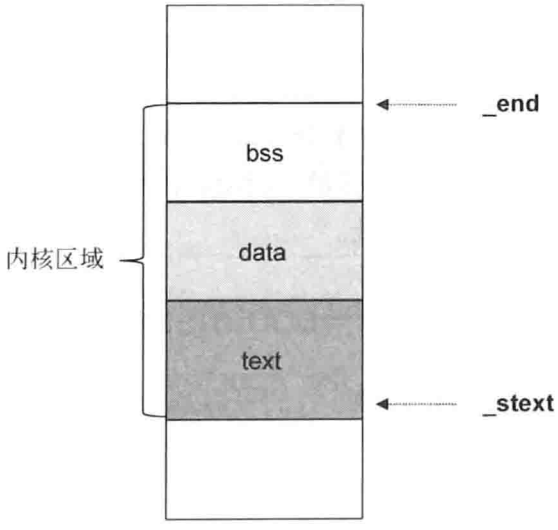


图15-5 内核区域的起始和末尾

通过代码15-11了解reserve_bootmem_node()函数（在对应于内核区域的_stext到_end之间）设置0号节点的页管理位图的方法。

:: 代码15-11 arch/arm/mm/mmu.c的reserve_node_zero()

```
void __init reserve_node_zero(pg_data_t *pgdat)
{
    unsigned long res_size = 0;

    /*text、data、bss区域设置*/
    reserve_bootmem_node(pgdat, __pa(_stext), _end - _stext,          ❶
                         BOOTMEM_DEFAULT);
```

```
    /*swapper_pg_dir: 页目录位置*/
    reserve_bootmem_node(pgdat, __pa(swapper_pg_dir),                    ❷
                         PTRS_PER_PGD * sizeof(pgd_t), BOOTMEM_DEFAULT);

    ...

}
```

在代码段❶中，将内核的text、data、bss区域的页显示为正在使用。在代码段❷中，将对应于页目录的页（在页管理器位图中）标记为正在使用。

_stext和swapper_pg_dir都是虚拟地址，要变更为物理地址以求出页帧号。对虚拟地址和物理地址进行转换的宏如下所示。

:: arch/arm/include/asm/memory.h

```
#define __pa(x)                         __virt_to_phys((unsigned long)(x))
#define __va(x)                         ((void *)__phys_to_virt((unsigned long)(x)))

#define __virt_to_phys(x)               ((x) - PAGE_OFFSET + PHYS_OFFSET)
#define __phys_to_virt(x)               ((x) - PHYS_OFFSET + PAGE_OFFSET)
```

将虚拟地址变更为物理地址的方法十分简单：从虚拟地址x减掉内核空间的起始位置PAGE_OFFSET（0xC0000000）后得到偏移量（offset），将它加到PHYS_OFFSET（物理内存的起始地址）即可获得虚拟地址x的物理地址。

15.6　排除虚拟内存盘节点——bootmem_reserve_initrd()

下面了解bootmem_reserve_initrd()函数。我们可以基于讲过的内容推断出，在页管理位图中，bootmem_reserve_initrd()函数将对应于虚拟内存盘区域的页设置为正在使用。代码15-12是bootmem_reserve_initrd()函数。

:: 代码15-12　arch/arm/mm/init.c的bootmem_reserve_initrd()

```
static void __init bootmem_reserve_initrd(int node)
{
#ifdef CONFIG_BLK_DEV_INITRD
    pg_data_t *pgdat = NODE_DATA(node);                                  ❶
    int res;

    res = reserve_bootmem_node(pgdat, phys_initrd_start,                 ❷
                phys_initrd_size, BOOTMEM_EXCLUSIVE);

    ...

#endif
}
```

在代码段❶中获得具有虚拟内存盘节点的节点描述符。

在代码段❷中调用reserve_bootmem_node()函数，将虚拟内存盘节点的页管理位图（pgdat->bdata->node_bootmem_map）从虚拟内存盘的起始页帧（phys_initrd_start）到随机内存盘页数（phys_initrd_size）大小的位标记为1，表示虚拟内存盘区域的页正在使用。

15.7 设置为无可用页——bootmem_free_node()

分析bootmem_free_node()函数前，再次观察曾调用该函数的代码15-1的代码段❺。

:: 代码15-1 bootmem_init()函数的代码段❺

```
    for_each_node(node)
        bootmem_free_node(node, mi);                                    ❺
```

这段代码对各节点调用bootmem_free_node()函数。此处可能发生问题，因为已经在代码15-3的代码段❼中通过free_bootmem_node()函数设置了可用区域。再次观察这段代码。

:: 代码15-3 bootmem_init_node()函数的代码段❼

```
    for_each_nodebank(i, mi, node) {                                    ❼
        struct membank *bank = &mi->bank[i];
        free_bootmem_node(pgdat, bank_phys_start(bank), bank_phys_size(bank));
        ...
    }
```

下面通过代码15-13了解函数名相似的bootmem_free_node()和free_bootmem_node有何差异。

:: 代码15-13 arch/arm/mm/init.c的bootmem_free_node()

```
static void __init bootmem_free_node(int node, struct meminfo *mi)
{
    unsigned long zone_size[MAX_NR_ZONES], zhole_size[MAX_NR_ZONES];
    unsigned long start_pfn, end_pfn;
    int i;

    pg_data_t *pgdat = NODE_DATA(node);                                 ❶

    /*求出节点的起始和末尾帧号*/
    start_pfn = pgdat->bdata->node_min_pfn;                             ❷
    end_pfn = pgdat->bdata->node_low_pfn;

    /*初始化zone_size、zhole_size*/
    memset(zone_size, 0, sizeof(zone_size));                            ❸
    memset(zhole_size, 0, sizeof(zhole_size));

    /*将全部页帧数保存到zone_size[0]*/
    zone_size[0] = end_pfn - start_pfn;                                 ❹
    zhole_size[0] = zone_size[0];

    /*求出相当于hole的页帧数*/
```

```
    for_each_nodebank(i, mi, node)                                          ❺
        zhole_size[0] -= bank_pfn_size(&mi->bank[i]);

    /*ARM默认不执行任何操作*/
    arch_adjust_zones(node, zone_size, zhole_size);

    /*在pgdat结构体的内容中初始化pgdat->node_zones->free_area*/
    free_area_init_node(node, zone_size, start_pfn, zhole_size);            ❻
}
```

在代码段❶中求出相应节点的节点描述符pgdat。

在代码段❷中访问pgdat->bdata，求出bootmem管理的node_min_pfn和node_low_pfn的值。node_min_pfn和node_low_pfn具有节点的起始和末尾页帧号。

在代码段❸中，对保存节点所属ZONE的大小zone_size和保存ZONE中包含的空洞大小的zhole_size数组进行初始化。[①]



补充知识点　zone_size[]和zhole_size[]的关系

zone_size[]数组用于保存不同ZONE类型具有的页数，zhole_size[]数组用于保存不同ZONE类型具有的空洞页数，如图15-6所示。

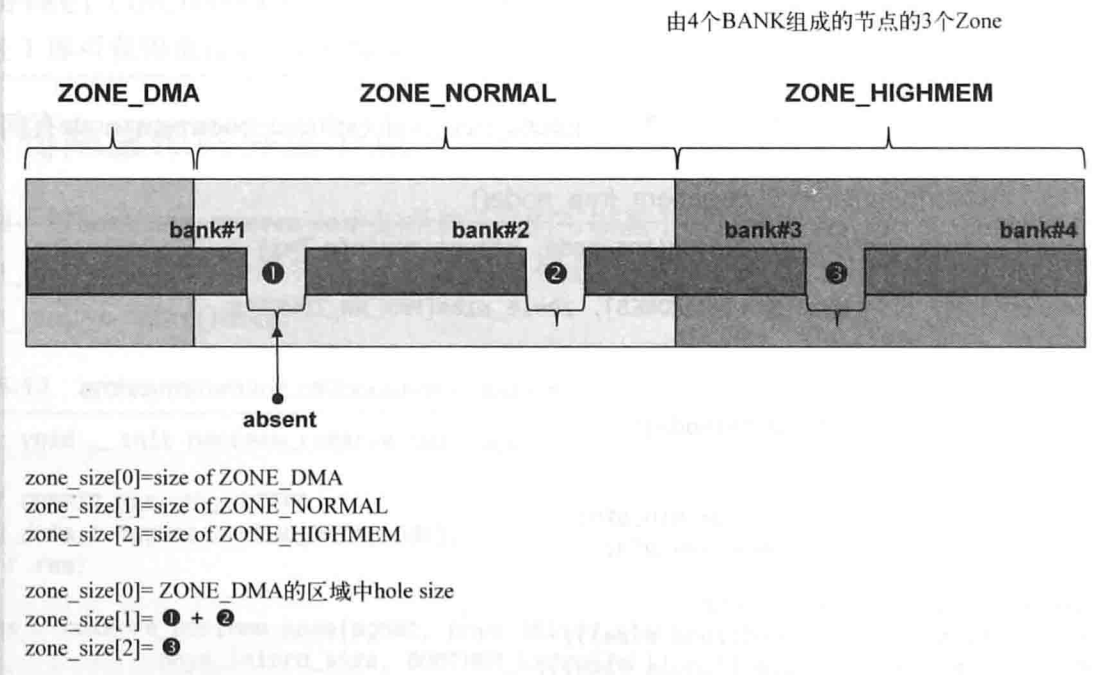


图15-6　zone_size[]和zhole_size[]的关系

① ZONE是节点中类型相同的页的集合。ZONE类型中有ZONE_DMA、ZONE_DMA32、ZONE_NORMAL、ZONE_HIGHMEM、ZONE_MOVABLE等。

在代码段❹中，将相应节点的页帧数保存到zone_size[0]。

在代码段❺中，将对应于ZONE的空洞的页数保存到zhole_size[0]。值得注意的是，bank_pfn_size()函数用于查找BANK的页帧数，不可能存在空洞。因此，在保存包括空洞的所有页帧数的zone_size[0]中减掉通过bank_pfn_size函数获得的结果值，即可求出相应页中的空洞页数。

代码段❻的free_area_init_node()函数就是bootmem_free_node()的核心函数。实际上，free_area_init_node()函数调用free_area_init_core()函数以初始化free_area。此处的free_area用于初始化pgdat结构体的zone类型成员变量node_zones的free_area区域。free_area_init_node()函数将在15.8.2节详细讲解。

bootmem_free_node()函数的作用概括如下。

- 设置未设置节点描述符pgdat的成员变量的值。
- 设置pgdat的zone结构体的值。
- 初始化zone结构体的free_area的链表，并将nr_free设置为0。
- 将属于ZONE的页帧对应的struct page结构体的flags设置为PG_reserved。

15.8 初始化 free_area 区域

free_area_init_node()函数调用free_area_init_core()函数对free_area进行初始化。学习free_area_init_node()函数前，先分析free_area结构体。

15.8.1 free_area结构体

先观察struct zone的结构和free_area结构体。struct zone和struct_free_area的定义如下。

:: include/linux/mmzone.h

```
#define MAX_ORDER 11

struct zone {
    /*页分配器 (page allocator) 主要使用的域*/
    unsigned long       pages_min, pages_low, pages_high;

    /*应对OOM (out of memory, 内存溢出), 确保下层内存*/
    unsigned long       lowmem_reserve[MAX_NR_ZONES];

    ...

    /*为每个CPU缓存1页, 不返回伙伴系统
     *每个CPU都拥有, 以快速分配
     */
#ifdef CONFIG_NUMA
    struct per_cpu_pageset  *pageset[NR_CPUS];
#else
    struct per_cpu_pageset  pageset[NR_CPUS];
#endif
```

```
    /*伙伴系统的骨干结构体，用于管理可用页*/
    struct free_area    free_area[MAX_ORDER];

    ...

    unsigned long       flags;          /*保存对zone的统计信息*/

    /*zone的起始页帧号*/
    atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];

    unsigned long       zone_start_pfn; /*zone的初始页帧号*/
    unsigned long       spanned_pages;  /*包含空洞的总页数*/
    unsigned long       present_pages;  /*减掉空洞的实际页数*/

    ...

} ___cacheline_internodealigned_in_smp;

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

free_area结构体是内核的内存分配器slab分配器的骨干结构，用于管理可用页。具有MAX_ORDER大小的项，各项以项索引为基准，管理2^{0}个到2^{MAX_ORDER - 1}个可用页。

例如，free_area[0]是1页的可用列表，free_area[1]是2^1 = 2页，而free_area[2]是2^2 = 4页。这个free_area作为伙伴系统（buddy system）的骨干结构，可视为该伙伴系统中实现了slab分配器。

各可用页根据页类型分为struct free_area的free_list进行管理。free_list是数组，按如下所示将页类型分为5种进行管理。

:: include/linux/mmzone.h

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

15.8.2　free_area_init_node()

下面正式分析free_area_init_node()函数，代码15-14是其定义。

:: 代码15-14　mm/page_alloc.c的free_area_init_node()

```
void __paginginit free_area_init_node(int nid, unsigned long *zones_size,
        unsigned long node_start_pfn, unsigned long *zholes_size)
{
    pg_data_t *pgdat = NODE_DATA(nid);                                    ❶
```

```
    pgdat->node_id = nid;
    pgdat->node_start_pfn = node_start_pfn;

    /*设置pgdat->node_spanned_pages、pgdat->node_present_pages*/
    calculate_node_totalpages(pgdat, zones_size, zholes_size);          ❷

    /*设置pgdat->node_mem_map*/
    alloc_node_mem_map(pgdat);                                          ❸

    ...

    /*设置struct zone*/
    free_area_init_core(pgdat, zones_size, zholes_size);                ❹
}
```

在代码段❶中求出相应节点的节点描述符pgdat，设置pgdat的成员变量节点号（node_id）和页的起始页帧号（node_start_pfn）。

代码段❷的calculate_node_totalpages()函数为pgdat的成员变量（包括空洞在内的总页数（node_spanned_pages））和除空洞外的页数（node_present_pages）设置值。

在代码段❸中设置pgdat的成员变量node_mem_map，node_mem_map在页数组中指向相应节点的页起始位置。此处的页数组管理系统页帧，与页帧和页数组项1:1映射。

在代码段❹中填充pgdat的zone结构体。其核心作用是设置free_area数组，并设置与该页帧对应的页数组标签（flag）。free_area_init_core()函数将在15.8.3节具体讲解。

15.8.3 free_area_init_core()

通过代码15-15了解free_area_init_core()函数。

:: 代码15-15 mm/page_alloc.c的free_area_init_core()

```
static void __paginginit free_area_init_core(struct pglist_data *pgdat,
        unsigned long *zones_size, unsigned long *zholes_size)
{

    ...

    for (j = 0; j < MAX_NR_ZONES; j++) {
        /*节点的下一个zone*/
        struct zone *zone = pgdat->node_zones + j;
        size = zone_spanned_pages_in_node(nid, j, zones_size);

        ...
        /*包含空洞的zone大小*/
        zone->spanned_pages = size;                                     ❶
        /*DMA、DMA32、NORMAL、MOVABLE之一*/
        zone->name = zone_names[j];
        zone->zone_pgdat = pgdat;    /*zone所属节点的pgdat*/
        zone->prev_priority = DEF_PRIORITY;
```

```
        zone->reclaim_stat.recent_rotated[0] = 0;
        zone->reclaim_stat.recent_rotated[1] = 0;
        zone->reclaim_stat.recent_scanned[0] = 0;
        zone->reclaim_stat.recent_scanned[1] = 0;
        zap_zone_vm_stats(zone);
        zone->flags = 0;

        /*求出zone的batchsize，初始化结构体zone的per-cpu pageset结构体
         *Struct per_cpu_pageset *pageset[NR_CPUS]
         */
        zone_pcp_init(zone);                                              ❷

        /*初始化zone的free_area结构体*/
        ret = init_currently_empty_zone(zone, zone_start_pfn,             ❸
                                        size, MEMMAP_EARLY);

        /*设置struct page结构体的flags成员的PG_reserved位*/
        memmap_init(size, nid, j, zone_start_pfn);                        ❹
    }
}
```

在代码段❶中设置zone结构体的成员变量值。[①]

在代码段❷中初始化zone结构体的per_cpu_pageset结构体变量pageset。per_cpu_pageset按CPU进行管理，缓存1页。它不直接返回伙伴系统，为快速分配而按不同CPU持有页。

在代码段❸中初始化zone结构体的free_area结构体。init_currently_empty_zone()函数初始化free_area的链表，将显示可用页的成员变量nr_free设置为0（15.8.4节）。

代码段❹调用memmap_init()函数，在与页帧具有1:1映射关系的页数组（page array）中，向相应页帧的page结构体的flags成员设置PG_reserved位。memmap_init()函数将在15.8.5节进行讲解。



补充知识点　伙伴系统

伙伴系统（buddy system）是管理连续内存的系统，以命令（order）为单位管理页。order=0时，管理2^{0}=1页；order=1时，管理2^{1}=2页；order=2时，管理2^{2}=4页。图15-7描述了伙伴系统以命令为单位管理页的过程。

为什么叫伙伴（buddy）呢？众所周知，伙伴指“好朋友”。伙伴系统将各页集成（grouping）后进行管理。页集成的条件是，必须是与自己相邻的页，正是这相邻页将成为自己的伙伴。也就是说，使用页后返回free_area并进行管理时，该页必须与自己的伙伴集成在一起。

① 为了提高代码可读性，我稍微调整了顺序。

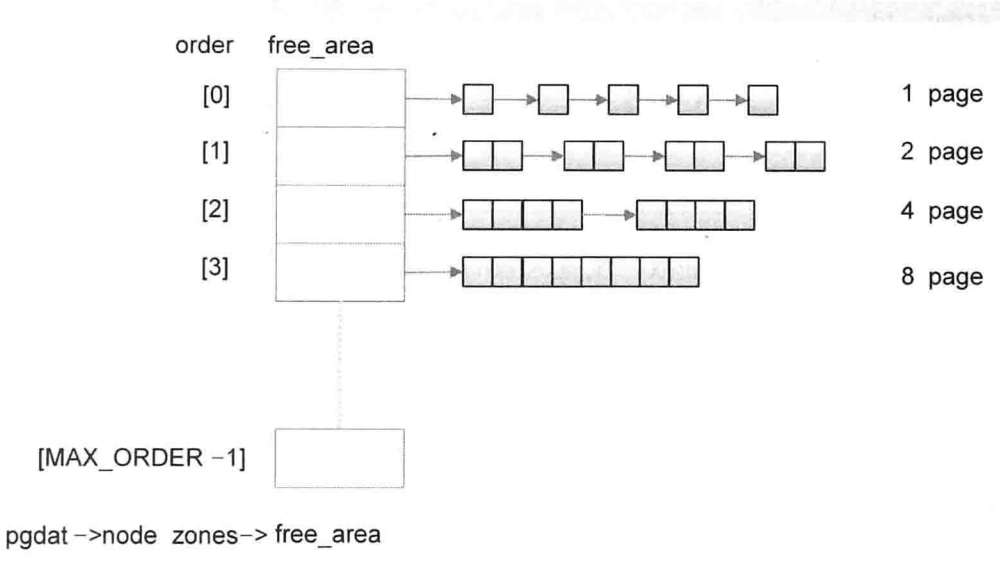


图15-7 伙伴系统的页管理

15.8.4 init_currently_empty_zone()

init_currently_empty_zone()函数如代码15-16所示，调用zone_init_free_lists()函数，初始化free_area的链表，将显示可用页的成员变量nr_free设置为0。

:: 代码15-16 mm/page_alloc.c的init_currently_empty_zone()和zone_init_free_lists()

```
__meminit int init_currently_empty_zone(struct zone *zone,
                    unsigned long zone_start_pfn, unsigned long size,
                    enum memmap_context context)
{
    /*zone所属节点*/
    struct pglist_data *pgdat = zone->zone_pgdat;

    ...

    /*利用数组特性求出对应zone的idx
     *ZONE_DMA时zone_idx()返回0
     *ZONE_NORMAL时zone_idx()返回1
     */
    pgdat->nr_zones = zone_idx(zone) + 1;                          ❶
    zone->zone_start_pfn = zone_start_pfn;

    /*初始化free_area的链表，使nr_free=0*/
    zone_init_free_lists(zone);                                    ❷

    return 0;
}

static void __meminit zone_init_free_lists(struct zone *zone)
{
```

```
    int order, t;
    for_each_migratetype_order(order, t) {
        INIT_LIST_HEAD(&zone->free_area[order].free_list[t]);
        zone->free_area[order].nr_free = 0;
    }
}
```

代码段❶保存属于节点的zone数，设置区域的起始页帧号（zone_start_pfn）。

代码段❷调用zone_init_free_lists函数，将显示free_area[]的各命令（order）的可用页数的值nr_free均初始化为0。

15.8.5　memmap_init()

下面了解memmap_init()函数。在与页帧具有1:1映射关系的页数组中，memmap_init()函数，对相应页帧的page结构体的flags成员设置PG_reserved位。page结构体的定义如下。

:: include/linux/mm_types.h

```
struct page {
    unsigned long flags;    /*Atomic flags*/
    atomic_t _count;        /*Usage count*/
    union {
        atomic_t _mapcount; /*当前映射的页表项的值*/
        ...
    };
    union {
        struct {
        /*page->flags中设置的值不同，用途也会改变
          例如，设置PG_buddy，表示伙伴系统的
          命令*/
        unsigned long private;

        /*页帧所属地址空间*/
        struct address_space *mapping;
        };
    ...
    };
};
```

flags成员表示页属性，不依赖结构特性的独立标签如表15-1所示。

表15-1　page->flag的标签列表

标　签	意　义
PG_reserved	不发生交换（swap）
PG_private	page->private为非NULL时，为I/O使用页时
PG_locked	锁住页。从文件向内存帧读入数据时，用于防止竞态条件（race condition）
PG_uptodate	表示页的内容是否有效，意味着页缓存的内容与磁盘文件相同

（续）

标　签	意　义
PG_referenced	浏览页
PG_reclaim	可回收以重新使用页
PG_error	该页发生I/O错误时
PG_arch_1	从属于结构的状态时
PG_highmem	位于各页的高端内存区域，永远不与内核空间的虚拟地址进行映射
PG_buddy	页处于可用状态，并位于伙伴系统
PG_dirty	变更页内容
PG_writeback	用块设备保存页内容
PG_compound	由多个页组成的compound page的一部分
PG_slab	在slab分配器中使用的页
PG_swapcache	页存在于交换缓存（swap cache）
PG_active	积极使用页

memmap_init()函数的定义如下所示。

:: mm/page_alloc.c

```
#ifndef __HAVE_ARCH_MEMMAP_INIT
#define memmap_init(size, nid, zone, start_pfn) \
    memmap_init_zone((size), (nid), (zone), (start_pfn), MEMMAP_EARLY)
#endif
```

最终调用memmap_init_zone()函数，memmap_init_zone()的定义如代码15-17所示。

:: 代码15-17　mm/page_alloc.c的memmap_init_zone()

```
void __meminit memmap_init_zone(unsigned long size, int nid, unsigned long zone,
        unsigned long start_pfn, enum memmap_context context)
{
    struct page *page;
    unsigned long end_pfn = start_pfn + size;
    unsigned long pfn;
    struct zone *z;

    if (highest_memmap_pfn < end_pfn - 1)                          ❶
        highest_memmap_pfn = end_pfn - 1;

    z = &NODE_DATA(nid)->node_zones[zone];                         ❷
    for (pfn = start_pfn; pfn < end_pfn; pfn++) {

        ...

        page = pfn_to_page(pfn);                                   ❸
        set_page_links(page, zone, nid, pfn);

        init_page_count(page);                                     ❹
```

```
            reset_page_mapcount(page);
            SetPageReserved(page);

            ...

            INIT_LIST_HEAD(&page->lru);                                    ❺

            ...
        }
    }
```

在代码段❶中调用每个zone，同时将highest_memmap_pfn更新为end_pfn - 1。highest_memmap_pfn是全局变量，原先被初始化为0。

代码段❷利用NODE_DATA()宏，访问当前节点的节点描述符后，获得相应区域的地址。

在代码段❸中利用页帧号获取相应页的页结构体。调用set_page_links()函数为page->flags设置与zone、node、section相关的位。

在代码段❹中先调用init_page_count()函数和reset_page_mapcount()函数，并设置page->count = 1，page->_mapcount = -1。_count域表示页的使用次数，用于保存使用页的浏览次数。_count为0时表示未使用该页。然后调用SetPageReserved()函数对struct page->flags设置PG_reserved位。

在代码段❺中将设置好的页添加到链表，添加好的页可利用链表对页进行访问搜索。

最后，通过bootmem_free_node()函数结束代码15-1的bootmem_init()函数。结束bootmem_init后，启动时可从bootmem分配内存。接收分配时利用alloc_bootmem_low_pages函数，以页为单位执行分配。



补充知识点　bootmem为了分配页而寻找位图中设置为0之处的方法

bootmem在内核启动时分配内存，它需要通过节点查找可用页，并通过位图进行管理。因此，为了查找可用页，必须找到位图中设置为0之处。为找到设置为0的最初的位，内核提供_find_first_zero_bit_le和_find_next_zero_bit_le函数。[①]这2个函数均由汇编代码编写而成。

∷ arch/arm/lib/findbit.S

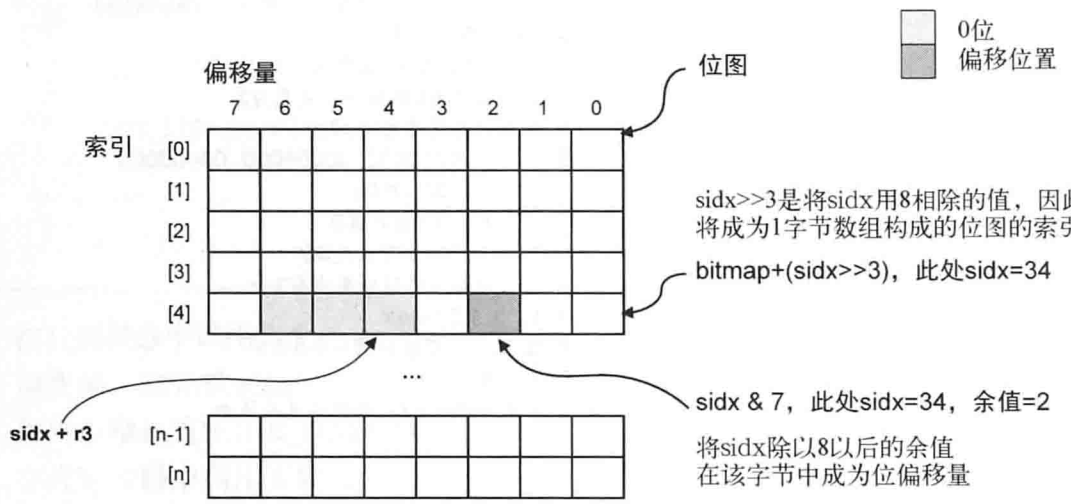
```
ENTRY(_find_first_zero_bit_le)
    teq r1, #0
    beq 3f
    mov r2, #0
1:  ldrb    r3, [r0, r2, lsr #3]
    eors    r3, r3, #0xff
    bne .L_found
    add r2, r2, #8                          @ 下一位
2:  cmp r2, r1                              @ 是否还有要比较的位?
    blo 1b
3:  mov r0, r1                              @ 无剩余的位
    mov pc, lr
```

① 这里的_le表示小端序（little endian），大端序时函数名以_be结束。

```
ENDPROC(_find_first_zero_bit_le)
.L_found:
#if __LINUX_ARM_ARCH__ >= 5
    rsb r1, r3, #0                      @ r1 = 0 - r3
    and r3, r3, r1                      @ r3 = r1 & r3运算后，r3的值是
                                        @ 只将位值为0的第一位设置为1
                                        @ 剩余的均设置为0，例如
                                        @ r3 = 00000000 00000000 00000000
                                              00000010
   clz r3, r3                           @ 遇到第一个1前计算0
   rsb r3, r3, #31                      @ 31 - r3，为什么是31?
                                        @ 位图的相应字节中至少有1个
                                        @ 1，因此0个max
                                        @ 值为31，可求出第一个0位的
                                        @ 偏移量
    add r0, r2, r3                      @ 为基准值（r2）上偏移量后返回
#else
    ....
#endif
    mov pc, lr

ENTRY(_find_next_zero_bit_le)
    teq r1, #0                          @ 如果maxbit(size)为0
    beq 3b                              @ 则跳转到backward 3
    ands    ip, r2, #7                  @ 检查r2（sidx）是否为8的倍数
                                        @ 如果是8的倍数，由于位图是通过
                                          _find_first_zero_bit_le
                                        @ 由8位组成的，因此
                                        @ 是8的倍数时可从1字节的第一位
                                        @ 开始检查，但不是8的倍数时
                                        @ 用8相除后处理余值
       beq 1b                           @ 是8的倍数时计算访问_find_first_zero_bit_le
       ldrb    r3, [r0, r2, lsr #3]     @ 的内存地址
                                        @ 从基准点r0起用r2、lsr #3
                                        @ 求出字节单位偏移量
       eor r3, r3, #0xff                @ 对r3和0xff执行EOR运算，将其结果值保存到r3
                                        @ 若r3为非0值，表示原r3的值中
                                        @ 有0位
       movs    r3, r3, lsr ip           @ 以r2除以8后的余值大小
                                        @ 向右移动
                                        @ 因为需要跳过
                                        @ 相当于余值的偏移量
                                        @ 例如，如果ip为6，需要在第6个以上的
                                        @ 位中找到是0的值
       bne .L_found                     @ 到此，如果r3的值依然非0
                                        @ 则表示r3中存在表示0的位
                                        @ 故无法分页
                                        @ 有可分配的页，因此
                                        @ 跳转到.L_found
       orr r2, r2, #7                   @ 若未找到
       add r2, r2, #1                   @ 则排列位指针
       b   2b                           @ 并执行循环
ENDPROC(_find_next_zero_bit_le)
```

用图像描述该过程，如图15-8所示。请查看按各步骤（step）提供的示例。



Step 1	Index [4] = r3 = 1 0 0 0 1 1 1 1
Step 2	0xff = 1 1 1 1 1 1 1 1
Step 3	r3 = r3 XOR 0xff = 0 1 1 1 0 0 0 0
Step 4	r3 = r3 >> (sidx & 7) = 0 0 0 1 1 1 0 0
Step 5	r1 = 0 – r3 = 1 1 1 0 0 1 0 0
Step 6	r3 = r1 and r3 = 0 0 0 0 0 1 0 0
Step 7	clz r3, r3 = 29, (**00000000 00000000 00000000 00000**100)
Step 8	r3 = 31 – r3 = 2

最初可用位的位置=sidx+r3

[步骤3] 如果R3 XOR 0xff非0
则r3包含0位

[步骤4] 移动2位，如果r3非0
则r3中有0位

[步骤5] 从底层位开始，只将最初位
1的位置设置为1
意思是步骤4中表示第一个1的位，
其1在步骤1中表示
第一个可用的位0

[步骤7] 对前导0（leading zero）
执行运算（counting），r3=29

[步骤8] 至少1个1位中会有1，因此
0的最大个数从31减掉r3后
sidx+r3在索引[4]中指向最初可用的位

图15-8　在位图中查找第一个0

第16章 mm_init_owner()~preempt_disable()

在start_kernel()函数调用的各函数中，本章对mm_init_owner()函数到smp_prepare_boot_cpu()函数进行分析。函数调用关系如图16-1所示。

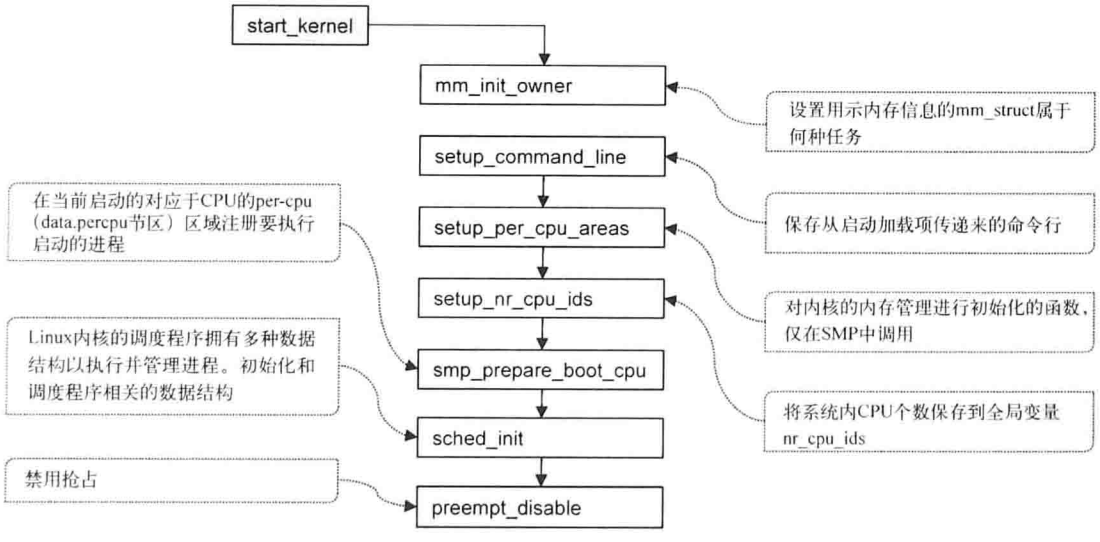


图16-1　从mm_init_owner()到preempt_disable()的调用关系

16.1　设置内存拥有者——mm_init_owner()

所有任务都具有自己的内存，启动阶段中的当前任务必须具备属于自己的内存。mm_init_owner()函数的作用正是如此。mm_init_owner()函数在start_kernel()函数中按如下方式调用，它传递init_mm和init_task参数。

:: init/main.c的start_kernel()

```
mm_init_owner(&init_mm, &init_task);
```

init_mm和init_task被声明为INIT_MM()和INIT_TASK()宏并初始化（10.1节、14.4.3节）。其

声明部分如下所示。

:: arch/arm/kernel/init_task.c

```
struct mm_struct init_mm = INIT_MM(init_mm);
```

:: arch/arm/kernel/init_task.c

```
struct task_struct init_task = INIT_TASK(init_task);
```

mm_init_owner()函数的定义如下所示。

:: kernel/fork.c

```
#ifdef CONFIG_MM_OWNER
void mm_init_owner(struct mm_struct *mm, struct task_struct *p)
{
    mm->owner = p;
}
#endif /* CONFIG_MM_OWNER */
```

该函数只在init_mm的owner成员变量中指定init_task，表示init_task是init_mm的拥有者。

16.2　保存命令行——setup_command_line()

命令行从启动加载项传递到内核。我们在12.2节讲解过parse_cmdline()函数，该函数分析内核命令行参数并调用符合各选项值的函数，以设置启动环境。

通过从最初的启动加载项接收的命令行和parse_cmdline()函数，setup_command_line()函数将处理的参数分为2个参数，以此创建启动过程中能够浏览的副本。从启动加载项接收的命令行，即原命令行称为untouched command line，执行参数分析后的命令行称为touched command line，这2个命令行会得到保存。

代码16-1表示setup_command_line()函数的作用，其结构十分简单。

:: 代码16-1　init/main.c的setup_command_line()

```
static void __init setup_command_line(char *command_line)
{
    saved_command_line = alloc_bootmem(strlen (boot_command_line)+1);          ❶
    static_command_line = alloc_bootmem(strlen (command_line)+1);
    strcpy (saved_command_line, boot_command_line);                            ❷
    strcpy (static_command_line, command_line);
}
```

在代码段❶中，分配相当于boot_command_line()和command_line大小的内存。在代码段❷中，将字符串复制到分配boot_command_line()和command_line的内存。

上述代码中比较重要的是，从启动加载项传递来的命令行（untouched command line）保存到saved_command_line变量，未能分析的参数信息（touched command line）通过parse_cmdline()函数（12.2节）保存到static_command_line。

16.3 初始化 per-cpu 数据——setup_per_cpu_areas()

setup_per_cpu_areas()是为了对内核的内存管理（memory management）进行初始化而调用的函数之一。只在SMP系统中调用，UP（Uni-Processor，单处理器）中不执行任何操作。setup_per_cpu_areas()函数为SMP的每个处理器生成per-cpu数据。各处理器具有的per-cpu空间通过初始分配器（bootmem allocator）动态接受分配，由属于__init节区的.data.percpu节区内容填充。

per-cpu数据按照不同CPU类别使用，以将性能低下引发的缓存一致性（cache coherency）问题减到最小。per-cpu数据由各CPU独立使用，即使不锁（lock）也可访问，十分有效。

:: 代码16-2 init/main.c的setup_per_cpu_areas()

```
unsigned long __per_cpu_offset[NR_CPUS] __read_mostly;                    ❶
...
static void __init setup_per_cpu_areas(void)
{
    ...
    unsigned long nr_possible_cpus = num_possible_cpus();                 ❷
    size = ALIGN(PERCPU_ENOUGH_ROOM, PAGE_SIZE);                          ❸
    /*按CPU数量分配*/
    ptr = alloc_bootmem_pages(size * nr_possible_cpus);                   ❹
    for_each_possible_cpu(i) {                                            ❺
        __per_cpu_offset[i] = ptr - __per_cpu_start;
        memcpy(ptr, __per_cpu_start, __per_cpu_end - __per_cpu_start);
        ptr += size;
    }
}
```

代码段❶中出现__read_mostly宏，观察Linux内核代码时会经常遇到。通过__read_mostly宏，程序员向编译器通知“该数据不经常修正，大部分只执行读取运算”。

考虑SMP的代码中经常见到，编译器将声明为__read_mostly的数据放入二进制文件的.data.read_mostly节区，使核心（core）能够对该区域中的数据副本分别执行缓存。只将读取为主的数据放入相同缓存线（cacheline），防止SMP内频繁发生缓存线弹跳（cacheline bouncing）问题。

在代码段❷中，通过num_possible_cpus()函数获得系统中可安装的最大CPU数，即SMP系统内可运行的最大核心数。

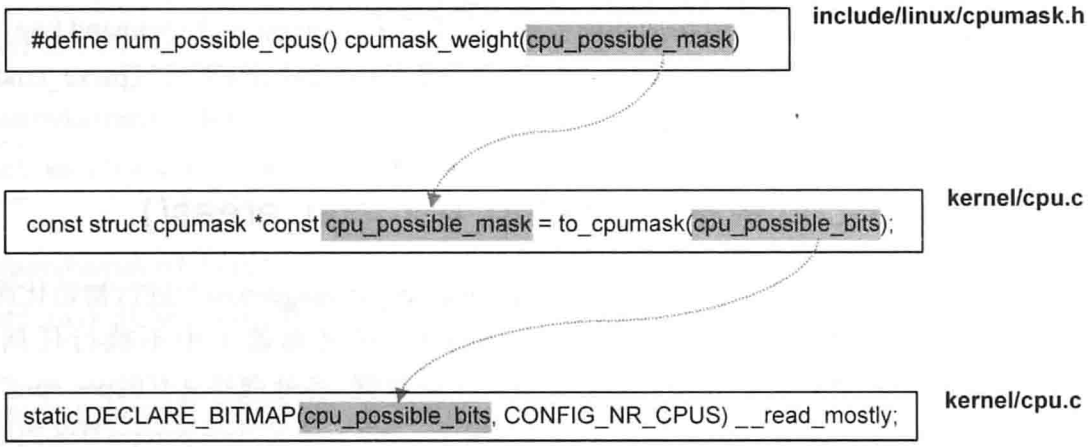


图16-2　num_possible_cpus()

请观察图16-2。可在系统内运行的CPU数是通过cpu_possible_mask()全局位图变量进行管理的。num_possible_cpus()利用cpumask_weight()函数返回位图内存在的1的个数。

在代码段❸中求出per-cpu区域的大小。请观察图16-3。

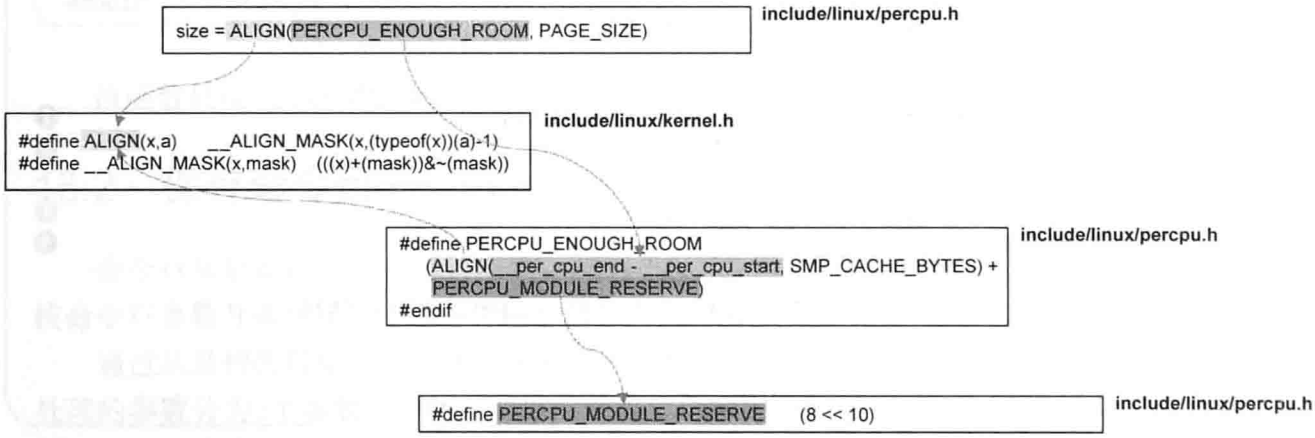


图16-3　ALIGN()

per-cpu区域是个值，它将.data.percpu区域（保存per-cpu数据）的大小和声明per-cpu数据的空间（PERCPU_MODULE_RESERVE:8KB）按页的大小（PAGE_SIZE:4KB）进行排列。

在代码段❹中，根据系统内存在的CPU数大小，利用初始分配器动态分配代码段❸中求出的per-cpu区域。

补充知识点　.data.percpu区域和DEFINE_PER_CPU()宏

如代码16-2所示，per-cpu数据位于.data.percpu节区。内核使用DEFINE_PER_CPU()宏声明per-cpu数据。

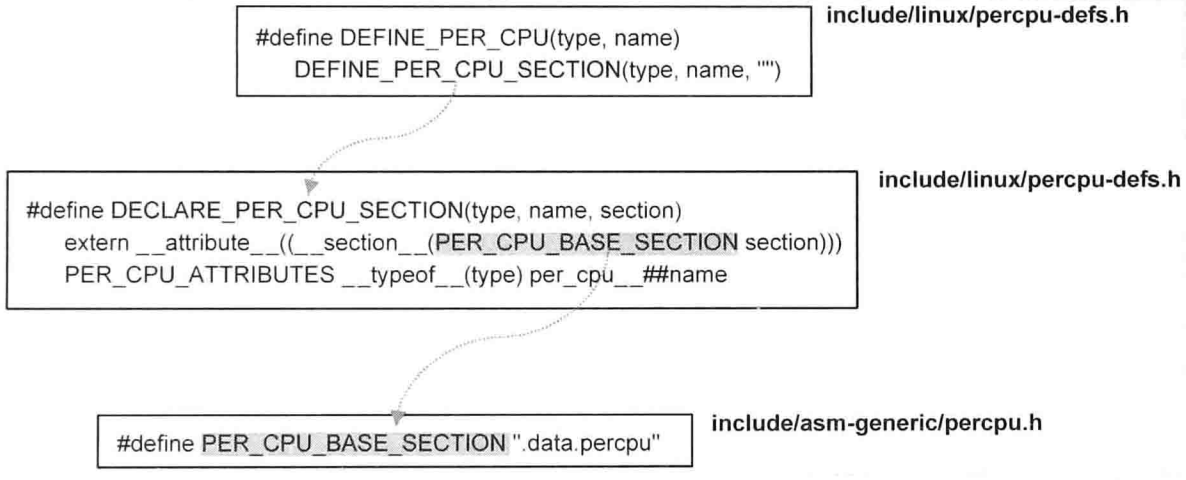


图16-4 DEFINE_PER_CPU()宏

从上图可以看到，利用DEFINE_PER_CPU()宏将type型数据结构声明为name名，并置于.data.percpu节区。此外，代码16-2代码段❸中提过的PERCPU_MODULE_RESERVE宏所指的8KB区域是保存内核代码内存在的所有DEFINE_PER_CPU宏代码的空间。

在代码段❺中，将.data.percpu节区的起始地址（__per_cpu_start）和动态分配的各处理器的per-cpu区域间的偏移量保存到__per_cpu_offset[]。只用求出的偏移量和__per_cpu_start可访问各处理器的per-cpu区域。然后，将保存在.data.percpu区域中的per-cpu数据复制到内存中的per-cpu区域。

16.4 求 CPU 个数——setup_nr_cpu_ids()

在setup_nr_cpu_ids()函数中，将系统内存在的CPU数保存到nr_cpu_ids全局变量。UP时其值为1，但SMP时具有2~32的值，在64位系统中具有2~64的值。

:: 代码16-3 init/main.c的setup_nr_cpu_ids()

```
int nr_cpu_ids __read_mostly = NR_CPUS;                                    ❶

static void __init setup_nr_cpu_ids(void)
{
    nr_cpu_ids = find_last_bit(cpumask_bits(cpu_possible_mask),NR_CPUS) + 1;   ❷
}
```

代码段❶中，nr_cpu_ids全局变量被声明为__read_mostly属性。nr_cpu_ids具有当前系统能具备的CPU数的信息。默认值为NR_CPUS值，NR_CPUS是编译时用户可设置的常量值。

NR_CPUS值并非当前系统内存在的CPU数的值，而是Linux内核能支持的CPU数的最大值。UP中是1，SMP中具有2~32的值。在64位内核中具有2~64的值。

在代码段❷中，利用cpu_possible_mask位图求出可存在的CPU数，并更新nr_cpu_ids值。cpu_possible_mask被全局定义，如图16-5所示。

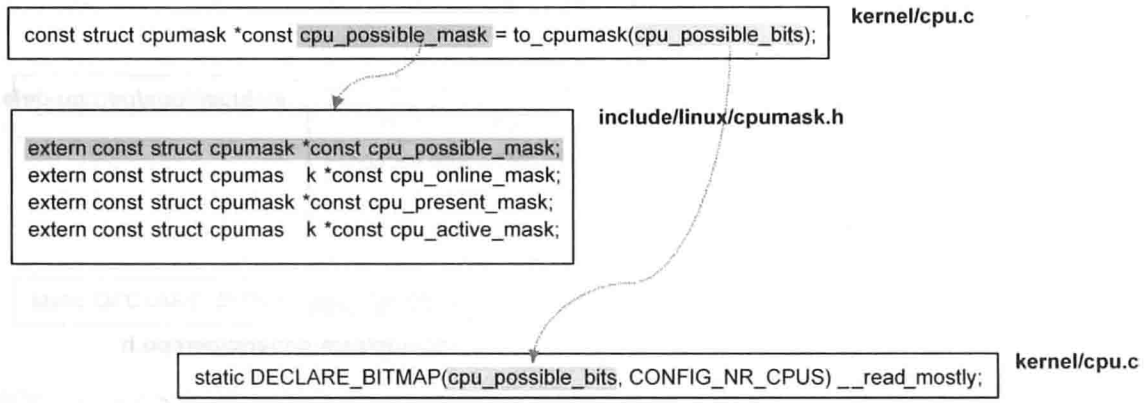


图16-5　cpu_possible_mask()

SMP系统中，Linux内核如表16-1所示，将CPU分为4种状态。这4种状态正如图16-5所示，被分别全局定义。

表16-1　CPU拓扑信息（topology information）

CPU拓扑信息	含　义
cpu_possible_mask	可安装的最多CPU的位图
cpu_present_mask	系统内安装的CPU的位图
cpu_online_mask	系统内安装的CPU中，正在使用的CPU的位图
cpu_active_mask	处于联机状态且可移动（migration）的CPU的位图
※ 若不支持CPU的热插拔（hotplug），则类似possible == present和active == online	

cpu_possible_mask是NR_CPUS大小的位图，具有可处于联机状态的CPU信息。利用find_last_bit()函数将可处于联机状态的CPU总数保存到nr_cpu_ids全局变量。

16.5　注册SMP上的启动进程——smp_prepare_boot_cpu()

SMP中存在多个CPU，启动Linux内核的只有1个。在smp_prepare_boot_cpu()中，为对应于当前启动的CPU的per-cpu（.data.percpu节区）区域注册要启动的进程。

:: 代码16-4　arch/arm/kernel/smp.c的smp_prepare_boot_cpu()

```
void __initsmp_prepare_boot_cpu(void)
{
    /*获得正在执行当前线程的cpu号*/
    unsigned intcpu = smp_processor_id();                    ❶

    per_cpu(cpu_data, cpu).idle = current;                   ❷
}
```

通过代码段❶获得正在执行当前线程的CPU号。在代码段❷中可以看到per_cpu()函数。通过上面的per_cpu（cpu_data、cpu），在.data.percpu节区内获得对应于执行当前线程的CPU号的cpu_data的指针。

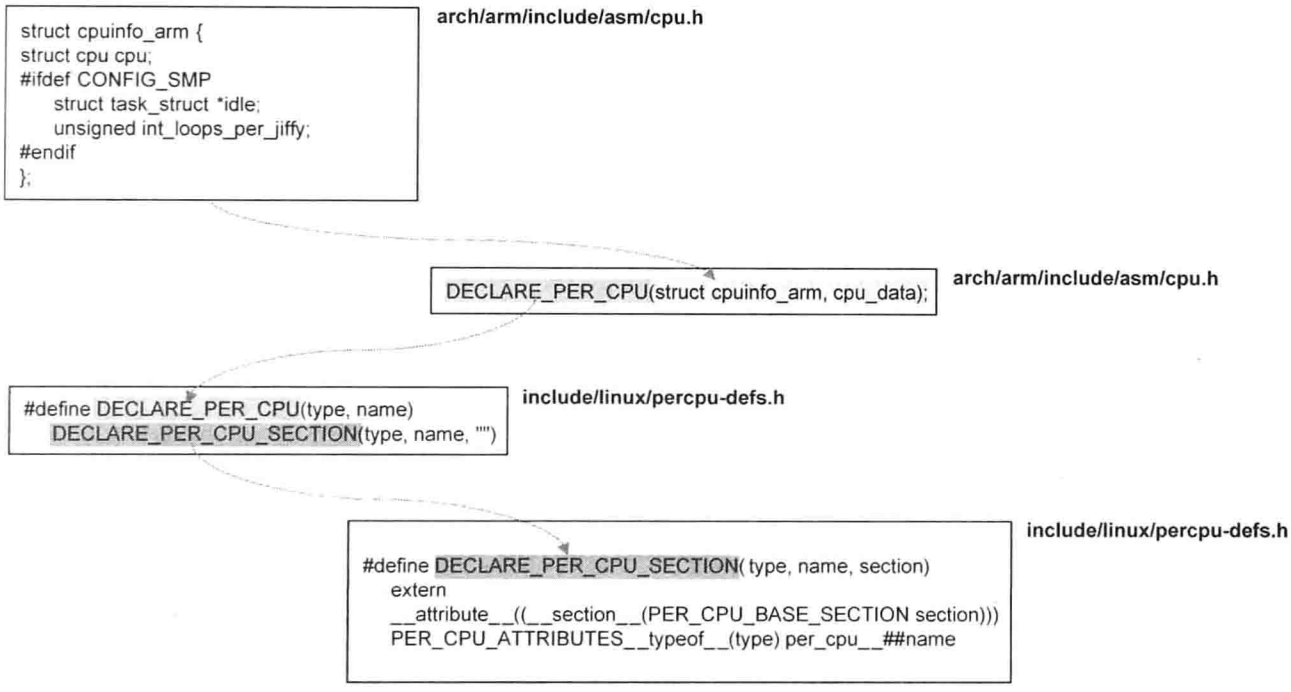


图16-6 cpu_data

通过图16-6可知，cpu_data是struct cpuinfo_arm结构体形式的全局变量，编译时位于.data.percpu节区。

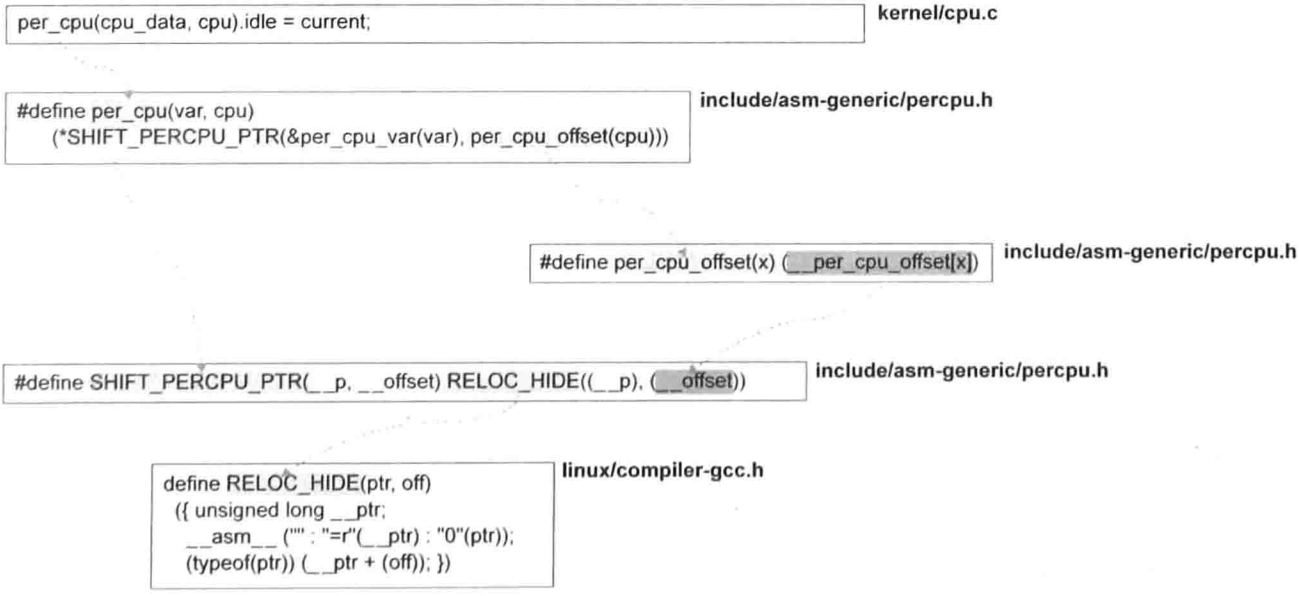


图16-7 per_cpu()

接下来观察图16-7。在per_cpu()中，将通过smp_processor_id()获得的cpuid用作.data.percpu内各CPU中的cpu_data的索引值。通过这种方式在当前CPU的per-cpu数据中访问cpu_data结构体，将当前任务的地址值保存到idle域。

16.6　初始化数据结构以使用调度程序——sched_init()

Linux内核的调度程序拥有如下多种数据结构以执行并管理进程。

- 集合调度时使用的task_group结构体
- 向进程实时分配CPU资源的rt_bandwidth结构体
- 显示CFS调度程序调度单位的sched_entity结构体
- 用于CPU就绪队列的runqueue结构体
- 实时调度进程时做出决策的root_domain结构体

sched_init()是初始化这些数据结构的函数。由于其不按顺序进行初始化，中间常出现混淆状况，所以很难理解。下面按照代码执行顺序查看。

16.6.1　为集合调度中使用的task_group的sched_entity结构体和runqueue结构体分配内存

task_group结构体是调度程序支持集合调度时使用的数据结构。内核编译时默认生成，具有默认提供的init_task_group，使用（通过user id集合进程的）CONFIG_USER_SCHED时另外提供root_task_group。激活集合调度之后，所有进程将属于1个task_group。此时，启动中生成的所有进程将属于init_task_group。

使用CONFIG_USER_SCHED时，root_task_group以后会被设置为init_task_group的父进程，不使用时按如下所示定义为init_task_group。

:: kernel/sched.c

```
#if CONFIG_USER_SCHED
#else
#define root_task_group init_task_group
#endif
```



补充知识点　调度程序相关内核设置

进入内核的设置画面后，可查看多种内核选项。其中，对集合进程的方法和集合对象的设置如下所示。

(1) 对集合进程方法的设置

CONFIG_CGROUP_SCHED：集合方法，使用cgroup。

CONFIG_USER_SCHED：集合方法，使用进程的UID。

(2) 对集合调度对象的设置

CONFIG_FAIR_GROUP_SCHED：激活对一般进程的集合调度。

CONFIG_RT_GROUP_SCHED：激活对实时进程的集合调度。

此处为task_group分配内存。准确地说，由task_group的成员sched_entity结构体变量se、rt_se和rq结构体变量cfs_rq、rt_rq分配指针。请观察函数。

:: 代码16-5 kernel/sched.c的sched_init() 1/4

```
void __init sched_init(void)
{
    ...
#ifdef CONFIG_FAIR_GROUP_SCHED                                    ❶
    alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
#ifdef CONFIG_RT_GROUP_SCHED
    alloc_size += 2 * nr_cpu_ids * sizeof(void **);
#endif
#ifdef CONFIG_USER_SCHED
    alloc_size *= 2;
#endif
    ...
    if (alloc_size) {                                             ❷
        ptr = (unsigned long)alloc_bootmem(alloc_size);
#ifdef CONFIG_FAIR_GROUP_SCHED                                    ❸
    init_task_group.se = (struct sched_entity **)ptr;
    ptr += nr_cpu_ids * sizeof(void **);
    init_task_group.cfs_rq = (struct cfs_rq **)ptr;
    ptr += nr_cpu_ids * sizeof(void **);
#ifdef CONFIG_USER_SCHED                                          ❹
    ...
#endif /* CONFIG_USER_SCHED */
#endif /* CONFIG_FAIR_GROUP_SCHED */

#ifdef CONFIG_RT_GROUP_SCHED                                      ❺
    init_task_group.rt_se = (struct sched_rt_entity **)ptr;
    ptr += nr_cpu_ids * sizeof(void **);
    init_task_group.rt_rq = (struct rt_rq **)ptr;
    ptr += nr_cpu_ids * sizeof(void **);
#ifdef CONFIG_USER_SCHED                                          ❻
    ...
#endif /* CONFIG_USER_SCHED */
#endif /* CONFIG_RT_GROUP_SCHED */
```

在代码段❶中，计算要为task_group的就绪队列和sched_entity分配的内存大小。要分配的内存大小根据添加的内核设置有所不同。

补充知识点　显示调度单位的sched_entity结构体

CFS调度程序与直接操作各进程的现有调度程序不同，它将schedulable entity抽象对象用作调度单位。导入这一概念后，不仅能够执行单一进程，也可以对任务集合执行调度。记述schedulable entity的结构体就是sched_entity。

CFS调度程序使用的红黑树（Red-Black Tree）中，该结构体用1个节点表示。进程或任务集合将sched_entity直接内置到记述自己的数据结构，可按红黑树内排列的状态罗列。

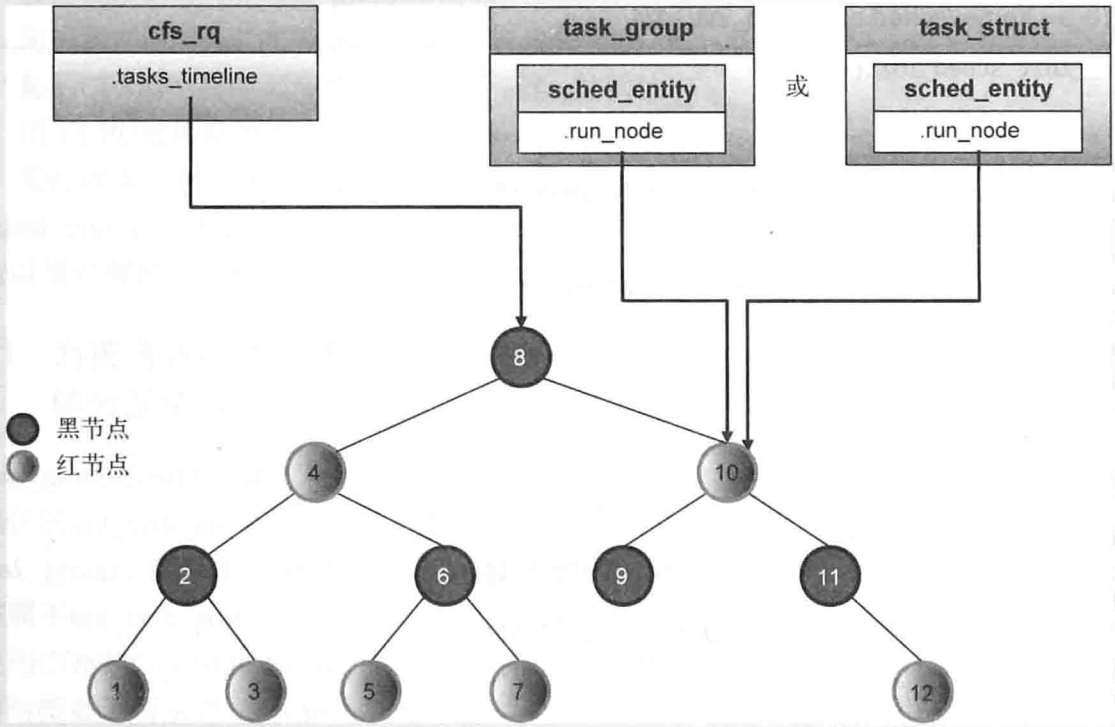


图16-8　红黑树和sched_entity的关系

调度程序不直接处理各进程，而是通过sched_entity处理抽象对象，以实现集合调度。

(1) 使用CONFIG_FAIR_GROUP_SCHED时

对一般进程使用集合调度。使用该内核设置时，编译时生成init_task_group。并加上init_task_group使用的指向用于一般进程的sched_entity、cfs runqueue[①]的指针大小。每个CPU均有1个，所以与CPU总数（nr_cpu_ids）相乘后再相加。

(2) 使用CONFIG_RT_GROUP_SCHED时

对实时进程使用集合调度。并加上init_task_group使用的指向用于实时进程的sched_entity、rt runqueue的指针大小。每个CPU均有1个，所以与CPU总数（nr_cpu_ids）相乘后再相加。

① task_group的se、rt_se、rq、rt_rq是指针数组，拥有系统全体CPU数大小的指针。

(3) 激活CONFIG_USER_SCHED时

额外使用名为root_task_group的task_group。需要添加分配前为init_task_group分配的内存大小，所以共需要2倍内存。

在代码段❷中使用bootmem分配器，以上面决定的大小分配内存。

(4) 使用CONFIG_FAIR_GROUP_SCHED时

在代码段❸中，为init_task_group的se、cfs_rq分别分配全体CPU数大小的指针。然后，给ptr加上分配的大小，使ptr能够指向未分配区域的起始地址。

(5) 使用CONFIG_RT_GROUP_SCHED时

在代码段❺中，为init_task_group的rt_se、rt_rq分别分配全体CPU数大小的指针。然后，给ptr加上分配的大小，使ptr能够指向未分配区域的起始地址。

(6) 使用CONFIG_USER_SCHED时

在代码段❹、❻中也额外为root_task_group分配内存，与给init_task_group分配时相同。

16.6.2 初始化root_domain、rt_bandwidth、task_group相关数据结构

本节主要对系统默认的root domain——def_root_domain、记述(可使用实时任务集合的)CPU使用量的rt_bandwidth进行初始化，并设置task_group间的链接信息。

:: 代码16-6 kernel/sched.c的sched_init() 2/4

```
#ifdef CONFIG_SMP
    init_defrootdomain();                                              ❶
#endif

    init_rt_bandwidth(&def_rt_bandwidth,                               ❷
                      global_rt_period(), global_rt_runtime());

#ifdef CONFIG_RT_GROUP_SCHED
    init_rt_bandwidth(&init_task_group.rt_bandwidth,                   ❸
                      global_rt_period(), global_rt_runtime());
#ifdef CONFIG_USER_SCHED
    init_rt_bandwidth(&root_task_group.rt_bandwidth,                   ❹
                      global_rt_period(), RUNTIME_INF);

#endif /* CONFIG_USER_SCHED */
#endif /* CONFIG_RT_GROUP_SCHED */

#ifdef CONFIG_GROUP_SCHED
    list_add(&init_task_group.list, &task_groups);                     ❺
    INIT_LIST_HEAD(&init_task_group.children);

#ifdef CONFIG_USER_SCHED
    INIT_LIST_HEAD(&root_task_group.children);                         ❻
    init_task_group.parent = &root_task_group;
```

```
    list_add(&init_task_group.siblings, &root_task_group.children);
#endif /* CONFIG_USER_SCHED */
#endif /* CONFIG_GROUP_SCHED */
```

在代码段❶中，调用init_rootdomain()函数初始化默认root_domain——def_root_domain。在ARM中初始化def_root_domain->cpupri，将所有CPU状态设置为CPUPRI_INVALID。cpupri是为了有效决定任务是否移动而引入的概念，此处不进行详细讲解。最后将def_root_domain的refcount设置为1，记录当前有1处正在执行浏览。



补充知识点　用于实时进程调度决策的root_domain

实时调度程序在进行与调度相关的决策时使用几种全局变量。例如，有时需要查找可运行的实时务是否多于CPU个数。此时会查看所有就绪队列状态，需要用到的变量就是rt_overloaded。调度程序每次调用schedule()时浏览该变量，每次增加全体CPU个数时，要浏览该变量的工作给CPU带来很多负载。

为了改善这种系统扩展性的大问题，出现了新概念root domain。可通过组建排他性的cpuset（exclusive cpuset）制作root domain。cpuset将由进程集合使用的cpu进行隔离，使其他cpuset中无法使用，这种cpuset称为排他性cpuset。

root_domain通过root_domain结构体记述，并将已使用的全局变量的访问范围缩小到个别root_domain。访问范围缩小后，对该变量的竞争状态也随之而降低，扩展性也得到提高。新的root_domain生成cpuset时，以属于cpuset的cpu的信息为基础。默认提供的def_root_domain以所有cpu的信息为基础生成。

最终，导入root_domain后，调度程序在特定root_domain内做出与调度相关的决策，从而缓解竞争状态。

在代码段❷中，用默认设置的调度周期和周期内可使用的时间对rt_bandwidth[①]结构体变量def_rt_bandwidth进行初始化。激活CONFIG_RT_GROUP_SCHED后，在代码段❸中也初始化init_task_group.rt_bandwidth。

激活CONFIG_USER_SCHED后，在代码段❹中将root_task_group.rt_bandwidth周期内可使用的时间设置为无限大。root_task_group是包含系统中运行的所有进程的任务集合，所以为其分配无限时间。

① 内核2.6.26中添加了与实时集合调度相关的几项功能，rt_bandwidth（实时带宽）就是其中之一，它让具有竞争关系的实时进程集合能够均衡使用CPU。现有的SCHED_FIFO是实时调度，具有相同优先顺序的进程相互竞争时，由占有当前CPU的进程获得优先权。换言之，当前执行的进程让出CPU资源前，即使是具有相同优先顺序的进程也需要等待。但是，添加这一概念后，如果某些SCHED_FIFO任务超过rt_bandwidth，即使是SCHED_FIFO任务也无法继续占有CPU。rt_bandwidth由集合允许的CPU时间和实际使用的CPU时间组成。

使用集合调度时（CONFIG_GROUP_SCHED），全局变量task_groups显示系统内所有任务集合的列表（list）。代码段❺中将init_task_group添加到该列表。使用CONFIG_USER_SCHED时，在代码段❻中将root_task_group指定为init_task_group的父进程，设置默认task_group数据结构的关系。

16.6.3 初始化系统上所有可用CPU的就绪队列

:: 代码16-7 kernel/sched.c的sched_init() 3/4

```
for_each_possible_cpu(i) {                                          ❶
    ...
    rq = cpu_rq(i);
    ...
    rq->nr_running = 0;
    init_cfs_rq(&rq->cfs, rq);
    init_rt_rq(&rq->rt, rq);

#ifdef CONFIG_FAIR_GROUP_SCHED                                      ❷
    init_task_group.shares = init_task_group_load;
    INIT_LIST_HEAD(&rq->leaf_cfs_rq_list);
#ifdef CONFIG_CGROUP_SCHED
    init_tg_cfs_entry(&init_task_group, &rq->cfs, NULL, i, 1, NULL);
#elif defined CONFIG_USER_SCHED
    root_task_group.shares = NICE_0_LOAD;
    init_tg_cfs_entry(&root_task_group, &rq->cfs, NULL, i, 0, NULL);
    init_tg_cfs_entry(&init_task_group, &per_cpu(init_cfs_rq, i),
                      &per_cpu(init_sched_entity, i),
                      i, 1, root_task_group.se[i]);
#endif
#endif /* CONFIG_FAIR_GROUP_SCHED */

    rq->rt.rt_runtime = def_rt_bandwidth.rt_runtime;
#ifdef CONFIG_RT_GROUP_SCHED                                        ❸
    INIT_LIST_HEAD(&rq->leaf_rt_rq_list);
#ifdef CONFIG_CGROUP_SCHED
    init_tg_rt_entry(&init_task_group, &rq->rt, NULL, i, 1, NULL);
#elif defined CONFIG_USER_SCHED
    init_tg_rt_entry(&root_task_group, &rq->rt, NULL, i, 0, NULL);
    init_tg_rt_entry(&init_task_group, &per_cpu(init_rt_rq, i),
                     &per_cpu(init_sched_rt_entity, i),
                     i, 1, root_task_group.rt_se[i]);
#endif
#endif

    for (j = 0; j < CPU_LOAD_IDX_MAX; j++)                          ❹
        rq->cpu_load[j] = 0;
#ifdef CONFIG_SMP                                                   ❺
    rq->sd = NULL;
    rq->rd = NULL;
    rq->active_balance = 0;
```

```
        rq->next_balance = jiffies;
        rq->push_cpu = 0;
        rq->cpu = i;
        rq->online = 0;
        rq->migration_thread = NULL;
        INIT_LIST_HEAD(&rq->migration_queue);
        rq_attach_root(rq, &def_root_domain);
    #endif
        init_rq_hrtick(rq);                                          ❻

        atomic_set(&rq->nr_iowait, 0);
    }
```

在代码段❶系统内的CPU中，对记录在cpu_possible_map的CPU执行循环，同时初始化就绪队列。先将插入就绪队列的进程数初始化为0，并将显示子就绪队列（cfs_rq、rt_rq）和就绪队列关系的成员变量也初始化。在代码段❷中初始化与init_task_group的task_group相关的就绪队列成员。使用的内核设置不同，初始化的对象也不同。

使用CONFIG_CGROUP_SCHED时

设置shares值，它决定init_task_group能够使用的CPU带宽，将init_task_group设置为rq->cfs的task_group。

使用CONFIG_USER_SCHED时

设置shares值，它决定root_task_group能够使用的CPU带宽。调用init_tg_rt_entry()，将root_task_group设置为rq->cfs的task_group。默认任务集合使用的各就绪队列的cfs_rq和sched_entity另行存在，最后会额外设置这2个数据结构和init_task_group的连接信息。

以后分配CPU带宽时，用此处的task_group的shares成员变量求出比例。

上述代码段初始化cfs就绪队列后，代码段❸将实时初始化就绪队列。在代码段❹中，对保存就绪队列最近CPU负载值的cpu_load成员变量进行初始化。

在代码段❺中，初始化就绪队列成员变量中与负载均衡相关的成员变量，并更新就绪队列的root_domain。在代码段❻中初始化与调度程序相关的计时器，注册回调函数。

16.6.4　初始化当前任务的调度相关值与注册针对负载均衡的中断处理器

:: 代码16-8　kernel/sched.c的sched_init() 4/4

```
        set_load_weight(&init_task);                                 ❶
    #ifdef CONFIG_PREEMPT_NOTIFIERS
        INIT_HLIST_HEAD(&init_task.preempt_notifiers);
    #endif
    #ifdef CONFIG_SMP
        open_softirq(SCHED_SOFTIRQ, run_rebalance_domains);          ❷
    #endif

    #ifdef CONFIG_RT_MUTEXES
        plist_head_init(&init_task.pi_waiters, &init_task.pi_lock);
```

```
#endif

    atomic_inc(&init_mm.mm_count);
    enter_lazy_tlb(&init_mm, current);
    init_idle(current, smp_processor_id());                    ❸
    current->sched_class = &fair_sched_class;
    ...
    scheduler_running = 1;
}
```

在代码段❶中设置init_task的负载（load weight）值（weight、inv_weight）。该值根据进程使用调度策略的不同而不同。在代码段❷中发生SCHED_SOFTIRQ中断时，调用run_rebalance_domains()函数，注册处理器使负载均衡。

该中断是由通过计时器函数最终调用的trigger_load_balance()引发的。负载均衡相关内容请参考30.1节的sched_init_smp()。

在代码段❸中用idle制作当前进程（current）。但紧接着将该sched_class设置为fair_sched_class，启动时就能显示为一般进程。最后，将scheduler_running设置为1，记录“已初始化调度程序”。

16.7 允许内核抢占和阻止抢占——preempt_enable()/preempt_disable()

内核2.4版的调度策略对于用户进程是抢占型，对内核代码是非抢占型。但从内核2.6版开始也允许对内核代码执行抢占，即支持抢占型内核，并包含于内核设置选项。为支持这种抢占型内核而添加了preempt_enable()和preempt_disable()，它们决定是否允许抢占。详情如代码16-9所示。

:: 代码16-9 include/linux/preempt.h的preempt_enable()和preempt_disable()

```
#ifdef CONFIG_PREEMPT                                          ❶

#define preempt_disable() \                                    ❷
do { \
    inc_preempt_count(); \
    barrier(); \
} while (0)

#define preempt_enable() \                                     ❸
do { \
    preempt_enable_no_resched(); \
    barrier(); \
    preempt_check_resched(); \
} while (0)

#else

#define preempt_disable()               do { } while (0)       ❹
#define preempt_enable()                do { } while (0)

#endif
```

代码段❶的定义决定了是否在内核代码中允许抢占。若已设置抢占，则运行preempt_enable()和preempt_disable()宏；否则如代码段❹所示，该宏不执行任何操作。

代码段❷的preempt_disable()宏禁用内核抢占，使抢占无法执行。例如，调度过程中内核不能被其他任务抢占，这种情况相当于需要调用preempt_disable()。为了在preempt_disable()中禁用抢占，调用inc_preempt_count()并依次调用以下宏函数。

:: include/linux/preempt.h

```
#define inc_preempt_count() add_preempt_count(1)
#define add_preempt_count(val) do { preempt_count() += (val); } while (0)
#define preempt_count() (current_thread_info()->preempt_count)
```

inc_preempt_count()宏将1作为参数传递给add_preempt_count()。add_preempt_count()宏将具有当前进程信息的thread_info结构体的preempt_count变量值加1，表示内核线程已被抢占，其他进程无法执行抢占，从而禁用内核抢占。

barrier()防止通过编译器混入汇编指令，确保内存结果值不被意外更换，相当于内存屏障。

与preempt_disable()相反，代码段❸的preempt_enable()宏激活内核抢占，以允许内核代码的抢占。

第17章 构建借用内存的后台

在Linux中管理内存时，分几个阶段进行抽象化，用数据结构管理。先用节点集合管理内存，然后用zone的集合管理节点，再用页的集合管理zone。此时使用的数据结构分别为pglist_data、zone、page结构体。pglist_data结构体定义为pg_data_t类型。

17.1 在 build_all_zonelist()中操作的一些数据结构

在init_bootmem()函数中，将各节点的结构体pglist_data的大部分进行了初始化。本章将初始化pglist_data的node_zonelist[]成员，如图17-1所示。

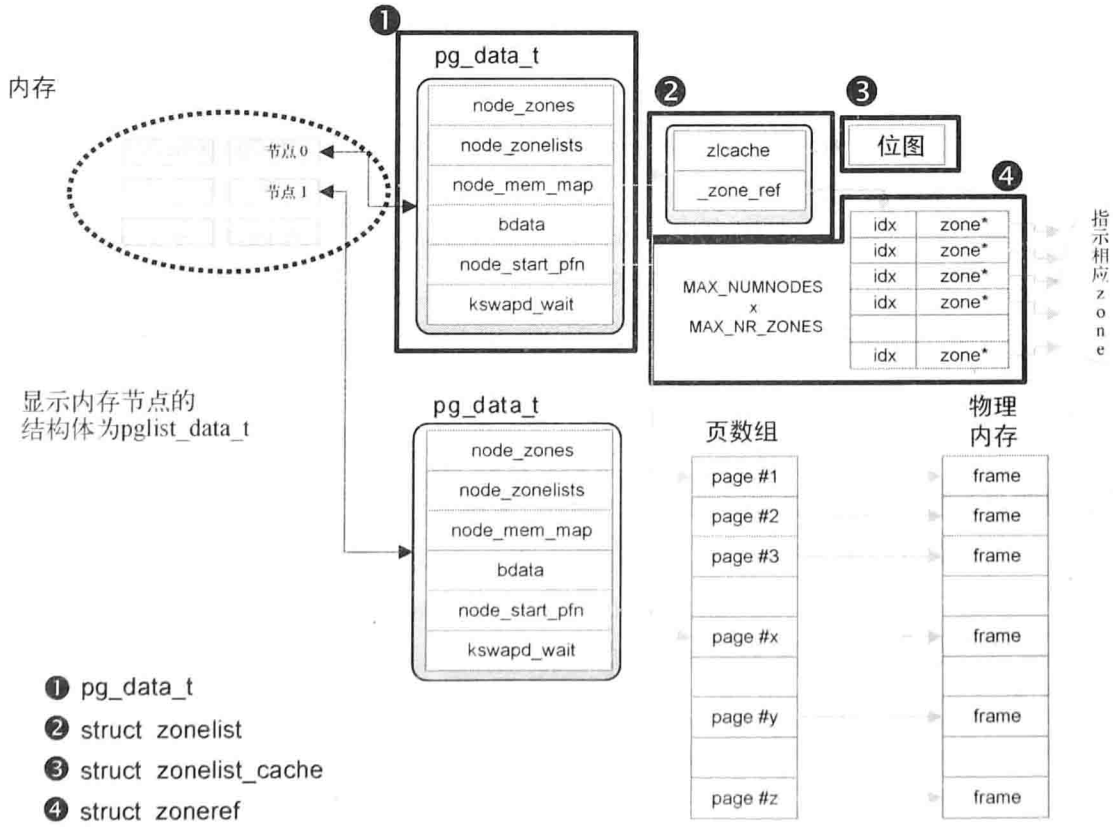


图17-1 通过build_all_zonelist()函数设置的各数据结构

node_zonelist[]数组是zone的备份列表（fallback list），在该zone中无可用内存时，用它查找其他节点的类型相同的zone以分配内存。换言之，分配内存时，如果该zone不再有可用内存，则用该列表保存优先顺序（应从哪个节点、哪个zone获取）。

pglist_data结构体中node_zone_list[]的类型的struct zonelist如下所示。

:: include/linux/mmzone.h

```
#define MAX_ZONELISTS 2

typedef struct pglist_data {

    ...
    struct zonelist node_zonelists[MAX_ZONELISTS];
    ...
} pg_data_t;


struct zonelist {
    struct zonelist_cache *zlcache_ptr;          /*NULL或&zlcache*/
    struct zoneref _zonerefs[MAX_ZONES_PER_ZONELIST + 1];
#ifdef CONFIG_NUMA
    struct zonelist_cache zlcache;               /*选项信息*/
#endif
};
```

node_zonelist[]数组具有2个元素，node_zonelist[0]保存备份列表，node_zonelist[1]构建的是相应节点的zone列表，而非备份列表。

备份列表的核心是_zonerefs[]数组。_zonerefs[]数组的大小为MAX_ZONES_PER_ZONELIST，最大节点数和节点可拥有的zone数的定义如下所示。

:: include/linux/mmzone.h

```
#define MAX_ZONES_PER_ZONELIST (MAX_NUMNODES * MAX_NR_ZONES)
```

各节点需要具有属于其他节点的zone列表，因此，不同节点具有MAX_ZONES_PER_ZONELIST个元素。_zonerefs[]数组的类型struct zoneref的定义如下所示。

:: include/linux/mmzone.h

```
struct zoneref {
    struct zone *zone;  /*实际的zone*/
    int zone_idx;       /*_zonerefs[]数组的索引*/
};
```

以上为本章需要填充的基本数据结构。下面正式分析build_all_zonelist()函数。

17.2 查看 build_all_zonelist()结构

代码17-1是本节要分析的build_all_zonelist()函数。

:: 代码17-1 mm/page_alloc.c的build_all_zonelists()

```
void build_all_zonelists(void)
{
    set_zonelist_order();                                          ❶

    if (system_state == SYSTEM_BOOTING) {                          ❷
        __build_all_zonelists(NULL);                               ❸
        mminit_verify_zonelist();                                  ❹
        cpuset_init_current_mems_allowed();                        ❺
    } else {
        stop_machine(__build_all_zonelists, NULL, NULL);           ❻
    }

    vm_total_pages = nr_free_pagecache_pages();                    ❼

    if (vm_total_pages < (pageblock_nr_pages * MIGRATE_TYPES))     ❽
        page_group_by_mobility_disabled = 1;
    else
        page_group_by_mobility_disabled = 0;

    printk("Built %i zonelists in %s order, mobility grouping %s.  "   ❾
            "Total pages: %ld\n",
            num_online_nodes(),
            zonelist_order_name[current_zonelist_order],
            page_group_by_mobility_disabled ? "off" : "on",
            vm_total_pages);
#ifdef CONFIG_NUMA
    printk("Policy zone: %s\n", zone_names[policy_zone]);
#endif
}
```

在代码段❶中调用set_zonelist_order()函数决定zone排列方式：按相同区域排列，还是以节点为基准排列（17.3节）。

代码段❷中，不同系统状态调用的函数有所不同。系统状态共分6种，其定义如下所示。

:: include/linux/kernel.h

```
extern enum system_states {
    SYSTEM_BOOTING,
    SYSTEM_RUNNING,
    SYSTEM_HALT,
    SYSTEM_POWER_OFF,
    SYSTEM_RESTART,
    SYSTEM_SUSPEND_DISK,
} system_state;
```

系统状态为启动阶段（SYSTEM_BOOTING）时，调用代码段❸至代码段❺。

在代码段❸中，系统处于启动阶段时调用__build_all_zonelist()函数，该函数构建备份列表（build_zonelists()函数），并为快速访问备份列表构建位图缓存（build_zonelist_cache()函数）（17.4节）。

在代码段❹中调用mminit_verify_zonelist()函数，输出系统内所有节点的备份列表信息（17.5节）。

在代码段❺中调用cpuset_init_current_mems_allowed()函数，该函数在当前任务current的mems_allowed位图中将系统内所有节点设置为1。mems_allowed位图决定处理当前任务中发生的页分配请求的节点。

在代码段❻中调用stop_machine()函数。此时，在非系统启动阶段的状态下，即SYSTEM_RUNNING、SYSTEM_HALT、SYSTEM_POWER_OFF、SYSTEM_RESTART、SYSTEM_SUSPENDED_DISK状态下调用了build_all_zonelists()函数，系统结束时没有理由调用build_all_zonelists()函数。因此，要理解这是在SYSTEM_RUNNING状态下调用了build_all_zonelists()函数。

stop_machine()函数让系统的所有CPU执行stop_machine()函数的第一个参数build_all_zonelists()函数。执行17.4节的__build_all_zonelists()函数时，要先关闭中断和抢占。因为系统处于运行状态，所以构建新的zone列表时不能有访问它的用户。

在代码段❼中调用nr_free_pagecache_pages()函数，该函数求出可处理的空页数（17.7节）。

在代码段❽中，以通过代码段❼的nr_free_pagecache_pages()函数求出的值（vm_total_pages）和页移动性（page mobility）为基准，决定是否激活集合（grouping）（17.8节）。

代码段❽只将执行过的内容信息打印到控制台。打印的信息输出内容为联机节点（online node）数、zone列表顺序，是否根据移动性对页执行集合（grouping）、vm_total_pages、NUMA时输出policy zone。①

17.3　决定zone的列表方式——set_zonelist_order()

本节决定zonelist的顺序，先考虑由2个节点构成的内存。假设节点0的zone类型由ZONE_NORMAL和ZONE_DMA构成，节点1由ZONE_NORMAL构成，可将节点0的zonelist按如下所示用两种类型构建。

类型A：Node(0) ZONE_NORMAL→Node(0) ZONE_DMA →Node(1) ZONE_NORMAL

类型B：Node(0) ZONE_NORMAL→Node(1) ZONE_NORMAL →Node(0) ZONE_DMA

类型A时，如果节点0的ZONE_NORMAL中无法分配内存，就从节点0的ZONE_DMA开始分配。虽然类型A提供内存局部性（memory locality），但ZONE_DMA区域的内存比较小，因此会发生ZONE_DMA的OOM（Out Of Memory，内存溢出）问题。

类型B时，如果节点0的ZONE_NORMAL中无法分配内存，就从节点1的ZONE_NORMAL开始分配。虽然不提供内存局部性，但不会像A那样容易发生DMA的OOM问题。因此，类型A为节

① policy zone以下的zone不适用policy。

点顺序（node order，节点的优先顺序），类型B为zone顺序（zone order，zone的优先顺序）。

set_zonelist_order()函数决定用节点顺序构建还是用zone顺序构建，其定义如下所示。

:: mm/page_alloc.c

```
static void set_zonelist_order(void)
{
    if (user_zonelist_order == ZONELIST_ORDER_DEFAULT)
        current_zonelist_order = default_zonelist_order();
    else
        current_zonelist_order = user_zonelist_order;
}
```

该函数检查user_zonelist_order是否为ZONELIST_ORDER_DEFAULT，user_zonelist_order具有以下3个值之一。

:: mm/page_alloc.c

```
#define ZONELIST_ORDER_DEFAULT  0
#define ZONELIST_ORDER_NODE     1
#define ZONELIST_ORDER_ZONE     2
```

如果user_zonelist_order就是ZONELIST_ORDER_DEFAULT，则调用default_zonelist_order()函数，决定将zonelist顺序作为节点顺序还是zone顺序。代码17-2是default_zonelist_order()函数。如果DMA区域超过一定大小，default_zonelist_order()函数就默认成为ZONELIST_ORDER_NODE；反之，则由ZONELIST_ORDER_ZONE使用。

:: 代码17-2 mm/page_alloc.c的default_zonelist_order()

```
static int default_zonelist_order(void)
{
    ...

    for_each_online_node(nid) {
        for (zone_type = 0; zone_type < MAX_NR_ZONES; zone_type++) {
            z = &NODE_DATA(nid)->node_zones[zone_type];
            if (populated_zone(z)) {
                if (zone_type < ZONE_NORMAL)                          ❶
                    low_kmem_size += z->present_pages;
                    total_size += z->present_pages;
                }
            }
    }

    if (!low_kmem_size ||  /*DMA区域不存在时*/                          ❷
        low_kmem_size > total_size/2) /*DMA/DMA32区域较大时*/
        return ZONELIST_ORDER_NODE;

    average_size = total_size /                                       ❸
```

```
            (nodes_weight(node_states[N_HIGH_MEMORY]) + 1);

    for_each_online_node(nid) {                                          ❹
        low_kmem_size = 0;
        total_size = 0;
        for (zone_type = 0; zone_type < MAX_NR_ZONES; zone_type++) {
            z = &NODE_DATA(nid)->node_zones[zone_type];
            if (populated_zone(z)) {
                if (zone_type < ZONE_NORMAL)
                    low_kmem_size += z->present_pages;
                total_size += z->present_pages;
            }
        }
        if (low_kmem_size &&                                             ❺
            total_size > average_size &&
            low_kmem_size > total_size * 70/100)
            return ZONELIST_ORDER_NODE;
    }
    return ZONELIST_ORDER_ZONE;
}
```

在代码段❶的low_kmem_size中，保存相当于ZONE_NORMAL以下类型（ZONE_DMA、ZONE_DMA32）的实际内存大小，在total_size中保存整个系统实际内存的大小。

代码段❷中，如果对应于ZONE_DMA的内存超过全部内存的一半，则将zonelist的顺序设置为节点顺序（ZONELIST_ORDER_NODE）。代码段❸将全部内存的大小除以联机状态的节点数，求出各节点的平均内存大小（average_size）。

代码段❹和代码段❶相同。代码段❺中有对应于ZONE_DMA的内存，已选节点的内存大小比节点的平均内存更大，且对应于ZONE_DMA的内存占全体内存的70%以上时，将zonelist的顺序设置为节点顺序（ZONELIST_ORDER_NODE）。

但是，如果这些条件都失败了，就将zonelist的顺序设置为zone顺序（ZONELIST_ORDER_ZONE）。

最终，defalt_zonelist_order()函数根据DMA的大小变更其备份列表的顺序。DMA较小的时候，OOM的概率会变大。因此，要对各节点的zone列表执行循环，同时查找DMA区域。

此时，备份列表为zone列表则比较有效。反之，DMA较大时，访问其他节点DMA区域的概率较小，在相应节点的DMA中分配内存的可能性变大，故将备份列表设置为节点顺序。

17.4　构建备用列表和备用位图——__build_all_zonelist()

通过代码17-3分析__defalt_all_zonelist()函数。

:: 代码17-3　mm/page_alloc.c的__build_all_zonelists()

```
static int __build_all_zonelists(void *dummy)
{
    int nid;
```

```
    for_each_online_node(nid) {
        pg_data_t *pgdat = NODE_DATA(nid);

        build_zonelists(pgdat);                                    ❶

        build_zonelist_cache(pgdat);                               ❷
    }
    return 0;
}
```

调用代码段❶的build_zonelists()函数制作2个zonelist，在current_zonelist_order中设置所有节点的所有zone，以此在pgdat->zonelist[0]中构建备份列表。在pgdat->zonelist[1]中，只将pgdat所属节点的全部zone重新构建1次（17.4.1节）。

在代码段❷中调用build_zonelist_cache()函数，用位图管理备份列表的全部zone。如果zone的所有内存正在使用，则将对应于该zone的位图的位设置为1。这样就能通过位图快速找到可用zone（17.4.5节）。

17.4.1 build_zonelists()

代码17-4表示build_zonelists()函数。

:: 代码17-4 mm/page_alloc.c的build_zonelists()

```
static void build_zonelists(pg_data_t *pgdat)
{
    ...

    struct zonelist *zonelist;
    int order = current_zonelist_order;

    for (i = 0; i < MAX_ZONELISTS; i++) {                          ❶
        zonelist = pgdat->node_zonelists + i;
        zonelist->_zonerefs[0].zone = NULL;
        zonelist->_zonerefs[0].zone_idx = 0;
    }

    local_node = pgdat->node_id;          /*节点id*/
    load = num_online_nodes();            /*联机节点数*/
    prev_node = local_node;

    ...

    while ((node = find_next_best_node(local_node, &used_mask)) >= 0)  ❷
    {

        ...

        if (order == ZONELIST_ORDER_NODE)
            build_zonelists_in_node_order(pgdat, node);            ❸
```

```
        else
            node_order[j++] = node; /*保存节点顺序，j=节点数*/                    ❹
    }

    if (order == ZONELIST_ORDER_ZONE) {
        build_zonelists_in_zone_order(pgdat, j);                              ❺
    }

    build_thisnode_zonelists(pgdat);                                          ❻
}
```

代码段❶从当前节点的节点描述符pgdat访问struct zonelist结构体类型node_zonelists，并初始化成员变量zone和zone_idx。17.1节的struct zonelist和zonelist的成员struct zoneref的定义如下所示。

:: include/linux/mmzone.h

```
struct zonelist {
    struct zonelist_cache *zlcache_ptr;          /*NULL或&zlcache*/
    struct zoneref _zonerefs[MAX_ZONES_PER_ZONELIST + 1];
#ifdef CONFIG_NUMA
    struct zonelist_cache zlcache;               /*选项信息*/
#endif
};
```

:: include/linux/mmzone.h

```
struct zoneref {
    struct zone *zone;  /* Pointer to actual zone */
    int zone_idx;       /* zone_idx(zoneref->zone) */
};
```

在struct zoneref中，zone成员变量具有zonelist中的zone信息，而zone_idx则是zone成员变量在zonelist内的索引。

代码段❷调用find_next_best_node()函数，该函数为添加当前节点的备份列表，以当前节点为基准查找最佳节点。查找最佳节点需要以下几个标准。

(1) 以当前节点为基准距离最短的节点

(2) 偏向下一节点而非之前节点

(3) 未处于使用状态的节点

(4) 负载（load）较少的节点

最佳节点在适用上述4项标准时选择具有最小值的节点。

代码段❸、❹、❺根据zone列表顺序构建实际备份列表。代码段❸中，由while循环查找当前节点的最佳节点的节点号，因此，如果zone列表顺序为节点顺序，就调用build_zonelists_in_node_order()函数，以节点顺序构建备份列表；如果是zone顺序，则在代码段❹中为node_order[]数组保存节点顺序，其定义如下所示。

:: mm/page_alloc.c

```
static int node_order[MAX_NUMNODES];
```

利用node_order[]数组保存的节点顺序在代码段❺中调用build_zonelists_in_zone_order()函数，用zone顺序构建备份列表。

构建各节点的zone列表，即备份列表的核心函数就是build_zonelists_in_node_order()函数和build_zonelists_in_zone_order()函数，17.4.2节和17.4.3节将依次讲解。

在代码段❻中调用build_thisnode_zonelists()函数，在node_zonelists[1].zonerefs[]数组中构建相应节点的zone列表（17.4.5节）。

17.4.2 build_zonelists_in_node_order()

代码17-5是build_zonelists_in_node_order()函数。

:: 代码17-5 mm/page_alloc.c的build_zonelists_in_node_order()

```
static void build_zonelists_in_node_order(pg_data_t *pgdat, int node)
{
    int j;
    struct zonelist *zonelist;

    zonelist = &pgdat->node_zonelists[0];
    for (j = 0; zonelist->_zonerefs[j].zone != NULL; j++)                    ❶

    j = build_zonelists_node(NODE_DATA(node), zonelist, j,                   ❷
                             MAX_NR_ZONES - 1);

    zonelist->_zonerefs[j].zone = NULL;                                      ❸
    zonelist->_zonerefs[j].zone_idx = 0;
}
```

该阶段以节点为单位构建备份列表，各节点的zone按顺序构建。具有这些zone的列表的数组就是zonelists的_zonerefs成员变量。

在代码段❶中，先找到_zonerefs的成员中zone为非NULL的索引。zone为非NULL表示该区域已添加到备份列表。因此，找到_zonerefs[j].zone为非NULL的索引j后，将相应节点的zone从_zonerefs[j]开始添加到数组即可。

在代码段❷中调用build_zonelists_node()函数，将相应节点的zone添加到_zonerefs[]数组。此时，将索引j传递给参数以接收结果值。

在节点中，zone可由多个类型构成，各类型分别映射到_zonerefs[]数组。因此，通过build_zonelists_node()函数将zone添加到_zonerefs[]数组时使用。如果已经将该节点的所有zone添加到_zonerefs[]数组，则返回添加下一节点zone的位置。本节最后部分将分析build_zonelists_node()函数。

在代码段❸中初始化zonelist->zoneref[j]的zone和zone_idx，以添加下一节点的zone。这样，为备份列表添加下一个最佳节点的zone时，就从zonelist->zonerefs[j]为NULL的位置开始。

此处还需了解build_zonelists_node()函数。build_zonelists_node的定义如代码17-6所示。

:: 代码17-6　mm/page_alloc.c的build_zonelists_in_node_order()

```
static int build_zonelists_node(pg_data_t *pgdat, struct zonelist *zonelist,
                int nr_zones, enum zone_type zone_type)
{
    struct zone *zone;
    ...

    zone_type++;    /*比zone类型数多+1*/

    do {
        zone_type--;    /*以HIGHMEM->NORMAL->DMA32->DMA顺序*/                    ❶
        zone = pgdat->node_zones + zone_type;
        if (populated_zone(zone)) {                                              ❷
            zoneref_set_zone(zone, &zonelist->_zonerefs[nr_zones++]);
            check_highest_zone(zone_type);
        }
    } while (zone_type);
    return nr_zones;        /*下一节点的备份列表中的起始索引*/
}
```

从代码段❶中可以看出，将节点的zone注册到备份列表时，zone类型是按逆序注册的。即HIGHMEM->NORMAL->DMA32->DMA的顺序。也就是说，如果HIGHMEM中没有内存，就从NORMAL开始分配；如果NORMAL中没有内存，则从DMA开始分配。这样做是为了减少分配内存时发生OOM的风险，最大限度降低对系统的影响。

代码段❷调用populated_zone()函数，只在相应zone中有实际物理内存时调用zoneref_set_zone()函数，将zone注册到_zonerefs[]数组。

check_highest_zone()函数只保存最高zone类型，其声明如下所示。

:: mm/mempolicy.c

```
enum zone_type policy_zone=0;
```



补充知识点　populated_zone()函数的技巧，return !!

```
static inline int populated_zone(struct zone *zone)
{
    return (!!zone->present_pages);
}
```

zone->present_pages是整型值，加上“!”或“!!”后转换成逻辑值。通过下列示例查看。

:: trick.c

```
#include <stdio.h>

void main()
{
    int present_pages = 100;

    printf("present_pages = %d\n", present_pages);
    printf("!present_pages = %d\n", !present_pages);
    printf("!!present_pages = %d\n", !!present_pages);
}
```

运行结果

```
code@kernel$ gcc -o trick trick.c
code@kernel$ ./trick
present_pages = 100
!present_pages = 0
!!present_pages = 1
code@kernel$
```

17.4.3 build_zonelists_in_zone_order()

我们已经了解了节点顺序时构建备份列表的方法。现在通过代码17-7了解zone顺序时构建备份列表的build_zonelists_in_zone_order()函数。

:: 代码17-7 mm/page_alloc.c的build_zonelists_in_zone_order()

```
static void build_zonelists_in_zone_order(pg_data_t *pgdat, int nr_nodes)
{
    ...

    zonelist = &pgdat->node_zonelists[0];
    pos = 0;

    for (zone_type = MAX_NR_ZONES - 1; zone_type >= 0; zone_type--) {          ❶
        for (j = 0; j < nr_nodes; j++) {                                       ❷
            node = node_order[j];
            z = &NODE_DATA(node)->node_zones[zone_type];
            if (populated_zone(z)) {                                           ❸
                zoneref_set_zone(z, &zonelist->_zonerefs[pos++]);
                check_highest_zone(zone_type);
            }
        }
    }
    zonelist->_zonerefs[pos].zone = NULL;                                      ❹
    zonelist->_zonerefs[pos].zone_idx = 0;
}
```

以zone类型为基准，通过代码段❶和代码段❷对所有节点执行循环。此处需要确认和节点顺序一样按照HIGHMEM、NORMAL、DMA32、DMA顺序执行循环。

在代码段❸中将zone注册到实际备份列表。在代码段❹中构建下一节点的备份列表时，指定下一节点备份列表_zonerefs[]数组内的起始位置。

如果节点0和节点1分别通过build_zonelists_in_zone_order()函数构建了HIGHMEM、NORMAL、DMA zone，将获得下列结构。

Node(0)ZONE_HIGHMEM → Node(1)ZONE_HIGHMEM → Node(0)ZONE_NORMAL → Node(1)ZONE_NORMAL → Node(0)ZONE_DMA → Node(1)ZONE_DMA

与节点顺序相比，zone顺序具有局部性（locality）这个缺点。但其优点在于，能通过内存分配防止DMA内存区域的快速消耗。

17.4.4　build_thisnode_zonelists()

通过代码17-8分析build_thisnode_zonelists()函数。

:: 代码17-8　mm/page_alloc.c的build_thisnode_zonelists()

```
static void build_thisnode_zonelists(pg_data_t *pgdat)
{
    int j;
    struct zonelist *zonelist;

    zonelist = &pgdat->node_zonelists[1];                                    ❶
    j = build_zonelists_node(pgdat, zonelist, 0, MAX_NR_ZONES - 1);

    zonelist->_zonerefs[j].zone = NULL;
    zonelist->_zonerefs[j].zone_idx = 0;
}
```

和17.4.2节的build_zonelists_in_node_order()函数类似。只调用build_zonelists_node()函数，在代码段❶中将node_zonelists[1]作为参数传递，从而在node_zonelists[1].zonerefs[]数组中构建相应节点的zone列表。

那么，何时使用node_zonelists[1]中构建的zone列表呢？分配内存时若使用__GFP_THISNODE标签，则无法使用已构建的备份列表，此时使用该zone列表。[①]换言之，只能用于在当前节点或特定节点中才能分配内存的情况，仅在NUMA中有意义。

17.4.5　build_zonelist_cache()

build_zonelist_cache()函数通过位图管理备份列表的所有zone，若zone的所有内存正在使用，就在位图中将对应于该区域的位设置为1。这样即可通过位图快速查找可用zone。

① GFP_THISNODE相关内容请参考17.7节“补充知识点”。

通过代码17-9分析build_zonelist_cache()函数。

:: 代码17-9 mm/page_alloc.c的build_zonelist_cache()

```
static void build_zonelist_cache(pg_data_t *pgdat)
{
    struct zonelist_cache *zlc;                                          ❶

    ...
    zonelist = &pgdat->node_zonelists[0];
    zonelist->zlcache_ptr = zlc = &zonelist->zlcache;

    bitmap_zero(zlc->fullzones, MAX_ZONES_PER_ZONELIST);                 ❷

    for (z = zonelist->_zonerefs; z->zone; z++)                          ❸
        zlc->z_to_n[z - zonelist->_zonerefs] = zonelist_node_idx(z);
}
```

为了通过位图找到能够快速分配内存的zone，需要了解代码段❶的struct zonelist_cache结构体。struct zonelist_cache的定义如下所示。该结构体的定义分为NUMA和UMA，此处假定为NUMA。

:: include/linux/mmzone.h

```
#ifdef CONFIG_NUMA

struct zonelist_cache {
    unsigned short z_to_n[MAX_ZONES_PER_ZONELIST];      /* zone->nid */
    DECLARE_BITMAP(fullzones, MAX_ZONES_PER_ZONELIST);  /* zone full? */
    unsigned long last_full_zap;        /* when last zap'd (jiffies) */
};

#else
#define MAX_ZONELISTS 1
struct zonelist_cache;
#endif
```

struct zonelist_cache结构体具有3个成员，各成员的作用如表17-1所示。

表17-1 zonelist_cache结构体成员变量的意义和作用

成员变量	意　义
z_to_n	z_to_n[]数组保存zonelist中各zone所属节点号
fullzones	DECLARE_BITMAP宏生成的位图，zonelist中的所有zone不存在可用内存时，在位图中将相应zone位置的位设置为1
last_full_zap	对zonelist中设置为1的zone进行检查，确认该zone是否可用，用jiffies表示最后的检查时间①

在代码段❷中将位图fullzones均初始化为0。

① 对无可用内存的zone进行检查而变成可用状态时，在位图中将该位从1变为0，用zap表示。

在代码段中，3将备份列表内的所有节点的节点号保存到z_to_n[]数组。这样构建（保存fullzones位图和节点号的）z_to_n[]数组的原因在于，如果在fullzones位图中找到设置为0的最初的位，则可将位图中的索引用作z_to_n[]数组的索引，从而查找相应zone的节点号。

17.5　输出备用列表信息——mminit_verify_zonelist()

下面分析mminit_verify_zonelist()函数。

:: mm/mm_init.c的mminit_verify_zonelist()

```
void mminit_verify_zonelist(void)
{
        ...
    /*对各节点*/
    for_each_online_node(nid) {

        ...

        /*对节点能具有的最大zone数*/
        for (i = 0; i < MAX_ZONELISTS * MAX_NR_ZONES; i++) {

            ...

            /*输出对zonelist的信息*/
            printk(KERN_DEBUG "mminit::zonelist %s %d:%s = ",
                listid > 0 ? "thisnode" : "general", nid,
                zone->name);

            /*对各zonelist输出zone名称*/
            for_each_zone_zonelist(zone, z, zonelist, zoneid) {
#ifdef CONFIG_NUMA
                printk(KERN_CONT "%d:%s ", zone->node, zone->name);
#else
                printk(KERN_CONT "0:%s ", zone->name);
#endif
            }
            printk(KERN_CONT "\n");
        }
    }
}
```

mminit_verify_zonelist()函数输出系统内所有节点的备份列表信息。该函数只执行for循环访问节点的备份列表，输出构成备份列表的zone的节点号和节点名。

17.6　指定处理页分配请求的节点——cpuset_init_current_mems_allowed()

通过代码17-10分析cpuset_init_current_mems_allowed()函数。

:: 代码17-10 kernel/cpuset.c的cpuset_init_current_mems_allowed()

```
void cpuset_init_current_mems_allowed(void)
{
    nodes_setall(current->mems_allowed);
}
```

cpuset_init_current_mems_allowed()函数只调用nodes_setall()函数，在当前任务current的mems_allowed位图中，将系统的所有节点设置为1。mem_allowed位图决定处理当前任务中发生的页分配请求的节点。

17.7 求空页数——nr_free_pagecache_pages()

下面分析nr_free_pagecache_pages()函数，如代码17-11所示，该函数只调用nr_free_zone_pages()函数。

:: 代码17-11 kernel/stop_machine.c的nr_free_pagecache_pages()

```
unsigned int nr_free_pagecache_pages(void)
{
    return nr_free_zone_pages(gfp_zone(GFP_HIGHUSER_MOVABLE));
}
```

调用nr_free_zone_pages()函数时，将gfp_zone（GFP_HIGHUSER_MOVABLE）的结果值作为参数传递，gfp_zone()函数对传递来的参数标签值（GFP_HIGHUSER_MOVABLE）进行检查并返回zone类型。



补充知识点 GFP位掩码

下面是对GFP_HIGHUSER_MOVABLE标签的定义。

:: include/linux/gfp.h

```
...
#define GFP_KERNEL   (__GFP_WAIT | __GFP_IO | __GFP_FS)
#define GFP_HIGHUSER_MOVABLE    (__GFP_WAIT | __GFP_IO | __GFP_FS | \
                __GFP_HARDWALL | __GFP_HIGHMEM | \
                __GFP_MOVABLE)
...
```

GFP_HIGHUSER_MOVABLE标签值表示高端内存区域的可移动页，由多个GFP的位掩码组构成。GFP的位掩码定义如下所示。

:: include/linux/gfp.h

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

#define __GFP_WAIT  ((__force gfp_t)0x10u)  /*分配内存时允许中断*/
#define __GFP_HIGH  ((__force gfp_t)0x20u)  /*紧急要求分配内存*/
#define __GFP_IO    ((__force gfp_t)0x40u)  /*查找内存时允许执行I/O*/
#define __GFP_FS    ((__force gfp_t)0x80u)  /*查找内存时允许文件系统命令/
#define __GFP_COLD  ((__force gfp_t)0x100u)          /*请求不存在于CPU缓存的页*/
#define __GFP_NOWARN    ((__force gfp_t)0x200u)      /*忽略失败警告*/
#define __GFP_REPEAT    ((__force gfp_t)0x400u)      /*失败时自动重试*/
#define __GFP_NOFAIL    ((__force gfp_t)0x800u)      /*一直执行直至成功*/
#define __GFP_NORETRY   ((__force gfp_t)0x1000u)     /*失败时不进行重试*/
#define __GFP_COMP  ((__force gfp_t)0x4000u)         /*compound页，即多重页*/
#define __GFP_ZERO  ((__force gfp_t)0x8000u)         /*用0填充的页*/
#define __GFP_NOMEMALLOC ((__force gfp_t)0x10000u)   /*禁止使用为紧急情况保留的页*/
#define __GFP_HARDWALL   ((__force gfp_t)0x20000u)   /*NUMA only，放置内存限制*/
#define __GFP_THISNODE  ((__force gfp_t)0x40000u)    /*仅在当前节点分配*/
#define __GFP_RECLAIMABLE ((__force gfp_t)0x80000u)  /*可回收*/
#define __GFP_MOVABLE   ((__force gfp_t)0x100000u)   /*可移动*/
```

像zone modifier区域、action modifier（或flag）区域一样，GFP位掩码共分为2个区域，位的位置和相应位掩码如图17-2所示。

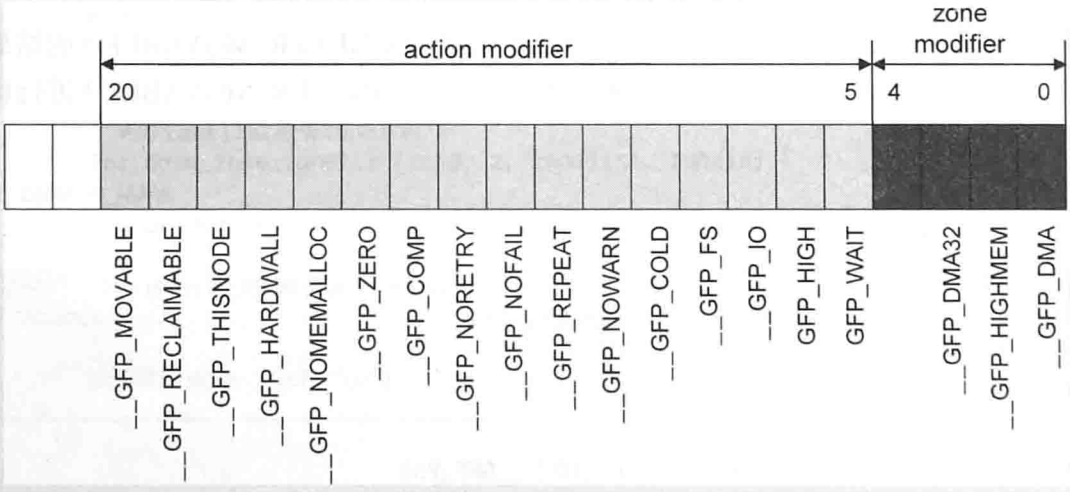


图17-2　GFP的位掩码

zone_modifier在GFP位掩码中对应于4位区域，表示在哪个zone分配空页。如果无法从该zone分配空页，则需要从其他zone分配。zone的检查顺序由zone_modifier中的3个位组合决定，其顺序如表17-2所示。

表17-2 基于zone modifier位结构的zone检查顺序

位	顺 序
0000	ZONE_NORMAL→ZONE_DMA
0001	ZONE_DMA
0011[①]	ZONE_DMA
0010	ZONE_HIGHMEM ® ZONE_NORMAL ® ZONE_DMA

Action modifier用于指定内存分配器的运行。例如，__GFP_WAIT位掩码时，表示可在请求内存分配时执行中断。

接下来求出gfp_zone()函数返回的zone类型中的可用页数，此处使用的函数就是代码17-12的nr_free_zone_pages()函数。

:: 代码17-12 mm/page_alloc.c的nr_free_zone_pages()

```
static unsigned int nr_free_zone_pages(int offset)
{
    ...

    /*对zonelist执行循环、同时只查找特定zone并加上size*/
    for_each_zone_zonelist(zone, z, zonelist, offset) {
        unsigned long size = zone->present_pages;
        unsigned long high = zone->pages_high;
        if (size > high)
            sum += size - high;                                    ❶
    }

    return sum;
}
```

代码段❶访问zonelist的所有zone，在sum中累积从zone->present_pages减掉zone->pages_high的值。zone->present_pages是相应zone中的物理页数，zone->pages_high变量用于决定相应zone是否为idle状态。若存在比pages_high更多的空页（free page），则当前zone变为idle状态。

可用内存不足时，内核将虚拟内存盘的页置换到硬盘，struct zone结构体的pages_min、pages_high、pages_low成员变量都是zone的页被置换到硬盘时浏览的值。

若zone的空页少于pages_low，则需要置换到硬盘；若空页少于pages_min，则执行页回收（reclaim），以在zone中分配请求的页。

如前所述，nr_free_pagecache_pages()函数用于求出可处理的空页数。[②]

① 如已设置__GFP_DMA，即使已设置__GFP_HIGHMEM位也忽略。

② 请确认非总的空页数。

17.8　页移动性

下面再次观察出现页移动性（page mobility）的代码段。

:: 代码17-1　build_all_zonelists()函数的代码段❽

```
    if (vm_total_pages < (pageblock_nr_pages * MIGRATE_TYPES))                    ❽
        page_group_by_mobility_disabled = 1;
    else
        page_group_by_mobility_disabled = 0;
```

这段代码检查特定条件，若通过（代码17-1的build_all_zonelists()函数的代码段❼）nr_free_pagecache_pages()函数求出的值（vm_total_pages）比pageblock_nr_pages×MIGRATE_TYPES小，就不允许以移动性（mobility）为基准执行页集合（page grouping）。

此处需要确认以下两点。

(1) pageblock_nr_pages和MIGRATE_TYPES是什么？

(2) 根据移动性执行的页集合是什么？

首先，pageblock_nr_pages和MIGRATE_TYPES的定义如下所示。

:: include/linux/mmzone.h

```
#define MAX_ORDER 11
```

:: include/linux/pageblock-flags.h

```
#define pageblock_order   (MAX_ORDER-1)①
```

:: include/linux/mmzone.h

```
#define MIGRATE_UNMOVABLE     0          /*不可移动*/
#define MIGRATE_RECLAIMABLE   1          /*可回收*/
#define MIGRATE_MOVABLE       2          /*可移动*/
#define MIGRATE_RESERVE       3          /*为紧急情况而保留*/
#define MIGRATE_ISOLATE       4          /*不可分配*/
#define MIGRATE_TYPES         5
```

MIGRATE_TYPES是表示移动类型的宏，其值为5，表示共5种移动类型。移动类型在Linux内核中表示移动性。

因此，“以移动性为基准执行页集合”是将移动类型相同的页进行组合，如图17-3所示，构建free_list时以移动性为基准执行页集合。

---

① 假定为不适用巨页（huge page）。使用巨页时pageblock_nr_pages定义如下。

```
#define pageblock_nr_pages (1UL << pageblock_order)
```

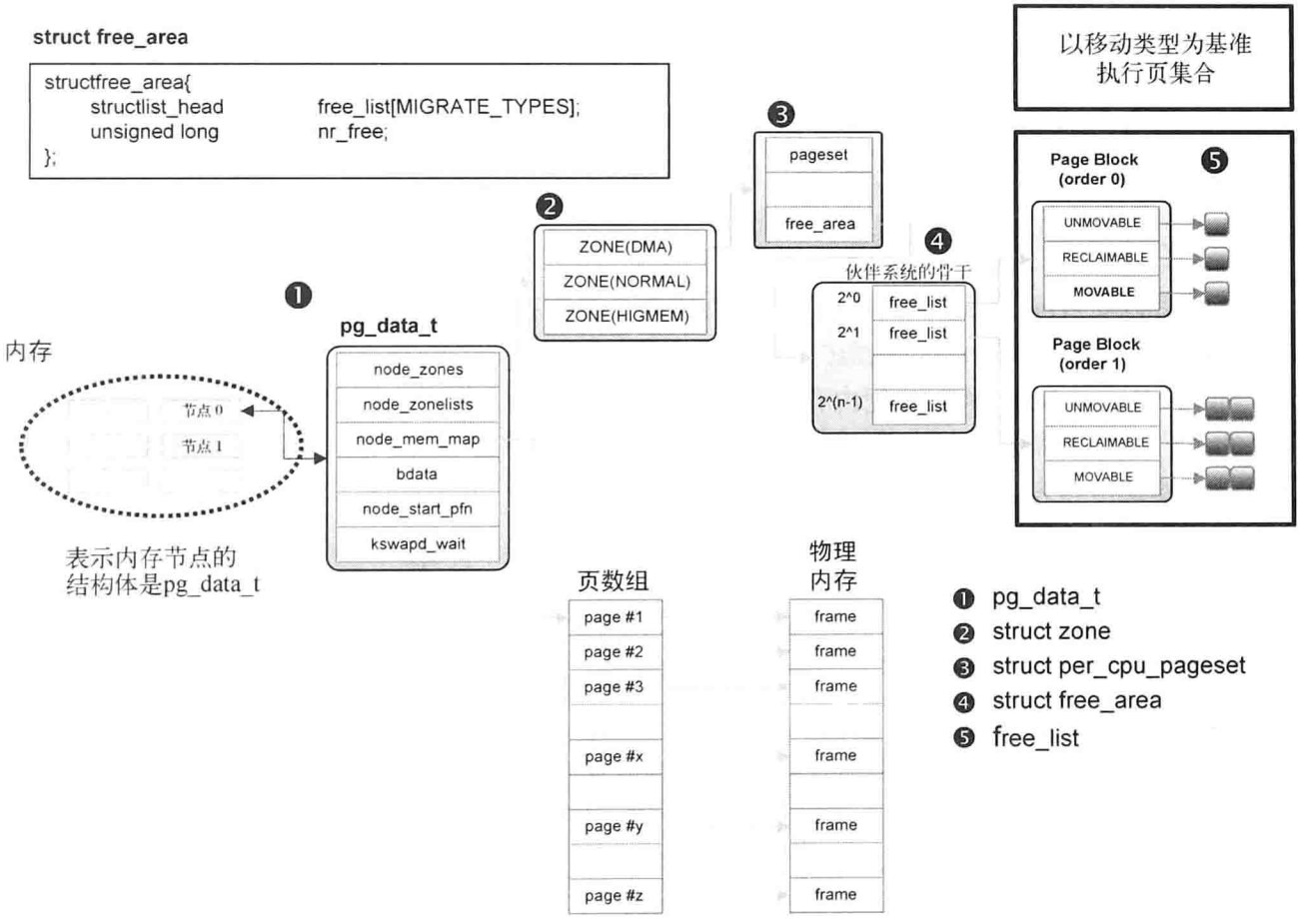


图17-3 构建free_list[]时以页移动性为基准执行页集合

通过集合可获得两大好处。

1. 防止内存碎片

以顺序为单位对具有相同移动属性的页执行集合，可防止内存碎片。例如，需要两个具备UNMOVABLE属性的连续的页时，查找已设置order=1的UNMOVABLE位的页块进行分配，从而防止页的碎片。

2. 分配大内存的方法

将具有相同移动属性的页集合在一处，使其能够在大内存分配中使用。例如，按如下所示对物理内存执行页集合。

- 内核内存（kmalloc）：UNMOVABLE
- 磁盘缓存（inode、dentry）：RECLAIMABLE
- 用户内存+页缓存：MOVABLE

完成页集合后，对类型相同的巨页发出请求时，可提高内存分配成功率。

补充知识点　在内核中防止内存碎片：反碎片（anti fragmentation）

内核将具有相同移动性的页进行组合以防止（prevent）碎片。此处使用“防止”这一术语的原因在于，我们使用的是反碎片（anti fragmentation）方式，而不是文件系统中使用的页碎片解决策略——对破碎的块执行重排序的碎片整理（defragmentation）方式。

我们可以通过build_all_zonelists()对Linux管理的各内存结构体进行初始化和设置操作。如图17-4所示，其中设置了struct zonelist、struct zonelist_cache、struct zoneref结构体成员。

build_all_zonelists()函数分配内存时，若相应zone的内存不足，则需要分配属于其他节点的zone的页。将这些信息构建好的zone列表称为备份列表。最后，build_all_zonelists()函数的主要作用就是构建备份列表。

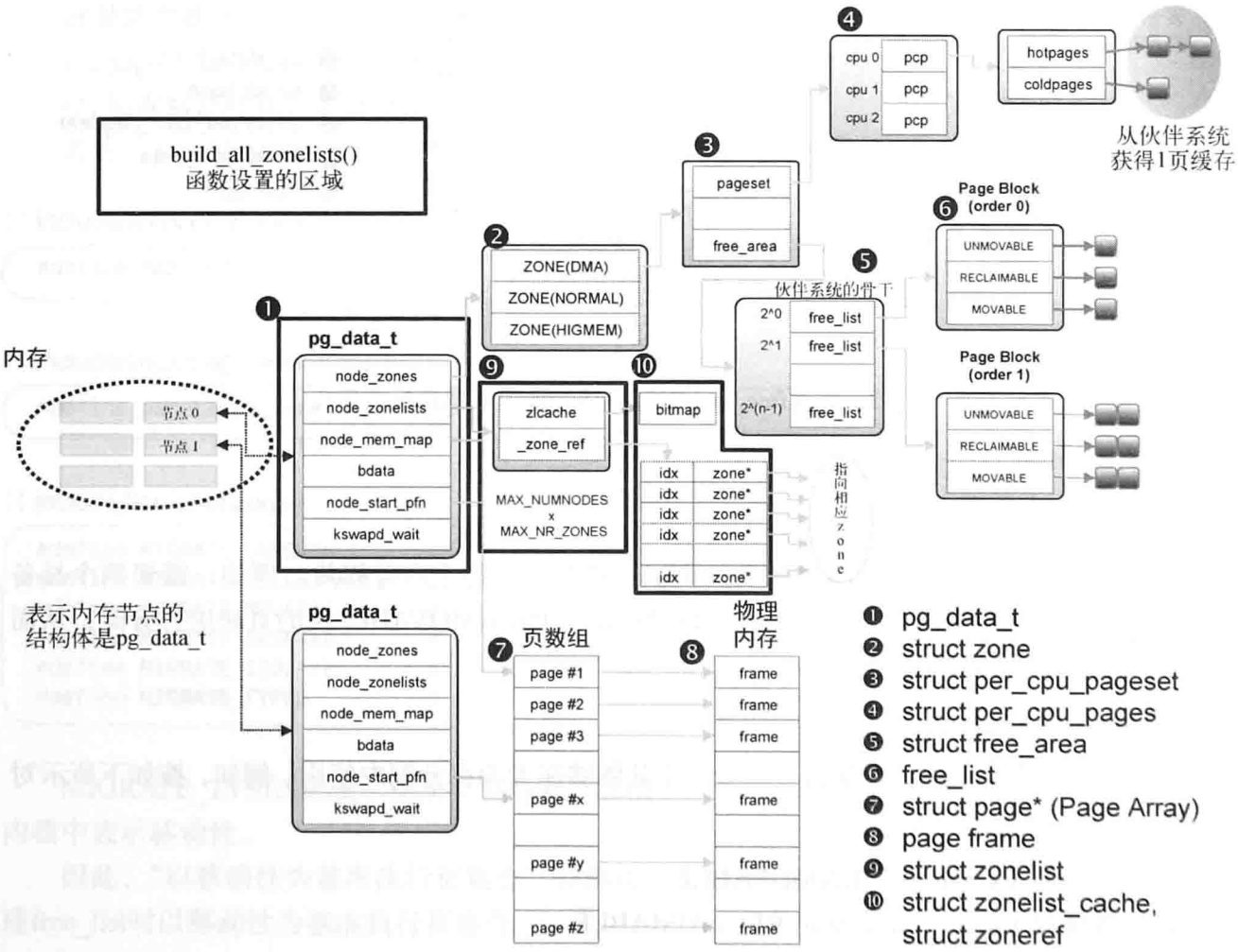


图17-4　build_all_zonelists()函数设置的各结构体及其关系

第 18 章 page_alloc_init()~pidhash_init()

本章将分析在start_kernel()函数中调用的函数，尤其是从page_alloc_init()函数到pidhash_init()函数这部分。图18-1表示这些函数的调用顺序。

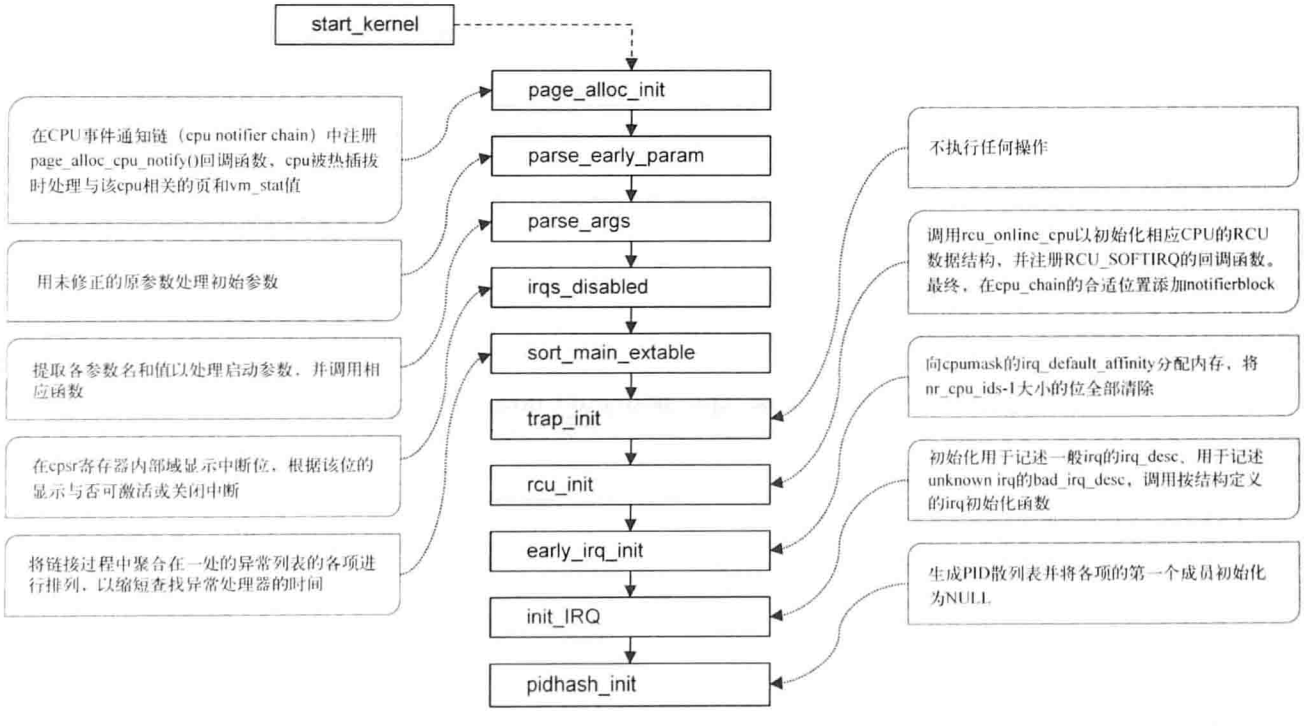


图18-1　从page_alloc_init()函数到pidhash_init()函数的调用关系

trap_init()函数在ARM中不执行任何操作，故省略。

18.1　处理用于热插拔 CPU 的页——page_alloc_init()

page_alloc_init()函数在CPU事件通知链中注册page_alloc_cpu_notify()回调函数，CPU被热插拔时[①]处理与该CPU相关的页。例如，CPU在运行状态下被拔出时，如果还有正在使用CPU的页，

① Linux内核中正式支持CPU热插拔的只有i386、ppc64、ia64、s390，在ARM中还处于试验阶段。

就必须都移到空白列表。因此，回调函数会获得与热插拔的CPU相关的ZONE的per_cpu_pageset结构体信息，并进行更新。

下面通过代码18-1了解page_alloc_init()函数。

:: 代码18-1　mm/page_alloc.c的page_alloc_init()

```
void __init page_alloc_init(void)
{
    hotcpu_notifier(page_alloc_cpu_notify, 0);
}
```

page_alloc_init()函数只调用hotcpu_notifier()函数。hotcpu_notifier()函数的定义如下所示。

:: include/linux/cpu.h

```
#define hotcpu_notifier(fn, pri) {                          \
    static struct notifier_block fn##_nb __cpuinitdata =    \
            { .notifier_call = fn, .priority = pri };       \
    register_cpu_notifier(&fn##_nb);                        \
}
```

制作notifier_block结构体类型的page_alloc_cpu_notify nb，将page_alloc_cpu_notify回调函数以优先顺序0注册到page_alloc_cpu_notify nb。将制作好的notifier，即page_alloc_cpu_notify nb最终注册到cpu_notifier，使其能够在CPU被热插拔时调用。

接下来通过代码18-2了解将page_alloc_cpu_notify nb notifier注册到cpu_notifier的register_cpu_notifier()函数。

:: 代码18-2　kernel/cpu.c的register_cpu_notifier()

```
int __ref register_cpu_notifier(struct notifier_block *nb)
{
    int ret;

    cpu_maps_update_begin();                                    ❶
    ret = raw_notifier_chain_register(&cpu_chain, nb);          ❷
    cpu_maps_update_done();                                     ❸
    return ret;
}
```

代码段❶和❸的cpu_maps_update_begin()、cpu_maps_update_done()函数是为了按顺序访问并更新cpu_online_mask和cpu_present_mask而使用的API，通过互斥体实现。代码段❷的raw_notifier_chain_register()对cpu_chain的列表执行循环，并将接收的notifier插入适当位置（priority大小降序）。

下面通过代码18-3了解实际回调函数page_alloc_cpu_notify()函数的实现部分。

:: 代码18-3 mm/page_alloc.c的page_alloc_cpu_notify()

```
static int page_alloc_cpu_notify(struct notifier_block *self,
                                 unsigned long action, void *hcpu)
{
    int cpu = (unsigned long)hcpu;

    if (action == CPU_DEAD || action == CPU_DEAD_FROZEN) {
        drain_pages(cpu);
        vm_events_fold_cpu(cpu);
        refresh_cpu_vm_stats(cpu);
    }
    return NOTIFY_OK;
}
```


该函数对CPU_DEAD和CPU_DEAD_FROZEN操作进行处理，共调用3个函数：drain_pages()函数将该CPU管理的页移到空白列表；vm_events_fold_cpu()函数将处于CPU_DEAD或CPU_DEAD_FROZEN状态的CPU的页相关事件进行移动，使之能够在执行当前代码的CPU中执行；refresh_cpu_vm_stats()函数获得与被热插拔的CPU相关zone的per_cpu_pageset结构体信息，并进行更新。

18.2 处理 console 参数——parse_early_param()

我们在12.2节中处理了对启动参数进行处理的部分，在代码18-4中用未修正的原参数boot_command_line处理初始参数。

:: 代码18-4 init/main.c的parse_early_param()

```
void __init parse_early_param(void)
{
    static __initdata int done = 0;
    static __initdata char tmp_cmdline[COMMAND_LINE_SIZE];

    if (done)
        return;

    strlcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
    parse_early_options(tmp_cmdline);                                    ❶
    done = 1;
}
```

这段代码判断parse_early_param是否为最初调用，之后调用代码段❶的parse_early_options()函数。

:: kernel/params.c

```
void __init parse_early_options(char *cmdline)
{
    parse_args("early options", cmdline, NULL, 0, do_early_param);
}
```

调用parse_args()，此处只需了解调用最后的参数do_early_param()函数，下一节将讲解parse_args()函数。下面是do_early_param()的代码。

:: 代码18-5　init/main.c的do_early_param()

```
static int __init do_early_param(char *param, char *val)
{
    struct obs_kernel_param *p;

    for (p = __setup_start; p < __setup_end; p++) {                    ❶
        if ((p->early && strcmp(param, p->str) == 0) ||                ❷
            (strcmp(param, "console") == 0 &&
            strcmp(p->str, "earlycon") == 0)
        ) {
            if (p->setup_func(val) != 0)                               ❸
                printk(KERN_WARNING
                       "Malformed early option '%s'\n", param);
        }
    }
    return 0;
}
```

在代码段❶中从__setup_start到__setup_end执行循环。该区域如下所示。

:: arch/arm/kernel/vmlinux.ids.S

```
__setup_start = .;
   *(.init.setup)
__setup_end = .;
```

__setup_start和__setup_end表示.init.setup节区的起始和末尾。那么.init.setup节区中会进入哪些值，又是怎么定义的呢？

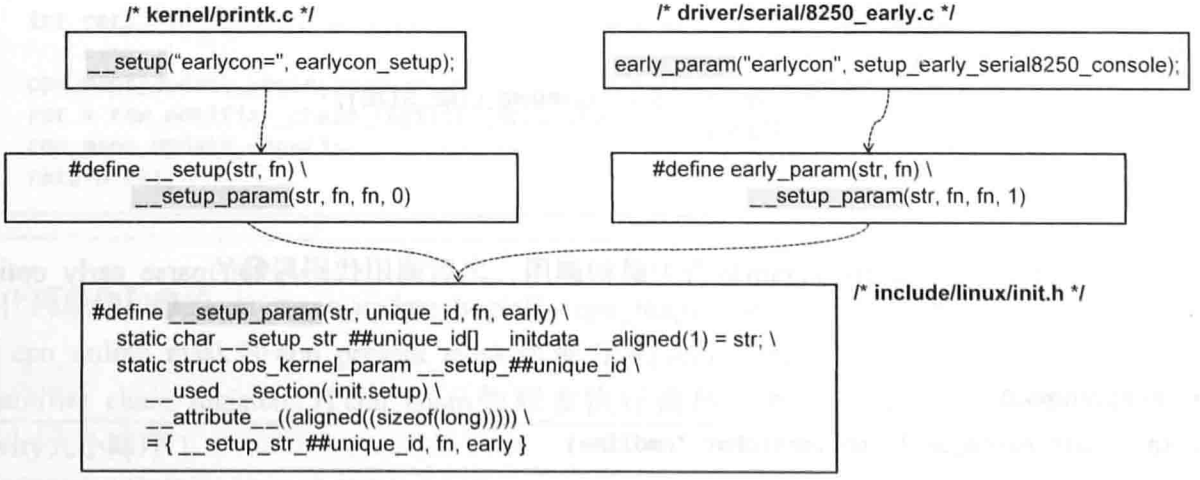


图18-2　__setup_param宏

图18-2是__setup_param宏的定义。.init.h文件的ifndef MODULE部分导致MODULE未定义时，这个宏才有效；已定义MODULE时不执行任何操作。定义MODULE时，该宏会声明obs_kernel_param结构体，初始化各成员变量，并将其置于.init.setup节区。

再次进入代码段❷。只有满足下面2个条件，才会如代码段❸所示，调用.init.setup中注册的obs_kernel_param结构体的setup_func函数。

(1) 定义early_param("console", console_setup)时

(2) 定义early_param("earlycon", earlycon_setup)时

Linux代码中不使用模块，而是将其包含到内核映像，仅在编译8250_early.c时才调用setup_early_serial8250_console()函数。

最终，parse_early_param()函数从内部调用do_early_param()，是只处理"console"参数的代码。下一节将仔细观察该函数如何被parse_args()函数调用。

18.3 处理特殊参数——parse_args()

我们在12.2节结束了对启动参数的处理，剩余未能处理的参数已复制到static_command_line。下面看看处理剩余的特别参数的过程。

为了处理parse_args启动参数，需要提取各参数名和值，并调用与之对应的函数。该函数是通用函数，因此，根据不同参数存在多种情况。由于最后参数unknown是函数指针，所以无法知晓调用哪个函数。该函数虽然在多处被调用，但此处假定在start_kernel()内的调用如下。先观察下列代码。

:: init/main.c

```
parse_args("Booting kernel", static_command_line, __start___param,
           __stop___param - __start___param,
           &unknown_bootoption);
```

在start_kernel()内如上所示实际调用parse_args()。先了解各参数：第一个参数static_command_line具有内核未能处理的启动参数，第二个和第三个参数__start___param和__stop___param如下所示。

:: include/asm-generic/vmlinux.lds.h

```
__param : AT(ADDR(__param) - LOAD_OFFSET) {                    \
    VMLINUX_SYMBOL(__start___param) = .;                       \
    *(__param)                                                 \
    VMLINUX_SYMBOL(__stop___param) = .;                        \
    . = ALIGN((align));                                        \
    VMLINUX_SYMBOL(__end_rodata) = .;                          \
}                                                              \
. = ALIGN((align));
```

__start___param表示__param节区的起始地址，__stop___param表示__param节区的结束地

址。使用module_param宏将之置于__param节区。

该宏如下图所示。

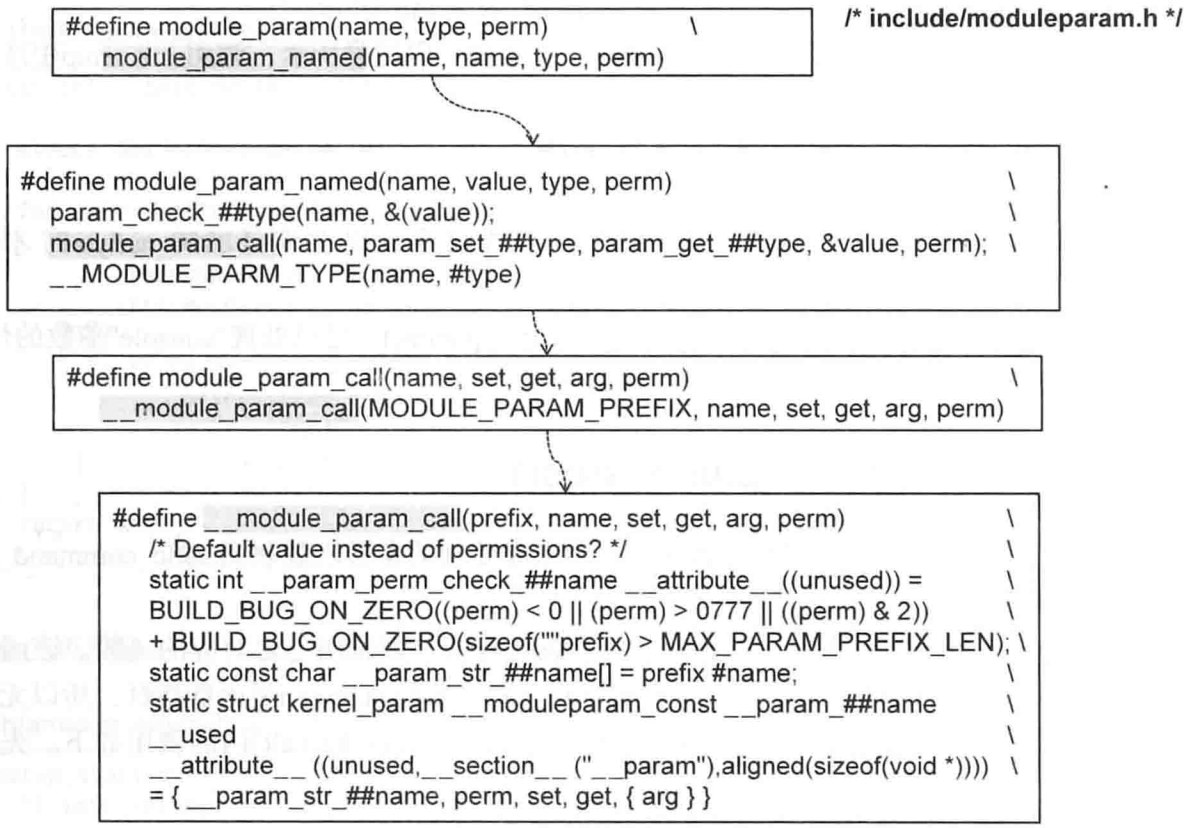


图18-3　module_param宏

这个宏声明并初始化kernel_param结构体，将其配置到param节区。这就是start_kernel内调用parse_args()时使用的参数。下面仔细观察parse_args()函数。

:: 代码18-6　kernel/params.c的parse_args()

```
int parse_args(const char *name,
               char *args,
               struct kernel_param *params,
               unsigned num,
               int (*unknown)(char *param, char *val))
{
    char *param, *val;

    DEBUGP("Parsing ARGS: %s\n", args);

    /* Chew leading spaces */
    while (*args == ' ')                                    ❶
        args++;

    while (*args) {
```

```
        int ret;
        int irq_was_disabled;

        args = next_arg(args, &param, &val);                                    ❷
        irq_was_disabled = irqs_disabled();
        ret = parse_one(param, val, params, num, unknown);                      ❸
        if (irq_was_disabled && !irqs_disabled()) {
            printk(KERN_WARNING "parse_args(): option '%s' enabled "
                    "irq's!\n", param);
        }
        ...

    return 0;
}
```


在代码段❶中检查前面有无空格，并查找空格之后的字符串。然后，在代码段❷的next_arg()函数中，将启动参数字符串中的参数字符串和值进行区分，并设置到param和val。

例如，将默认启动参数假定为"console=ttySAC0,115200 root=/dev/ram init=/bin/bash initrd=0x51000000,4M"。第一次调用next_arg时将变为param=console，val= ttySAC0,115200。[①]

通过循环语句再次调用next_arg()函数，之后param=root，val=/dev/ram。剩下的参数值也按这种方式取值。在代码段❸中，将已取得的param值和val值用作parse_one()的参数以调用函数。parse_one()函数如下所示。

:: 代码18-7 kernel/params.c的parse_one()

```
static int parse_one(char *param,
                     char *val,
                     struct kernel_param *params,
                     unsigned num_params,
                     int (*handle_unknown)(char *param, char *val))
{
    unsigned int i;

    /* Find parameter */
    for (i = 0; i < num_params; i++) {
        if (parameq(param, params[i].name)) {
            DEBUGP("They are equal!  Calling %p\n",
                   params[i].set);
            return params[i].set(val, &params[i]);                              ❶
        }
    }

    if (handle_unknown) {
        DEBUGP("Unknown argument: calling %p\n", handle_unknown);
        return handle_unknown(param, val);                                      ❷
    }
```

① 这只是为示例而使用的值。这些参数实际在parse_cmdline中处理。

```
    DEBUGP("Unknown argument `%s'\n", param);
    return -ENOENT;
}
```

若需要处理的启动参数存在于__param节区，则调用代码段❶。该函数通过下列module_param宏调用已定义的函数。

:: arch/arm/mach-pxa/cpufreq-pxa2xx.c

```
module_param(pxa255_turbo_table, uint,0);
```

module_param()宏函数已如上所示声明时，通过图18-3跟踪宏函数。观察后发现，形成了module_param()调用uint参数、module_param_named()调用uint参数、module_param_call()调用param_set_uint参数、__module_param_call()调用param_set_uint参数的顺序。最终，向set变量注册param_set_uint函数名，在代码段❶中调用param_set_uint()函数。

虽然传递了启动参数，但若__param节区中不存在需要处理的值，则调用代码段❷，此时调用unknown_bootoption()函数。在unknown_bootoption()内将未能处理的参数设置到init进程的argv值，使之能够在init进程中处理。

18.4　确认中断处理是否激活——irqs_disabled()

ARM中有用于显示当前状态的CPSR寄存器。在CPSR寄存器的内部域显示中断位，根据该位显示与否可激活或禁用中断。当前irqs_disabled()函数被定义成确认是否禁用IRQ位的宏，该宏如下所示。

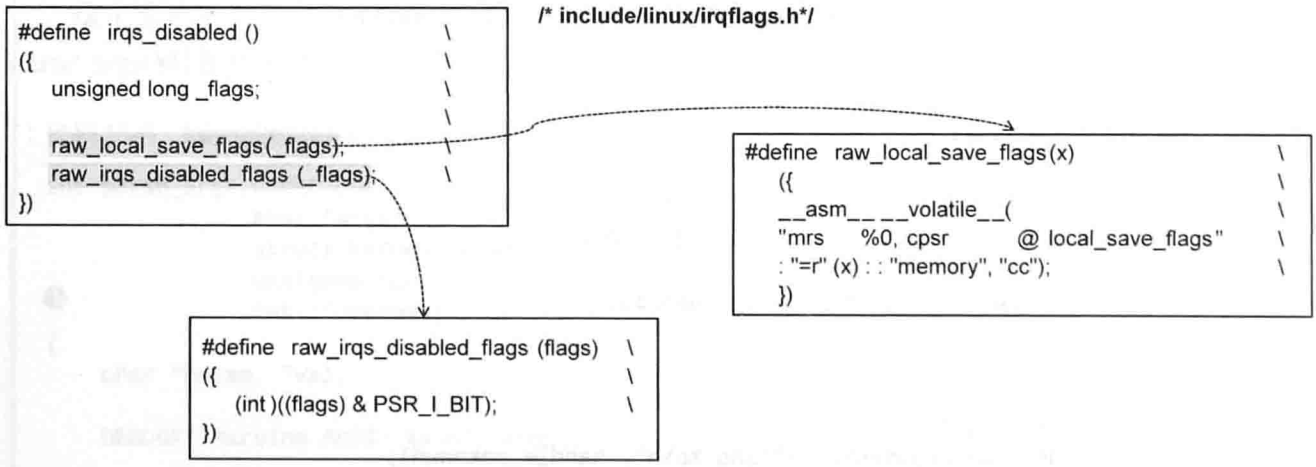


图18-4　irqs_disabled宏

该宏被置换后显示如下内容。我们为了代码可读性而省略了中括号，下面逐一观察。

:: 代码18-8 include/linux/irqflags.h的变换的irqs_disabled()

```
{
    unsigned long _flags;

    __asm____volatile__(
        "mrs    %0, cpsr        @ local_save_flags"                    ❶
        :"=r"(_flags) : : "memory", "cc");

    (int)((_flags) & PSR_I_BIT);                                       ❷
}
```

在代码段❶中使用mrs指令，将CPSR寄存器保存到_flags。在代码段❷中，将_flags和PSR_I_BIT执行位运算，以检查中断位。与进程状态寄存器相关的值定义如下，可以看出PSR_I_BIT被定义为0x00000080。

:: arch/arm/include/asm/ptrace.h

```
#define PSR_T_BIT   0x00000020
#define PSR_F_BIT   0x00000040
#define PSR_I_BIT   0x00000080
#define PSR_A_BIT   0x00000100
#define PSR_J_BIT   0x01000000
#define PSR_Q_BIT   0x08000000
#define PSR_V_BIT   0x10000000
#define PSR_C_BIT   0x20000000
#define PSR_Z_BIT   0x40000000
#define PSR_N_BIT   0x80000000
```

若已设置_flags的第8位，则中断处于禁用状态；反之则表示中断处于激活状态。图18-5表示CPSR的中断位。

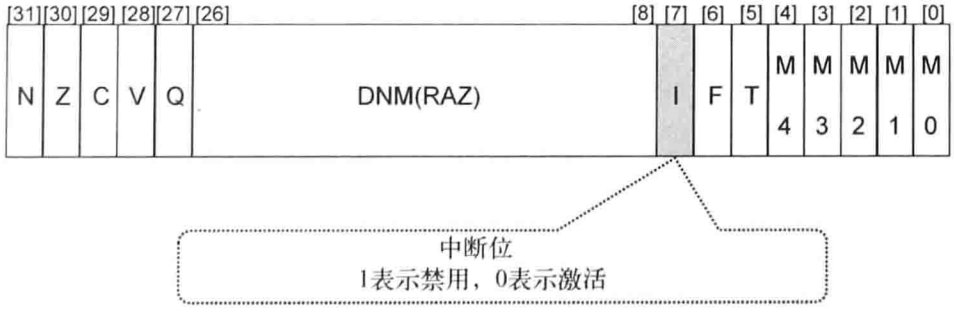


图18-5 CPSR的中断位

18.5 内核异常列表定义——sort_main_extable()

内核异常列表（exception table）是指，内核模式下对用户空间的数据访问失败时，对此进行

处理的处理器所在项（fixup handler entry）。实际上，异常列表的各项并不聚集在一处，而是四处分散的，因此在链接阶段集合在一个位置。

但由于未进行排列，内核运行过程中可能会为查找相应项而耗费较多时间。因此，如果将项集合在一处，可缩短查找异常处理器的时间。而这正是sort_main_extable()函数的作用。sort_main_extable()函数的定义如下所示。

:: kernel/extable.c

```
void __init sort_main_extable(void)
{
    sort_extable(__start___ex_table, __stop___ex_table);
}
```

sort_main_extable()函数只将__start___ex_table和__stop___ex_table用作参数调用sort_extable()函数。__start___ex_table和__stop___ex_table是链接描述符中定义的标签地址，是内核内定义的异常列表所在位置。最终，__start___ex_table和__stop___ex_table之间的异常处理器在sort_extable()函数中排列。

实际排列异常列表的sort_extable()函数定义如下所示。

:: lib/extable.c

```
void sort_extable(struct exception_table_entry *start,
                  struct exception_table_entry *finish)
{
    sort(start, finish - start, sizeof(struct exception_table_entry),
         cmp_ex, NULL);
}
```

内核提供的sort()函数不执行递归调用，可在O(nlog n)时间内排列。通过cmp_ex()函数实际比较异常列表项（struct exception_table_entry）。exception_table_entry结构体和cmp_ex()函数的定义如下所示。

:: arch/arm/include/asm/uaccess.h

```
struct exception_table_entry
{
    unsigned long insn, fixup;
};
```

:: lib/extable.c

```
static int cmp_ex(const void *a, const void *b)
{
    const struct exception_table_entry *x = a, *y = b;
    /* avoid overflow */
    if (x->insn > y->insn)
```

```
        return 1;
    if (x->insn < y->insn)
        return -1;
    return 0;
}
```


可以看出，根据cmp_ex()函数比较并传递指令号（inst）的不同结果，异常列表以指令号为基准进行排列。

18.6 初始化 RCU 机制——rcu_init()

在3.0版内核或开发版内核中，有很多同步的部分被替换成RCU系统。该函数中执行与Linux内核的RCU机制相关的初始化。

:: 代码18-9 kernel/rcupdate.c的rcu_init()

```
void __init rcu_init(void)
{
    __rcu_init();                                                    ❶
    hotcpu_notifier(rcu_barrier_cpu_hotplug, 0);                     ❷
}
```

在代码段❶中，__rcu_init()将当前启动的CPU和CPU_UP_PREPARE用作参数调用rcu_cpu_notify()。该函数最终调用rcu_online_cpu()，初始化RCU相关per-cpu变量。作为发生RCU_SOFTIRQ时调用的回调函数，设置rcu_process_callbacks()。

在代码段❷中，每次发生CPU热插拔事件时管理RCU相关变量。为注册及解除回调函数，将rcu_nb注册到cpu_chain。



补充知识点 RCU

RCU（Read Copy Update，读复制更新）是Linux中使用的同步技术之一，在2002年添加到Linux内核。RCU最大的特点是，reader在执行读取操作的过程中，可更新writer。在一般的锁结构中，writer执行写入操作时，reader需要等待，直到操作完成；反之，reader执行读取操作时，writer也需要等待。

在RCU中可同时执行读写的原因在于制作并使用数据副本。如果reader读取（read）数据的过程中，writer要更新（update）数据，那就利用reader读取的数据副本（copy）。reader的操作完成后，通过writer更新的数据将成为最终版本（原件），被reader读取过的数据则被回收（图18-6）。

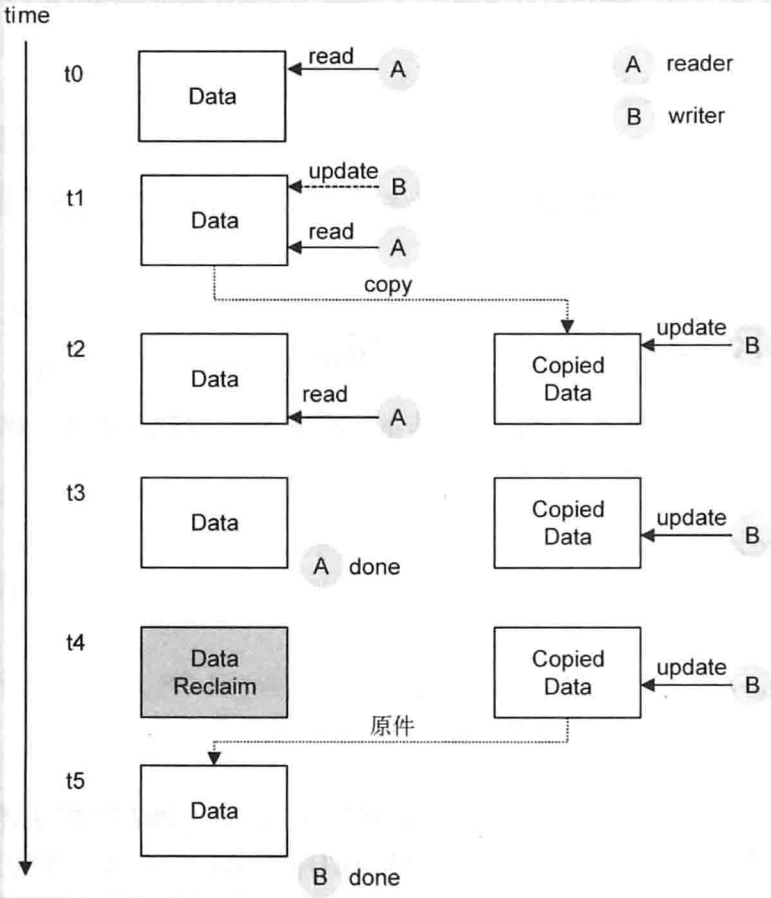


图18-6　RCU基本概念

使用RCU后，无需使用数据读写锁，因此可减少取锁时的系统开销，提高系统扩展性（scalability）。

但RCU也有缺点，即更新时发生的系统开销过多。要更新数据时，如果reader正在读取数据，那么完成读取前需要维护相应数据。何况reader完成读取操作后，为了不再次使用之前的数据，要进行处理。因此，RCU在对读取为主的数据进行同步时显示出最优化的性能。

在RCU中，各reader正在读取数据时，若writer执行更新，那么各reader完成读取的同时回收数据区域。RCU中称之为宽限期（grace period），目的在于让writer更新数据前进入的所有reader能够完成读取操作。

通过图18-7了解宽限期及其相关函数。

图18-7中，“删除”表示writer通过list_del_rcu()等函数开始更新（此处为删除）数据，宽限期调用synchronize_rcu()函数进行设置。最后，“回收”通过kfree()函数表示回收数据（图中是已删除数据的情况）。

图18-7表示1个reader在宽限期中覆盖（overlap）了整个宽限期的情形，在RCU中这是不允许的。

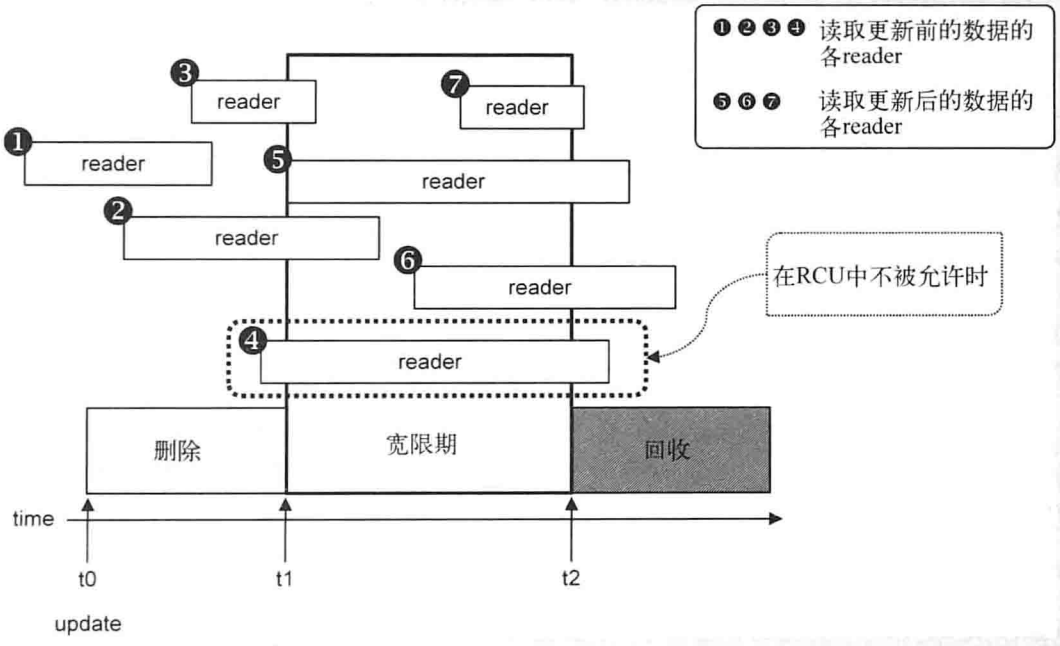


图18-7 在RCU中不被允许的示例

若发生这种情况，就延长宽限期，使writer更新前进入的所有reader能够完整读取数据，如图18-8所示。

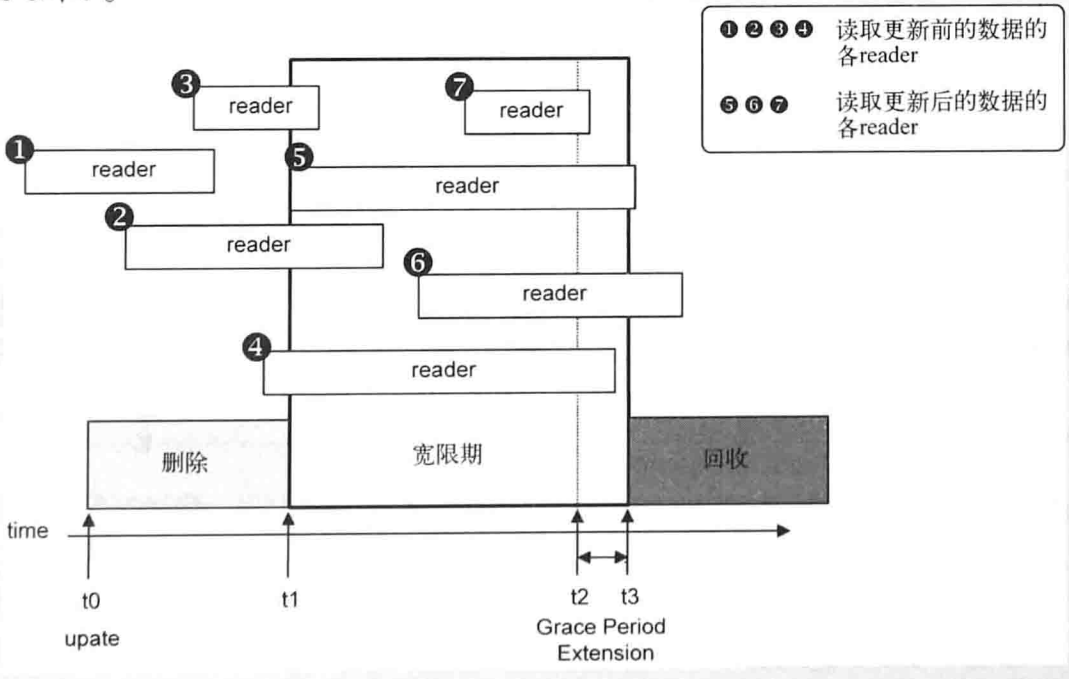


图18-8 延长宽限期

※ 参考文献：The design of preemptible read-copy-update（http://lwn.net/Articles/253651/）RCU机制相关文章（http://www2.rdrop.com/~paulmck/RCU）

18.7　准备使用 IRQ——early_irq_init()

early_irq_init()函数根据CONFIG_SPARSE_IRQ定义与否有所不同。此处假设未定义CONFIG_SPARSE_IRQ进行讲解。

若发生中断，则需要调用符合相应编号的中断处理函数。当前Linux中利用irq_desc结构体处理中断处理所需数据。在early_irq_init()函数中，注册irq_desc结构体前会经过必要的初始化过程。观察下列代码。

:: 代码18-10　kernel/irq/handle.c的early_irq_init()

```
int __init early_irq_init(void)
{
    struct irq_desc *desc;
    int count;
    int i;

    init_irq_default_affinity();                                    ❶

    printk(KERN_INFO "NR_IRQS:%d\n", NR_IRQS);

    desc = irq_desc;                                                ❷

    count = ARRAY_SIZE(irq_desc);

    for (i = 0; i < count; i++) {
        desc[i].irq = i;
        init_alloc_desc_masks(&desc[i], 0, true);                   ❸
        desc[i].kstat_irqs = kstat_irqs_all[i];                     ❹
    }
    return arch_early_irq_init();                                   ❺
}
```

代码段❶的init_irq_default_affinity()函数只在已设置CONFIG_SMP和CONFIG_GENERIC_HARDIRQS时为irq_default_affinity分配位图，并将nr_cpu_ids大小设置为1。否则不执行任何操作。

在代码段❷中可以看到irq_desc全局变量，以NR_IRQS大小被初始化为默认值。irq_desc的声明如下所示。

:: kernel/irq/handle.c

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
    [0 ... NR_IRQS-1] = {
        .status = IRQ_DISABLED,
        .chip = &no_irq_chip,
        .handle_irq = handle_bad_irq,
        .depth = 1,
        .lock = __SPIN_LOCK_UNLOCKED(irq_desc->lock),
    }
};
```

以NR_IRQS大小被静态初始化为默认值。如前所述，拥有处理IRQ所需信息的结构体就是irq_desc，其定义如下所示。

:: include/linux/irq.h

```
struct irq_desc {                                   /*中断描述符*/
    unsigned int        irq;                        /*对该描述符的中断号*/
    struct timer_rand_state *timer_rand_state;      /*指向timer_rand_state的指针*/
    unsigned int            *kstat_irqs;            /*对各CPU的IRQ状态*/
#ifdef CONFIG_INTR_REMAP
    struct irq_2_iommu      *irq_2_iommu;           /*与该IRQ相关的iommu*/
#endif
    irq_flow_handler_t  handle_irq;                 /*高级IRQ事件处理器（如果是NULL则_do_IRQ）*/

    struct irq_chip     *chip;                      /*访问硬件的低级中断*/
    struct msi_desc     *msi_desc;                  /*MSI描述符*/
    void                *handler_data;              /*用于irq_chip函数的各IRQ数据*/
    void                *chip_data;                 /*共享芯片函数及实现的芯片信息的依赖于平台的内部数据*/
    struct irqaction    *action;                    /*IRQ操作链（列表）*/
    unsigned int        status;                     /*IRQ状态信息*/
    unsigned int        depth;                      /*用于重叠的irq_disable()函数的值（disable-depth）*/
    unsigned int        wake_depth;                 /*用于set_irq_wake()的值（enable-depth）*/
    unsigned int        irq_count;                  /*用于查找被stall的IRQ的状态位*/
    unsigned long       last_unhandled;             /*未处理的aging计时器*/
    unsigned int        irqs_unhandled;             /*未处理的中断状态位*/
    spinlock_t          lock;                       /*用于SMP的锁（lock）变量*/
#ifdef CONFIG_SMP
    cpumask_var_t       affinity;                   /*SMP上的IRQ Affinity*/
    unsigned int        cpu;                        /*用于均衡的CPU索引*/
#ifdef CONFIG_GENERIC_PENDING_IRQ
    cpumask_var_t       pending_mask;               /*重新调整的中断待决（pending）掩码*/
#endif
#endif
    atomic_t            threads_active;             /*当前执行的线程的irqaction数*/
    wait_queue_head_t   wait_for_threads;           /*等待线程处理器的sync_irq的等待队列*/
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry   *dir                    /*proc文件系统的/proc/irq项*/
#endif
    const char      *name;                          /*用于向proc/intrrupts输出的处理器名*/
} ___cacheline_internodealigned_in_smp;
```

再次观察代码段❸，以NR_IRQS大小执行循环的同时，为irq_desc结构体的成员变量affinity分配NR_CPU大小的位图，并设置为1。在代码段❸中调用init_alloc_desc_masks()函数，如果最后的参数值为真，则从bootmem接收内存分配；如果是假，则利用slab分配器。

分配好的位图若支持CPU热插拔技术，则将IRQ移动到其他CPU。由于当前ARM不支持热插拔技术，故省略。现在，在代码段❹中利用全局数据kstat_irqs_all设置irq_desc结构体的成员变量kstat_irqs。kstats_irqs如下所示。

:: kernel/irq/handle.c

```
static unsigned int kstat_irqs_all[NR_IRQS][NR_CPUS];
```

设置好的irq_desc结构体的katat_irq变量计算每个CPU发生的中断数。

最后调用代码段❺，不执行任何操作，只返回0。若需要执行依赖于平台的初始化，（由于被声明为__weak）可重新定义该函数以执行必要功能。



补充知识点　查看中断相关设置

SMP中的IRQ affinity

/proc/irq/IRQ#/smp_affinity对现有IRQ代码明示要执行的CPU。这是允许执行的各CPU的位掩码值。这些值不能使所有CPU禁止中断。此外，若IRQ控制器不支持IRQ Affinity，则该值为0xFFFFFFF且保持不变。

/proc/irq/default_smp_affinity是对禁用的IRQ适用的默认位掩码值。若IRQ被激活，那么Affinity位掩码值就被设置为默认掩码值。可通过下列命令查看该值。

表示代码段❶的irq_default_affinity值，下面是nr_cpu_ids值为4时设置的值。

```
code@kernel$ cat /proc/irq/default_smp_affinity
f

code@kernel$
```

可通过下列命令查看代码段❹的kstat_irqs值。

```
code@kernel$ cat /proc/interrupts

           CPU0       CPU1       CPU2       CPU3
  0:         47          0          0          0   IO-APIC-edge      timer
  1:      12552         52          2          0   IO-APIC-edge      i8042
  6:          3          0          0          0   IO-APIC-edge      floppy
  7:          0          0          0          0   IO-APIC-edge      parport0
  8:          0          0          0          0   IO-APIC-edge      rtc0
  9:          0          0          0          0   IO-APIC-fasteoi   acpi
 12:        265       6405          0        832   IO-APIC-edge      i8042
 14:          0          0          0          0   IO-APIC-edge      ata_piix
 15:      11105        803        105         60   IO-APIC-edge      ata_piix
 16:          0          0          0          0   IO-APIC-fasteoi   vmci,
Ensoniq AudioPCI
 17:      13315        301        223        117   IO-APIC-fasteoi   ehci_hcd:usb1, ioc0
 18:         41         12          0          0   IO-APIC-fasteoi   uhci_hcd:usb2
 19:        700         12          6          6   IO-APIC-fasteoi   eth0
NMI:          0          0          0          0   Non-maskable interrupts
LOC:     177537     110351      73101      67992   Local timer interrupts
SPU:          0          0          0          0   Spurious interrupts
PMI:          0          0          0          0   Performance monitoring interrupts
IWI:          0          0          0          0   IRQ work interrupts
RES:       5277       5020       3746       4786   Rescheduling interrupts
```

```
CAL:        227        3486        4946        2591   Function call interrupts
TLB:        464         436         368         501   TLB shootdowns
TRM:          0           0           0           0   Thermal event interrupts
THR:          0           0           0           0   Threshold APIC interrupts
MCE:          0           0           0           0   Machine check exceptions
MCP:         14          14          14          14   Machine check polls
ERR:          0
MIS:          0

code@kernel$
```


18.8 初始化中断——init_IRQ()

Linux使用irq_desc结构体记述IRQ状态。该函数初始化用于记述一般IRQ的irq_desc和用于记述unknown IRQ的bad_irq_desc，并调用各结构中定义的IRQ初始化函数。

:: 代码18-11 arch/arm/kernel/irq.c的init_IRQ()

```
void __init init_IRQ(void)
{
    int irq;

    for (irq = 0; irq < NR_IRQS; irq++)                          ❶
        irq_desc[irq].status |= IRQ_NOREQUEST | IRQ_NOPROBE;

#ifdef CONFIG_SMP
    cpumask_setall(bad_irq_desc.affinity);                       ❷
    bad_irq_desc.cpu = smp_processor_id();
#endif
    init_arch_irq();                                             ❸
}
```

此处将IRQ_NOREQUEST设置为默认值。若已设置该标签，则无法通过我们编写设备驱动程序时常用的内核API——request_irq()使用相应IRQ。观察request_irq()可以发现调用requst_threaded_irq()，若已设置该标签，则返回-EINVAL并结束。

:: include/linux/interrupt.h

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn, unsigned long irqflags,
                         const char *devname, void *dev_id)
{
    ...
    if (desc->status & IRQ_NOREQUEST)
        return -EINVAL;
    ...
}
```

为了使用默认不可用的IRQ，需要将IRQ_VALID标签用作参数调用set_irq_flags()。虽然每个平台都不一样，但在S3C6410中，需要在启动过程将几个IRQ设置为可请求状态，如下所示。

:: arch/arm/plat-s3c64xx/irq.c

```
static void __init s3c64xx_uart_irq(struct uart_irq *uirq)
{
    ...
        set_irq_flags(irq, IRQF_VALID);
    ...
}

void __init s3c64xx_init_irq(u32 vic0_valid, u32 vic1_valid)
{
    for (irq = IRQ_TIMER0; irq <= IRQ_TIMER4; irq++) {
        ...
        set_irq_flags(irq, IRQF_VALID);
    }
    ...
}
```

:: arch/arm/plat-s3c64xx/irq-eint.c

```
static int __init s3c64xx_init_irq_eint(void)
{
    ...
    for (irq = IRQ_EINT(0); irq <= IRQ_EINT(27); irq++) {
        ...
        set_irq_flags(irq, IRQF_VALID);
    }
    ...
}
```

为了使用被初始化为不可用的IRQ，需要另外执行初始化任务。

重新回到代码18-11，在代码段❶中以NR_IRQS[①]大小对初始化过程执行循环，该值显示最大IRQ个数。在代码段❷中，为了能够在其他所有CPU中处理IRQ，将特定CPU中发生的unknown IRQ设置bad_irq_desc Affinity。然后，通过要处理该IRQ的CPU分配正在处理当前启动的CPU。在相应CPU无法处理特定CPU中发生的IRQ时，Affinity用于选择代替处理IRQ的CPU。

最后，在代码段❸中，init_arch_irq()调用根据平台另行编写的IRQ初始化函数以初始化中断。该函数一般初始化VIC（Vectored Interrupt Controller，向量中断控制器），设置符合IRQ的处理器，并分配计时器。

init_arch_irq()被定义为函数指针，初期被初始化为NULL。该函数指针在kernel/setup.c的setup_arch()中设置为系统的mdesc->init_irq。下面观察S3C6410中定义的函数。

① 该值在include/asm-arch/irq.h中根据不同平台有不同定义。

:: 代码18-12 arch/arm/kernel/irq.c的s3c64xx_init_irq()

```
void __init s3c64xx_init_irq(u32 vic0_valid, u32 vic1_valid)
{
    ...
    vic_init(S3C_VA_VIC0, S3C_VIC0_BASE, vic0_valid);                    ❶
    vic_init(S3C_VA_VIC1, S3C_VIC1_BASE, vic1_valid);

    set_irq_chained_handler(IRQ_TIMER0_VIC, s3c_irq_demux_timer0);       ❷
    set_irq_chained_handler(IRQ_TIMER1_VIC, s3c_irq_demux_timer1);
    set_irq_chained_handler(IRQ_TIMER2_VIC, s3c_irq_demux_timer2);
    set_irq_chained_handler(IRQ_TIMER3_VIC, s3c_irq_demux_timer3);
    set_irq_chained_handler(IRQ_TIMER4_VIC, s3c_irq_demux_timer4);

    for (irq = IRQ_TIMER0; irq <= IRQ_TIMER4; irq++) {
        set_irq_chip(irq, &s3c_irq_timer);
        set_irq_handler(irq, handle_level_irq);
        set_irq_flags(irq, IRQF_VALID);
    }

    ...
}
```

在代码段❶中初始化VIC，在代码段❷中调用set_irq_handler()和set_irq_chained_handler()注册几种计时器中断的中断处理器。从现在起，发生计时器中断时，通过generic_handle_irq()调用此处注册的处理器。

18.9 构建迅速搜寻进程信息的结构——pidhash_init()

进程具有各自固有的ID——PID，内核中经常需要利用PID查找相应进程信息（进程描述符）。例如，kill()系统调用时，要查找对应于PID的进程描述符，以获得用于记录信号的数据结构指针。

此时逐一搜索进程列表会导致效率低下，为了快速执行内核搜索，如图18-9制作PID散列表。该散列表的项包括进程描述符指针，将下列pid_hashfn()宏用作散列函数，将PID转换成散列表的slot索引。hash_long()只是对输入值执行移位运算以制作散列值的函数，此处省略。

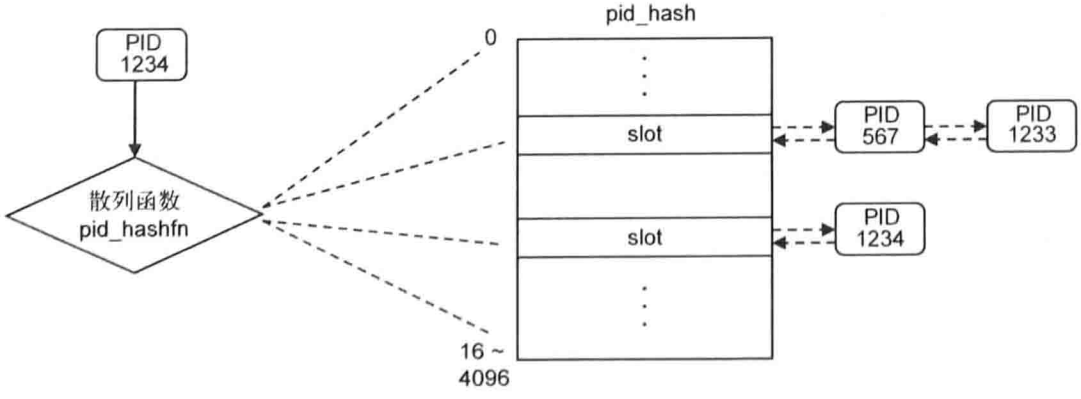


图18-9 散列函数和pid_hash列表

:: kernel/pid.c

```
#define pid_hashfn(nr, ns)    \
        hash_long((unsigned long)nr + (unsigned long)ns, pidhash_shift)
```

此处PID和散列表slot之间不存在1:1对应关系，相同slot中分配的PID间会发生冲突。为了解决这些冲突，PID生成双向链表执行链接。这种分配并初始化散列表中使用的内存的过程是在内核起始函数中执行的，其过程如代码18-13所示。

:: 代码18-13　kernel/pid.c的pidhash_init()

```
void __init pidhash_init(void)
{
    int i, pidhash_size;
    unsigned long megabytes = nr_kernel_pages >> (20 - PAGE_SHIFT);

    pidhash_shift = max(4, fls(megabytes * 4));                          ❶
    pidhash_shift = min(12, pidhash_shift);
    pidhash_size = 1 << pidhash_shift;

    ...

    pid_hash = alloc_bootmem(pidhash_size * sizeof(*(pid_hash)));        ❷
    ...
    for (i = 0; i < pidhash_size; i++)                                   ❸
        INIT_HLIST_HEAD(&pid_hash[i]);
}
```

首先，在代码段❶中将PID散列表的大小设置到pidhash_size变量。散列表的大小根据机器内核内存大小而不同，由最少16个slot到最多4096个slot组成。上述代码中的megabytes变量表示内核内存大小。

在代码段❷中，为指向PID散列表的变量pid_hash分配slot个数（pidhash_size）乘以各slot大小（sizeof(*pid_hash)）的内存。全局变量pid_hash由双向链表组成，其声明如下所示。

:: kernel/pid.c

```
static struct hlist_head① *pid_hash;
```

最后，为了对pid_hash列表内存进行初始化，在代码段❸中使用下列INIT_HLIST_HEAD()宏为链表分配NULL。

:: include/linux/list.h

```
#define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
```

① 链表相关数据结构和函数在include/linux/list.h中定义。

第 19 章 init_timers()~page_cgroup_init()

本章将分析在start_kernel()函数中调用的函数，尤其是从init_timers()函数到cgroup_init()函数这部分。图19-1表示这些函数的调用顺序。



图19-1　从init_timers()函数到page_cgroup_init()函数的调用关系

此处profile_init()为使用内核的默认分析器而初始化。该函数被分配分析器[①]使用的缓冲，并初始化分析CPU掩码图（profiling cpu mask map）。下面逐一讲解其余函数。

19.1　初始化计时器——init_timers()

计时器是给定时间间隔后调用特定函数的技术。在Linux内核中利用jiffies变量测定，jiffies具有系统启动后发生的时钟周期的次数。一般情况下，周期是根据中断而增加的值，硬件决定了1个周期大概是多少时间。

计时器中有显示计时器期满时点的域。设置计时器时，为当前jiffies值加上所需周期数，从而向该域设置值。内核检查计时器时，比较计时器期满时间域和当前jiffies值，若jiffies值大于等于期满时间，则计时器期满，执行已设置的函数。

本节的init_timers()为使用计时器而执行整体需要的初始化任务。完成任务后初始化各计时器以使用，如代码19-1所示。

:: 代码19-1　kernel/timer.c的init_timers()

```
void __init init_timers(void)
{
    int err = timer_cpu_notify(&timers_nb, (unsigned long)CPU_UP_PREPARE,
                               (void *)(long)smp_processor_id());                    ❶

    ...
    register_cpu_notifier(&timers_nb);                                               ❷
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);                                  ❸
}
```

首先，在代码段❶中为使用计时器做准备。timer_cpu_notify()根据CPU的运行执行特定任务后，如声明所示，注册为notifier。其作用是设置指定的time vector base，以根据不同CPU设置计时器。

:: kernel/timer.c的notifier block的声明

```
static struct notifier_block __cpuinitdata timers_nb = {
    .notifier_call  = timer_cpu_notify,
};
```

在代码段❷中，CPU开始运行或中止时，为了能够执行特定任务，将notiffier注册到chain列表型数据结构。在代码段❸中，将发生TIMER_SOFTIRQ时调用的操作函数指定为run_timer_softirq()。

下面仔细观察上述3个代码段。

① 内核分析器负责收集程序执行期间发生的数据，以分析性能或提供内核自身运行方式信息。

19.1.1 timers_cpu_notify()

代码19-1的代码段❶中，将CPU_UP_PREPARE和PID作为参数传递而调用timer_cpu_notify()。CPU_UP_PREPARE表示已准备好CPU，用于设置对应于按CPU赋予的PID的计时器值。通过代码19-2仔细观察该函数。

:: 代码19-2 kernel/timer.c的timer_cpu_notify()

```
static int __cpuinit timer_cpu_notify(struct notifier_block *self,
                                      unsigned long action, void *hcpu)
{
    long cpu = (long)hcpu;
    switch(action) {
    case CPU_UP_PREPARE:
    case CPU_UP_PREPARE_FROZEN:
        if (init_timers_cpu(cpu) < 0)                                    ❶
            return NOTIFY_BAD;
            break;
#ifdef CONFIG_HOTPLUG_CPU
    case CPU_DEAD:
    case CPU_DEAD_FROZEN:
    migrate_timers(cpu);                                                 ❷
    break;
#endif
    ...
}
```

代码段❶在CPU_UP_PREPAR或CPU_UP_PREPARE_FROZEN操作时执行，是CPU开始运行的情形。此时，调用init_timers_cpu()并初始化作为参数传递的对应于PID的CPU计时器。此处调用的init_timers_cpu()内部为timer vector base的变量分配当前jiffies值。

代码段❷相当于中止CPU的情形，只在根据ifdef CONFIG_HOTPLUG_CPU语句使用可动态插入或删除CPU的热插拔功能时定义。中止CPU时调用migrate_timers()，将属于相应CPU的计时器移动到其他CPU。

19.1.2 register_cpu_notifier()

CPU开始或中止运行时，为了通知该运行并执行特定任务，注册notifier。代码19-1的代码段❷中，作为参数传递的timers_nb具有执行特定任务的notifier函数指针，该函数指针为如下notifier block数据结构形态。除notifier函数指针外，还包括链表指针和优先顺序值。

:: include/linux/notifier.h

```
struct notifier_block {
    int (*notifier_call)(struct notifier_block *, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};
```

notifier block在register_cpu_notifier()内部作为cpu_chain的新项添加。仔细观察代码19-3。

:: 代码19-3　kernel/cpu.c的register_cpu_notifier()

```
int __ref register_cpu_notifier(struct notifier_block *nb)
{
    int ret;
    cpu_maps_update_begin();                                    ❶
    ret = raw_notifier_chain_register(&cpu_chain, nb);          ❷
    cpu_maps_update_done();                                     ❸
    return ret;
}
```

CPU相关掩码值作为特定锁已执行同步。因此，为了更新某个CPU掩码值，需要先锁住互斥锁，代码段❶的cpu_maps_update_begin()执行该操作。

在代码段❷中，将根据CPU状态变化而调用的回调函数notifier注册到cpu_chain。cpu_chain具有发生通知时要执行的列表，是raw notifier head形态的全局变量。raw notifier head只具有notifier block的head，对注册或解除callback没有限制，调用方必须通过locking进行保护。

:: include/linux/notifier.h

```
struct raw_notifier_head {
    struct notifier_block *head;
};
```

在代码段❸中，使用cpu_mask_update_done()解除已锁住的互斥体锁。

19.1.3　open_softirq()

代码19-1的代码段❸中，将TIMER_SOFTIRQ和run_timer_softirq()函数作为参数传递并调用open_softirq()。该函数的作用非常简单，如代码19-4所示。

:: 代码19-4　kernel/softirq.c的open_softirq()

```
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;                            ❶
}
```

softirq_vec是全局变量，具有用于处理软件中断softirq的向量处理器softirq_action数据结构，其声明如下所示。

:: include/linux/interrupt.h

```
struct softirq_action
{
    void    (*action)(struct softirq_action *);
};
```

:: kernel/softirq.c

```
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

在代码段❶中，将TIMER_SOFTIRQ值用作向量处理器数组softirq_vec的索引，将相应中断发生时要采取的操作设置为run_timer_softirq()。

19.2 初始化高分辨率计时器——hrtimers_init()

Linux内核计时器中有HR计时器（High Resolution timer）。普通计时器以jiffies为单位运行，而HR计时器以纳米秒为单位运行。

各计时器需要标准周期以计算时间。普通计时器以计时控制器中周期性发生的中断计算jiffies值，并以该jiffies值为基准运行。HR计时器需要能够计算纳米秒的值，将计时器中断发生后的时间以纳米秒为单位保存到计时控制器，读取该值以设置具有纳米秒单位的时间格式值的ktime[①]值，并以ktime中设置的值为基准运行HR计时器。

初始化HR计时器的程序与初始化普通计时器的程序相同，只有notify block和SOFTIRQ变量值有差异，如代码19-5所示。

:: 代码19-5 kernel/hrtimer.c的hrtimers_init()

```
void __init hrtimers_init(void)
{
    hrtimer_cpu_notify(&hrtimers_nb, (unsigned long)CPU_UP_PREPARE,
                       (void *)(long)smp_processor_id());                    ❶
    register_cpu_notifier(&hrtimers_nb);                                     ❷
#ifdef   CONFIG_HIGH_RES_TIMERS
    open_softirq(HRTIMER_SOFTIRQ, run_hrtimer_softirq);                      ❸
#endif
}
```

与普通计时器类似，在代码段❶中为使用HR计时器做准备。hrtimer_cpu_notify()为根据CPU运行执行特定任务，如下列代码所示注册为notifier，为了按照不同CPU设置计时器，设置指定的timer vector base。

:: kernel/hrtimer.c

```
static struct notifier_block __cpuinitdata hrtimers_nb = {
    .notifier_call = hrtimer_cpu_notify,
};
```

在代码段❷中，为了能在CPU开始或中止运行时执行特定操作，将HR计时器的notifier与普通计时器一样注册到相同的chain数据结构。在代码段❸中，将HRTIMER_SOFTIRQ发生时调用

① 定义于include/linux/ktime.h。

的操作函数指定为run_hrtimer_softirq()。



补充知识点　内核的范围 HR计时器的实现

高分辨率计时器（hrtimer）在十分精密的时间间隔（interval）后调用内核代码。与普通计时器（regular timer）不同，我们用它精密控制微秒（micro second，用μs标记）或纳米秒（nano second，用ns标记）等时间。

但与这种需要在特定时间内准确反应的情况相比，通常只需在特定时间范围（range）内反应。例如，HR计时器被设置为某个任务需要在20μs唤醒（wakeup），但实际上，即使在更短的10μs时wakeup也没有关系。内核为这种情况提供了范围HR计时器（range hrtimer）。

为了显示计时器期满的时间，范围HR计时器具有2个期满时间，软时间（soft expiration time）和硬时间（hard expiration time）。软时间和硬时间的差就是范围，wakeup发生在范围内时使用（图19-2）。

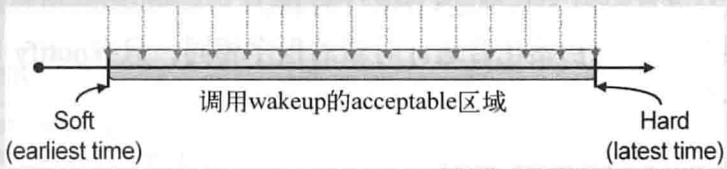


图19-2　范围HR计时器的软时间和硬时间

范围HR计时器的目的在于，发生计时器中断时执行内核的HR计时器代码，对期满软时间的所有任务引发wakeup，只用1次中断处理可同时清除软件时间期满的其他计时器中发生的计时器中断，从而减少中断数和电力消耗。

当前Linux内核提供范围HR计时器，但不能对软件时间期满的所有的任务引发wakeup。这是当前内核使用的数据结构带来的限制，通过下面示例了解一下。

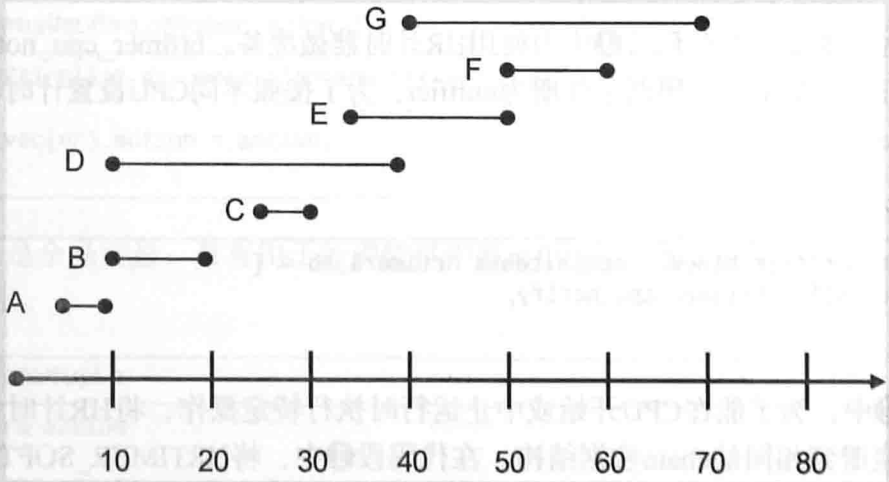


图19-3　计时器的软时间和硬时间

假定我们有图19-3的从A到G共7个计时器，7个计时器每10μs硬时间期满。

计时器A中，虽然硬时间的期满时间为10μs，但软时间5μs时也可期满。计时器A在10μs期满时，由于计时器B的软时间期满为10μs，若同时期满则可清除计时器B的中断。而计时器D的软时间为10μs，所以计时器A期满时需要同时使计时器B和计时器D期满。但当前内核中无法使D期满，因为内核将HR计时器用红黑树进行管理，且搜索期满时间时使用深度优先搜寻（depth first search）。用红黑树将图19-3构建为图19-4。

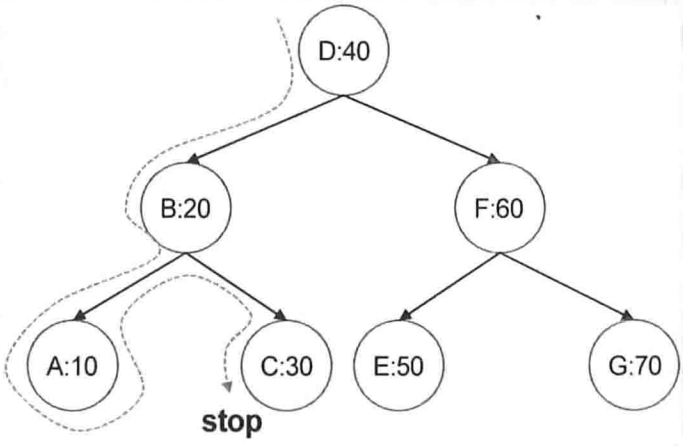


图19-4 用红黑树构建的各HR计时器

虽然可以在10μs使计时器A和B期满，但计时器C的软时间为25μs，因此不再进行搜索。计时器D虽然能够在10μs期满，但以当前的内核代码无法实现。

为了解决这些问题，内核3.2版将利用为当前数据结构红黑树添加附加信息的Augmented红黑树。这种方式不仅可以记述硬时间，也可以记述子树内最快的软时间。将图19-4用Augmented红黑树表现为图19-5。

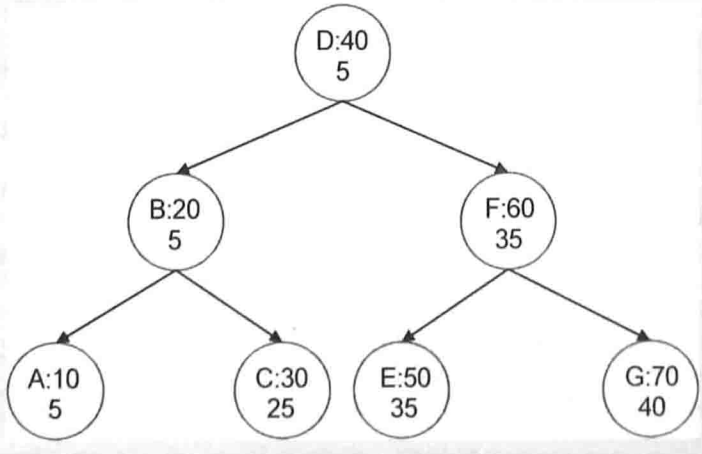


图19-5 要管理HR计时器的Augmented红黑树（预计内核3.2中包含）

管理范围HR计时器的Augmented红黑树请参考下列参考文献。

※参考文献：Linux Weekly Network（lwn.net），Greedy hrtimer

19.3　注册 softirq 的回调函数——softirq_init()

硬件中断处理器不能花费太多时间。Linux内核将这项任务分成2部分，在硬件中断处理器中进行最少量的处理，其余任务则延迟处理。在旧的内核版本中，通过底半部（bottom half）机制处理延迟任务，但未能获得多处理器的好处，所以转换成softirq机制。softirq被静态声明，9个softirq定义如下。

:: include/linux/interrupt.h

```
enum
{
    HI_SOFTIRQ=0,                    /*优先顺序高的tasklet*/
    TIMER_SOFTIRQ,                   /*计时器中断*/
    NET_TX_SOFTIRQ,                  /*网络数据包传输*/
    NET_RX_SOFTIRQ.                  /*网络数据包接收*/
    BLOCK_SOFTIRQ,                   /*I/O相关处理*/
    TASKLET_SOFTIRQ,                 /*tasklet*/
    SCHED_SOFTIRQ,                   /*日程表相关处理*/
    HRTIMER_SOFTIRQ,                 /*hrtimer中断*/
    RCU_SOFTIRQ,                     /*RCU相关处理*/
    NR_SOFTIRQS
}
```

softirq每次由多个CPU执行，因此需要同步。softirq中有tasklet，用于执行延迟任务，基于softirq运行。与softirq的不同之处在于，它保证1个tasklet只能在1个CPU中执行。相同类型的tasklet按顺序执行，不同类型的tasklet并列执行。

softirq_init()函数中，各CPU均初始化tasklet和用于softirq的链表，并注册回调函数。下面通过代码19-6进行分析。

:: 代码19-6　kernel/softirq.c的softirq_init()

```
void __init softirq_init(void)
{
    int cpu;

    for_each_possible_cpu(cpu) {
        int i;

        per_cpu(tasklet_vec, cpu).tail =
            &per_cpu(tasklet_vec, cpu).head;                    ❶
        per_cpu(tasklet_hi_vec, cpu).tail =
            &per_cpu(tasklet_hi_vec, cpu).head;

        for (i = 0; i < NR_SOFTIRQS; i++)
            INIT_LIST_HEAD(&per_cpu(softirq_work_list[i], cpu));  ❷
    }

    register_hotcpu_notifier(&remote_softirq_cpu_notifier);     ❸
```

```
    open_softirq(TASKLET_SOFTIRQ, tasklet_action);                    ❹
    open_softirq(HI_SOFTIRQ, tasklet_hi_action);
}
```

在代码段❶中初始化链表以处理tasklet，在代码段❷中按各softirq初始化链表。此处会连接实际需要处理的延迟任务。

在代码段❸中，注册发生CPU热插拔时执行的回调函数remote_softirq_cpu_notify()函数。

在代码段❹中，为softirq类型中对应于HI_SOFTIRQ和TASKLET_SOFTIRQ的softirq_vec注册回调函数tasklet_action和task_hi_action。由于per_cpu宏已经对tasklet_vec、tasklet_hi_vec这2个类型执行了调用，因此，为了按照不同CPU处理tasklet，具有2个列表。

通过代码19-7可知，发生CPU热插拔时执行的回调函数remote_softirq_cpu_notify()函数会进行哪些softirq相关处理。

:: 代码19-7 kernel/softirq.c的remote_softirq_cpu_notify()

```
static int __cpuinit remote_softirq_cpu_notify(struct notifier_block *self,
                                               unsigned long action, void *hcpu)
{

    if (action == CPU_DEAD || action == CPU_DEAD_FROZEN) {
        int cpu = (unsigned long) hcpu;
        int i;

        local_irq_disable();
        for (i = 0; i < NR_SOFTIRQS; i++) {
            struct list_head *head = &per_cpu(softirq_work_list[i], cpu);
            struct list_head *local_head;

            if (list_empty(head))
                continue;

            local_head = &__get_cpu_var(softirq_work_list[i]);

            list_splice_init(head, local_head);                       ❶

            raise_softirq_irqoff(i);                                  ❷
        }
        local_irq_enable();
    }

    return NOTIFY_OK;
}
```

发生CPU热插拔时，对softirq_work_list执行循环，同时检查CPU_DEAD和CPU_DEAD_FROZEN时要执行的操作中有无softirq。如果有，则在代码段❶中对当前CPU的softirq_work_list挂上发生热插拔的CPU的soft_work_list，并初始化为空白列表。然后在代码段❷中调用raise_softirq_irqoff()函数，唤醒ksoftirqd。

raise_softirq_irqoff()函数定义如代码19-8所示。

:: 代码19-8　kernel/softirq.c的raise_softirq_irqoff()

```
inline void raise_softirq_irqoff(unsigned int nr)
{
    __raise_softirq_irqoff(nr);

    if (!in_interrupt())
        wakeup_softirqd();
}
```

首先，通过__raise_softirq_irqoff()函数为__softirq_pending变量设置位，使ksoftirqd能够处理任务。然后，如果其后不是中断上下文，则调用wakeup_softirqd()函数，否则设置位的任务会在返回IRQ或softirq后处理。wakeup_softirqd()函数的定义如下所示，用于确认ksoftirqd的任务状态，非运行时才唤醒任务。

:: kernel/softirq.c

```
void wakeup_softirqd(void)
{
    /* Interrupts are disabled: no need to stop preemption */
    struct task_struct *tsk = __get_cpu_var(ksoftirqd);

    if (tsk && tsk->state != TASK_RUNNING)
        wake_up_process(tsk);
}
```

raise_softirq()函数引发softirq时，从内部调用__do_softirq()函数，又从该函数调用注册到softirq_vec的各函数。

19.4　设置 xtime——timekeeping_init()

在timekeeping_init()函数中初始化需要和时钟代码共同管理的时间相关值。适用NTP并设置保存当前时间的xtime，初始化xtime_cache以及total_sleep_time值，用于求出内核的当前时间或启动时间。通过代码19-9进行分析。

:: 代码19-9　kernel/time/timekeeping.c的timekeeping_init()

```
void __init timekeeping_init(void)
{
    unsigned long flags;
    unsigned long sec = read_persistent_clock();                    ❶

    write_seqlock_irqsave(&xtime_lock, flags);
    ntp_init();                                                     ❷
```

```
    clock = clocksource_get_next();                                    ❸
    clocksource_enable(clock);
    clocksource_calculate_interval(clock, NTP_INTERVAL_LENGTH);        ❹
    clock->cycle_last = clocksource_read(clock);                       ❺

    xtime.tv_sec = sec;
    xtime.tv_nsec = 0;

    set_normalized_timespec(&wall_to_monotonic,                        ❻
        -xtime.tv_sec, -xtime.tv_nsec);

    update_xtime_cache(0);                                             ❼

    total_sleep_time = 0;
    write_sequnlock_irqrestore(&xtime_lock, flags);
}
```


代码段❶的read_persistent_clock()函数的每个结构都要实现，若还未实现则返回默认值0。当前ARM结构中尚未实现。

在代码段❷的ntp_init()函数中初始化NTP状态变量，leap timer基于以当前时间为基准计算时间的CLOCK_REALTIME，被初始化为使用绝对时间，注册了计算leap second的函数。

然后，代码段❸的clocksource_get_next()函数在启动结束前返回clocksource_jiffies时钟。clock_source为clocksource_jiffies时，由于jiffies值已经运行，故无需激活。

代码段❹的clocksource_calculate_interval()函数通过NTP_INTERVAL_LENGTH设置clock的成员变量xtime_interval、low_interval、cycle_interval。NTP_INTERVAL_LENGTH是jiffies值增加1时所需纳米秒单位的值。

代码段❺中，从clocksource_read()函数获取jiffies值，代入clock->cycle_last并初始化xtime值。

在代码段❻中设置timespec变量的second、nanoseconds域，用timespec保存格式进行正规化操作。wall_to_monotinic是为获得单调时间而需要与xtime相加的值。单调时间在系统启动时间中设置为0，虽然wall_to_monotonic的值为负数，但tv_nsec值始终保持为正数，因此可用于正规化操作。wall_to_monotonic在单调计时器中止后重启时运行。为获得实际启动时间，将total_sleep_time与wall_to_monotonic相加。

在代码段❼中将xtime的值如实复制到xtime_cache，并添加作为参数传递的nsec值0。xtime_cache用于求出当前内核时间。

在代码段❽中，将total_sleep_time初始化为0。下面通过代码简单了解timekeeping_init()函数调用函数中的ntp_init()函数和set_narmalized_timespec()函数。

先了解set_normallized_timespec()函数，其定义如下所示。

:: kernel/time.c

```
void set_normalized_timespec(struct timespec *ts, time_t sec, long nsec)
{
    while (nsec >= NSEC_PER_SEC) {
```

```
        nsec -= NSEC_PER_SEC;
        ++sec;
    }
    while (nsec < 0) {
        nsec += NSEC_PER_SEC;
        --sec;
    }
    ts->tv_sec = sec;
    ts->tv_nsec = nsec;
}
EXPORT_SYMBOL(set_normalized_timespec);
```

作为参数传递的sec、nsec被设置为0，而NSEC_PER_SEC具有1000000000值，因此wall_to_monotonic的tv_sec_nsec被设置为0。

接下来了解ntp_init()函数，其定义如下所示。

:: kernel/time/ntp.c

```
void __init ntp_init(void)
{
    ntp_clear();
    hrtimer_init(&leap_timer, CLOCK_REALTIME, HRTIMER_MODE_ABS);
    leap_timer.function = ntp_leap_second;
}
```

通过hrtimer_init()函数初始化计时器。可设置的时钟有CLOCK_REALTIME和CLOCK_MONOTONIC。REALTIME表示当前系统的realtime，MONOTONIC表示系统启动时从0开始的时钟。时钟模式中可使用相对时间和绝对时间，相对时间表示相对于当前时间的时间。leap_timer时钟设置为REALTIME和绝对时间。

向leap_timer注册ntp_leap_second函数，这是为了适应被增减的闰秒（leap second），进而调整平均太阳时和原子时的误差。向UTC插入闰秒时，发生23:59:60时间而增加1秒，因此将xtime延迟1秒以调整时间。反之，闰秒被清除1秒时，由于时间变快，故需要将xtime增加1秒。



补充知识点　NTP

NTP（Net Time Protocol，网络时间协议）是连接到网络的电脑为了同步时钟时刻而使用的协议。NTP使用UTC（Universal Time Coordinated，协调世界时）将电脑的时钟时间同步到1/1000秒以下。通过整个电脑网络保持准确时刻是十分重要的，因为即使是几十分之1秒的差异也可能导致大问题。

例如，有了UTC，即使在地域分散的业务处理环境也可准确维持任务顺序，安全机制也基于全网UTC。多台电脑修改同一文件系统也需要依赖于准确同步化的时钟时刻。

19.5 初始化硬件计时器——time_init()

我们在该函数中初始化硬件时钟，设置计时器中断周期，注册中断发生时调用的处理器。该函数十分依赖于特定硬件，下面以S3C6410为例观察代码。

:: 代码19-10 arch/arm/kernel/time.c的time_init()

```
void __init time_init(void)
{
#ifndef CONFIG_GENERIC_TIME
    if (system_timer->offset == NULL)
        system_timer->offset = dummy_gettimeoffset;
#endif

        system_timer->init();                                    ❶
}
```

time_init()函数只调用全局变量system_timer的init()。system_timer在之前的setup_arch()函数中被设置为指向mdesc->timer。结果，每个机器结构都拥有不同的值。因此，要想在新设备中封装Linux内核，就需要编写能够将对应于timer_init()的system_timer->init设置为符合相应设备的代码。S3C6410时，机器描述符设置如下。

:: arch/arm/mach-s3c6410/mach-smdk6410.c

```
MACHINE_START(SMDK6410, "SMDK6410")
    .phys_io    = S3C_PA_UART & 0xfff00000,
    .io_pg_offst    = (((u32)S3C_VA_UART) >> 18) & 0xfffc,
    .boot_params    = S3C64XX_PA_SDRAM + 0x100,

    .init_irq   = s3c6410_init_irq,
    .map_io     = smdk6410_map_io,
    .init_machine   = smdk6410_machine_init,
    .timer      = &s3c24xx_timer,
MACHINE_END
```

机器描述符的timer变量被设置为s3c24xx_timer，因此代码段❶中调用s3c24xx_timer->init()函数。s3c24xx_timer的值如下所示。

:: arch/arm/plat-s3c/time.c

```
struct sys_timer s3c24xx_timer = {
        .init       = s3c2410_timer_init,
        .offset     = s3c2410_gettimeoffset,
        .resume     = s3c2410_timer_setup
};
```

最终调用的函数将成为s3c2410_timer_init()。下列代码依赖于特定硬件，先简单了解一下。

:: 代码19-11　arch/arm/plat-s3c/time.c的s3c2410_timer_init()

```
static void __init s3c2410_timer_init(void)
{
    s3c2410_timer_resources();                                    ❶
    s3c2410_timer_setup();                                        ❷
    setup_irq(IRQ_TIMER4, &s3c2410_timer_irq);                    ❸
}
```

在代码段❶中进行计时器时钟相关初始化。然后在代码段❷中激活计时器中断，设置计时周期并开始计时。最后，在代码段❸中注册计时器中断发生时要处理的处理器函数。

中断发生时会调用irqaction结构体的成员变量.handler中设置的函数，在s3c2410_timer_irq中如下所示。

:: arch/arm/plat-s3c/time.c

```
static struct irqaction s3c2410_timer_irq = {
    .name       = "S3C2410 Timer Tick",
    .flags      = IRQF_DISABLED | IRQF_TIMER | IRQF_IRQPOLL,
    .handler    = s3c2410_timer_interrupt,
};
```

最终，中断发生时调用的函数将成为s3c2410_timer_interrupt()。

中断处理方法请参考12.9节。

19.6　初始化时钟时间——sched_clock_init()

Linux内核中，每个CPU都有sched_clock_data结构体。利用该结构体计算就绪队列的时钟值，该值在各调度器函数中决定是否调度。在sched_clock_init()函数中初始化各CPU具有的sched_clock_data结构体。

但是，未定义CONFIG_HAVE_UNSTABLE_SCHED_CLOCK时不使用上述结构体。不仅如此，当前函数仅将sched_clock_running设置为1，不执行任何操作。ARM的默认设置文件中未进行任何定义，下面简单了解一下。

以下就是定义CONFIG_HAVE_UNSTABLE_SCHED_CLOCK时调用的代码。

:: 代码19-12　kernel/sched_clock.c的sched_clock_init()

```
void sched_clock_init(void)
{

    u64 ktime_now = ktime_to_ns(ktime_get());                     ❶
    int cpu;

    for_each_possible_cpu(cpu) {
        struct sched_clock_data *scd = cpu_sdc(cpu);              ❷
```

```
        scd->lock = (raw_spinlock_t)__RAW_SPIN_LOCK_UNLOCKED;
        scd->tick_raw = 0;
        scd->tick_gtod = ktime_now;
        scd->clock = ktime_now;
    }
    sched_clock_running = 1;                                    ❸
}
```


代码段❶中返回获得单调时间（monotonic time）并转换成纳米秒后的值。返回的值在代码段❷中初始化各CPU中保存的sched_clock_data。按照不同CPU进行初始化后，在代码段❸中将sched_clock_running设置为1。设置好的值决定是否在sched_clock_cpu()中0调用sched_clock()。

若调用__update_sched_clock()函数，那么按照不同CPU初始化的sched_clock_data结构体则更新为新值。该函数调用过程如图19-6所示。

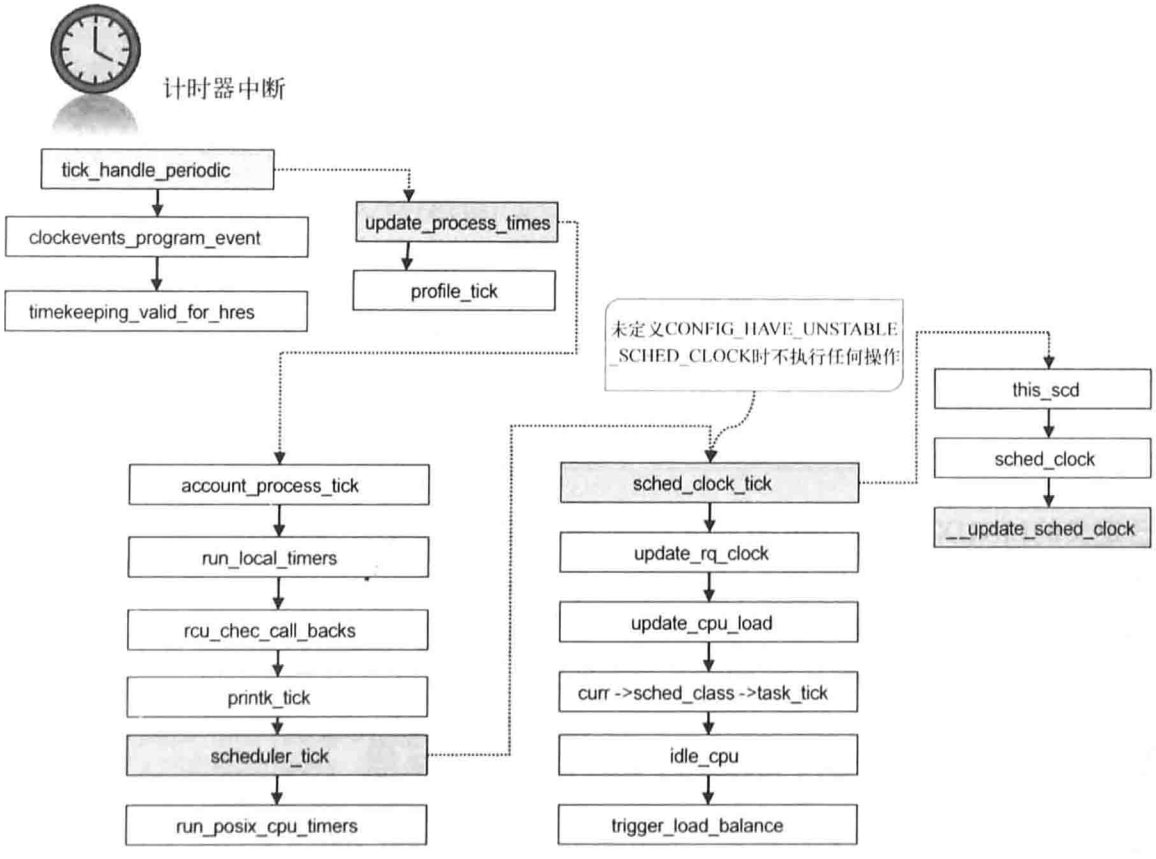


图19-6 __update_sched_clock调用过程

每次发生计时器中断时，各CPU的sched_clock_data结构体的成员变量会更新为新值。将新值重置为相应CPU的就绪队列的成员变量——时钟值。

updata_curr()或updata_curr_rt()函数中，就绪队列的时钟值用于计算当前任务的执行时间以及CPU使用时间等。如果任务的执行时间比基准时间长，则调用resched_task()，标记为需要重新调

度。计时器中断结束时（参考12.9.2节），利用标记的值调用schedule()函数，转换成新任务。



补充知识点　Linux的多种时间检测

Linux内核提供多种方法以检测并表示时间。

经过时间（wall time）

指人可以认知的日常生活中的时间，比如人们看墙上的挂钟就可得知时间之类的情形。从电脑的角度讲，它表示电脑完成任务的实际时间。实际上wall time可以发生变化，因为它不仅可由用户调整，而且系统调整时间时会略有不一致。就像挂钟读秒一样，以秒为单位的时间可能表示得并不准确。

单调时间（monotonic time）

Linux中启动后线性增长的时间。单调时间能够保证启动后严格按照线性增加，这在需要准确计算2个对象之间发生的时间差时十分有用。

19.7　激活 CPU 的中断处理——local_irq_enable()

顾名思义，local_irq_enable()函数用于激活进入当前执行代码的CPU的中断。由于该函数决定CPU是否接受中断，所以大部分代码依赖于处理器，并由汇编代码编写而成。

:: include/linux/irqflags.h

```
#define local_irq_enable() \
        do { trace_hardirqs_on(); raw_local_irq_enable(); } while (0)
}
```

已定义的LINUX_ARM_ARCH不同，raw_local_irq_enable()代码也不一样，但总的来说是清除CPSR的I位以激活IRQ。

执行该函数后激活中断，此时可处理中断。中断发生时跳转到中断向量表，在IRQ描述符中查找并调用该中断的处理器。

19.8　检测用作根文件系统的 init 虚拟内存盘

如果之前在内核命令行中已设置“initrd=xxx”，那么initrd_start变量中可能已经设置了值。

:: init/main.c

```
if (initrd_start && !initrd_below_start_ok &&
    page_to_pfn(virt_to_page((void *)initrd_start)) < min_low_pfn) {
        printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
            "disabling it.\n",
            page_to_pfn(virt_to_page((void *)initrd_start)),
```

```
                min_low_pfn);
    initrd_start = 0;
}
```

此处只确认initra是否存在于正确的内存位置，如果不在正确位置，则禁用initrd；否则维持已激活的状态。

补充知识点 initrd

initrd是initial ramdisk的缩写，内核2.6中添加了initramfs(cpio)。然而人们依然使用initrd，主要用于在小型系统等的虚拟内存盘中直接使用根文件系统（也可以使用initramfs）。

初始化bootmem的过程中，要另外预约（reserve）initrd中使用的内存。如果内核被设置为“以后调用rootfs_initcall节区时使用initrd”，则挂载默认initramfs，将initrd预约的内存数据复制到initramfs内部文件，之后释放initrd预约的内存。

19.9 初始化以分配动态内存——vmalloc_init()

vmalloc()函数用于在内核区域中分配动态内存。和用户区域中用于分配动态内存的malloc()函数一样，该函数将不连续的物理内存分配成连续的逻辑内存。vmalloc_init()函数为能够使用这些vmalloc()函数而进行初始化。

:: 代码19-13 mm/vmalloc.c的vmalloc_init()

```
void __init vmalloc_init(void)
{
    struct vmap_area *va;                                          ❶
    struct vm_struct *tmp;
    int i;

    for_each_possible_cpu(i) {                                     ❷
        struct vmap_block_queue *vbq;

        vbq = &per_cpu(vmap_block_queue, i);
        spin_lock_init(&vbq->lock);
        INIT_LIST_HEAD(&vbq->free);
        INIT_LIST_HEAD(&vbq->dirty);
        vbq->nr_dirty = 0;
    }

    for (tmp = vmlist; tmp; tmp = tmp->next) {                     ❸
        va = alloc_bootmem(sizeof(struct vmap_area));
        va->flags = tmp->flags | VM_VM_AREA;
        va->va_start = (unsigned long)tmp->addr;
        va->va_end = va->va_start + tmp->size;
        __insert_vmap_area(va);
```

```
    }
    vmap_initialized = true;                                    ❹
}
```

代码段❶中最先看到的是vmap_area结构体和vm_struct结构体。通过vmalloc()分配的内核内存区域通过vm_struct结构体显示。各vm_struct结构体为了能快速访问而由红黑树管理，红黑树的各节点显示为vmap_area结构体。这种结构与用户区域中mm_struct利用mmap和mm_rb管理vm_area_struct类似。

另外，vm_area_struct结构体用于对用户区域中进程的虚拟内存进行管理。vm_struct和vm_area_struct没有关系。在代码段❷中对per-cpu列表vmap_block_queue执行初始化。

在代码段❸中对vm_list执行循环，同时对当前存在的vm_struct构建红黑树。每个vm_struct生成vmap_area，并将vmap_area结构体的va_start、va_end、flags的信息以该vm_struct的内容为基础进行构建。然后通过__insert_vmap_area()函数，向以vmap_area_root为根的红黑树插入（insert）各节点（vmap_area）。

以vmap_area作为节点构建红黑树后，将代码段❹中的全局变量vmap_initialized变更为真（true）。

19.10　预先初始化目录项和索引节点缓存——vfs_caches_init_early()

在vfs_caches_init_early()函数中制作用于目录项缓存（dentry cache）和索引节点缓存（inode cache）的散列表并进行初始化。文件访问（file access）显示出空间局部性（spatial locality）和时间局部性（temporal locality），可有效使用目录项缓存和索引节点缓存。

vfs_caches_init_early()函数制作用于目录项缓存和索引节点缓存的散列表，并初始化散列表项。此处调用dcache_init_early()和inode_init_early()函数。

:: 代码19-14　fs/dcache.c的vfs_caches_init_early()

```
void __init vfs_caches_init_early(void)
{
    dcache_init_early();                                        ❶
    inode_init_early();                                         ❷
}
```

在dcache_init_early()函数中制作并初始化目录项缓存散列表，制作散列表的目的在于快速访问与给定文件路径相关的目录项对象。由于散列表能够占据很多内存空间，所以如果生成散列表的内存分配失败，就将散列表的项数减少一半并继续分配。

:: 代码19-15　fs/dcache.c的dcache_init_early()

```
static void __init dcache_init_early(void)
{
    int loop;
```

```
    if (hashdist)                                                    ❶
        return;
    dentry_hashtable =
        alloc_large_system_hash("Dentry cache",                      ❷
                    sizeof(struct hlist_head),
                    dhash_entries,
                    13,
                    HASH_EARLY,
                    &d_hash_shift,
                    &d_hash_mask,
                    0);

    for (loop = 0; loop < (1 << d_hash_shift); loop++)               ❸
        INIT_HLIST_HEAD(&dentry_hashtable[loop]);
}
```

观察代码段❷可以发现，通过alloc_large_system_hash内核API生成“Dentry cache”散列表。通过dhash_entries变量传递要生成的散列表项数信息。该函数使用bootmem、vmalloc、get_free_pages()函数之一为散列表分配内存。若当前系统具有64位的NUMA环境，则如代码段❶所示，hashdist的值变为“真”并返回。hashdist的值由下列代码决定。

:: include/linux/bootmem.h

```
#if defined(CONFIG_NUMA) && defined(CONFIG_64BIT)
#define HASHDIST_DEFAULT 1
#else
#define HASHDIST_DEFAULT 0
#endif
extern int hashdist;
```

:: mm/page_alloc.c

```
int hashdist = HASHDIST_DEFAULT;
```

64位NUMA系统中，要分配的散列表需经多个节点执行分配，因此要通过vmalloc()执行操作。vmalloc()可用前延迟散列表的分配，最终通过vfs_caches_init()函数执行操作。

通过代码段❸对分配的散列表进行初始化。

:: 代码19-16 mm/page_alloc.c的alloc_large_system_has()

```
void *__init alloc_large_system_hash(const char *tablename,
                    unsigned long bucketsize,
                    unsigned long numentries,
                    int scale,
                    int flags,
                    unsigned int *_hash_shift,
                    unsigned int *_hash_mask,
```

```
                    unsigned long limit)
{
    ...
    if (!numentries) {                                                    ❶

        numentries = nr_kernel_pages;
        numentries += (1UL << (20 - PAGE_SHIFT)) - 1;

        numentries >>= 20 - PAGE_SHIFT;
        numentries <<= 20 - PAGE_SHIFT;
        if (scale > PAGE_SHIFT)
            numentries >>= (scale - PAGE_SHIFT);
        else
            numentries <<= (PAGE_SHIFT - scale);

        if (unlikely((numentries * bucketsize) < PAGE_SIZE))
            numentries = PAGE_SIZE / bucketsize;
    }
    numentries = roundup_pow_of_two(numentries);                          ❷
    ...
    log2qty = ilog2(numentries);                                          ❸

    do {                                                                  ❹
        size = bucketsize << log2qty;                                     ❺

        if (flags & HASH_EARLY)                                           ❻
            table = alloc_bootmem_nopanic(size);
        else if (hashdist)
            table = __vmalloc(size, GFP_ATOMIC, PAGE_KERNEL);
        else {
            unsigned long order = get_order(size);
            table = (void*) __get_free_pages(GFP_ATOMIC, order);

            if (table) {
                unsigned long alloc_end = (unsigned long)table +
                    (PAGE_SIZE << order);
                unsigned long used = (unsigned long)table +
                PAGE_ALIGN(size);
                split_page(virt_to_page(table), order);
                while (used < alloc_end) {
                    free_page(used);
                        used += PAGE_SIZE;
                    }
                }
            }
    } while (!table && size > PAGE_SIZE && --log2qty);

    ...

    return table;
}
```

先观察代码段❶。该函数在开始阶段确认接收的函数参数numentries值。希望各位记住之前的dcache_init_early()函数中用numentries值传递了散列表的项数。该条件语句乍一看有些复杂，其作用却十分简单。若接收的numentries值为0，则以当前系统中的页数和页大小为基础，将numentries值设为合适的值。然后通过代码段❷和代码段❸，（为方便以后的运算）四舍五入让numentries值成为2的指数幂，并求出以2为底的对数值。

代码段❹中出现的循环语句（do ~ while()）参考之前决定的numentries值和当前系统的内存状况分配散列表。如果bootmem管理器是最先通过代码段❻执行内存分配的当前时点的内存管理器，则传递HASH_EARLY标签，从bootmem分配器接收散列表。

当前的内核流程是通过dcache_init_early()调用的，若不是64位NUMA系统，通常会执行现在的条件语句。如果hashdist的值为“真”，则使用伙伴分配器。通过vmalloc()接收散列表，这由后面要介绍的dcache_init()函数执行。上述2种情况均失败时，就通过__get_free_pages()接受分配。

19.11 初始化 cpuset 子系统——cpuset_init_early()

结构简单的电脑中，即使没有精确控制进程要使用的CPU和内存，也能充分保障其性能。但系统越大、内存结构越复杂，越需要更加精确分离的系统资源分配策略。

原因在于，许多进程同时访问多个CPU和内存时发生的竞争，或访问内存时浪费的时间随系统增大而按比例增加。cpuset为了让每个进程集合能够在已定的CPU和内存资源中运行而加以限制，能够有效降低随着系统增大而增加的系统开销。

cpuset是将CPU和内存资源以逻辑性和层次性集成的一种机制。这是cgroup中使用的子系统之一，向由cgroup集成的进程集合适用cpuset后，属于cpuset的进程只能在已定的CPU中运行，也只能访问已定的内存。

由于和cgroup共同使用，因此cpuset自然具有层次结构，各节点表现为cpuset结构体。在cpuset_init_early()中初始化指向系统顶层cpuset的top_cpuset，观察下列函数。

:: 代码19-17 kernel/cpuset.c的cpuset_init_early()

```
int __init cpuset_init_early(void)
{
    alloc_bootmem_cpumask_var(&top_cpuset.cpus_allowed);                ❶
    top_cpuset.mems_generation = cpuset_mems_generation++;              ❷
    return 0;
}
```

如前所述，cpuset为进程集合指定CPU和内存，该集合仅使用指定的CPU和内存。记录此时可用CPU的位图就是cpuset结构体的成员变量cpus_allowed[①]。在代码段❶中，有必要的情况下为cpus_allowed分配内存。

① top_cpuset的cpus_allowed和用于记录系统联机状态的CPU的cpu_online_map保持一致，因发生热插拔事件而变更CPU状态时也一样。

虽然此处并不另行设置，但cpuset结构体不仅具有cpus_allowed，还具有记录可用（可接受分配的）内存节点的mems_allowed位图。属于该cpuset的进程只能在该位图中记录的节点发送分配请求。

在属于该cpuset的进程结构体中，cpuset的内存策略也被另外保存为数据结构。因此，其中1个内存策略发生变更时，需要采取措施使策略保持一致，此时使用cpuset的mems_generation成员变量和进程的cpuset_mems_generation。

在代码段❷中，将top_cpuset.mems_generation初始化为cpuset_mems_generation，并将cpuset_mems_generation加1。属于top_cpuset的进程以后比较自身的cpuset_mems_generation值和top_cpuset.mems_generation值后，就与cpuset的内存策略执行同步。



补充知识点　利用成员变量的cpuset和进程间内存策略同步机制

如果cpuset的mems_allowed变更，则在变更后的cpuset的mems_generation中保存全局变量cpuset_mems_generation的值，并将cpuset_mems_generation加1。此时，在属于cpuset的进程角度上讲，进程的cpuset_mems_generation和cpuset的mems_generation会变得不一样。这表示需要将进程的内存策略与cpuset执行同步。

就这样，如果内存策略发生变更，变更事项并不直接适用于属于cpuset的任务，而是在伙伴分配器分配内存时最终调用的__alloc_pages_internal()中检查发送分配请求的进程的cpuset_mems_generation，并执行更新。

cpuset的内存策略具有代表性的变更情况如下所示。

(1) 新建cpuset时

(2) 已变更NUMA策略相关标签时

(3) 已变更mems_allowed位图时

发生变更的具有代表性的函数如下所示。

(1) cpuset_init()

(2) cpuset_create()

(3) cpuset_write_u64()

(4) update_nodemask()

19.12　初始化内存子系统——page_cgroup_init()

内存子系统限制属于特定cgroup的进程的内存使用，自动生成由集合内的进程使用的内存资源报告。为了使用cgroup的子系统之一——内存，该函数对系统所有联机节点执行循环，同时分配全部页数大小的page_cgroup结构体并进行初始化。不使用内存子系统时不执行任何操作，直接结束。

:: 代码19-18 mm/page_cgroup.c的page_cgroup_init()

```
void __init page_cgroup_init(void)
{
    int nid, fail;
    if (mem_cgroup_disabled())                                          ❶
        return;

    for_each_online_node(nid)  {                                        ❷
        fail = alloc_node_page_cgroup(nid);
        if (fail)
            goto fail;
    }

    printk(KERN_INFO "allocated %ld bytes of page_cgroup\n", total_usage);   ❸
    printk(KERN_INFO "please try cgroup_disable=memory option if you"
                     " don't want\n");
    return;
fail:
    printk(KERN_CRIT "allocation of page_cgroup was failed.\n");
    printk(KERN_CRIT "please try cgroup_disable=memory boot option\n");
    panic("Out of memory");
}
```

在代码段❶中查看内存子系统是否关闭。[①]若已关闭则不执行任何操作，直接结束。

在代码段❷中对联机状态的节点执行循环，同时按照不同节点分配节点的全部页数（包括空洞）大小的page_cgroup结构体，并进行初始化。若内存分配成功，则在代码段❸中输出为page_cgroup分配的内存大小并结束；反之则输出错误信息，引发内核恐慌（panic）。

下面仔细观察实际分配page_cgroup时使用的alloc_node_page_cgroup()。

:: 代码19-19 mm/page_cgroup.c的alloc_node_page_cgroup()

```
static int __init alloc_node_page_cgroup(int nid)
{
    struct page_cgroup *base, *pc;
    unsigned long table_size;
    unsigned long start_pfn, nr_pages, index;

    start_pfn = NODE_DATA(nid)->node_start_pfn;
    nr_pages = NODE_DATA(nid)->node_spanned_pages;

    if (!nr_pages)
        return 0;

    table_size = sizeof(struct page_cgroup) * nr_pages;
    base = __alloc_bootmem_node_nopanic(NODE_DATA(nid),                 ❶
                table_size, PAGE_SIZE, __pa(MAX_DMA_ADDRESS));
```

① mem_cgroup_disabled()参考mem_cgroup_subsys的disable成员变量判断激活与否，该变量为全局变量，因此disable被默认初始化为0，mem_cgroup_disabled()返回假。

```
    if (!base)
        return -ENOMEM;

    for (index = 0; index < nr_pages; index++) {                    ❷
        pc = base + index;
        __init_page_cgroup(pc, start_pfn + index);
    }
    NODE_DATA(nid)->node_page_cgroup = base;
    total_usage += table_size;
    return 0;
}
```

在该函数中参照相应节点的node_spanned_pages求出节点具有的全部页数（包括空洞）。利用求出的全部页数在代码段❶中求出节点的全部page_cgroup结构体的大小，然后分配内存。在代码段❷中，对之前生成的所有page_cgroup结构体执行循环，同时初始化结构体的变量。

最后，将分配的内存起始地址保存到节点的node_page_cgroup，并将分配的内存大小加到total_usage变量。完成对所有节点的分配后，total_usage输出分配的总内存大小。

第20章 终止bootmem分配器并替换为伙伴系统

系统启动阶段使用bootmem内存分配器，Linux中由slab分配器执行bootmem的功能。但是，slab分配器是在伙伴系统中实现的，因此，原本由bootmem管理的内存必须由伙伴系统管理。本章要讲解的mem_init()函数就帮助后者完成这项任务。

20.1 mem_init()函数的调用关系及其与数据结构的相互关系

mem_init()函数和初始化bootmem的函数一样具有十分复杂的结构，函数的调用关系如图20-1所示。其中最核心的部分就是代码20-9的__free_one_page()函数。

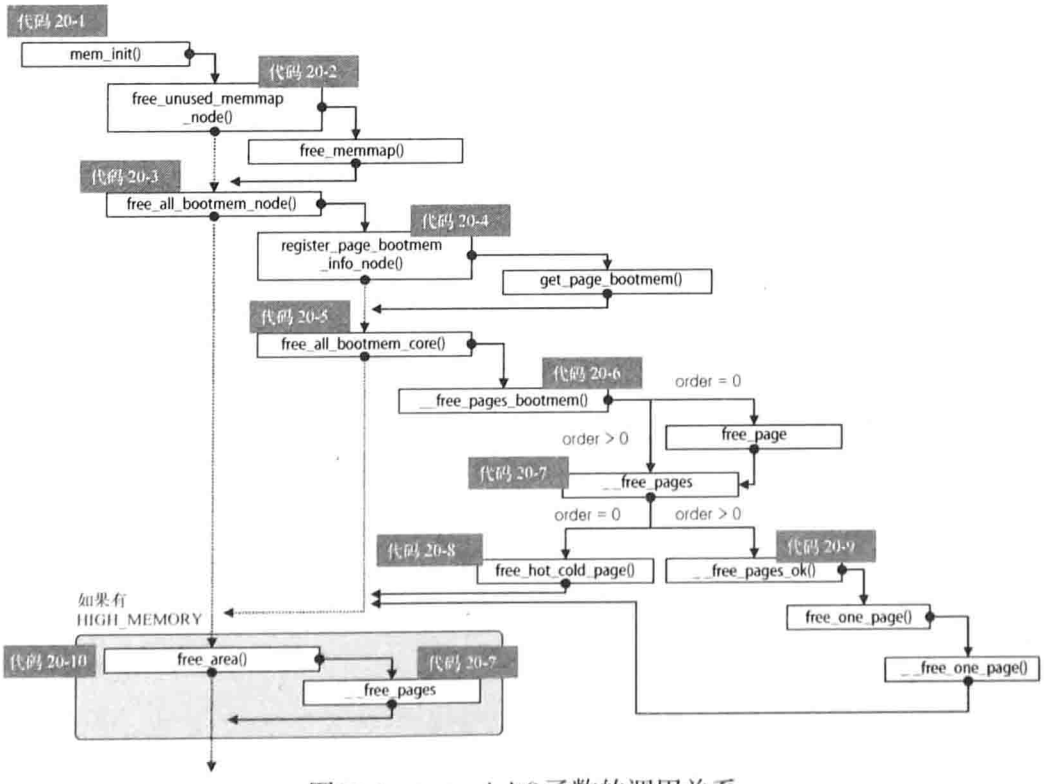


图20-1 mem_init()函数的调用关系

本章要重点了解的伙伴系统并不是额外的系统，zone结构体的成员变量free_area[]数组正是构建伙伴的核心数据结构。图20-2在我们目前了解的内存相关结构体之间的关系图中表示free_area[]区域。mem_init()函数将bootmem管理过的空白页在free_area[]结构体中按顺序（order）列表进行管理。

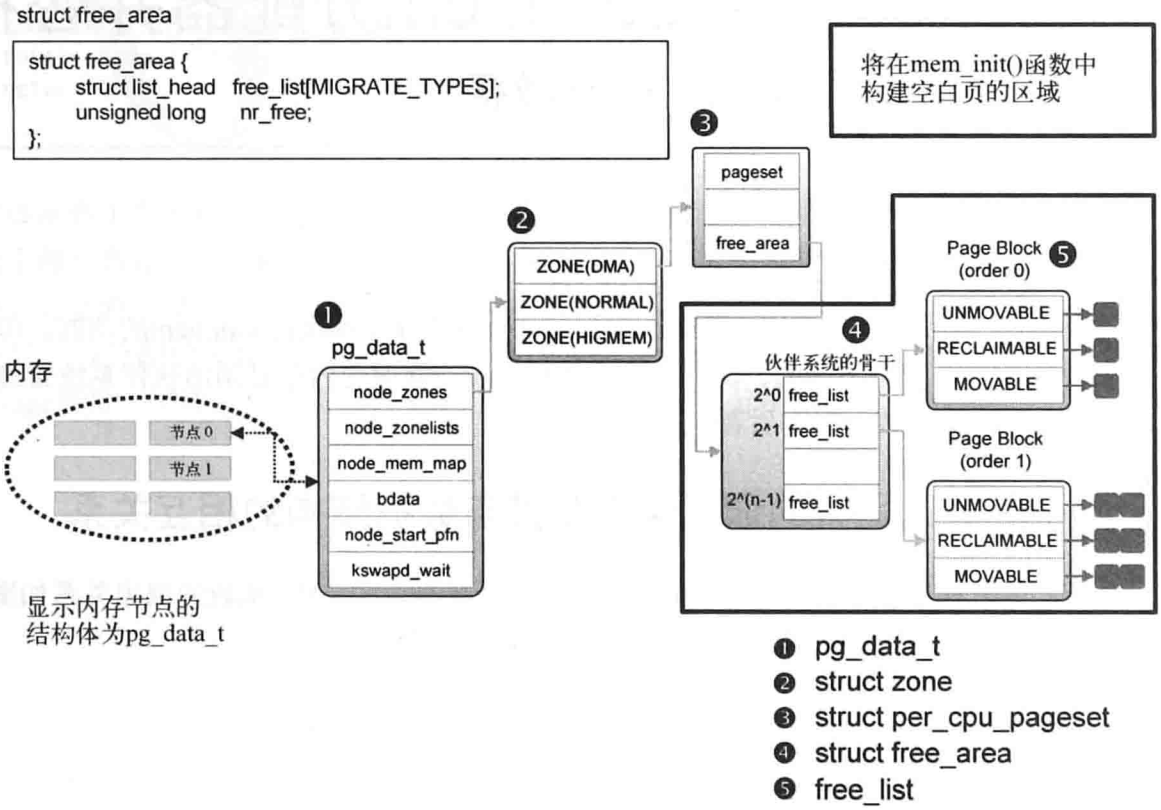


图20-2　在mem_init()函数中构建的区域（伙伴系统的空白页列表）

20.2　查看 mem_init()结构

下面通过分析代码正式了解其运行原理。代码20-1是mem_init()函数。

:: 代码20-1　arch/arm/mm/init.c的mem_init()

```
void __init mem_init(void)
{
    ...

    for_each_online_node(node) {
    pg_data_t *pgdat = NODE_DATA(node);

    free_unused_memmap_node(node, &meminfo);                    ❶

    if (pgdat->node_spanned_pages != 0)
```

```
        totalram_pages += free_all_bootmem_node(pgdat);                    ❷
    }
    ...
#ifdef CONFIG_HIGHMEM
    /*在高端内存页中设置空白*/
    for_each_online_node(node) {
        for_each_nodebank (i, &meminfo, node) {
            unsigned long start = bank_pfn_start(&meminfo.bank[i]);
            unsigned long end = bank_pfn_end(&meminfo.bank[i]);
            if (start >= max_low_pfn + PHYS_PFN_OFFSET)
                totalhigh_pages += free_area(start, end, NULL);            ❸
        }
    }
    totalram_pages += totalhigh_pages;
#endif

    ...

    }
}
```

mem_init()函数大体分为3个区域。

在代码段❶中调用free_unused_memmap_node()函数，将物理上不存在的页（hole）在页管理位图中全部记录为“不使用”。free_unused_memmap_node()函数将在20.3节再次讲解。

在页管理位图中记录为“不使用”后，代码段❷中调用free_all_bootmem_node()函数进行释放，使其能够在伙伴系统中管理空白页。free_all_bootmem_node()函数将在20.4节再次讲解。

在代码段❸中将高端内存区域释放到伙伴系统，使其能够管理空白页。调用free_area()函数执行这项操作，最终在代码段❷和代码段❸内部调用__free_pages()函数，执行相同逻辑。free_area()函数定义如下。

:: arch/arm/mm/init.c

```
static inline int free_area(unsigned long pfn, unsigned long end, char *s)
{
    unsigned int pages = 0, size = (end - pfn) << (PAGE_SHIFT - 10);

    for (; pfn < end; pfn++) {
        struct page *page = pfn_to_page(pfn);
        ClearPageReserved(page);        /*清除page->flags的PG_reserved位*/

        init_page_count(page);          /*page->_count = 0 */

        __free_page(page);              /*释放到伙伴系统，参考代码8;
                                          在内部调用__free_pages()函数*/

        pages++;                        /*下一页*/
    }
    ...
    return pages;
}
```

free_area()函数在内部调用__free_pages()函数，将空白页和一般内存区域共同释放到伙伴系统。

20.3　记录到不存在的内存位图——free_unused_memmap_node()

通过代码20-2了解free_unused_memmap_node()函数，在bootmem按照不同节点管理的页管理位图中，该函数将不使用的页（BANK之间的空白空间）设置为空白页。

:: 代码20-2　arch/arm/mm/init.c的free_unused_memmap_node()

```
static void __init free_unused_memmap_node(int node, struct meminfo *mi)
{
    ...
    for_each_nodebank(i, mi, node) {
        struct membank *bank = &mi->bank[i];
        bank_start = bank_pfn_start(bank);                              ❶
        if (bank_start < prev_bank_end) {
            printk(KERN_ERR "MEM: unordered memory banks.  "
                "Not freeing memmap.\n");
            break;
        }

        if (prev_bank_end && prev_bank_end != bank_start)               ❷
            free_memmap(node, prev_bank_end, bank_start);

        prev_bank_end = bank_pfn_end(bank);                             ❸
    }
}
```

在代码段❶中通过bank_pfn_start()和bank_pfn_end()宏获得BANK的起始页帧号，其定义如下所示。

:: arch/arm/include/asm/setup.h

```
#define bank_pfn_start(bank)  __phys_to_pfn((bank)->start)
#define bank_pfn_end(bank)    __phys_to_pfn((bank)->start + (bank)->size)
```

在代码段❷中，检查前一个BANK的末尾（prev_bank_end）和BANK的起始页帧号（bank_start）是否一致。如果不一致，则以前的内存和现在的内存之间存在空间。因此，对不使用的空间调用free_memmap()函数，在节点管理的页管理位图中将该页帧位设置为0。

free_memmap()函数的定义如下所示，其核心函数为free_bootmem_node()（15.4.7节）。

:: arch/arm/mm/init.c

```
static inline void
free_memmap(int node, unsigned long start_pfn, unsigned long end_pfn)
{
```

```
    struct page *start_pg, *end_pg;
    unsigned long pg, pgend;

    /*变更为页指针*/
    start_pg = pfn_to_page(start_pfn);
    end_pg = pfn_to_page(end_pfn);

    /*以页单位排列*/
    pg = PAGE_ALIGN(__pa(start_pg));
    pgend = __pa(end_pg) & PAGE_MASK;

    if (pg < pgend)
        free_bootmem_node(NODE_DATA(node), pg, pgend - pg);
}
```


分析bootmem_init()时讲解过该函数，在页管理位图中记录“相应页可分配”。

代码20-2的代码段❸和当前BANK的尾页帧号 + 1的效果相同。

20.4 移交至普通空白页伙伴系统——free_all_bootmem_node()

下面通过代码20-3了解free_all_bootmem_node()函数。

:: 代码20-3 mm/bootmem.c的free_all_bootmem_node()

```
unsigned long __init free_all_bootmem_node(pg_data_t *pgdat)
{
    register_page_bootmem_info_node(pgdat);                    ❶

    return free_all_bootmem_core(pgdat->bdata);                ❷
}
```

在代码段❶中调用register_page_bootmem_info_node()函数，对具有元信息（附加信息）的各页指定属性，表示该页具有元信息。例如，在具有节点描述符pglist_data结构体的页中，指定该页包含节点信息。

除节点信息外，元信息还具有（以页的集合——节区为单位管理页时）节区初始页信息。register_page_bootmem_info_node()函数将在20.4.1节再次讲解。

在代码段❷中调用free_all_bootmem_core()函数，将节点描述符的成员变量pglist_data->bdata作为参数进行传递。pglist_data->bdata是bootmem_data结构体类型，而bootmem_data是系统启动时在内存分配器bootmem中管理内存的结构体。

bootmem_data结构体中设置的所有数据现在要移到伙伴系统，free_all_bootmem_core()函数执行这项操作（20.4.2节）。

20.4.1 register_page_bootmem_info_node()

代码20-4是在页中设置元信息的register_page_bootmem_info_node()函数。

:: 代码20-4　mm/memory_hotplug.c的register_page_bootmem_info_node()

```
void register_page_bootmem_info_node(struct pglist_data *pgdat)
{
    ...

    /*pgdlist_data占据的页数*/
    nr_pages = PAGE_ALIGN(sizeof(struct pglist_data)) >> PAGE_SHIFT;
    /*pglist_data所处页的page结构体地址*/
    page = virt_to_page(pgdat);

    for (i = 0; i < nr_pages; i++, page++)                          ❶
        get_page_bootmem(node, page, NODE_INFO);

    zone = &pgdat->node_zones[0];

    for (; zone < pgdat->node_zones + MAX_NR_ZONES - 1; zone++) {   ❷
        if (zone->wait_table) {
            nr_pages = zone->wait_table_hash_nr_entries
                * sizeof(wait_queue_head_t);
            nr_pages = PAGE_ALIGN(nr_pages) >> PAGE_SHIFT;
            page = virt_to_page(zone->wait_table);

            for (i = 0; i < nr_pages; i++, page++)
                get_page_bootmem(node, page, NODE_INFO);
        }
    }

    /*节点的起始pfn，最终pfn*/
    pfn = pgdat->node_start_pfn;
    end_pfn = pfn + pgdat->node_spanned_pages;

    for (; pfn < end_pfn; pfn += PAGES_PER_SECTION)                 ❸
        register_page_bootmem_info_section(pfn);

}
```

在代码段❶中使用节点描述符pglist_data结构体的大小求出所需页数，对相应页标记NODE_INFO。NODE_INFO的定义如下所示，表示页具有何种元信息。

:: include/linux/memory_hotplug.h

```
#define SECTION_INFO                    (-1 - 1)
#define MIX_SECTION_INFO                (-1 - 2)
#define NODE_INFO                       (-1 - 3)
```

那么，页的这些元信息保存到哪里呢？其答案可在get_page_bootmem()函数中找到。

:: include/linux/memory_hotplug.h

```
static void get_page_bootmem(unsigned long info,  struct page *page, int type)
```

```
{
    atomic_set(&page->_mapcount, type);
    SetPagePrivate(page);
    set_page_private(page, info);
    atomic_inc(&page->_count);
}
```

get_page_bootmem()函数调用set_page_private()函数，将值保存到page->private成员，并为引用计数（_count）加1。page->private成员变量的用途十分广泛，也用于记录伙伴系统的页顺序。

代码段❷中如果有zone->wait_table，则为保存相应列表的所有页，将NODE_INFO记录到page->private。zone->wait_table是等待相应区域的页处于可分配状态的各进程的等待队列（wait queue）。

代码段❸表示相应的页为节区起始，节区由多个页组成，因此，为节区起始页调用register_page_bootmem_info_section()函数进行处理。register_page_bootmem_info_section()函数在内部调用get_page_bootmem（node、page、SECTION_INFO）。

以上就是register_page_bootmem_info_section()函数，简言之，该函数在保存节点元信息的页中为page->private变量设置“相应页保存着元信息”。

20.4.2 free_all_bootmem_core()

下面分析free_all_bootmem_core()函数。free_all_bootmem_core()函数将节点描述符的成员变量pglist_data->bdata作为参数接收，pglist_data->bdata是bootmem_data结构体类型。bootmem_data是系统启动时在内存分配器bootmem中管理内存的结构体。

free_all_bootmem_core()函数将bootmem_data结构体中设置的所有数据移到伙伴系统。接下来通过代码20-5进行分析。

:: 代码20-5 mm/memory_hotplug.c的free_all_bootmem_core()

```
static unsigned long __init free_all_bootmem_core(bootmem_data_t *bdata)
{
    ...

    start = bdata->node_min_pfn;     /*起始页帧*/
    end = bdata->node_low_pfn;       /*末尾页帧*/
    aligned = !(start & (BITS_PER_LONG - 1));   /*是否以词单位进行排列*/

    ...

    while (start < end) {
        unsigned long *map, idx, vec;
        map = bdata->node_bootmem_map;       /*页管理位图*/
        idx = start - bdata->node_min_pfn;  /*要清空的页的索引*/
        vec = ~map[idx / BITS_PER_LONG];                                    ❶

        if (aligned && vec == ~0UL && start + BITS_PER_LONG < end) {        ❷
```

```
            int order = ilog2(BITS_PER_LONG);                                    ❸
            __free_pages_bootmem(pfn_to_page(start), order);                     ❹

            count += BITS_PER_LONG; /*已清空的页数*/
        } else {                                                                 ❺
            unsigned long off = 0;
            while (vec && off < BITS_PER_LONG) {
                if (vec & 1) {      /*对空白页*/
                    page = pfn_to_page(start + off);
                    __free_pages_bootmem(page, 0);
                    count++;
                }
                vec >>= 1;
                off++;
            }
        }
        start += BITS_PER_LONG;
    }

    /*查找用于页管理的页数*/
    pages = bdata->node_low_pfn - bdata->node_min_pfn;                           ❻

    pages = bootmem_bootmap_pages(pages);
    count += pages;

    while (pages--)                                                              ❼
        __free_pages_bootmem(page++, 0);

    ...

    return count;
}
```

先分析代码段❶。页管理位图由unsigned long[]数组构成，用于查找要清空（free）的页属于哪个元素。32位的BITS_PER_LONG显示类型long的位数，其定义如下所示。

:: arch/arm/include/asm/types.h

```
#define BITS_PER_LONG 32
```

如果idx=0，则第一个元素将成为map[0]；如果idx=1，则第二个元素将成为map[1]。对求出的值执行NOT(~)运算并保存到vec变量。例如，若map[0]的值为0，则表示共32位是0，即32个页都是空白页。因此，vec的值将成为0XFFFFFFFF（~MAP[0]）。

代码段❷检查释放页时是否1次释放多页。在重要条件中，vec==~OUL指vec全部为0xFFFFFFFF时才为“真”。即vec值为代码段❶中求出的值，具有32个空白页的情况下才具有0xFFFFFFFF值。

需要同时释放多页时执行代码段❸和代码段❹，否则执行代码段❺每次释放1页（顺序0）。

代码段❸将需要释放的页数变更为顺序单位，因为伙伴系统以顺序为单位管理各页。

例如，32个页为log_2^{32} = 5，即顺序变成5。

代码段❹中执行的代码__free_pages_bootmem()才是实际释放页的函数，以作为参数接收的顺序单位释放页。该函数实际上最重要，20.4.3节将再次讲解。

在代码段❻中求出保存页管理位图的页数，查找起始页并释放这些页。bootmem_bootmap_pages()函数是bootmem_init()函数（15.2节）的代码，用于获取保存页管理位图的页数。利用__free_pages_bootmem()函数释放获得的页管理位图的页数和起始页。

20.4.3 __free_pages_bootmem()

通过代码20-6分析以顺序单位释放页的__free_pages_bootmem()函数。

:: 代码20-6 mm/page_alloc.c的__free_pages_bootmem()

```
void __meminit __free_pages_bootmem(struct page *page, unsigned int order)
{
    if (order == 0) {  /*以1页为单位释放*/
        __ClearPageReserved(page);                                    ❶
        set_page_count(page, 0);     /* page->_count = 0 */
        set_page_refcounted(page);   /* page->_count = 1*/
        __free_page(page);                                            ❷
    } else {     /*以顺序为单位释放*/

        ...

        for (loop = 0; loop < BITS_PER_LONG; loop++) {
            struct page *p = &page[loop];                             ❸
            __ClearPageReserved(p);
            set_page_count(p, 0);
        }
        set_page_refcounted(page);                                    ❹

        __free_pages(page, order);                                    ❺
    }
}
```

代码段❶清空page->flags的PG_reserved位。__ClearPageReserved()函数由宏生成，对页标签执行位运算。设置页的使用计数器page->count=1。

根据接收的参数order值，将代码段❷的__free_page()函数和代码段❺的__free_pages()函数作为实际释放页的函数进行调用。但是__free_page()函数和__free_pages(page,0)相同，其定义如下所示。

:: include/linux/gfp.h

```
#define __free_page(page) __free_pages((page),0)
```

这样，__free_pages()函数就成为__free_pages_bootmem()函数的核心（20.4.4节）。

在代码段❸中，以顺序单位释放页时（此处和BITS_PER_LONG一样共32个页），清空各页page->flags的PG_reserved位。设置page->_count=0后，在代码段❹中只对第一页设置page->count=1。设置为1使页作为相应顺序的起始页处于使用状态，并避免在内存中被清除。



补充知识点　page->flags位

页标签（page->flags）值表示页属性，其定义如下所示。

:: include/linux/page-flags.h

```
enum pageflags {
    PG_locked,                  /*页被锁*/
    PG_error,                   /*I/O中发生错误*/
    PG_referenced,              /*正在使用页*/
    PG_uptodate,                /*从块设备中成功读取页*/
    PG_dirty,                   /*页的内容被修正*/
    PG_lru,                     /*lru列表中存在页*/
    PG_active,                  /*页使用活跃*/
    PG_slab,                    /*slab分配器的页*/
    PG_arch_1,                  /*从属结构的页状态位*/
    PG_reserved,                /*被预约的页，不可交换*/
    PG_private,                 /*page->private中有值*/
    PG_writeback,               /*页正被写入块设备*/
    PG_head,                    /*首（head）页*/
    PG_tail,                    /*尾（tail）页*/
    PG_compound,                /*由多页构成的混合页的一部分*/
    PG_swapcache,               /*用于交换的页（swap page）*/
    PG_mappedtodisk,            /*用于写入的块被分配①*/
    PG_reclaim,                 /*预计页将被回收*/
    PG_buddy,                   /*伙伴系统内的空白页*/
    PG_swapbacked,              /*swapbacked页*/

    ...

};
```

页属性不从属于结构或CPU类型，使用独立标签表示。②



补充知识点　宏定义的函数原理

__ClearPageReserved()函数负责清除并设置对应于页标签的位，它由宏定义。下面观察该函数如何定义及生成，相关内核代码如下所示。

① 为了将回写（writeback）到块设备时添加的页写入块设备，需要决定写入该页的块设备内的位置。如果已设置PG_mappedtodisk位，说明已决定用于写入的块设备内的位置。※参考文献：http://studyfoss.egloos.com/5512112

② 已设置PG_arch_1时，从属于结构。

:: include/linux/page-flags.h

```
#define __CLEARPAGEFLAG(uname, lname)                    \
static inline void __ClearPage##uname(struct page *page)        \
            { __clear_bit(PG_##lname, &page->flags); }

__CLEARPAGEFLAG(Reserved, reserved)
```

首先，__CLEARPAGEFLAG(Reserved, reserved)用__ClearPageReserved()替代__ClearPage##uname。使用##从接收的变量名参数制作函数名，也就是直接使用字符串制作函数名。用相同方法生成调整页标签的其他函数。

通过下列示例观察##的使用方法，在内核中就像这样利用##定义函数，所以搜索__ClearPageReserved()函数时无法找到定义该函数的地方，因为内核代码内根本就没有。

:: macro-test.c

```
#include <stdio.h>

#define __PAGEFLAG(uname) TESTPAGEFLAG(uname)    \
    __SETPAGEFLAG(uname)  __CLEARPAGEFLAG(uname)


#define TESTPAGEFLAG(uname)                      \
static inline void TestClearPage##uname()        \
{printf("function: %s\n",  __FUNCTION__);}


#define __SETPAGEFLAG(uname)                     \
static inline void SetPage##uname()              \
{printf("function: %s\n",  __FUNCTION__);}


#define __CLEARPAGEFLAG(uname)                           \
static inline void ClearPage##uname()                    \
{printf("function: %s\n",  __FUNCTION__);}

__PAGEFLAG(Reserved)

void main()
{
    TestClearPageReserved();
    SetPageReserved();
    ClearPageReserved();
}
```

运行结果

```
code@kernel$ gcc -o macro-test macro-test.c
code@kernel$
code@kernel$ ./macro-test
function: TestClearPageReserved
function: SetPageReserved
function: ClearPageReserved
code@kernel$
```

20.4.4　__free_pages()

下面分析__free_pages()函数，如代码20-7所示。

:: 代码20-7　mm/page_alloc.c的__free_pages()

```
void __free_pages(struct page *page, unsigned int order)
{
    if (put_page_testzero(page)) {                          ❶

        if (order == 0)
            free_hot_page(page);                            ❷
        else
            __free_pages_ok(page, order);                   ❸
    }
}
```

在代码段❶中，检查将页次数page->_count减1后的值是否为0。仅在page->_count=0时释放页。

代码段❷中顺序为0时，即释放1页时调用free_hot_page()函数。free_hot_page()函数在内部只调用free_hot_cold_page()函数（20.4.5节）。

在代码段❸中调用__free_pages_ok()函数，将页以顺序单位释放（20.4.6节）。

20.4.5　free_hot_cold_page()

下面通过代码20-8分析free_hot_cold_page()函数。

:: 代码20-8　mm/page_alloc.c的free_hot_cold_page()

```
static void free_hot_cold_page(struct page *page, int cold)
{
    struct zone *zone = page_zone(page);
    struct per_cpu_pages *pcp;

    ...

    pcp = &zone_pcp(zone, get_cpu())->pcp;                  ❶

    if (cold)                                               ❷
        list_add_tail(&page->lru, &pcp->list);
```

```
    else
        list_add(&page->lru, &pcp->list);

    ...

}
```

代码段❶的zone_pcp()函数获得对应于get_cpu()的CPU的页列表zone->pageset->pcp。[①]这是不同CPU具有的1页列表，这些页不会直接返回伙伴系统，而是持有不同CPU的页以快速执行分配（16.3节）。

在代码段❷中传递参数cold值时，在pcp->list的末尾添加相应页。若不是cold而是hot，则直接插入pcp->list头部之后。[②]图20-3表示list_add_tail()函数和list_add()函数的执行原理。

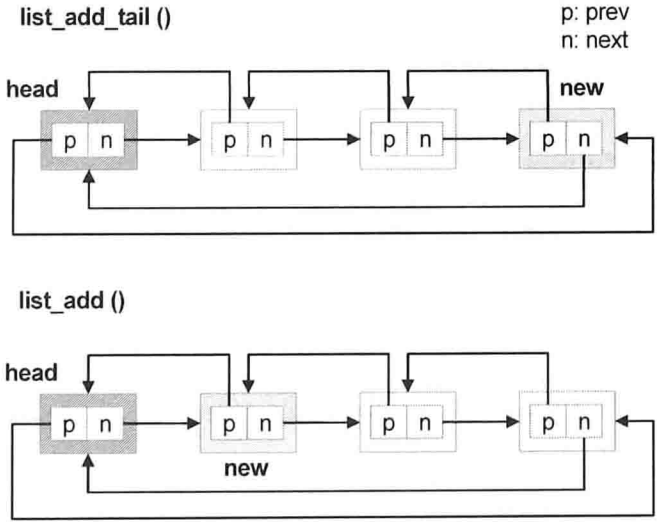


图20-3 通过list_add_tail()函数和list_add()函数添加项目

20.4.6 __free_pages_ok()

__free_pages_ok()函数以顺序单位释放页，要经过几个阶段的函数调用，其调用关系如下所示。

```
__free_pages_ok() → free_one_page() → __free_one_page()
```

最终释放页的函数是__free_one_page()函数，通过代码20-9分析一下。

:: 代码20-9 mm/page_alloc.c的__free_pages_ok()

```
static void __free_pages_ok(struct page *page, unsigned int order)
{
```

① pcp是per cpu pages的简称。
② 更准确地说，向pcp->list添加页列表。

```
    ...

    free_one_page(page_zone(page), page, order);
}

static void free_one_page(struct zone *zone, struct page *page, int order)
{
    ...

    __free_one_page(page, zone, order);
}

static inline void __free_one_page(struct page *page,
        struct zone *zone, unsigned int order)
{
    unsigned long page_idx;
    int order_size = 1 << order;
    int migratetype = get_pageblock_migratetype(page);                    ❶
    ...

    page_idx = page_to_pfn(page) & ((1 << MAX_ORDER) - 1);                ❷

    ...

    while (order < MAX_ORDER-1) {
        unsigned long combined_idx;
        struct page *buddy;

        buddy = __page_find_buddy(page, page_idx, order);                 ❸

        if (!page_is_buddy(page, buddy, order))                           ❹
            break;

        list_del(&buddy->lru);                                            ❺
        zone->free_area[order].nr_free--;
        rmv_page_order(buddy);

        combined_idx = __find_combined_index(page_idx, order);            ❻
        page = page + (combined_idx - page_idx);
        page_idx = combined_idx;

        order++;                                                          ❼
    }

    set_page_order(page, order);                                          ❽

    list_add(&page->lru,                                                  ❾
        &zone->free_area[order].free_list[migratetype]);
    zone->free_area[order].nr_free++;
}
```

在代码段❶中获得页的迁移类型，即移动类型，因为伙伴系统的骨干free_list[]数组按迁移类型管理页（17.8节）。

在代码段❷中求出page_idx值，将内存分为2^{MAX_ORDER}单位的页块时，该值显示对应于2^{MAX_ORDER}个页中第几页的索引。伙伴系统以顺序单位管理页，最多将2^{MAX_ORDER}个页捆绑为1个可用块。因此，某页要想找到与自身统一的伙伴，需要自己所属的2^{MAX_ORDER}页内的索引，该索引就是page_idx，（参考图20-4）。

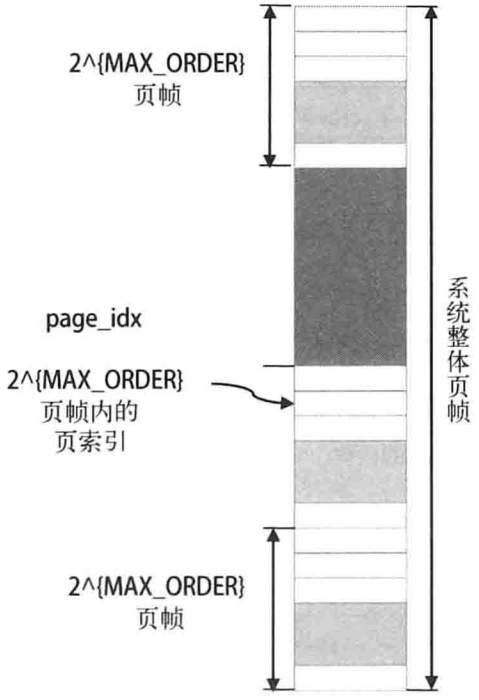


图20-4 内存分割与已分割内存中的索引

在代码段❸中以page_idx为基准查找接收的参数page的伙伴，查找算法将在“补充知识点”中详细讲解。

在代码段❹中，由于不存在伙伴时无需与伙伴统一，因此退出循环。

在代码段❺中从伙伴所属列表取出伙伴，并将伙伴所属列表的空白页块的值zone->free_area[order].nr_free减少1个。此外，利用rmv_page_order()函数删除保存伙伴顺序信息的page->private值。

在代码段❻中求出与伙伴统一后使用的新索引combined_idx。根据page_idx和buddy值的不同，combined_idx可能小于等于page_idx值，具体原因参见“补充知识点”。

求出与伙伴统一的combined_idx后，将该值保存到page_idx。在代码段❼中增加1个顺序，并检查有无可统一的伙伴。若可以统一，则继续执行相同任务直到MAX_ORDER。

在代码段❽中调用set_page_order()函数，将顺序记录到相应顺序第一页的page->private变量。此处的顺序可能是未能找到要统一的伙伴而调用__free_one_page()函数时传递的顺序值参数，也可能是找到要统一的伙伴时增加的顺序值。

在代码段❾中，通过链表将该页链接到伙伴系统的核心free_area[order].free_list[]数组。free_list[]数组按页的迁移类型进行区分并管理，这样可大量分配类型相同的页，并防止内存碎片。



补充知识点　查找伙伴

Linux内核中查找伙伴的算法由十分简单的代码构成，但理解起来并不容易。如图20-5所示给定page_idx时，查看page_idx的伙伴位置。

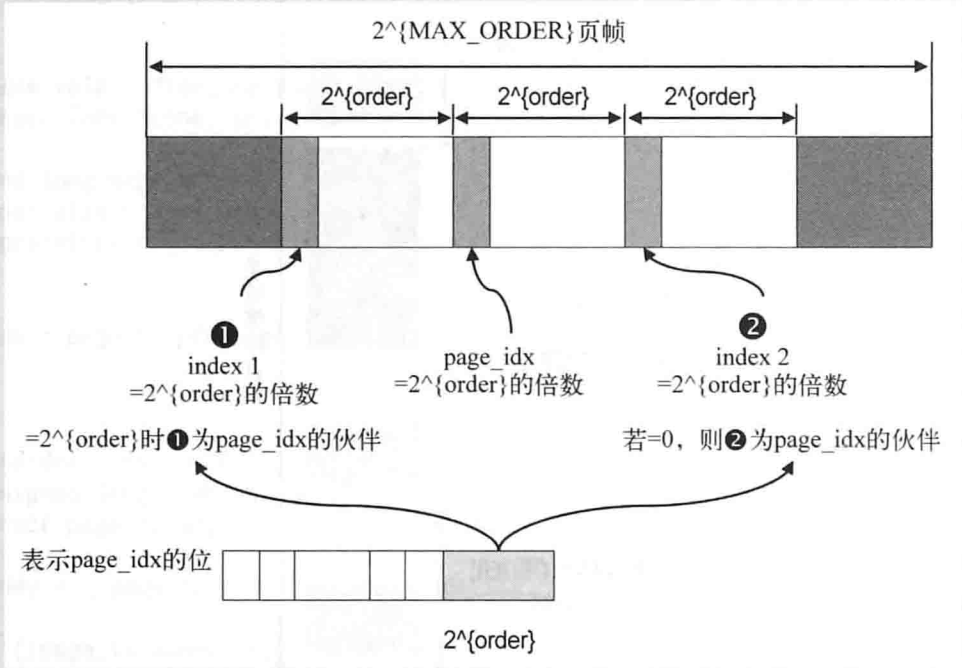


图20-5　page_idx及其可能的伙伴位置

将内存以2^{MAX_ORDER}单位分割时，page_idx是2^{MAX_ORDER}个页帧内的索引。而对应于page_idx的页将成为相应顺序的第一页，因为统一或查找伙伴时，将该顺序的第一页作为对象以检查顺序。其余页不具备顺序的信息（page->private）。

如图20-5所示，以page_idx为基准，距-2^{order}大小的(1)或者距+2^{order}大小的(2)将成为page_idx的伙伴。给定page_idx值时该如何选择伙伴呢？该答案可通过page_idx值确定。由于page_idx是顺序的第一个页帧号，所以page_idx值将是顺序的倍数。

用位表现的话，在page_idx值中对应于下级顺序的位被设置为顺序值或变成0。例如，order=2时，对应于page_idx的顺序值将成为下列二者之一。

```
page_idx = 0b????????????000     2^{order} = 2^{2} = 0b100
page_idx = 0b????????????100     2^{order} = 2^{2} = 0b100
```

在page_idx值中，对应于顺序的下级位均为0时，需要与距+2^{order}的伙伴统一；对应于顺序的下级位与顺序相同时，需要与距-2^{order}的前方的伙伴统一。

最终原理是：如果对应于page_idx值顺序的下级位与顺序相同，则减少该顺序大小；如果下级位为0，则增加该顺序大小。执行该任务的就是XOR运算。例如，order=2时，按如下所示查找对应于2种page_idx的伙伴。

```
0b??????????????000的伙伴        = 0b??????????????000 ^ 0b100
                                  = 0b??????????????100
0b??????????????100的伙伴        = 0b??????????????100 ^ 0b100
                                  = 0b??????????????000
```

该原理在Linux中如下所示。

:: mm/page_alloc.c

```
static inline struct page *__page_find_buddy(struct page *page,
unsigned long page_idx, unsigned int order)
{
    unsigned long buddy_idx = page_idx ^ (1 << order);
    return page + (buddy_idx - page_idx);
}
```

通过图20-6分别查看page_idx=1且order=0时、page_idx=8且order=1时的情况。

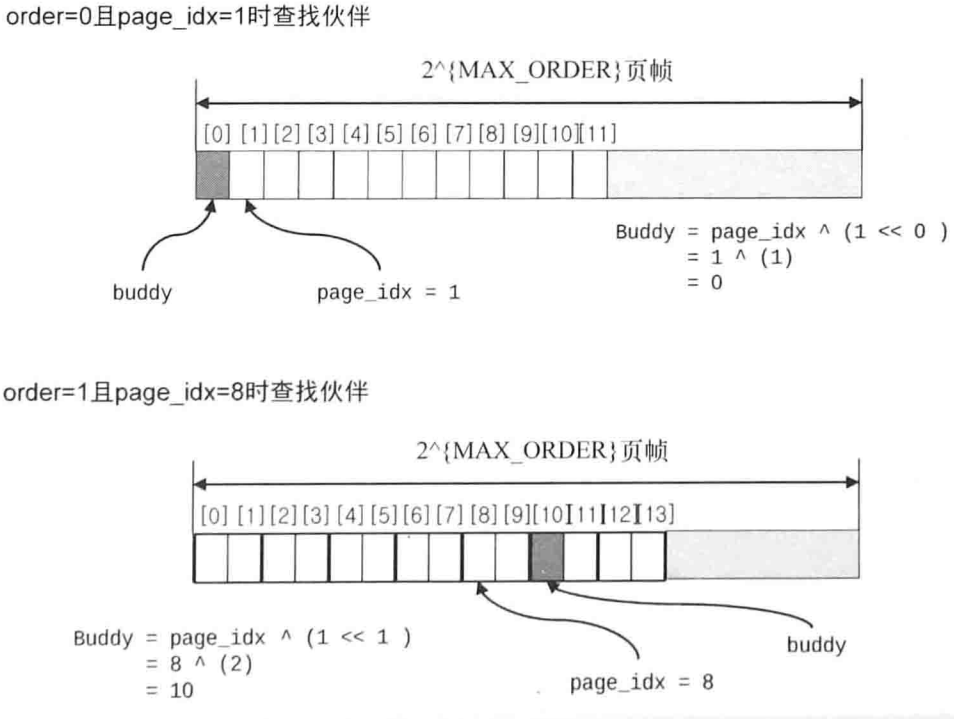


图20-6 order=0且page_idx=1时、order=1且page_idx=8时的伙伴

20.5　移交到高端内存空白页伙伴系统——free_area()

再次通过代码20-1的代码段❸分析free_area()函数。

```
#ifdef CONFIG_HIGHMEM
    /*在高端内存页中设置空白*/
    for_each_online_node(node) {
        for_each_nodebank (i, &meminfo, node) {
            unsigned long start = bank_pfn_start(&meminfo.bank[i]);
            unsigned long end = bank_pfn_end(&meminfo.bank[i]);
            if (start >= max_low_pfn + PHYS_PFN_OFFSET)
                totalhigh_pages += free_area(start, end, NULL);                    ❸
        }
    }
    totalram_pages += totalhigh_pages;
#endif
```

存在高端内存时，该代码求出高端内存的起始页帧和尾页帧号，然后调用free_area()函数使伙伴系统管理空白页。

free_area()函数在内部调用__free_page()函数，将空白页和一般内存区域共同释放到伙伴系统。代码20-10是free_area()函数的定义。

:: 代码20-10　arch/arm/mm/init.c的free_area()

```
static inline int free_area(unsigned long pfn, unsigned long end, char *s)
{
    unsigned int pages = 0, size = (end - pfn) << (PAGE_SHIFT - 10);

    for (; pfn < end; pfn++) {
        struct page *page = pfn_to_page(pfn);
        ClearPageReserved(page);    /*清空page->flags的PG_reserved位*/

        init_page_count(page);      /*page->_count = 0 */

        __free_page(page);          /*向伙伴系统释放，参考代码20-7
                                      在内部调用__free_pages()函数*/

        pages++;                    /*下一页*/
    }

    ...

    return pages;
}
```

本章的mem_init()函数结束启动时的内存分配器bootmem，在伙伴系统的free_list[]数组将bootmem管理的所有空白页以顺序单位构建列表。构建好的伙伴系统将为Linux的内存分配器slab分配器提供空白页。

第21章 初始化以支持CPU热插拔

本章讲解CPU热插拔（hotplug），该功能即使在SMP系统运行（running）过程中也能挂起（suspend）并恢复（resume）CPU运行。

21.1 初始化 cpu_hotplug 成员变量——cpu_hotplug_init()

cpu_hotplug_init()函数初始化CPU热插拔相关结构体，了解该函数前先了解支持CPU热插拔功能的设置。

在Linux中支持CPU热插拔功能需要设置Kconfig的HOTPLUG_CPU。当然，使用CPU热插拔功能还需要支持SMP。

:: arch/arm/Kconfig

```
config HOTPLUG_CPU
    bool "Support for hot-pluggable CPUs (EXPERIMENTAL)"
    depends on SMP && HOTPLUG && EXPERIMENTAL
    help
      Say Y here to experiment with turning CPUs off and on.
      CPUs can be controlled through /sys/devices/system/cpu.
```

CPU热插拔功能处于激活状态时，可访问/sys/devices/system/cpu目录控制CPU。例如，要将处于联机状态的1号CPU变更为脱机状态，可按如下所示修改联机文件的值，从而将CPU1动态地变更为脱机。[①]

```
code@kernel$ echo 0 > /sys/devices/system/cpu/cpu1/online
```

当前不需要执行CPU热插拔功能，可通过初始化CPU热插拔相关结构体的cpu_hotplug_init()函数了解其概念。

① 变更CPU的联机/脱机状态都是按照逻辑变更的。为了添加/删除实际物理CPU，供应商要同时提供BIOS（Basic Input Output System，基本输入输出系统）和添加/删除功能。

:: 代码21-1　kernel/cpu.c的cpu_hotplug_init()

```
void __init cpu_hotplug_init(void)
{
    cpu_hotplug.active_writer = NULL;
    mutex_init(&cpu_hotplug.lock);
    cpu_hotplug.refcount = 0;
}
```

cpu_hotplug_init()函数将初始化管理CPU热插拔的结构体cpu_hotplug的成员变量，cpu_hotplug结构体的静态定义如下所示。

:: kernel/cpu.c

```
static struct {
    struct task_struct *active_writer;          /*使用中的任务*/
    struct mutex lock;                          /*锁住对refcount成员的访问*/
    int refcount;
} cpu_hotplug;
```

active_writer是正在积极使用CPU的任务指针，refcount成员变量用于显示能否执行CPU热插拔命令。refcount值为0时可执行CPU热插拔相关命令，执行过程中无法根据互斥体锁（cpu_hotplug.lock）变更refcount值。[①]

21.2　CPU的联机→脱机转换处理

接下来了解CPU从联机变更为脱机时发生哪些事件、要执行哪些操作。

- **从联机变更为脱机时发生的事件**
  发生CPU_DOWN_PREPARE事件或CPU_DOWN_PREPARE_FROZEN事件。各任务冻结（frozen）状态下，CPU要脱机时发生CPU_DOWN_PREPARE_FROZEN事件。
- **执行的操作**
  - 需要将脱机CPU中执行的所有进程移动到新CPU（利用各进程的cpuset选择）。
  - 使新CPU中处理脱机CPU挂载的所有中断。
  - 使应当在脱机CPU中运行的计时器、底半部、tasklet在新CPU中运行。
  - 将脱机CPU中执行的所有服务移动到新CPU，然后调用__cpu_disable()函数关闭CPU。
- **完成脱机时发生的事件**
  成功完成所有操作后，将发生CPU_DEAD或CPU_DEAD_FROZEN事件。

① cpu_hotplug.refcount值调用kernel/cpu.c定义的get_online_cpus()函数以增加，用put_online_cpus()函数以减少。



补充知识点 顶半部（top half）、底半部、softirq、tesklet、工作队列

顶半部、底半部、softirq、tesklet、工作队列等术语均与中断有关。这些术语十分混乱，理解这些概念前先要理解“顶半部”和“底半部”的概念。

发生中断时需要调用处理中断的处理器，由于中断处于时间临界（time critical）的情况较多，且不在进程上下文中执行，因此具有不被中断的特性。要想在中断处理器中同时处理所有中断，就需要检测系统性能是否低下，故分2阶段处理中断。

第一阶段为顶半部，第二阶段为底半部。顶半部执行需要在发生中断时迅速反应的时间临界任务，该任务在中断处理器中执行。不在中断处理器中执行的其余任务在底半部执行。

将顶半部不执行的任务移交给底半部，这些任务又称“延迟任务”。所以把处理中断时的2个区域分别称为顶半部和底半部。

如果顶半部为中断处理器，那么需要另一种中断处理器处理对应于底半部区域的工作。实现底半部的就是softirq、tesklet、工作队列。[①] 这3个底半部的不同之处整理如表21-1所示。

表21-1 底半部的种类

底半部	特　征
softirq	■ 已考虑到线程（thread）且安全时 ■ 发生时间临界且常用时 ■ 编译时静态生成 ■ 在中断上下文中执行 ■ 激活所有中断时执行 ■ 相同softirq可在不同处理器中执行 ■ 对相同softirq提供共享数据锁机制 ■ 无法睡眠（sleep）
tasklet	■ 通过softirq实现 ■ 未考虑线程时 ■ 静态或动态生成 ■ 在中断上下文中执行 ■ 激活所有中断时执行 ■ 相同tasklet无法在不同处理器中执行 ■ 无需提供共享数据锁机制 ■ 无法睡眠
workqueue	■ 将延迟任务移交给内核线程，基于内核线程实现 ■ 在进程上下文中执行 ■ 可调度并睡眠

① Linux内核2.4版前曾有BH（Bottom Half）中断处理器。当时，这是能处理各中断延迟任务的唯一方法，因此叫作BH。后来BH因性能问题被任务队列（task queue）取代，而任务队列缺乏灵活性，且用于联网等对速度较为敏感的子系统时容量过于庞大。因此在内核2.3版引入与BH接口对应的softirq和tesklet的底半部，内核2.5版中由提供任务队列接口的新的底半部工作队列取代了任务队列。

第 22 章 激活slab内存分配器——kmem_cache_init()

本章分析激活slab内存分配器的阶段，slab内存分配器基于伙伴系统实现，kmem_cache_init()函数负责激活这个Linux的内存分配器。

伙伴系统将内存以页为单位进行分配，页单位的内存分配会造成内存浪费。假如需要96字节内存时分配了4KB内存，则浪费4000字节。将内存分解成更小的单位使用就能解决这个问题，而这正是slab分配器的作用。

22.1 slab 分配器的概念及结构体

slab分配器管理slab列表（slab list），slab由页组成，其页由slab管理的具有特定大小的对象组成。根据内存分配请求的不同，对象状态可能为“正在使用”或“尚未使用”。slab管理的这些对象可能是内核中广泛使用的task结构体、page结构体等，或是具有特定大小的内存。

图22-1是slab分配器大致结构图，首先要理解几个重点术语。cache具有管理slab的信息，这些信息大多是增加或减少slab大小的单位，或是slab内管理的对象的数。

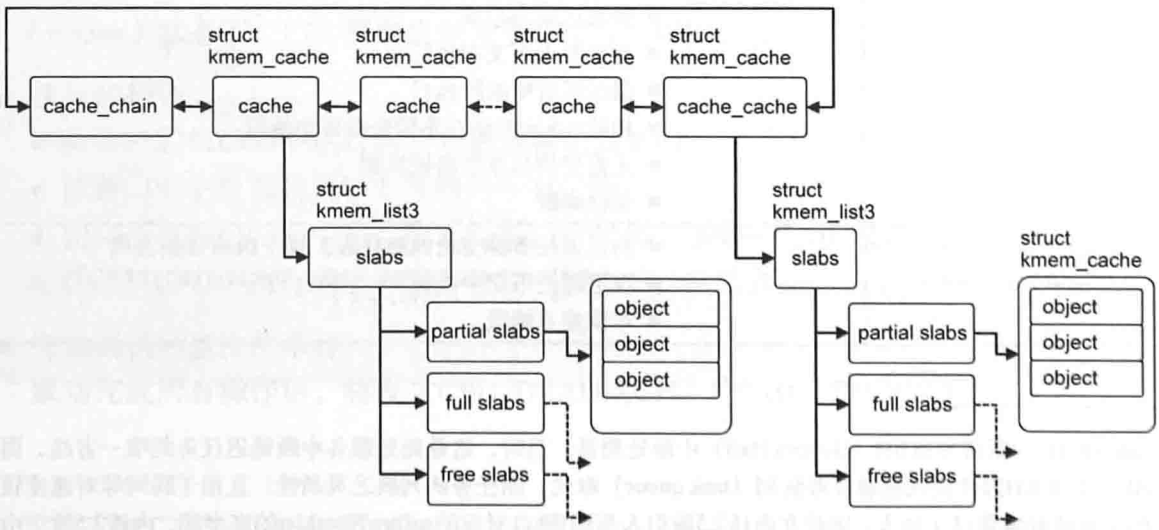


图22-1 slab分配器的概念图

此处使用cache这个词的原因是，在slab中事先制作各对象，使能够及时分配对应于该对象的内存，如同对各对象执行缓存（caching）一样。①

图22-1中，cache_cache管理cache描述符kmem_cache结构体的slab。若要制作struct task的slab，需要生成管理该slab的cache。slab分配器制作cache时需要分配kmem_cache结构体大小的内存。

因此先生成kmem_cache结构体对象，每次生成cache时，slab分配器要提供cache_cache管理的kmem_cache结构体类型的内存对象。②

图22-1中，cache_chain通过各cache的链（chain）访问cache的表头（list head），用于访问cache链表（linked list）。

cache利用kmem_list3结构体管理slab。kmem_list3结构体将slab分为3种类型：请求内存时可分配对象的闲置的slab（partial）、slab内所有内存对象因分配请求而耗尽的slab（full）、尚未执行任何分配的slab（free or empty），并通过链表管理各slab。③

22.2 slab 分配器的重要结构体——kmem_cache 和 kmem_list3

要想理解slab分配器，首先需要了解slab分配器的核心结构体kmem_cache结构体和kmem_list3结构体。kmem_cache结构体的定义如下所示。

:: mm/slab.c

```
struct kmem_cache {
    struct array_cache *array[NR_CPUS];      /*per-cpu数据，不同CPU拥有的数据*/
    unsigned int batchcount;                 /*可用per-spu同时获得的对象数*/
    unsigned int limit;                      /*per-cpu列表管理的最大对象数*/
    unsigned int shared;
    unsigned int buffer_size;                /*包含所有对象的cache的大小*/
    u32 reciprocal_buffer_size;              /*按Newton-Raphson方式求出buffer_size
                                               时使用的值④*/
    unsigned int flags;                      /*缓存属性标签，CFLGS_OFF_SLAB = offslab*/
    unsigned int num;                        /*每个slab具有的对象数*/
    unsigned int gfporder;                   /*页分配单位，2^gfporder*/
    gfp_t gfpflags;                          /*GFP标签*/
    size_t colour;                           /*cache colour数*/
    unsigned int colour_off;                 /*colour offset*/
    struct kmem_cache *slabp_cache;          /*off slab的slab mgnt*/
    unsigned int slab_size;                  /*slab head的大小*/
    unsigned int dflags;                     /*dynamic flags*/
    void (*ctor)(void *obj);                 /*生成对象时执行的函数*/
    const char *name;                        /*cache名称，ex：cat /proc/slabinfo*/
    struct list_head next;                   /*连接cache的链表*/
```

① cache是对一种类型的缓存，相应slab由不同类型对象组成。例如，如果cache是针对struct task的，那么所有对象将成为struct task类型。

② 由于是表示cache的对象缓存，故命名为cache_cache。

③ kmem_list3中的“3”表示将slab分为3种进行管理。

④ Newton-Raphson将除运算改成乘运算求解，因为乘运算比除运算速度快。

```
    ...

    /*按节点排列kmem_list3项*/
    struct kmem_list3 *nodelists[MAX_NUMNODES];
};
```

kmem_cache结构体中的最后成员变量nodelist[]数组具有MAX_NUMNODES个kmem_list3类型指针，将slab按节点进行管理，使其能从相应节点分配对象。

kmem_cache结构体的array成员变量对不同CPU管理的cache内各对象列表进行管理。array_cache结构体的定义如下所示。

:: mm/slab.c

```
struct array_cache {
    unsigned int avail;                    /*per-cpu数据中可用对象数*/
    unsigned int limit;                    /*最大对象数*/
    unsigned int batchcount;               /*可从slab同时获得的对象数*/
    unsigned int touched;                  /*显示是否用过cache*/
    spinlock_t lock;
    void *entry[ ];                        /*不同per-cpu使用的slab的对象地址列表*/
};
```

按照不同CPU放置array_cache是为了在内存分配时提高性能。

上述就是kmem_cache结构体及其成员结构体array_cache，二者关系如图22-2所示。

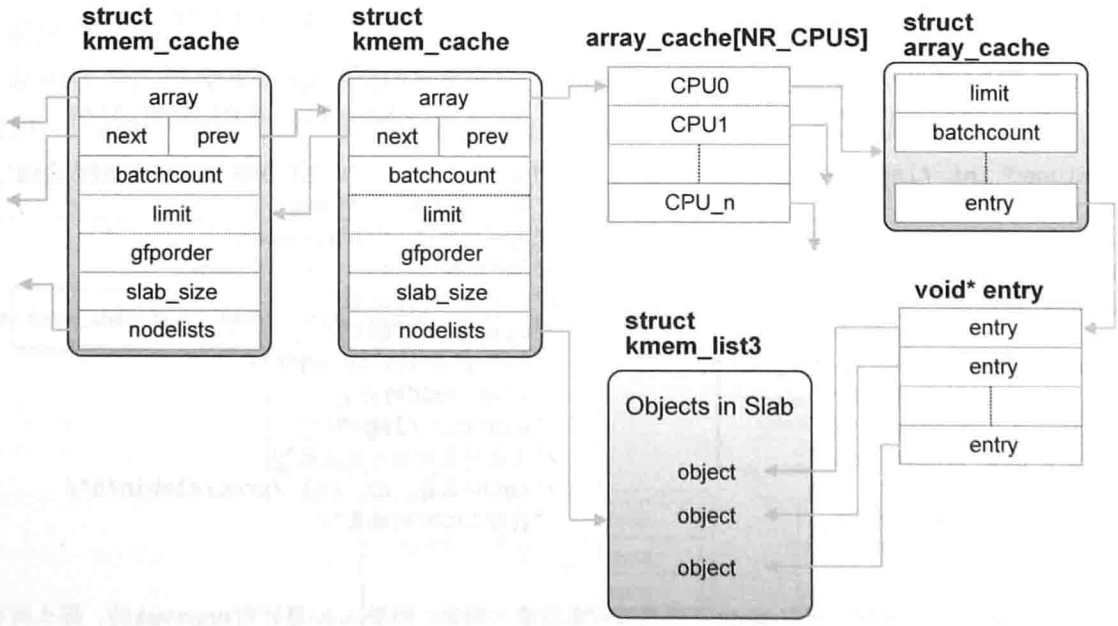


图22-2　kmem_cache结构体和array_cache结构体的关系

接下来讲解slab分配器的核心结构体kmem_list3，其定义如下所示。

:: mm/slab.c

```
struct kmem_list3 {
    struct list_head slabs_partial;     /*partial slab 列表*/
    struct list_head slabs_full;        /*full slab 列表*/
    struct list_head slabs_free;        /*free slab 列表*/
    unsigned long free_objects;         /*可分配的对象数*/
    unsigned int free_limit;            /*尚未分配的可允许最大对象数*/
    unsigned int colour_next;           /*各节点的cache colouring*/
    spinlock_t list_lock;
    struct array_cache *shared;         /*在相同节点内共享*/
    struct array_cache **alien;         /*在不同节点内共享*/
    unsigned long next_reap;            /*更新时间*/
    int free_touched;                   /*显示是否用过cache*/
};
```

图22-3表示kmem_list3结构体，如图所示用3个链表管理slab。

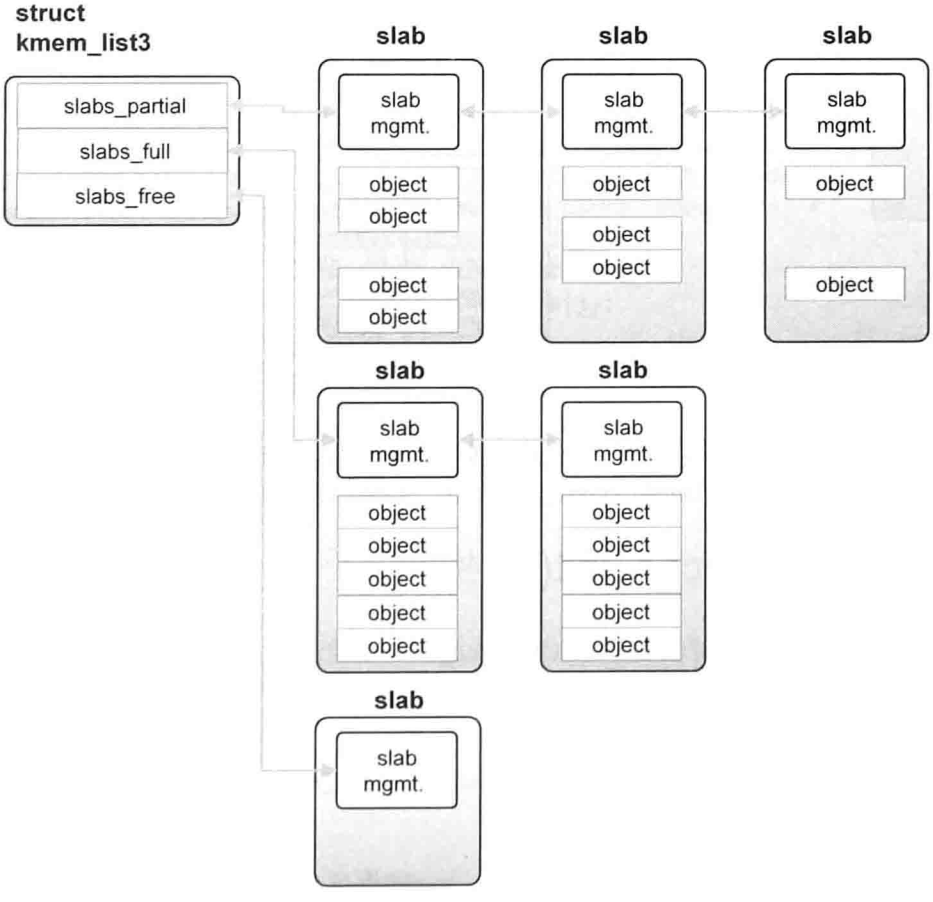


图22-3 kmem_list3结构体

通过分析代码了解各slab的组成内容。以之前分析的数据结构为基础，对slab分配器中使用的主要结构体kmem_cache、array_cache、kmem_list3之间的关系进行整体指向，如图22-4所示。

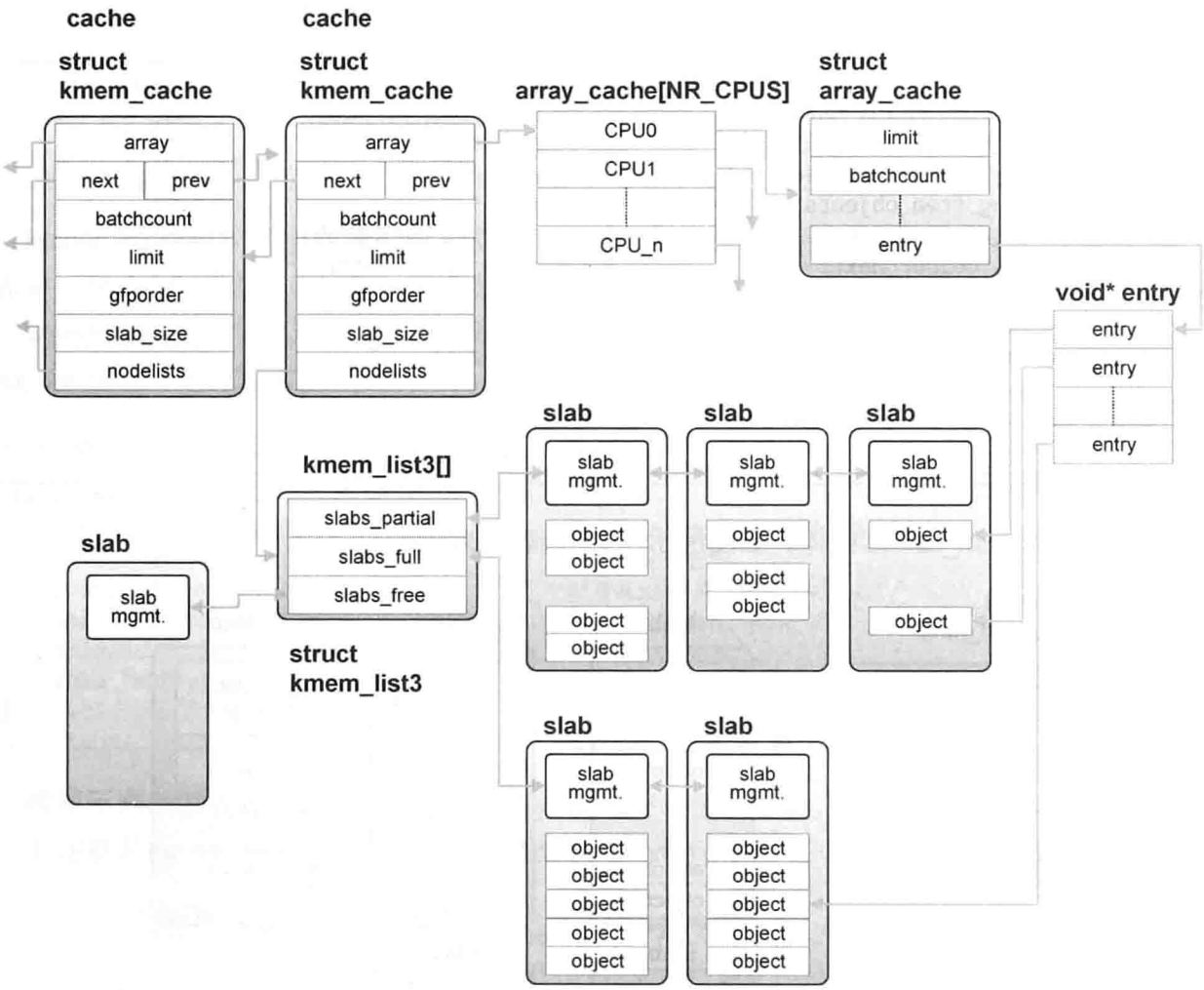


图22-4　slab分配器主要结构体关系图

22.3　查看 kmem_cache_init()结构

kmem_cache_init()函数相对较长，在内部调用多个函数。但从整体流程看，共执行6个阶段的任务。

(1) 初始化cache_cache cache

(2) 生成array_cache & kmem_list3 cache

(3) 生成kmalloc cache

(4) 将已生成cache_cache及kmalloc cache的__init数据区域用内存分配（kmalloc）方法替代

(5) 将__init数据区域中生成的cache_cache的kmem_list3通过kmalloc替代

(6) 重置各cache的头数组（head array）大小

通过代码22-1了解激活slab分配器的kmem_cache_init()函数。

:: 代码 22-1 mm/slab.c的kmem_cache_init()

```
void __init kmem_cache_init(void)
{
    struct cache_sizes *sizes;
    struct cache_names *names;

    ...

    /*1阶段：初始化cache_cache cache*/
    for (i = 0; i < NUM_INIT_LISTS; i++) {                                    ❶
        kmem_list3_init(&initkmem_list3[i]);
        if (i < MAX_NUMNODES)
            cache_cache.nodelists[i] = NULL;
    }

    set_up_list3s(&cache_cache, CACHE_CACHE) ;                                ❷
    ...
    node = numa_node_id();                                                    ❸
    INIT_LIST_HEAD(&cache_chain);
    list_add(&cache_cache.next, &cache_chain);

    cache_cache.colour_off = cache_line_size();                               ❹
    cache_cache.array[smp_processor_id()] = &initarray_cache.cache;
    cache_cache.nodelists[node] = &initkmem_list3[CACHE_CACHE + node];
    cache_cache.buffer_size = offsetof(struct kmem_cache, nodelists) +
                nr_node_ids * sizeof(struct kmem_list3 *);
    cache_cache.buffer_size = ALIGN(cache_cache.buffer_size,
                                    cache_line_size());
    cache_cache.reciprocal_buffer_size =
        reciprocal_value(cache_cache.buffer_size);

    for (order = 0; order < MAX_ORDER; order++) {                             ❺
        cache_estimate(order, cache_cache.buffer_size,
                       cache_line_size(), 0, &left_over, &cache_cache.num);
        if (cache_cache.num)
            break;
    }
    cache_cache.gfporder = order;
    cache_cache.colour = left_over / cache_cache.colour_off;
    cache_cache.slab_size = ALIGN(cache_cache.num * sizeof(kmem_bufctl_t) +
                      sizeof(struct slab), cache_line_size());

    /*2阶段：生成array_cache & kmem_list3 cache*/
    sizes = malloc_sizes;                                                     ❻
    names = cache_names;

    sizes[INDEX_AC].cs_cachep = kmem_cache_create(names[INDEX_AC].name,       ❼
                                sizes[INDEX_AC].cs_size,
                                ARCH_KMALLOC_MINALIGN,
                                ARCH_KMALLOC_FLAGS|SLAB_PANIC,
                                NULL);
```

```
    if (INDEX_AC != INDEX_L3) {
        sizes[INDEX_L3].cs_cachep =                                    ❽
            kmem_cache_create(names[INDEX_L3].name,
            sizes[INDEX_L3].cs_size,
            ARCH_KMALLOC_MINALIGN,
            ARCH_KMALLOC_FLAGS|SLAB_PANIC,
            NULL);
    }

    /*3阶段：生成kmalloc cache*/
    while (sizes->cs_size != ULONG_MAX) {                              ❾
        if (!sizes->cs_cachep) {
            sizes->cs_cachep = kmem_cache_create(names->name,
                               sizes->cs_size,
                               ARCH_KMALLOC_MINALIGN,
                               ARCH_KMALLOC_FLAGS|SLAB_PANIC,
                               NULL);
        }

        ...

        sizes++;
        names++;
    }

    /*4阶段：将已生成cache_cache及kmalloc cache的__init数据区域用内存分配（kmalloc）方法提到*/
    {                                                                  ❿
        struct array_cache *ptr;
        ptr = kmalloc(sizeof(struct arraycache_init), GFP_KERNEL);
        ...
        memcpy(ptr, cpu_cache_get(&cache_cache),
               sizeof(struct arraycache_init));
        ...
        cache_cache.array[smp_processor_id()] = ptr;
        ...
        ptr = kmalloc(sizeof(struct arraycache_init), GFP_KERNEL);
        ...
        memcpy(ptr, cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep),
               sizeof(struct arraycache_init));
        ...
        malloc_sizes[INDEX_AC].cs_cachep->array[smp_processor_id()] =
            ptr;
    }

    /*5阶段：将__init数据区域中生成的cache_cache的kmem_list3通过kmalloc替代*/
    {                                                                  ⓫
        int nid;

        for_each_online_node(nid) {
            init_list(&cache_cache, &initkmem_list3[CACHE_CACHE + nid], nid);
```

```
            init_list(malloc_sizes[INDEX_AC].cs_cachep,
                      &initkmem_list3[SIZE_AC + nid], nid);

            if (INDEX_AC != INDEX_L3) {
                init_list(malloc_sizes[INDEX_L3].cs_cachep,
                &initkmem_list3[SIZE_L3 + nid], nid);
            }
        }
    }

    /*6阶段: 重置各cache的头数组 (head array) 大小*/
    {                                                                    ⓬
        struct kmem_cache *cachep;
        mutex_lock(&cache_chain_mutex);
        list_for_each_entry(cachep, &cache_chain, next)
             if (enable_cpucache(cachep))
            BUG();
        mutex_unlock(&cache_chain_mutex);
    }
    ...
    g_cpucache_up = FULL;                                                ⓭
    register_cpu_notifier(&cpucache_notifier);
}
```

在代码段❶中调用kmem_list3_init()函数，初始化initkmem_list3[]数组和cache_cache.nodelists[]数组（22.4节）。

在代码段❷中调用set_up_list3s()函数，将cache_cache.nodelists[]数组与kmem_list3结构体的initkmem_list3[]数组的元素进行连接（22.5节）。

在代码段❸中，初始化与各cache的链表进行连接时起到头作用的cache_chain列表的表头，并为cache_chain添加cache_cache。需要注意，实际添加的并不是cache_cache，而是cache_cache的列表成员变量cache_cache.next。在内核中找到列表地址后获取实际列表所属结构体。

在代码段❹中指定cache_cache的成员变量的值。设置cache clouring中要使用的单位colour_off值，以及保存不同CPU使用对象指针的array[]数组，包含所有对象cache大小的buffer_size。设置按NewtonRaphson方式求出buffer_size时要使用的reciprocal_buffer_size的值。

其中，cache_cache的array_cache数组cache_cache.array[]指向initarray_cache.cache的内存区域，管理cache_cache的各节点所有对象的slab列表的cache_cache.nodelists[node]指向initkmem_list3[CACHE_CACHE+node]。[①]但这2个内存区域会在启动后消失，所以在代码段❿和代码段⓫中接收内存分配以进行替代。

initarray_cache的声明如下所示。

① cache_cache.nodelist[]数组已经根据代码段❷的set_up_list3s(&cache_cache，CACHE_CACHE)完成映射。

:: mm/slab.c

```
static struct arraycache_init initarray_cache __initdata =
    { .{0, BOOT_CPUCACHE_ENTRIES, 1, 0} };
```

在代码段❺中，通过cache_estimate()函数求出cache扩展或缩小时使用的页的顺序单位，利用顺序设置gfporder、colour、slab_size成员变量的值（22.6节）。

在代码段❻中，让malloc_sizes和cache_names指向sizes和names，为批量生成预定义的kmalloc cache，malloc_sizes填充cache_sizes结构体数组的值，cache_names填充cache_names结构体数组的值（22.7节）。

代码段❼、❽、❾的代码结构是相同的。调用kmem_cache_create()函数，通过对应于INDEX_AC的arraycache_init结构体的cache、对应于INDEX_L3的kmem_list3结构体的cache、保存在cache_name[]数组的名称生成各cache。kmem_cache_create()函数相关内容请参考22.8节，代码段❼、❽的arraycache_init、kmem_list3 cache的生成请参考22.9节。

接下来分析代码段❿到代码段⓭。这部分代码的核心是，用kmalloc()分配的内存替代启动时为临时使用而静态生成的内存区域。

在代码段❿中，接收代码段❹中静态分配的initarray_cache.cache的值和代码段❼中分配的initarray_generic.cache的内存，以进行替代（22.10节）。

在代码段⓫中，将静态生成的kmem_list3类型数组initkmem_list3[]执行内存分配并复制，修改管理过它的结构体成员变量的链接。完成该阶段后就不再需要启动时使用过的initkmem_list3[]数组，因此启动后回收内存。在代码段⓫的init_list()函数中分配内存并修改链接。

在代码段⓬中对cache_chain的所有cache执行循环，同时调用enable_cpucache()函数，重置cache的与array_cache相关的大小和成员变量等。

在代码段⓭中设置g_cpucache_up=UP，表示slab分配器已激活，并将cpucache_notifier注册到cpu_chain。注册cpucache_notifier是为了在新CPU处于联机状态时初始化新添加CPU的array_cache。

22.4　初始化 initkmem_list3[]、cache_cache、nodelist[]

initkmem_list3[]是所有对象的slab列表数组，cache_cache是具有所有cache的描述符kmem_cache结构体的cache。生成cache时分配cache_cache具有的kmem_cache结构体。initkmem_list3[]数组和cache_cache的定义如下所示。

:: mm/slab.c

```
#define NUM_INIT_LISTS (3 * MAX_NUMNODES)
struct kmem_list3 __initdata initkmem_list3[NUM_INIT_LISTS];

static struct kmem_cache cache_cache = {
    .batchcount = 1,
    .limit = BOOT_CPUCACHE_ENTRIES,
    .shared = 1,
```

```
    .buffer_size = sizeof(struct kmem_cache),
    .name = "kmem_cache",
};
```

此处需要确认两点：第一，NUM_INIT_LISTS被定义为“3×MAX_NUMNODES”，之所以是“3”是为了在启动时临时管理slab分配器的核心数据结构kmem_cache、array_cache、kmem_list3结构体数组；第二，数组中指定了__initdata属性，该数据区域会在启动后消失，以后将利用kmalloc()函数从slab分配器接收内存分配以重新构建。图22-5表示initkmem_list3[]数组结构。

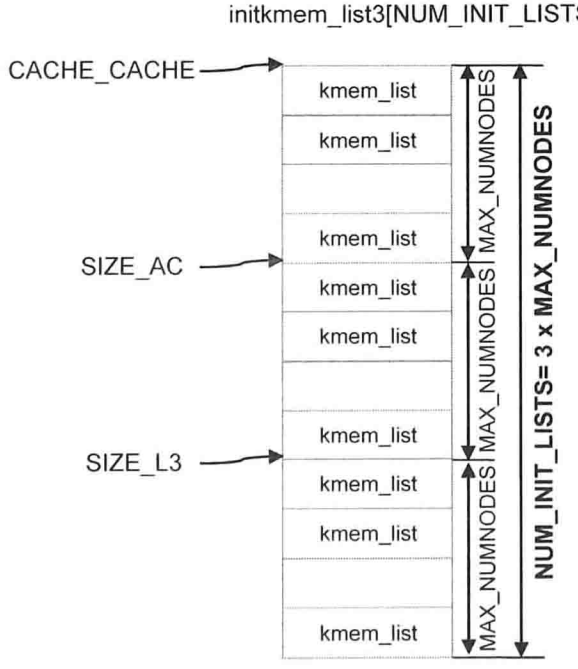


图22-5 initkmem_list3[]数组结构

函数kmem_list3_init()对initkmem_list3[]数组进行初始化，其定义如下所示。

:: mm/slab.c

```
static void kmem_list3_init(struct kmem_list3 *parent)
{
    INIT_LIST_HEAD(&parent->slabs_full);
    INIT_LIST_HEAD(&parent->slabs_partial);
    INIT_LIST_HEAD(&parent->slabs_free);
    parent->shared = NULL;
    parent->alien = NULL;
    parent->colour_next = 0;
    spin_lock_init(&parent->list_lock);
    parent->free_objects = 0;
    parent->free_touched = 0;
}
```

该函数将初始化传递来的kmem_list3结构体类型参数。

22.5　连接 kmem_list3 数组并决定 cache 压缩时间——set_up_list3s()

下面讲解set_up_list3s()函数，其定义如下所示。

:: mm/slab.c

```
static void __init set_up_list3s(struct kmem_cache *cachep, int index)
{
    int node;

    for_each_online_node(node) {
        cachep->nodelists[node] = &initkmem_list3[index + node];
        cachep->nodelists[node]->next_reap = jiffies +
            REAPTIMEOUT_LIST3 +      /* REAPTIMEOUT_LIST3 (4*HZ) -> 4秒 */
            ((unsigned long)cachep) % REAPTIMEOUT_LIST3;
    }
}
```

将cache_cache.nodelists[]数组与kmem_list3结构体initkmem_list3[]数组的元素进行连接，利用CACHE_CACHE索引仅连接管理kmem_cache结构体的slab的kmem_list3区域。用作索引的CACHE_CACHE、SIZE_AC、SIZE_L3的定义如下所示。

:: mm/slab.c

```
#define CACHE_CACHE 0
#define SIZE_AC MAX_NUMNODES
#define SIZE_L3 (2 * MAX_NUMNODES)
```

然后将array_cache和管理kmem_list3的slab的malloc_sizes[INDEX_AC]、malloc_sizes[INDEX_L3]进行连接。

除连接数组外，还设置缩小cache的时间next_reap成员变量。next_reap成员变量是为了避免slab频繁缩小/扩大造成的性能低下而定义的时间周期（time interval）。

执行set_up_list3s()函数后，cache_cache.nodelists[]数组和对应于initkmem_list3[CACHE_CACHE]的各元素执行映射。图22-6是执行set_up_list3s()函数后的cache_cache和initkmem_list3[]数组的关系。

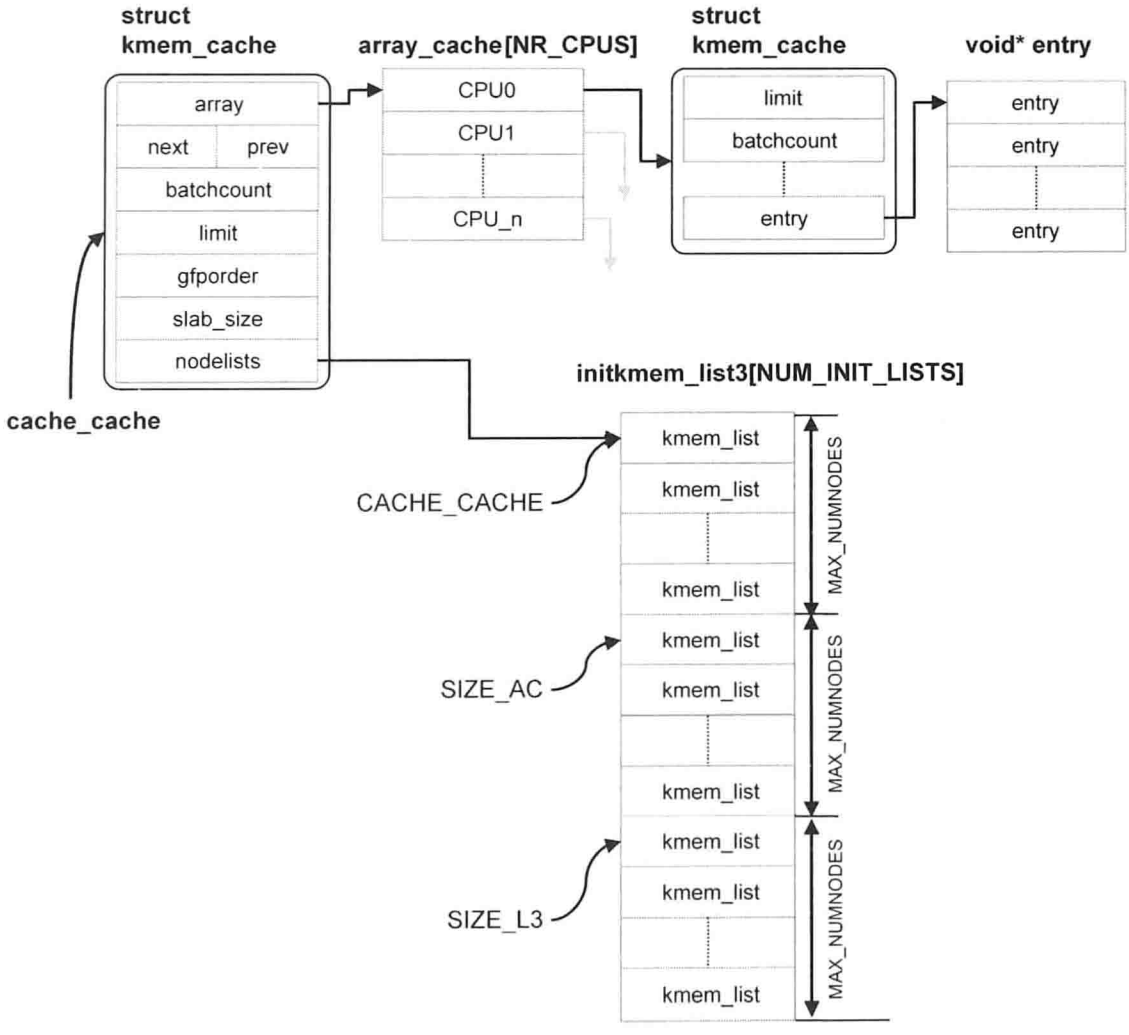


图22-6 cache_cache.nodelist[]数组和initkmem_list3[]数组间的映射

22.6 求出用于 cache 扩展/压缩的页顺序——cache_estimate()

通过cache_estimate()函数求出cache扩展/缩小时使用的页顺序单位，其定义如代码22-2所示。

:: 代码22-2 mm/slab.c的cache_estimate()

```
static void cache_estimate(unsigned long gfporder, size_t buffer_size,
             size_t align, int flags, size_t *left_over,
             unsigned int *num)
{
    int nr_objs;
    size_t mgmt_size;
    size_t slab_size = PAGE_SIZE << gfporder;

    if (flags & CFLGS_OFF_SLAB) {                    /*off slab时*/          ❶
```

```
        mgmt_size = 0;
        nr_objs = slab_size / buffer_size;

        if (nr_objs > SLAB_LIMIT)
            nr_objs = SLAB_LIMIT;
    } else {                                    /*on slab时*/                    ❷
        nr_objs = (slab_size - sizeof(struct slab)) /
              (buffer_size + sizeof(kmem_bufctl_t));

        /*若排列slab管理信息时传递了slab大小，则减少1个slab对象数*/
        if (slab_mgmt_size(nr_objs, align) + nr_objs*buffer_size
               > slab_size)
            nr_objs--;

        if (nr_objs > SLAB_LIMIT)
            nr_objs = SLAB_LIMIT;

         /*求出用于管理的数据大小*/
        mgmt_size = slab_mgmt_size(nr_objs, align);
    }

    *num = nr_objs;          /*对象总数*/

    /*该顺序中分配对象后的剩余空间*/
    *left_over = slab_size - nr_objs*buffer_size - mgmt_size;
}
```

cache_estimate()函数中出现了off slab和on slab的概念。将代码段❶和代码段❷分为off slab和on slab，求出对象数nr_objs和剩余空间left_over的值。off slab是管理slab的管理信息脱离slab的情况，而on slab指管理信息包含于slab的情形。

图22-7表示可构成slab的2种类型on slab和off slab的概念。

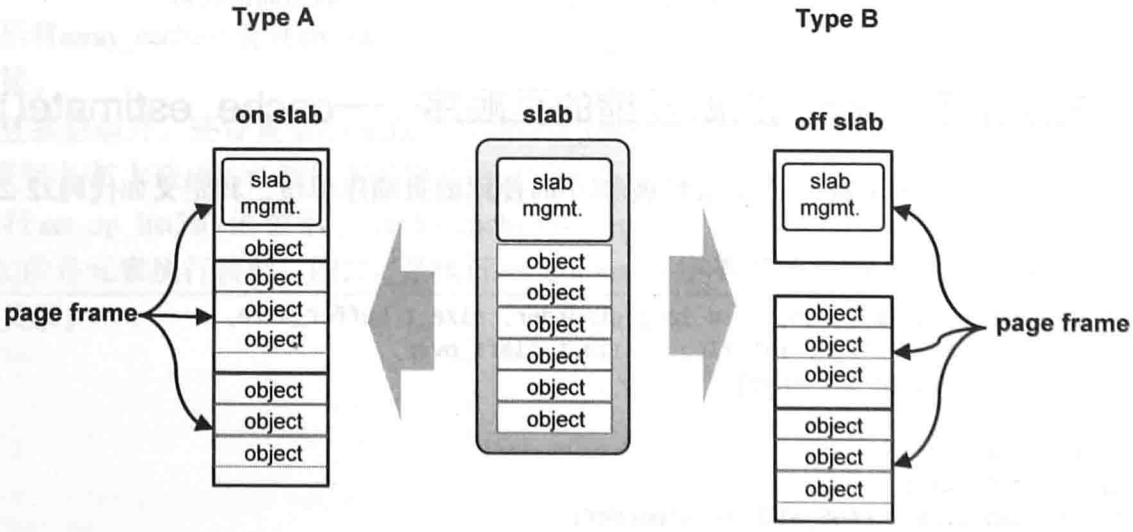


图22-7　on slab vs. off slab

因此，on slab和off slab时，slab中管理的对象数不同。off slab时，由于管理信息分离，故将用slab分配的页除以cache的大小buffer_size，求出slab中管理的对象数。此时必须除去slab管理信息区域才能求出对象数。

可通过slab_mgmt_size()函数了解slab管理信息是如何构成的。下面是slab_mgmt_size()函数的定义。

:: mm/slab.c

```
static size_t slab_mgmt_size(size_t nr_objs, size_t align)
{
    return ALIGN(sizeof(struct slab)+nr_objs*sizeof(kmem_bufctl_t), align);
}
```

如图22-7所示，通过slab_mgmt_size()函数可确认slab管理信息（slab management）由slab结构体和slab管理的对象数大小的kmem_bufctl_t类型数组构成。slab结构体和kmem_bufctl_t的定义如下所示。

:: mm/slab.c

```
struct slab {
    struct list_head list;                  /*full、partial或free slab列表*/
    unsigned long colouroff;                /*slab colouring偏移量*/
    void *s_mem;                            /*slab中第一个对象*/
    unsigned int inuse;                     /*正在使用的对象数*/
    kmem_bufctl_t free;                     /*第一个空白对象*/
    unsigned short nodeid;                  /*节点号*/
};
```

:: mm/slab.c

```
typedef unsigned int kmem_bufctl_t;
#define BUFCTL_END  (((kmem_bufctl_t)(~0U))-0)
#define BUFCTL_FREE (((kmem_bufctl_t)(~0U))-1)
#define BUFCTL_ACTIVE   (((kmem_bufctl_t)(~0U))-2)
#define SLAB_LIMIT  (((kmem_bufctl_t)(~0U))-3)
```

图22-8是slab管理信息的结构。

```
struct slab {
    struct list_head list;
    unsigned long colouroff;
    void * s_mem;
    unsigned int inuse;
    kmem_bufctl_t free;
    unsigned short nodeid;
};
```

```
typedef unsigned int kmem_bufctl_t;
#define BUFCTL_END  (((kmem_bufctl_t)(~0U))-0)
#define BUFCTL_FREE (((kmem_bufctl_t)(~0U))-1)
#define BUFCTL_ACTIVE   (((kmem_bufctl_t)(~0U))-2)
```

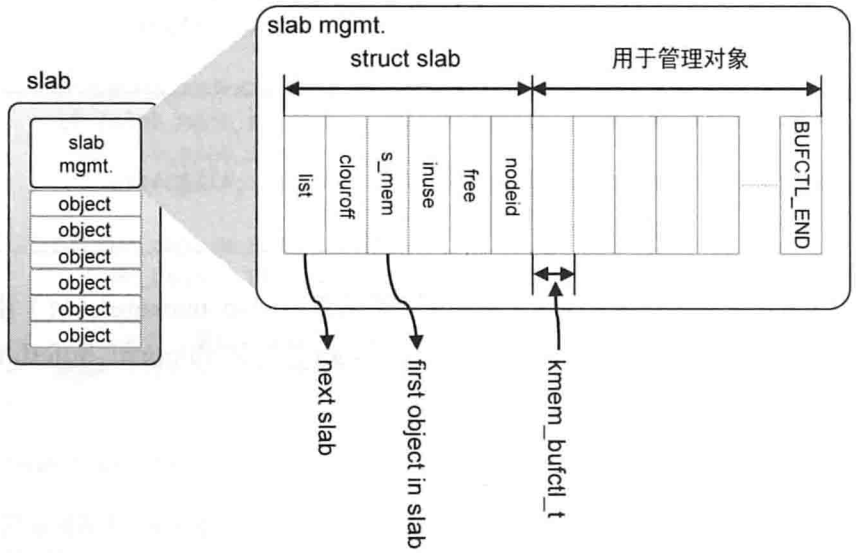


图22-8　slab结构体和kmem_bufctl_t类型数组构成的slab管理信息

如图22-9所示管理slab中保存的对象。

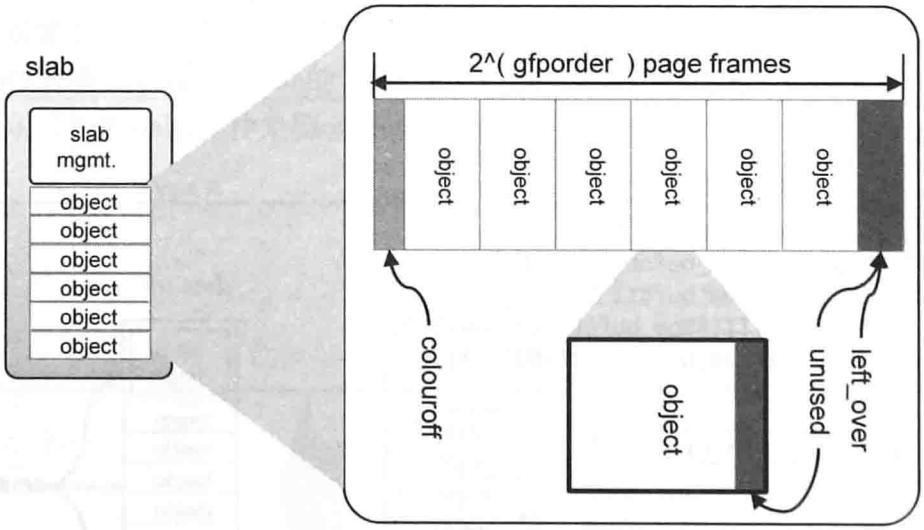


图22-9　slab中管理的对象

22.7　malloc_sizes 和 cache_names

malloc_sizes和cache_names的定义如下所示。

:: mm/slab.c

```
struct cache_sizes malloc_sizes[ ] = {
#define CACHE(x) { .cs_size = (x) },
#include <linux/kmalloc_sizes.h>
    CACHE(ULONG_MAX)
#undef CACHE
};
EXPORT_SYMBOL(malloc_sizes);

static struct cache_names __initdata cache_names[ ] = {
#define CACHE(x) { .name = "size-" #x, .name_dma = "size-" #x "(DMA)" },
#include <linux/kmalloc_sizes.h>
    {NULL,}
#undef CACHE
};
```

为了批量生成预定义的各kmalloc cache，malloc_sizes向cache_sizes结构体数组填充值，cache_names填充cache_names结构体数组的值。cache_sizes和cache_names结构体的定义如下所示。

:: include/linux/slab_def.h

```
/* Size description struct for general caches. */
struct cache_sizes {
    size_t          cs_size;
    struct kmem_cache   *cs_cachep;
#ifdef CONFIG_ZONE_DMA
    struct kmem_cache   *cs_dmacachep;
#endif
};
```

:: mm/slab.c

```
struct cache_names {
    char *name;
    char *name_dma;
};
```

malloc_sizes[]数组和cache_names[]数组利用linux/kmalloc_sizes.h中记述的值填充，如下所示。

:: include/linux/kmalloc_sizes.h

```
#if (PAGE_SIZE == 4096)
    CACHE(32)
#endif
    CACHE(64)
#if L1_CACHE_BYTES < 64
    CACHE(96)
#endif
```

```
    CACHE(128)
#if L1_CACHE_BYTES < 128
    CACHE(192)
#endif
    CACHE(256)
    CACHE(512)
    CACHE(1024)
    CACHE(2048)
    CACHE(4096)
    CACHE(8192)
    CACHE(16384)
    CACHE(32768)
    CACHE(65536)
    CACHE(131072)

    ...
```



补充知识点　通过/proc/slabinfo了解kmalloc cache

可通过/proc/slabinfo查看malloc_sizes[]数组和cache_names[]数组中保存的值，以及各kmalloc cache的名称、大小等。

```
code@kernel~$ cat /proc/slabinfo ¦ awk -v OFS="\t" -F' ' '{print $1,$3,$4,$5,$6}'
slabinfo version: 2.1

#name            <active_objs>      <num_objs>        <objsize>         <objperslab>

...

size-8192        92                 8192              4                 8
size-4096        216                4096              8                 8
size-2048        400                2048              16                8
size-1024        1344               1024              32                8
size-512         2080               512               32                4
size-256         1120               256               32                2
size-128         2848               128               32                1
size-64          7040               64                64-               1
...
code@kernel~$
```

22.8　生成 cache——kmem_cache_create()

生成cache的函数就是kmem_cache_create()函数，如代码22-3所示。

:: 代码22-3 mm/slab.c的kmem_cache_create()

```
struct kmem_cache *
kmem_cache_create (const char *name, size_t size, size_t align,
    unsigned long flags, void (*ctor)(void *))
{
    size_t left_over, slab_size, ralign;
    struct kmem_cache *cachep = NULL, *pc;

    ...

    cachep = kmem_cache_zalloc(&cache_cache, GFP_KERNEL);                    ❶

    ...

    left_over = calculate_slab_order(cachep, size, align, flags);            ❷
    ...
    slab_size = ALIGN(cachep->num * sizeof(kmem_bufctl_t)
              + sizeof(struct slab), align);
    ...
    cachep->colour_off = cache_line_size();
    ...
    cachep->colour = left_over / cachep->colour_off;
    cachep->slab_size = slab_size;
    cachep->flags = flags;
    cachep->gfpflags = 0;
    ...
    cachep->buffer_size = size;
    cachep->reciprocal_buffer_size = reciprocal_value(size);
    ...
    cachep->ctor = ctor;
    cachep->name = name;

    if (setup_cpu_cache(cachep)) {                                           ❸
        __kmem_cache_destroy(cachep);
        cachep = NULL;
        goto oops;
    }

    list_add(&cachep->next, &cache_chain);                                   ❹

    ...

}
EXPORT_SYMBOL(kmem_cache_create);
```

代码段❶调用kmem_cache_zalloc()函数，从cache_cache分配kmem_cache结构体类型的对象（22.8.1节）。

代码段❷调用calculate_slab_order()函数，并调用cache_estimate()函数（22.6节）计算slab的页顺序和每个slab对象数。

代码段❸设置cache的不同CPU具有的array_cache，调用setup_cpu_cache()函数执行（22.8.3节）。

代码段❹将生成的cache添加到cache_chain链表，完成cache生成操作。更准确地说，将list_head结构体类型的cache.next添加到list_head类型的链表cache_chain。

22.8.1　kmem_cache_zalloc()

调用kmem_cache_zalloc()函数以在cache_cache中分配对象，其调用关系如图22-10所示。

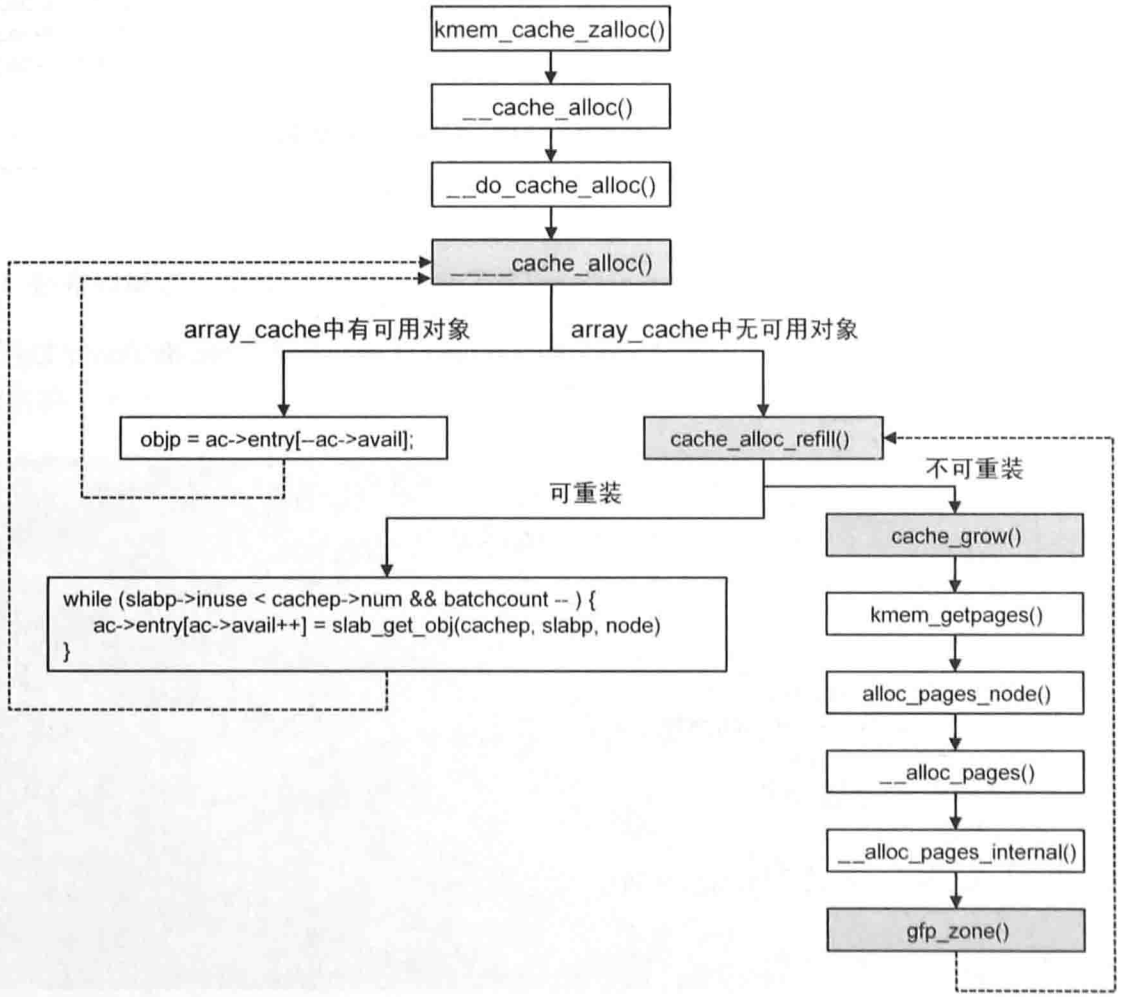


图22-10　在cache中分配对象时的函数调用关系图

下面讲解函数调用关系图中出现的几种用于分配对象的重要函数。

首先，相应cache的array_cache中有可用对象时，____cache_alloc()函数减少1个可用对象（--ac->avail），并直接分配对象。若array_cache中无可用对象，则从slab调用batchcount大小的slab_get_obj()函数以进行重装（refill）。若无法从slab执行重装，则调用cache_grow()函数扩展slab。

观察cache_grow()函数的调用关系可知，最终调用的是gfp_zone()函数，该函数是在伙伴系统

以顺序单位分配页的核心函数。

22.8.2 calculate_slab_order()

calculate_slab_order()函数计算slab的页顺序和每个slab对象数，其定义如代码22-4所示，从内部调用代码22-2的cache_estimate()函数以进行求解。

:: 代码22-4 mm/slab.c的calculate_slab_order()

```
static size_t calculate_slab_order(struct kmem_cache *cachep,
            size_t size, size_t align, unsigned long flags)
{
    ...
    for (gfporder = 0; gfporder <= KMALLOC_MAX_ORDER; gfporder++) {
        unsigned int num;
        size_t remainder;

        cache_estimate(gfporder, size, align, flags, &remainder, &num);
        if (!num)
            continue;

        ...

        cachep->num = num;
        cachep->gfporder = gfporder;
        left_over = remainder;

        ...

        if (left_over * 8 <= (PAGE_SIZE << gfporder))
            break;
    }
    return left_over;
}
```

22.8.3 setup_cpu_cache()

setup_cpu_cache()函数设置cache的不同CPU具有的array_cache。全局变量g_cpucache_up变量值不同，setup_cpu_cache()函数中执行的操作有些不同。分析代码前分析g_cpucache_up变量的含义。

使用g_cpucache_up变量是因为需要生成cache的CPU数据array_cache，而array_cache又是内存，所以需要从内存分配器slab分配器接收内存分配。但由于尚未激活slab分配器，因此将slab分配器的激活阶段保存到g_cpucache_up变量，使生成array_cache的方法根据不同阶段而有所不同。

g_cpucache_up能具有的值如下所示。

:: mm/slab.c

```
static enum {
    NONE,
    PARTIAL_AC,
    PARTIAL_L3,
    FULL
} g_cpucache_up;
```

NODE阶段无法分配array_cache，PARTIAL_AC阶段可分配array_cache，PARTIAL_L3阶段可分配kmem_list3，FULL阶段可通过slab分配器分配内存。

下面通过代码22-5分析setup_cpu_cache()函数。

:: 代码 22-5　mm/slab.c的setup_cpu_cache()

```
static int __init_refok setup_cpu_cache(struct kmem_cache *cachep)
{
    if (g_cpucache_up == FULL)                                          ❶
        return enable_cpucache(cachep);

    if (g_cpucache_up == NONE) {                                        ❷
        cachep->array[smp_processor_id()] = &initarray_generic.cache;
        set_up_list3s(cachep, SIZE_AC);
        ...

    } else {    /*PARTIAL_AC或PARTIAL_L3时*/
        cachep->array[smp_processor_id()] =                             ❸
            kmalloc(sizeof(struct arraycache_init), GFP_KERNEL);
        if (g_cpucache_up == PARTIAL_AC) {
            set_up_list3s(cachep, SIZE_L3);
            g_cpucache_up = PARTIAL_L3;
        } else {    /*PARTIAL_L3时*/                                  ❹
            int node;
            for_each_online_node(node) {
                cachep->nodelists[node] =
                    kmalloc_node(sizeof(struct kmem_list3),
                        GFP_KERNEL, node);
                BUG_ON(!cachep->nodelists[node]);
                kmem_list3_init(cachep->nodelists[node]);
            }
        }
    }

    cachep->nodelists[numa_node_id()]->next_reap =                      ❺
            jiffies + REAPTIMEOUT_LIST3 +
            ((unsigned long)cachep) % REAPTIMEOUT_LIST3;

    cpu_cache_get(cachep)->avail = 0;
    cpu_cache_get(cachep)->limit = BOOT_CPUCACHE_ENTRIES;
    cpu_cache_get(cachep)->batchcount = 1;
    cpu_cache_get(cachep)->touched = 0;
```

```
    cachep->batchcount = 1;
    cachep->limit = BOOT_CPUCACHE_ENTRIES;
    return 0;
}
```

代码段❶是g_cpucache_up==FULL阶段，可使用slab分配器分配array_cache，并调用enable_cpucache()函数设置cache的array_cache（22.8.4节）。

代码段❷是g_cpucache_up==NODE的情况，将静态生成的array_cache即initarray_generic.cache指定为cache的array_cache。

代码段❸是g_cpucache_up为PARTIAL_AC或PARTIAL_L3的情形，调用kmalloc()函数分配array_cache。kmalloc()函数调用kmem_cache_zalloc()函数中的__cache_alloc()函数。

代码段❹是g_cpucache_up ==PARTIAL_L3阶段，调用kmalloc_node()函数生成各节点的kmem_list3，并连接到cache.nodelist[]，调用kmem_list3_init()函数进行初始化。

在代码段❺中调整cache的各成员变量的值。

22.8.4 enable_cpucache()

enable_cpucache()函数的定义如代码22-6所示。代码22-6根据cache的buffer_size值保存cache的array_cache中可指向的对象数，并调用do_tune_cpucache()函数设置cache的array_cache。

:: 代码22-6 mm/slab.c的enable_cpucache()

```
static int enable_cpucache(struct kmem_cache *cachep)
{
    ...

    if (cachep->buffer_size > 131072)
        limit = 1;
    else if (cachep->buffer_size > PAGE_SIZE)
        limit = 8;
    else if (cachep->buffer_size > 1024)
        limit = 24;
    else if (cachep->buffer_size > 256)
        limit = 54;
    else
        limit = 120;

    ...

    err = do_tune_cpucache(cachep, limit, (limit + 1) / 2,
    ...

    return err;
}
```

22.9　生成 arraycache_init,kmem_list3 cache

通过kmem_cache_create()函数的代码段❼和代码段❽生成arraycache_init cache和kmem_list3 cache。

:: 代码22-1　kmem_cache_init()函数的代码段❼、❽

```
    sizes[INDEX_AC].cs_cachep = kmem_cache_create(names[INDEX_AC].name,          ❼
                        sizes[INDEX_AC].cs_size,
                        ARCH_KMALLOC_MINALIGN,
                        ARCH_KMALLOC_FLAGS|SLAB_PANIC,
                        NULL);

        if (INDEX_AC != INDEX_L3) {
            sizes[INDEX_L3].cs_cachep =                                          ❽
                kmem_cache_create(names[INDEX_L3].name,
                    sizes[INDEX_L3].cs_size,
                    ARCH_KMALLOC_MINALIGN,
                    ARCH_KMALLOC_FLAGS|SLAB_PANIC,
                    NULL);
        }
```

代码段❼、❽的代码结构相同。调用kmem_cache_create()函数，从函数名可以看出，从cache_cache为cache分配kmem_cache结构体。首先，代码段❼中生成用于arraycache_init结构体对象的cache，代码段❽中生成用于kmem_list3结构体对象的cache。2个代码段中使用的INDEX_AC和INDEX_L3的定义如下所示。

:: mm/slab.c

```
#define INDEX_AC index_of(sizeof(struct arraycache_init))
#define INDEX_L3 index_of(sizeof(struct kmem_list3))
```

代码22-7是index_of()函数，该函数返回kmalloc_caches中最接近相应结构体大小的缓存大小索引。

:: 代码22-7　mm/slab.c的index_of()

```
static __always_inline int index_of(const size_t size)
{
    extern void __bad_size(void);

    if (__builtin_constant_p(size)) {
        int i = 0;

#define CACHE(x)    \
    if (size <=x) \
        return i; \
    else \
        i++;
#include <linux/kmalloc_sizes.h>
```

```
#undef CACHE
        __bad_size();
    } else
        __bad_size();
    return 0;
}
```

因此，在代码段❼和代码段❽中调用kmem_cache_create()函数，生成arraycache_init结构体对象和用于kmem_list3结构体对象的最近的kmalloc cache。需要确认一点，虽然通过kmem_cache_create()函数生成cache，但启动时会利用静态声明的内存，因此最后均分配内存取而代之。代码段❼在setup_cpu_cache()函数中使用如下静态声明的内存。

```
cachep->array[smp_processor_id()] = &initarray_generic.cache;
```

代码段❼和代码段❽中使用的静态生成的内存将分别通过kmalloc()分配内存并被替代（22.10节）。

代码段❼和代码段❽完成时，与cache_cache、malloc_sizes[]、initkmem_list3[]数组的关系如图22-11所示。

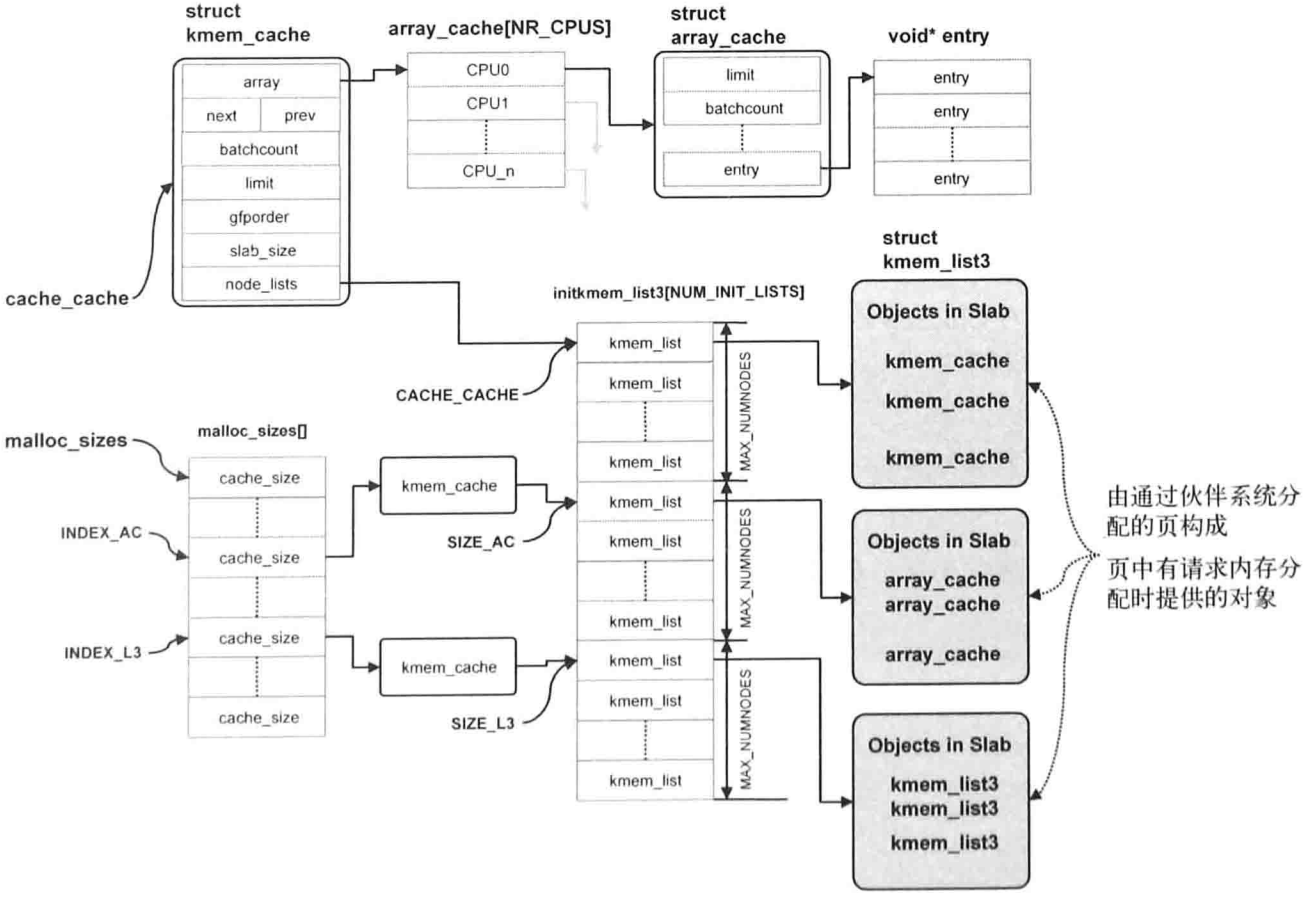


图22-11 映射cache_cache.nodelists[]、malloc_sizes[]数组和initkmem_list3[]数组

22.10　用 kmalloc()函数分配的内存替代静态分配的内存

cache_cache的array_cache和cache_cache的kmem_list3类型的nodelists[]中指定的内存使用静态生成的内存区域，故分配内存后进行替代。用kmalloc()分配的内存取代启动时临时使用的静态生成的内存区域。在代码22-1所示的kmem_cache_init()函数内的如下代码段中静态分配。

:: kmem_cache_init()函数内

```
for (i = 0; i < NUM_INIT_LISTS; i++) {
    kmem_list3_init(&initkmem_list3[i]);
    if (i < MAX_NUMNODES)
        cache_cache.nodelists[i] = NULL;
}

cache_cache.array[smp_processor_id()] = &initarray_cache.cache;

sizes[INDEX_AC].cs_cachep = kmem_cache_create(names[INDEX_AC].name,
                    sizes[INDEX_AC].cs_size,
                    ARCH_KMALLOC_MINALIGN,
                    ARCH_KMALLOC_FLAGS¦SLAB_PANIC,
                    NULL);
```

通过kmalloc()函数分配内存，移动initkmem_list3[]、initarray_cache.cache、initarray_generic.cache中保存过的数据。

通过内存分配器分配静态生成的内存后，进行替代时的cache_cache和malloc_sizes[]数组的关系如图22-12所示。不再使用initkmem_list3[]数组。

本章通过代码查看了Linux中使用的内存分配器slab分配器是如何激活的。分配内存时需要激活内存分配器，故发生Chicken-Egg问题。

为解决该问题，利用显示slab激活阶段的全局变量g_cpucache_up表示slab的激活阶段。按不同阶段适当利用静态生成的内存以执行激活。预先生成一般kmalloc中使用的内存，并将静态生成的内存区域全部替代为kmalloc()函数分配的内存，使各静态内存在启动后能回收并重新使用。

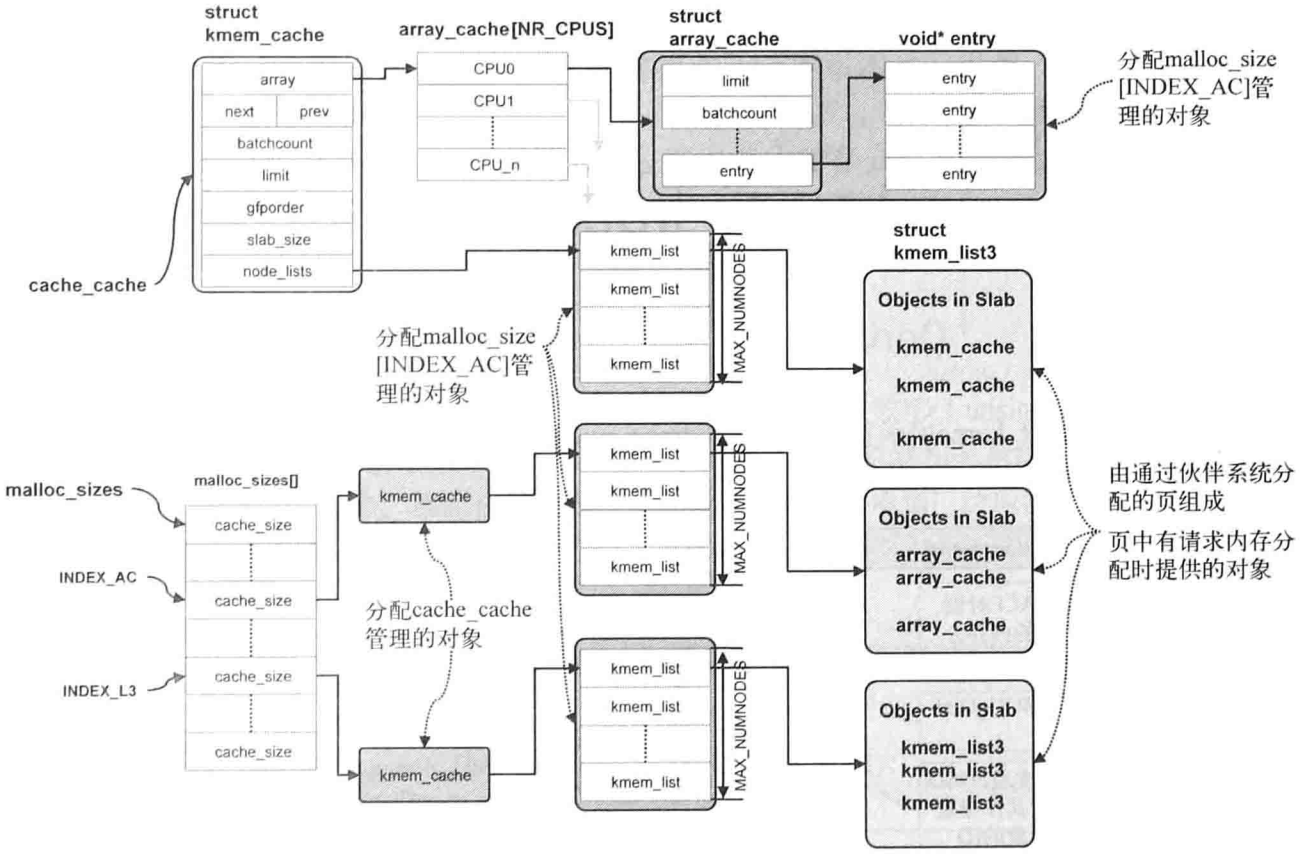


图22-12 替代内存后的cache_cache、malloc_sizes[]数组



补充知识点 slob、slub内存分配器

之前介绍了Linux的主内存分配器是slab分配器，但除slab分配器外，Linux还提供多种内存分配器，slob和slub就是其中之一。slab内存分配器通常是比较理想的内存分配器，但在内存系统过小的嵌入式系统（embedded system）或使用大容量内存的并行计算系统（parallel computing system）环境下无法发挥最佳性能。

需要极小内存footprint的嵌入式环境使用slob分配器，由多个节点构成的具有大容量内存的服务器环境使用slub分配器。

第 23 章 kmem_trace_init()~security_init()

本章将讲解start_kernel()函数内的kmem_trace_init()函数到security_init()函数部分，函数调用顺序及执行业务如图23-1所示。

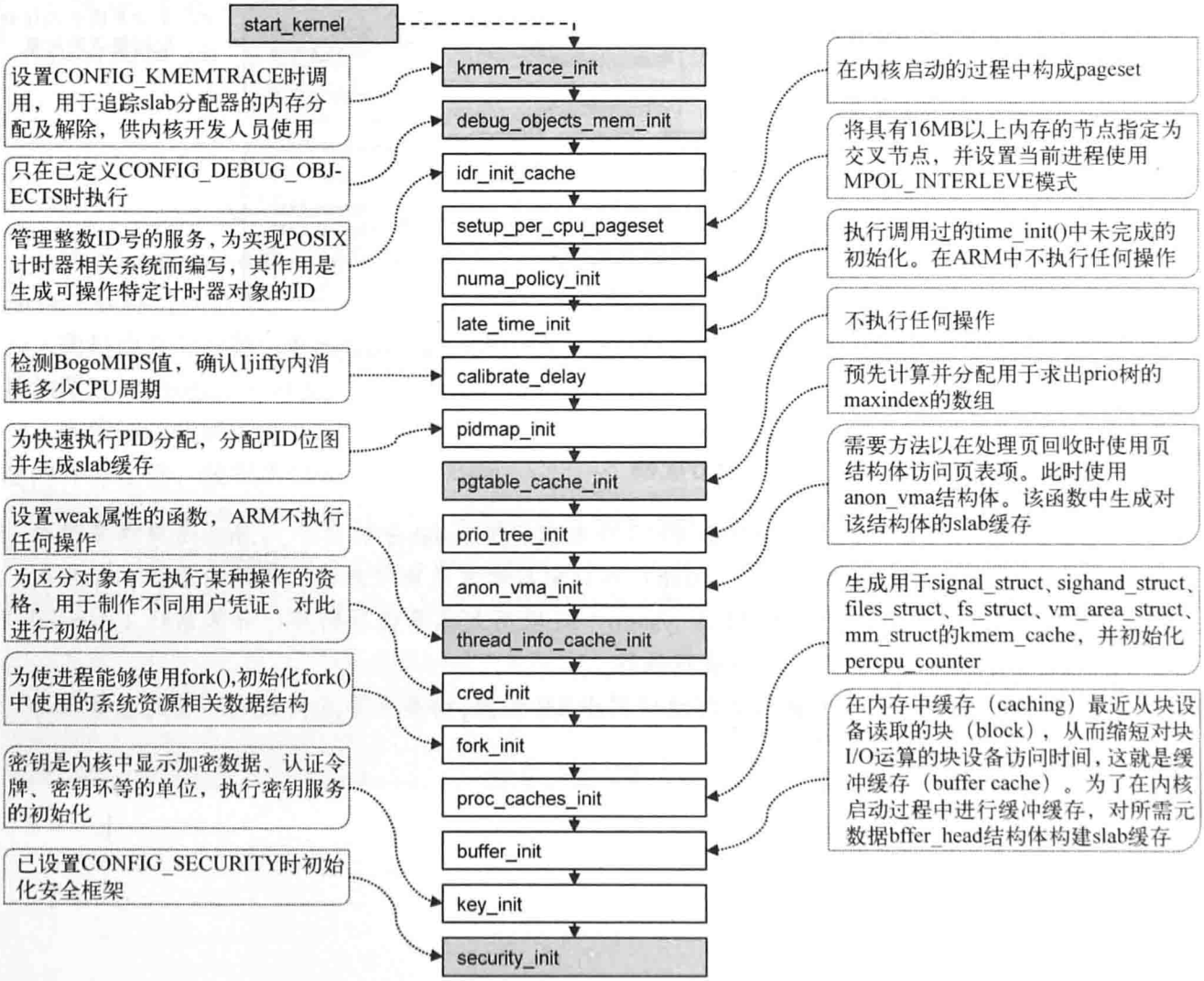


图23-1　从kmem_trace_init()函数到security_init()函数的函数调用顺序

其中，kmem_trace_init()、debug_objects_mem_init()、pgtable_cache_init()、thread_info_cache_init()、security_init()函数在ARM内不执行任何操作，故本章不进行讲解，只简单概括一下kmem_trace_init()函数和security_init()函数。

kmem_trace_init()函数帮助内核开发人员追踪内核的内存分配器如何运行，尤其是追踪kmalloc()、kfree()、kmem_cache_alloc、kmem_cache_free()等函数。security_init()函数是内核中指定CONFIG_SECURITY时调用的函数，使Linux内核使用其他安全模式而非默认安全模式。

23.1 生成 ID alloccator 缓存——idr_init_cache()

IDR是内核提供的一种基数树（radix tree），其作用是连接整数ID（unsigned long）和特定指针（void*）。IDR是管理整数ID的服务，也是为了实现POSIX计时器相关系统而编写的。它生成可操作特定计时器对象的ID，目前在各种设备驱动及VFS层中也得到广泛应用。

idr_init_cache()函数定义如下。

:: lib/idr.c

```
void __init idr_init_cache(void)
{
    idr_layer_cache = kmem_cache_create("idr_layer_cache",
                sizeof(struct idr_layer), 0, SLAB_PANIC, NULL);
}
```

首先生成idr_layer结构体缓存，idr_layer具有ID信息。若idr_layer_cache缓存区域生成失败，则kmem_cache_create()函数中会发生内核恐慌。

23.2 初始化 pageset——setup_per_cpu_pageset()

如前所述，内核对常用对象（object）预先分配内存以维持slab缓存，从而对频繁发生的分配请求和解除迅速提供服务。同样，为了对单页的分配请求和解除迅速提供服务，内核预先为每个本地CPU分配页（page），从而维持per-cpu缓存。

这些单页缓存称为pageset，此处要介绍的setup_per_cpu_pageset函数在内核启动过程中构建pageset。用于分配单页的缓存pageset将预先分配好的页分为热页（hot page）和冷页（cold page）。

热页指本地CPU缓存内的页，比RAM上的冷页访问速度更快。pageset可称为per-cpu热&冷页列表。

:: 代码23-1 linux/mmzone.h的per_cpu_pages结构体

```
struct per_cpu_pages {
    int count;          ❶
    int high;           ❷
    int batch;          ❸
    struct list_head list;
};
```

上述代码是pageset的描述符（descriptor）。作为链表，热&冷页列表通过list成员进行管理。热页位于链表前部，冷页位于链表后部。

代码段❶的count成员变量显示与热&冷页列表连接的页数，代码段❷的high成员变量是对count的水印（watermark）值。为有效使用内存，内核使热&冷页列表中的已缓冲页数不超过high。代码段❸的batch是为了根据high水印值在pageset列表中执行页的添加/删除而从伙伴系统请求的内存大小。

:: 代码23-2　mm/page_alloc.c的setup_per_cpu_pageset()

```
void __init setup_per_cpu_pageset(void)
{
    int err;
    err = process_zones(smp_processor_id());                    ❶
    BUG_ON(err);
    register_cpu_notifier(&pageset_notifier);                   ❷
}
```

对per-cpu数据pageset的分配和初始化任务实际上是通过process_zones()函数执行的。在代码段❶中对SMP系统内正在启动的CPU执行pageset初始化。

通过代码段❷，在cpu notifier chain中注册pageset的notifier block，从而对SMP系统内准备的其他所有CPU执行pageset初始化。

:: 代码23-3　mm/page_alloc.c的process_zones()

```
static int __cpuinit process_zones(int cpu)
{
    struct zone *zone, *dzone;
    int node = cpu_to_node(cpu);                                ❶
    node_set_state(node, N_CPU);
    for_each_populated_zone(zone) {

        zone_pcp(zone, cpu) = kmalloc_node(sizeof(struct per_cpu_pageset),   ❷
                    GFP_KERNEL, node);
        if (!zone_pcp(zone, cpu))
            goto bad;
        setup_pageset(zone_pcp(zone, cpu), zone_batchsize(zone));            ❸
        ...
    }
    ...
}
```

首先，在代码23-3的代码段❶中通过cpu_to_node()函数返回当前处理器正在使用的节点。

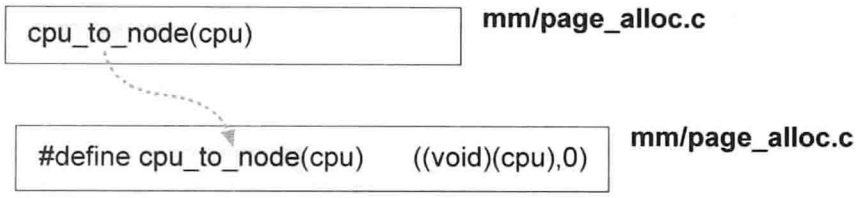


图23-2 cpu_to_node()

从图23-2可知，cpu_to_node()宏始终返回0，因为使用ARM处理器的系统中没有使用NUMA模式的系统，且始终假定UMA。

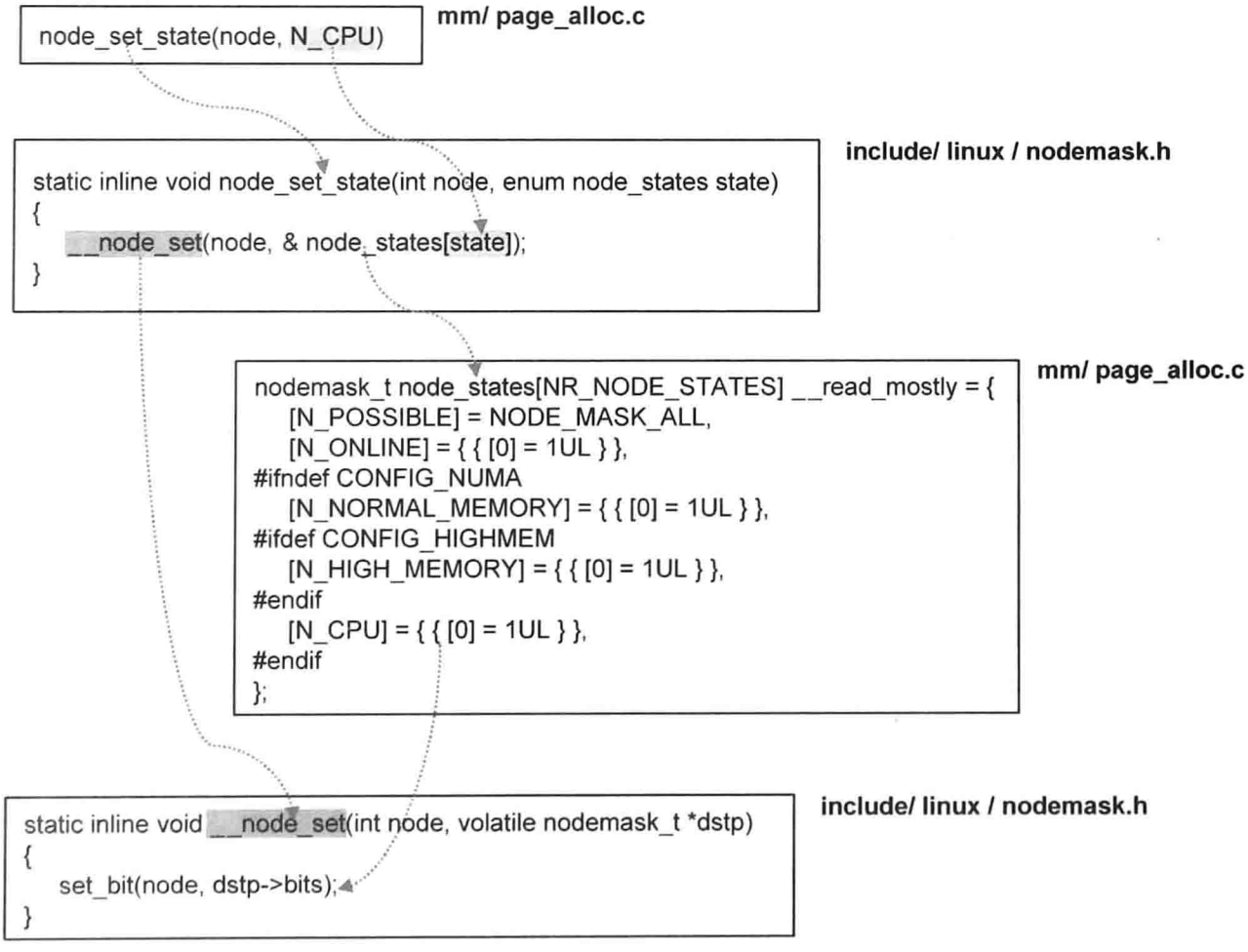


图23-3 node_set_state()

接下来通过node_set_state()函数设置当前节点由处理器（CPU）使用。若当前系统内有多个节点，为提供各节点状态信息，Linux会具有全局位图nodemask_t node_states。从图23-3可知，浏览node_states[NR_CPUS]位图并显示CPU持有的当前节点。

下面观察本代码中最核心的代码段❷。通过zone_pcp()函数浏览当前zone的per-cpu pageset，通过kmalloc_node()动态分配内存。在代码段❸中，通过setup_pageset()函数对之前新分配的代码

23-1的per_cpu_pages结构体执行初始化。

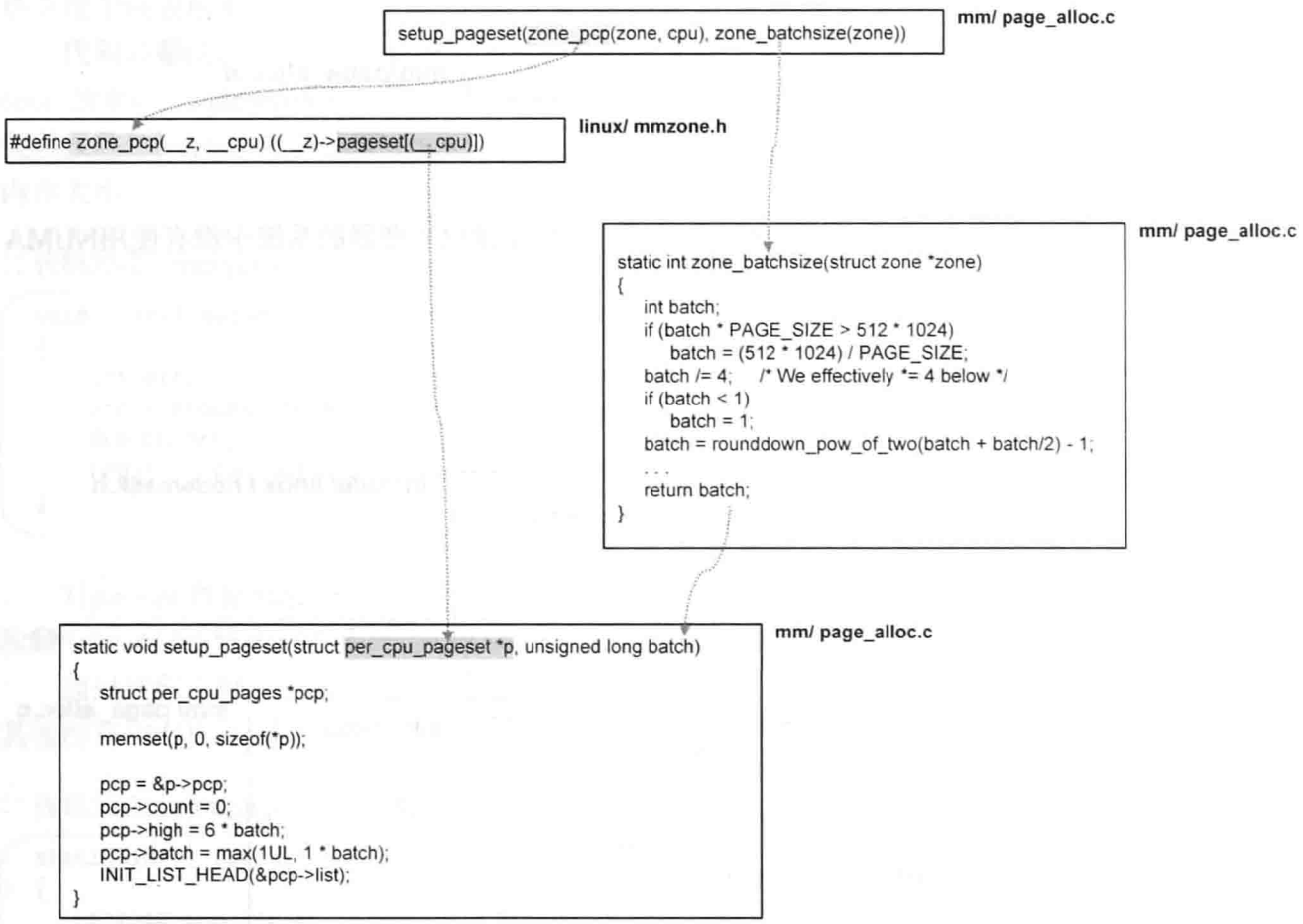


图23-4　setup_pageset()

图23-4表示代码段❸的整体执行过程。

由setup_pageset()初始化的per_cpu_pages结构体的域中，最重要的部分是决定batch大小。batch大小是根据high水印值从伙伴系统执行分配/解除的内存大小的单位。为有效管理内存，决定batch大小十分重要。batch大小通过zone_batchsize()函数形成，如图23-4所示。现在仔细观察zone_batchsize()函数。

:: 代码23-4　mm/page_alloc.c的zone_batchsize()

```
static int zone_batchsize(struct zone *zone)
{
    int batch;
    batch = zone->present_pages / 1024;                    ❶
    if (batch * PAGE_SIZE > 512 * 1024)
        batch = (512 * 1024) / PAGE_SIZE;
    batch /= 4;
    if (batch < 1)
        batch = 1;
```

```
    batch = rounddown_pow_of_two(batch + batch/2) - 1;                    ❷
    ...
    return batch
}
```

在代码段❶中求出当前zone中的相当于所有页的1/1024的页数，使其大小不超过512MB。将相当于该值25%大小的页数设置为batch的大小。

通过代码段❷使用rounddown_pow_of_two()函数，将求出的batch大小按2^n-1的值进行排列。和2^n-1形态的大小对齐是因为根据经验最小化缓存别名（cache aliasing）问题。batch的大小由当前zone内的页数（present_pages）的量决定。

但将其大小限制为不超过512MB。让热&冷页列表大小不超过512MB且用2^n-1表示是因为考虑到L2缓存大小，由此作为单页缓存以提高性能。

per-cpu数据pageset是每个CPU持有的单页缓存，我们已经完成了对正在启动的CPU的pageset设置。如果是SMP系统，则需要对其他所有CPU执行pageset设置。

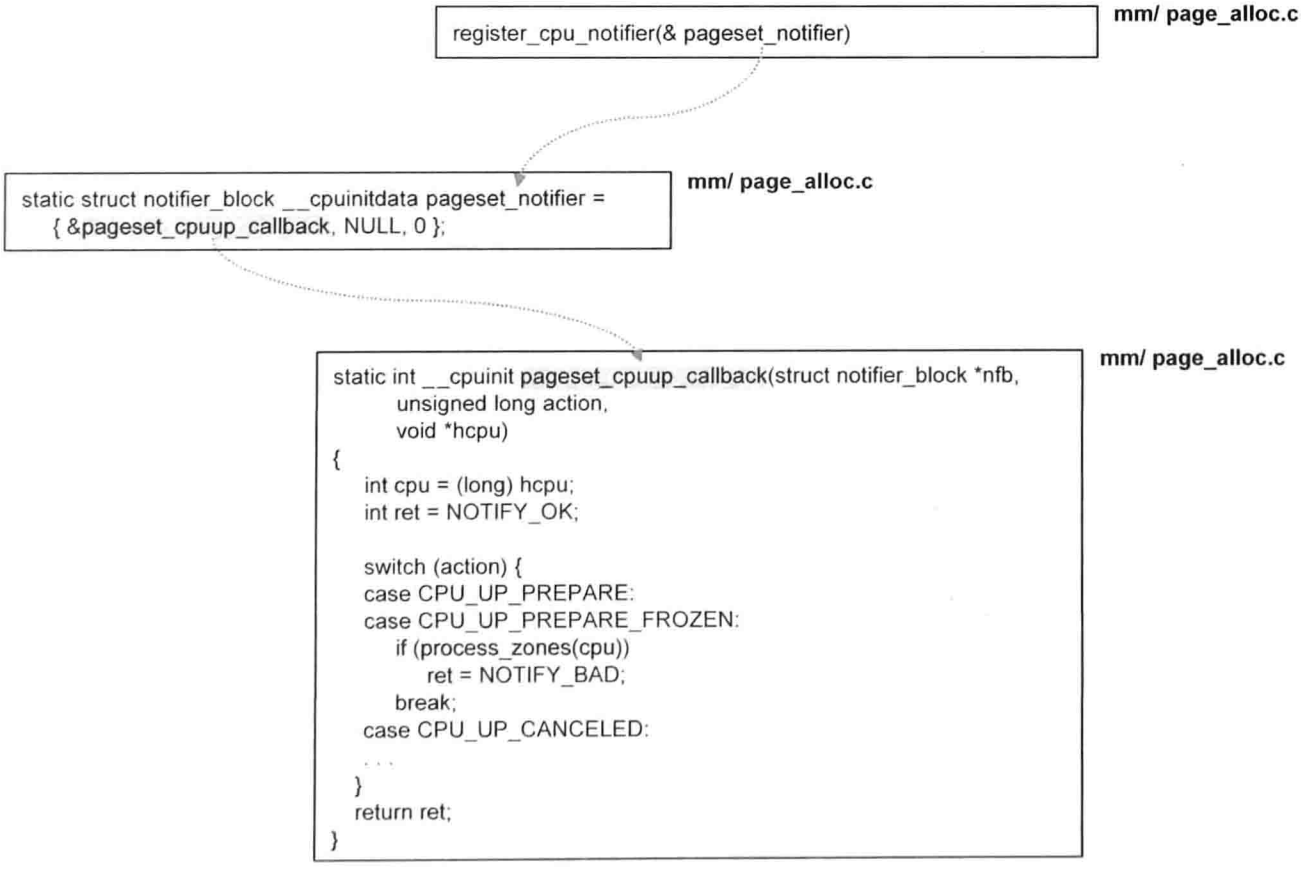


图23-5 pageset_notifier

为了对SMP内的其他所有CPU设置pageset，在代码23-2的代码段❷中注册pageset_notifier。通过图23-5可了解pageset_notifier的执行流程。如果SMP系统内的其他CPU处于准备启动状态，就按之前的步骤进行，通过process_zone()函数对相应的本地CPU执行pageset分配及初始化。

23.3　指定交叉节点——numa_policy_init()

NUMA系统中的内存策略决定内核将在哪个节点分配内存，内存策略表现为下列mempolicy结构体，由策略模式、标签、节点标记组成。

:: include/linux/mempolicy.h

```
struct mempolicy {
    ...
    unsigned short mode;
    unsigned short flags;
    union {
        short            preferred_node;
        nodemask_t       nodes;
    } v;        // system nodemask
    ...
};
```

初始启动时，MPOL_DEFAULT模式被设置为默认策略。但该策略试图在zone节点中执行内存分配，因此，启动过程中发生的内存分配会导致特定节点过载。

所以在numa_policy_init()函数中，将具有16MB以上内存的节点指定为交叉节点，并设置当前进程使用MPOL_INTERLEAVE模式。设置后可防止启动过程中的分配被分散到多个节点，使分配请求聚集到特定节点。该函数中还分配内存策略相关slab缓存，详细内容请参考代码23-5。

:: 代码23-5　mm/mempolicy.c的numa_policy_init()

```
void __init numa_policy_init(void)
{
    nodemask_t interleave_nodes;
    unsigned long largest = 0;
    int nid, prefer = 0;

    policy_cache = kmem_cache_create("numa_policy",                    ❶
                                     sizeof(struct mempolicy),
                                     0, SLAB_PANIC, NULL);

    sn_cache = kmem_cache_create("shared_policy_node",                 ❷
                                 sizeof(struct sp_node),
                                 0, SLAB_PANIC, NULL);

    nodes_clear(interleave_nodes);
    for_each_node_state(nid, N_HIGH_MEMORY) {                          ❸
        unsigned long total_pages = node_present_pages(nid);
```

```
        /* Preserve the largest node */
        if (largest < total_pages) {
            largest = total_pages;
            prefer = nid;
    }

    /* Interleave this node? */
    if ((total_pages << PAGE_SHIFT) >= (16 << 20))
        node_set(nid, interleave_nodes);
    }

    /* All too small, use the largest */
    if (unlikely(nodes_empty(interleave_nodes)))                    ❹
        node_set(prefer, interleave_nodes);

    if (do_set_mempolicy(MPOL_INTERLEAVE, 0, &interleave_nodes))    ❺
        printk("numa_policy_init: interleaving failed\n");
}
```

在代码段❶、❷中生成mem_policy、sp_node结构体的slab缓存。代码段❸中对包含高端内存区域的所有节点执行循环。此时，如果内核未使用CONFIG_HIGHMEM设置，则N_HIGH_MEMORY被定义为N_NORMAL_MEMORY，如下所示。

∷ include/linux/nodemask.h

```
enum node_states {
    ...
    N_NORMAL_MEMORY,
#ifdef CONFIG_HIGHMEM
    N_HIGH_MEMORY,
#else
    N_HIGH_MEMORY = N_NORMAL_MEMORY,
#endif
    ...
};
```

此处对N_HIGH_MEMPRY节点执行循环的同时找到最大节点，将节点号和页数分别保存到prefer、largest。这是为了在没有16MB以上的节点时退而求其次，将最大节点用作交叉节点。

具有16MB以上大小的节点时，为表示其为交叉节点，在interleave_nodes位图中设置对应于节点号的位。如果在代码段❹中即使对所有节点执行循环也未设置interleave_nodes位图，那就在interleave_nodes位图中设置退而求其次求出的最大节点。

最后，在代码段❺中调用do_set_mempolicy()生成内存策略并适用于任务。适用内存策略后，该进程的内存分配将根据指定的内存策略执行。

通过下列代码仔细观察do_set_mempolicy()是如何运行的。

:: 代码23-6　mm/mempolicy.c的do_set_mempolicy()

```
static long do_set_mempolicy(unsigned short mode, unsigned short flags,
                             nodemask_t *nodes)
{
    struct mempolicy *new;
    struct mm_struct *mm = current->mm;

    new = mpol_new(mode, flags, nodes);                                    ❶
    if (IS_ERR(new))
        return PTR_ERR(new);

    if (mm)
        down_write(&mm->mmap_sem);
    mpol_put(current->mempolicy);                                          ❷
    current->mempolicy = new;

    mpol_set_task_struct_flag();                                           ❸

    if (new && new->mode == MPOL_INTERLEAVE &&                             ❹
        nodes_weight(new->v.nodes))
            current->il_next = first_node(new->v.nodes);

    if (mm)
        up_write(&mm->mmap_sem);

    return 0;
}
```

在代码段❶中调用mpol_new()，生成用于记述内存策略的结构体。代码段❷中有使用过的策略时，返回mempolicy slab缓存。然后将代码段❶中生成的内存策略反映到当前进程。

在代码段❸中为当前进程的task_struct设置PF_MEMPOLICY标签，表示内存策略已适用于当前任务中。代码段❹中，若生成的策略使用MPOL_INTERLEAVE模式且具有交叉节点，则在当前进程信息结构体的il_next成员变量中保存最快编号的交叉节点。



补充知识点　NUMA内存策略

MPOL_DEFAULT

该模式只在内部API中使用，表示返回系统现有策略。使用下列3种模式之一后被清除时，使用该策略返回。默认策略使用zone分配。

MPOL_BIND

该模式表示只在已定的节点包中分配内存。内存分配在离分配请求发生最近的节点（属于该节点包的节点之一）进行。分配失败时不在其他节点另外分配。

MPOL_PREFERRED

该模式表示内存分配必须在策略给定的1个节点中形成。分配失败之后，尝试在附近其

他节点执行分配。

MPOL_INTERLEAVE

将多个节点组成1个节点包后执行内存分配。分配失败后在其他节点重新分配。

23.4 结束计时器初始化——late_time_init()

执行调用过的time_init()中未能完成的初始化。late_time_init按如下所示被定义为函数指针，根据结构设置其他函数的地址以进行调用。ARM中默认不设置函数，所以是NULL，不会被调用。

:: init/main.c

```
void (*__initdata late_time_init)(void);
if (late_time_init)
    late_time_init();
```

23.5 测定 BogoMIPS——calibrate_delay()

calibrate_delay()函数检测BogoMIPS值，该值显示1jiffy内消耗多少CPU周期。BogoMIPS是李纳斯想到的，在Linux系统中用作显示处理器性能的相对测定值。MIPS（Milions of Instruction Per Second，每秒百万条指令）是最常用的电脑标准，但仅用MIPS显示电脑性能存在一定的局限性。

BogoMIPS通过计时器中断和循环编程检测1jiffy需要多少CPU时间。通过代码23-7了解calibrate_delay()函数。

:: 代码23-7 init/main.c的calibrate_delay()

```
void __cpuinit calibrate_delay(void)
{

        ...
        while ((loops_per_jiffy <<= 1) != 0) {                    ❶
            /* wait for "start of" clock tick */
            ticks = jiffies;
            while (ticks == jiffies)
                /* nothing */;
            /* Go .. */
            ticks = jiffies;
            __delay(loops_per_jiffy);   /* busy waiting */
            ticks = jiffies - ticks;
            if (ticks)
                break;
        }

        loops_per_jiffy >>= 1;
        loopbit = loops_per_jiffy;
```

```
        while (lps_precision-- && (loopbit >>= 1)) {                    ❷
            loops_per_jiffy |= loopbit;
            ticks = jiffies;
            while (ticks == jiffies)
                /* nothing */;
            ticks = jiffies;
            __delay(loops_per_jiffy);
            if (jiffies != ticks)              /* longer than 1 tick */
                loops_per_jiffy &= ~loopbit;
        }
    }

    printk(KERN_CONT "%lu.%02lu BogoMIPS (lpj=%lu)\n",
            loops_per_jiffy/(500000/HZ),
            (loops_per_jiffy/(5000/HZ)) % 100, loops_per_jiffy);
}
```

在代码段❶中将loops_per_jiffy值设置为4096，然后每增加2倍的同时求出1jiffy内可执行的最大count值——loops_per_jiffy值。

在代码段❷中，将loops_per_jiffy值除以2之后的loopbit值每减少2倍的同时求出更精确的值。__delay()函数是每个结构均实现的函数，若该时间内未发生计时器中断，则jiffies值和ticks值相同。

23.6　制作位图以分配进程识别符（ID）——pidmap_init()

进程分别拥有固定账号——PID。为分配固有PID值，pidmap_init()中会生成位图并进行初始化。

内核2.6.24版以后引入PID命名空间，分析代码前先了解PID命名空间。

一般情况下，命名空间用于对某个相同的名称进行区域分离，以防发生冲突。该概念同样适用于PID命名空间。相同命名空间内无法存在相同PID。但不同命名空间中会存在相同PID，且相同PID间无任何关联。

Linux中默认使用init_pid_ns命名空间，也可以生成新的命名空间以管理PID。PID命名空间作为具有父子关系的层次结构进行管理。接下来观察下列代码。

:: 代码23-8　kernel/pid.c的pidmap_init()

```
void __init pidmap_init(void)
{
    init_pid_ns.pidmap[0].page = kzalloc(PAGE_SIZE, GFP_KERNEL);        ❶
    set_bit(0, init_pid_ns.pidmap[0].page)                              ❷
    atomic_dec(&init_pid_ns.pidmap[0].nr_free)                          ❸
    init_pid_ns.pid_cachep = KMEM_CACHE(pid,                            ❹
            SLAB_HWCACHE_ALIGN | SLAB_PANIC);
}
```

为了分配PID，代码段❶向init_pid_ns命名空间结构体中的pidmap数组的第一个元素分配位

图。在代码段❷中通过set_bit设置0号位，表示正在使用PID 0。既然已经分配了PID 0，那么在代码段❸将具有可分配值的nr_free成员变量的值减1。PID 0是idle进程，可想象为内核自身。最后，在代码段❹中生成名为“pid”的slab缓存以结束pidmap_init()。

pidmap_init()的执行过程如图23-6所示。

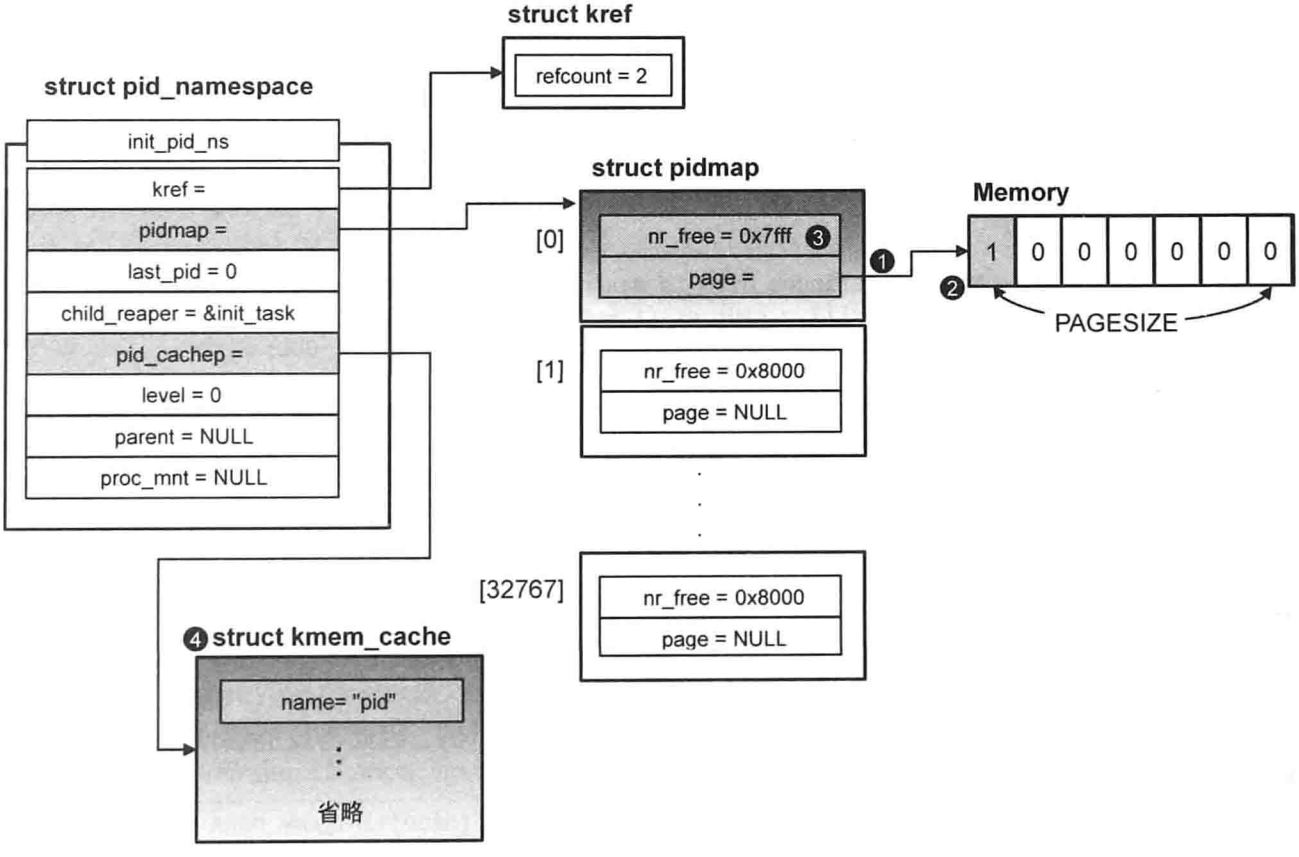


图23-6 执行pidmap_init()

补充知识点 swapper

0号进程有时也称为swapper。Linux内核的每个进程都具有lifetime。现代的冯•诺依曼体系结构中，程序必须积攒到内存后才能执行，但内存容量是限定的。Linux内核中执行多个进程，其中有等待用户响应的或处于sleep状态的进程，这些进程无需立即积攒到内存。

此时，swapper用磁盘对内存中积攒的进程执行换出，以确保需要立即执行的进程内存。需要再次通过磁盘执行移动的进程时，就重新从磁盘向内存执行换入。这就是swapper的作用。swapper是BSD中出现的术语，据说BSD中为0号进程赋予了page swapping功能。

但Linux内核中通过kswapd内核线程执行该功能。准确地说，具有pid 0的进程更适合称为idle进程而非swapper。

23.7　初始化优先树的数据结构——prio_tree_init()

优先搜索树（priority search tree）将堆（heap）和基数（radix）树组合在一起。PST与现有树不同，它拥有2个值（起始地址、最终地址）而不是1个键值。最终地址用于堆索引，具有最大堆索引的节点是顶端节点，整体结构与堆类似。Linux中因系统内存不足而需要回收页缓存时使用该数据结构。

:: lib/prio_tree.c

```
void __init prio_tree_init(void)
{
    unsigned int i;

    for (i = 0; i < ARRAY_SIZE(index_bits_to_maxindex) - 1; i++)
        index_bits_to_maxindex[i] = (1UL << (i + 1)) - 1;
    index_bits_to_maxindex[ARRAY_SIZE(index_bits_to_maxindex) - 1] = ~0UL;
}
```

在该函数中预先计算并分配用于求出PST的maxindex的数组。该数组中进入2^n−1的值，如1、3、7、15、…2^31−1、2^32−1。

23.8　生成 anon_vma slab 缓存——anon_vma_init()

Linux中可分配的内存不足时需要回收页，此时会优先回收最近未使用的页。在页表项中检查最近是否使用过页。页回收处理是通过搜索page结构体执行的，因此需要能够使用该结构体访问页表项的方法，这就是anon_vma数据结构。[①]

① 内核版本2.6.6中，page结构体内有指向自身页表项的成员变量pte，但需要相当多的内存，因此2.6.7开始用anon_vma结构体替代。

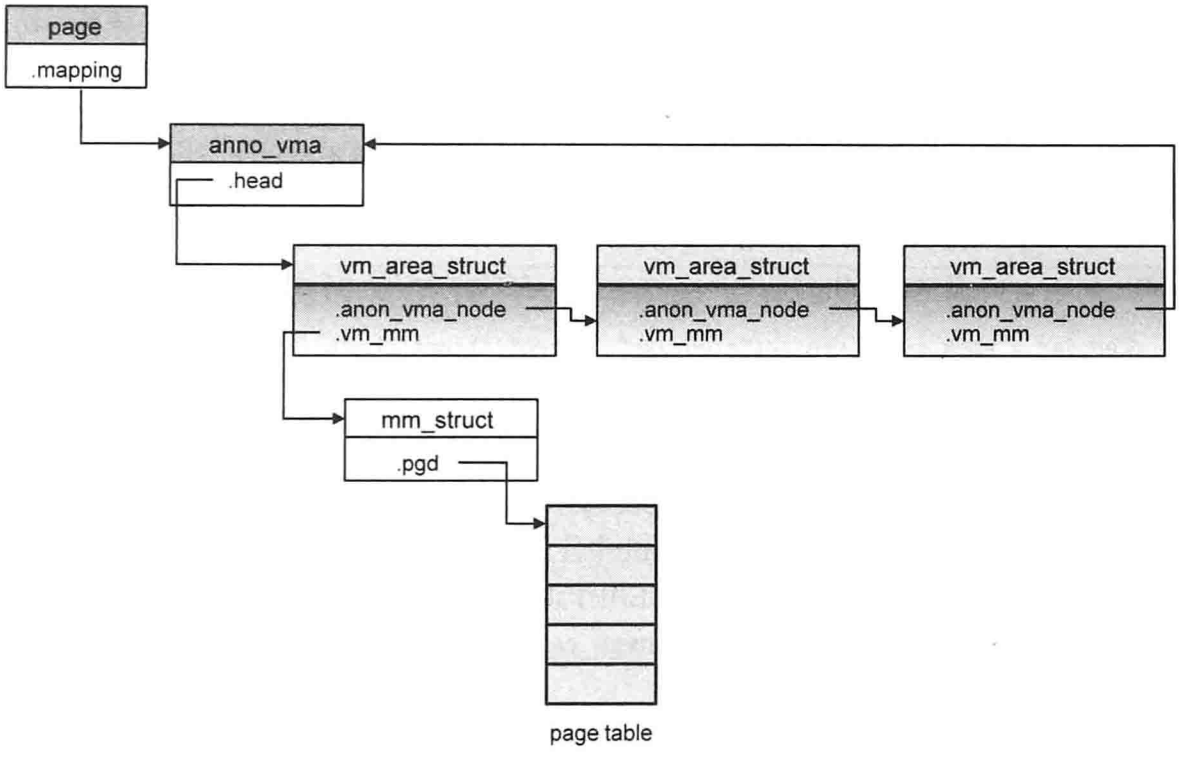


图23-7 通过anon_vma搜索页表

anon_vma结构体指向vm_area_struct结构体列表，各vm_area_struct的vm_mm成员变量指向mm_struct，最终可访问各页表项。在该函数中生成该结构体的slab缓存。

:: 代码23-9 mm/rmap.c的anon_vma_init()

```
void __init anon_vma_init(void)
{
    anon_vma_cachep = kmem_cache_create("anon_vma", sizeof(struct anon_vma),          ❶
                0, SLAB_DESTROY_BY_RCU|SLAB_PANIC, anon_vma_ctor);
}
```

在代码段❶中生成anon_vma结构体的slab缓存，作为生成器指定anon_vma_ctor()。

23.9 为对象的每个用户赋予资格——cred_init()

Linux中为了区分对象有无执行某项操作的资格，按照不同用户制作并使用凭证（credential）。此处的“对象”是任务、文件、套接字、消息队列、共享内存段、信号量等。凭证在多用户系统中十分重要，它将对象连接到特定用户和特定用户组。通过凭证决定各对象能否执行操作，从而保障各用户数据的完整性和整体系统的稳定性。

要想使用凭证，就必须得到“要访问对象的数据结构”和“要保护对象的资源”这两方面的支持。先考虑在任务中读写特定文件的情形。文件系统中，各文件由特定用户持有并属于几个用

户组。文件持有者分为自身、文件用户组、其他所有用户以赋予文件访问权限。在任务中访问文件时，VFS根据任务凭证和文件凭证允许/阻止访问。

用于管理各任务凭证的数据结构和函数位于kernel/cred.c中，从start_kernel()调用对其进行初始化的函数。观察代码23-10。

:: 代码23-10　kernel/cred.c的cred_init()

```
void __init cred_init(void)
{
    cred_jar = kmem_cache_create("cred_jar", sizeof(struct cred),                    ❶
                                 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
}
```

cred_init()进行初始化以使用凭证，代码段❶将可保存凭证的slab保存到全局变量。作为第一个参数，为了指称该slab缓存，将/proc/slabinfo中使用的名称命名为“cred_jar”。将硬件缓存线中排列缓存内对象的SLAB_HWCACHE_ALINE用作slab的标签。

全局变量cred_jar按如下所示被声明为kmem_cache结构体的指针变量，用于全部cred.c文件。

:: kernel/cred.c

```
static struct kmem_cache *cred_jar;
```

23.10　初始化数据结构以使用 fork()函数——fork_init()

该函数为了使用fork()而初始化fork()内使用的系统资源相关数据。仔细观察代码23-11。

:: 代码23-11　kernel/fork.c的fork_init()

```
void __init fork_init(unsigned long mempages)
{
    task_struct_cachep =                                                          ❶
        kmem_cache_create("task_struct", sizeof(struct task_struct),
        ARCH_MIN_TASKALIGN, SLAB_PANIC, NULL);

    ...

    max_threads = mempages / (8 * THREAD_SIZE / PAGE_SIZE);                       ❷
    if(max_threads < 20)
        max_threads = 20;

    init_task.signal->rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;                ❸
    init_task.signal->rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
    init_task.signal->rlim[RLIMIT_SIGPENDING] =                                   ❹
                            init_task.signal->rlim[RLIMIT_NPROC];
}
```

代码段❶生成task_struct结构体的slab缓存，代码段❷利用系统的全部页帧数求出系统中可生成的最大线程数。此外指定最小值为20个。用max_threads检查copy_process()中生成线程时是否超过最大数。

代码段❸对限制进程资源使用的rlimit结构体变量进行初始化。最后，在代码段❹中设置可延迟信号的最大数。

23.11 初始化生成进程的缓存——proc_caches_init()

内核运行过程中，系统生命周期内经常生成并结束进程。有多种结构体可生成进程，且需要经过多个步骤分配/解除。

为此，proc_caches_init()利用生成进程所需的各结构体slab缓存解决普遍发生的碎片问题，并防止反复进行初始化。此外，由于slab缓存支持硬件缓存排列和coloring，故可提高系统活用度并改善性能。通过代码分析proc_caches_init()中生成哪些slab缓存。

:: 代码23-12 kernel/fork.c的proc_caches_init()

```
void __init proc_caches_init(void)
{
    sighand_cachep = kmem_cache_create("sighand_cache",                    ❶
            sizeof(struct sighand_struct), 0,
            SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_DESTROY_BY_RCU,
            sighand_ctor);
    signal_cachep = kmem_cache_create("signal_cache",
            sizeof(struct signal_struct), 0,
            SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
    files_cachep = kmem_cache_create("files_cache",
            sizeof(struct files_struct), 0,
            SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
    fs_cachep = kmem_cache_create("fs_cache",
            sizeof(struct fs_struct), 0,
            SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
    mm_cachep = kmem_cache_create("mm_struct",
            sizeof(struct mm_struct), ARCH_MIN_MMSTRUCT_ALIGN,
            SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);
    vm_area_cachep = KMEM_CACHE(vm_area_struct, SLAB_PANIC);

    mmap_init();                                                           ❷
}
```

代码段❶通过kmem_cache_create()函数（22.8节）生成多种slab缓存。

表23-1是代码段❶中生成的slab缓存和请求从slab缓存分配内存的函数。

表23-1　proc_caches_init()中生成的slab缓存

slab缓存名	相应对象（结构体）	请求分配的函数
"sighand_cache"	struct sighand_struct	copy_sighand
"signal_cache"	struct signal_struct	copy_signal
"files_cache"	struct files_struct	copy_files
"fs_cache"	struct fs_struct	copy_fs
"mm_struct"	struct mm_struct	copy_mm
"vm_area_struct"	struct vm_area_struct	copy_vma

生成slab缓存后，在代码段❷中调用mmap_init。mmap_init()如图23-8所示。

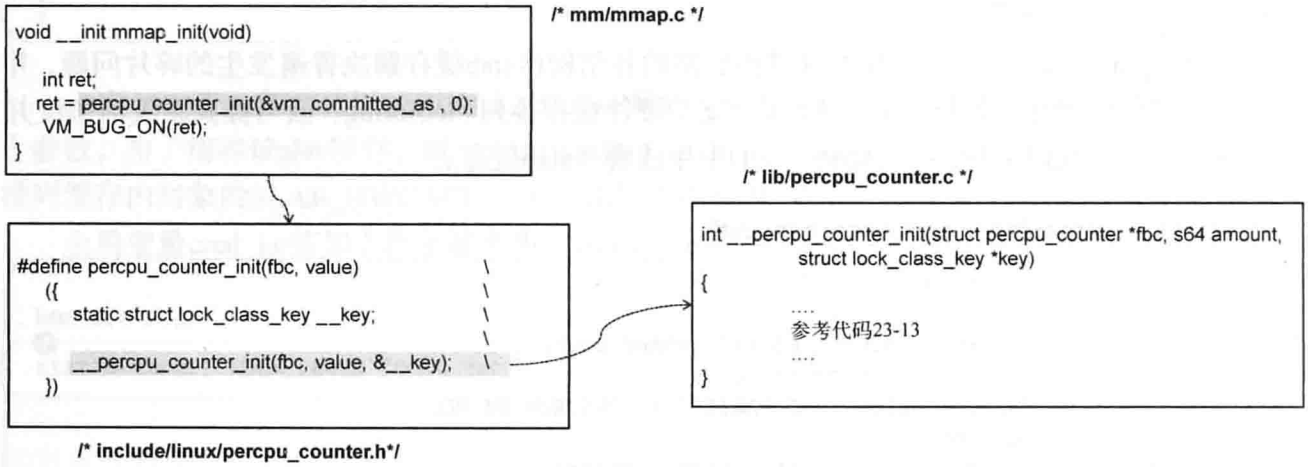


图23-8　调用mmap_init()

最终调用__percpu_counter_init()函数，如代码23-13所示。

:: 代码23-13　lib/percpu_counter.c的__percpu_counter_init()

```
int __percpu_counter_init(struct percpu_counter *fbc, s64 amount,
                struct lock_class_key *key)
{
    spin_lock_init(&fbc->lock);
    lockdep_set_class(&fbc->lock, key);
    fbc->count = amount;                                    ❶
    fbc->counters = alloc_percpu(s32);
    if (!fbc->counters)
        return -ENOMEM;
    ...
    return 0;
}
```

参数fbc是vm_committed_as。最终，在代码段❶中为vm_committed_as->count设置0，在percpu区域动态分配s32类型（signed int）的内存以保存vm_committed_as->counters。

此处初始化的vm_committed_as变量在进程分配新的虚拟内存时反映vm_acct_memory()函数

中分配的内存值。此外，也用于查看__vm_enough_memory()函数中根据overcommit设置的映射内存是否充足。

可通过/proc/meminfo确认该值。通过cat命令确认的结果如下所示。

```
code@kernel$ cat /proc/meminfo
...
CommitLimit:     1429732 kB
Committed_AS:     326164 kB
VmallocTotal:     122880 kB
VmallocUsed:        7488 kB
VmallocChunk:     110124 kB
HardwareCorrupted:     0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       4096 kB
DirectMap4k:       24568 kB
DirectMap4M:      884736 kB

code@kernel$
```


补充知识点 overcommit

内存overcommit是一种内核功能，能够分配比应用程序实际可用内存更多的内存。某些应用程序在执行过程中获得较多内存，但不怎么使用。因此，应用程序不使用已分配的内存时，overcommit可执行比实际内存大小更多的应用程序。

如果应用程序在这种情况下使用内存，内核将结束应用程序。可在内核中调整这种overcommit操作。详细内容请参考相关内核文件（Documentation/vm/overcommit-accounting）。

Linux内核中的overcommit有3种选项。

0：决定内核是否允许overcommit时采用启发式（Heuristic）方法。该值为默认值。

1：始终执行overcommit。可意外结束应用程序。

2：防止超过特定水印。该水印可在/proc/sys/vm/overcommit_ratio中设置值并调整。所有commit不得超过交换空间大小+overcommit_ratio*磁盘大小。overcommit_ratio的默认值为50。

该值可通过如下两种方法适用。

1. `code@kernel:~$ echo 'vm.overcommit_memory = 1' >> /etc/sysctl.conf`
2. `code@kernel:~$ sysctl vm.overcommit_memory = 1`

大部分运行在默认模式中无特别状况，但第2个能提供更好的保护。

23.12 初始化缓冲缓存——buffer_init()

Linux内核以块（block）为单位执行与块设备进行的所有I/O运算。如果说页是所有与内存相

关的I/O运算中使用的逻辑（logical）单位，块则是与块设备的I/O运算中使用的物理（physical）单位。从块设备读取的块通过buffer_head元数据维持在内存中。

将最近从块设备读取的块在内存上进行缓存，可缩短对块I/O运算的块设备访问时间。这就是缓冲缓存。内核启动过程中，buffer_init()为缓冲缓存对必要的元数据buffer_head结构体构建slab缓存。

:: 代码23-14　fs/buffer.c的buffer_init()

```
void __init buffer_init(void)
{
    int nrpages;
    bh_cachep = kmem_cache_create("buffer_head",                                    ❶
            sizeof(struct buffer_head), 0,
                (SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|
                SLAB_MEM_SPREAD),
                init_buffer_head);

    nrpages = (nr_free_buffer_pages() * 10) / 100;                                 ❷
    max_buffer_heads = nrpages * (PAGE_SIZE / sizeof(struct buffer_head));
    ...
}
```

首先在代码段❶中利用kmem_cache_create()函数分配buffer_head结构体的slab缓存。其生成器为init_buffer_head()函数，只将新分配的buffer_head结构体初始化为0。

在代码段❷通过nr_free_buffer_pages()求出构建ZONE NORMAL区域的所有页中相当于当前可用页10%的页数，并保存到nrpages变量。这是用于在系统内执行缓冲缓存的页数。

接下来在代码段❸中利用nrpages求出可存在于系统内的buffer_head结构体的最大个数，并保存到max_buffer_heads。这些max_buffer_heads对缓冲缓存起到水印作用，若系统内存在max_buffer_heads以上的buffer_head结构体，则对已修正的“脏块”（dirty block）执行块设备回写（write-back）。

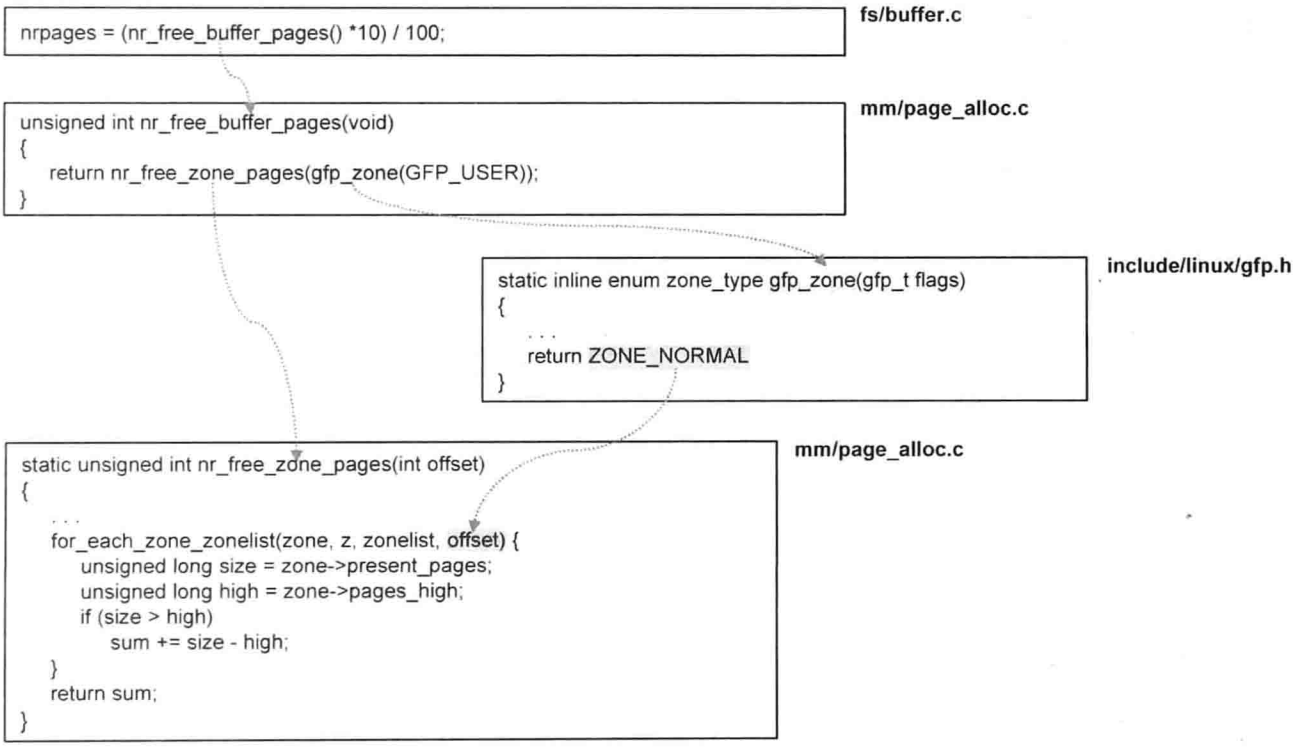


图23-9 nr_free_buffer_pages

观察图23-9可以看到，代码段❷的nr_free_buffer_pages()如前所述求出并返回可在ZONE_NORMAL区域中使用的页数。

:: 代码23-15 mm/page_alloc.c的nr_free_zone_pages()

```
static unsigned int nr_free_zone_pages(int offset)
{
    ...
    for_each_zone_zonelist(zone, z, zonelist, offset) {                    ❶
        unsigned long size = zone->present_pages;
        unsigned long high = zone->pages_high;
        if (size > high)
            /*size - high值在相应zone中释放（free）的页*/
            sum += size - high;
    }

    return sum;
}
```

先观察代码段❶。偏移量接收传递来的函数参数ZONE_NORMAL。通过for_each_zone_zonelist对zonelist执行循环，求出ZONE_NORMAL区域中的空白页（free page）数。此处空白页指，zone内除空洞（hole）外的所有页数（zone->present_pages）中，除去zone内保存的页的最大值（zone->pages_high）后的剩余页空间。

23.13　准备密钥——key_init()

内核中的密钥是显示加密数据、认证令牌、密钥环（keyring）等的单位。密钥在内核内部表现为key结构体的数据结构，具有如下属性。

(1) 序列号（serial）：为各密钥赋予的固有数字

(2) 类型（type）：根据密钥执行的操作定义类型

(3) 描述（description）：在密钥搜索中配对的字符串

(4) 访问控制信息（user）：密钥持有者的用户ID或组ID

(5) 结束时间（expiry）：密钥结束时间，可永久指定

(6) 有效载荷（payload）：表现为实际密钥的数据，在密钥环中是密钥列表

(7) 状态（flags）：各密钥的默认状态值，具有和Uninstantiated、Instantiated、Negative、Expired、Revoked、Dead相同的值。

:: include/linux/key.h

```
struct key {
    key_serial_t                serial;
    struct key_type             *type;
    struct key_user             *user;
    time_t                      expiry;
    unsigned long               flags;
    char                        *description;

    union {
        unsigned long                       value;
        void                            *data;
        truct keyring_list              *subscriptions;
    } payload;

    ...
};
```

内核内部具有保留安全密钥的机制，内核文件中把这种机制命名为内核密钥保留服务（kernel key retention service）。通过该服务使用加密密钥和认证令牌，支持按不同用户赋予权限或在文件系统和内核内部保存管理权限的密钥等操作。还可以使用密钥环，钥匙圈形态的密钥环是一种密钥类型，具有对其他密钥的链接。

通过下列Linux内核设置激活密钥服务。

```
"Security options" / "Enable access key retention support" (CONFIG_KEYS)
```

在start_kernel()中调用key_init()以初始化密钥服务，如代码23-16所示。

:: 代码23-16 security/keys/key.c的key_init()

```
void __init key_init(void)
{
    key_jar = kmem_cache_create("key_jar", sizeof(struct key),                    ❶
                  0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);

    list_add_tail(&key_type_keyring.link, &key_types_list);                       ❷
    list_add_tail(&key_type_dead.link, &key_types_list);
    list_add_tail(&key_type_user.link, &key_types_list);

    /* record the root user tracking */
    rb_link_node(&root_key_user.node,                                             ❸
                 NULL,
                 &key_user_tree.rb_node);

    rb_insert_color(&root_key_user.node,
                        &key_user_tree);

}
```

在代码段❶中，将可保存密钥的slab分配到全局变量。第一个参数将指称该slab缓存的名称指定为“key_jar”，该名称在/proc/slabinfo中使用。将硬件缓存线中排列缓存内对象的SLAB_HWCACHE_ALIGN用作slab的标签。

在代码段❷中，为具有密钥类型的列表key_types_list添加keyring、dead、user等特殊密钥类型。文件系统等内核服务使密钥类型需要在内核内部注册，用户空间中无法直接注册新类型。

在代码段❸中，为管理密钥的数据结构key_user_tree红黑树添加root_key_user.node。rb_link_node()函数的第二个参数指定树结构的父节点（parent），将该参数添加为NULL的root_key_user.node将成为树的根节点。既然添加了新节点，就调用rb_insert_color函数重新调整（rebalance）树的平衡。



补充知识点 红黑树

红黑树（red-black tree）是在插入、删除时保持树平衡的一种平衡树（balanced tree或B-tree）。具有树的平衡表示即使在最坏的情况下也能在一定时间内结束搜索。[①]

红黑树并不是独立显示的树，而是为完善其他平衡树而诞生的。其系谱如下所示。

AVL树→2-3树→2-3-4树→红黑树

平衡树的鼻祖是AVL树，是1962年由2名苏联科学家G.M.Adelson-Velskii和E.M.Ladies提出的算法，因此命名为AVL。AVL树左边的子高度低于等于右边的子高度，各节点具有平衡度（balance factor）值。平衡度较大时，将节点执行回转以保持平衡。AVL的缺点是，每次

① 树的搜索时间在n个节点组成的树中保障$O(\log_2 n)$。

插入、删除时都要对平衡度的变化执行追踪，具有较多插入、删除的情况。

2-3树的每个节点最多可具有3个子节点，节点中的值就是密钥（key）。左侧子节点的值比密钥小，右侧子节点的值比密钥大。2-3树的特征是每个节点可具有2个值，通过分割（split）使通往叶节点（leaf node）的路径长度均保持一致。2-3树的缺点是，叶节点从3节点执行分割时，若父节点为3节点，则需要对插入路径执行回溯。删除时还需要考虑叶节点、内部节点、2节点、3节点等多种情况。

2-3-4树的概念基本上和2-3树相同，但其具有3个密钥和4个子链接。插入、删除、搜索方法和2-3树类似，但保持平衡的分割方法与2-3树不同。2-3树将中间的密钥提升为父进程，小的放在父节点左侧，大的放在父节点右侧。而2-3-4树的分割发生于向4节点插入新数据时。此外，2-3-4树的插入密钥不是分割对象，因此无需执行回溯。所以，2-3-4树的优点是，AVL树预备插入或删除后，无需执行平衡度检查，且无需保持2-3树预备插入路径。

红黑树就是2-3-4树的二进制树（binary tree）版本。即，2-3-4树的基本概念直接适用于红黑树中。但由于红黑树为二进制树，因此用红色与黑色表现，颜色含义如下所示。

- 红节点：父节点，在2-3-4树中表示3节点或4节点
- 黑节点：根节点（root node）、叶节点（leaf node）

红黑树中执行插入、删除时颜色会变，黑节点具有2个红的子节点时，将2个子节点变为黑节点，自己则变为红节点。

第24章 初始化VFS中使用的多种缓存——vfs_cache_init()

内核提供VFS（Virtual File System，虚拟文件系统）文件系统抽象层，VFS使用多种缓存技术以提高性能。通过vfs_caches_init()函数为使用缓存做准备。

:: 代码24-1 fs/dcache.c的vfs_caches_init()

```
void __init vfs_caches_init(unsigned long mempages)
{
    unsigned long reserve;

    reserve = min((mempages - nr_free_pages()) * 3/2, mempages - 1);          ❶
    mempages -= reserve;

    names_cachep = kmem_cache_create("names_cache", PATH_MAX, 0,              ❷
                                     SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);

    dcache_init();                                                            ❸
    inode_init();                                                             ❹
    files_init(mempages);                                                     ❺
    mnt_init();                                                               ❻
    bdev_cache_init();                                                        ❼
    chrdev_init();                                                            ❽
}
```

vfs_caches_init()函数接收mempages参数。mempages是全局变量num_physpages值，具有当前系统所有物理内存的页数信息。num_physpages值在启动过程中决定，详细过程请参考mem_init()函数（第20章）。

在代码段❶中，将相当于当前内核大小1.5倍的页视为被预约（reserve）区域，从系统内具有所有页数信息的num_physpages值中排除。仔细观察代码段❶。通过min()函数将预约区域作为当前系统内存空间与内核大小1.5倍内存空间之间的最小值。通常选择内核大小1.5倍的空间。

从图24-1可以看出，通过nr_free_pages()访问vm_stat全局数组。内核通过vm_stat维持系统内对页的统计数值信息。内核通过“enum zone_stat_item”定义了页可能位于的所有状态类型，vm_stat通过它维持属于各状态的页数信息。

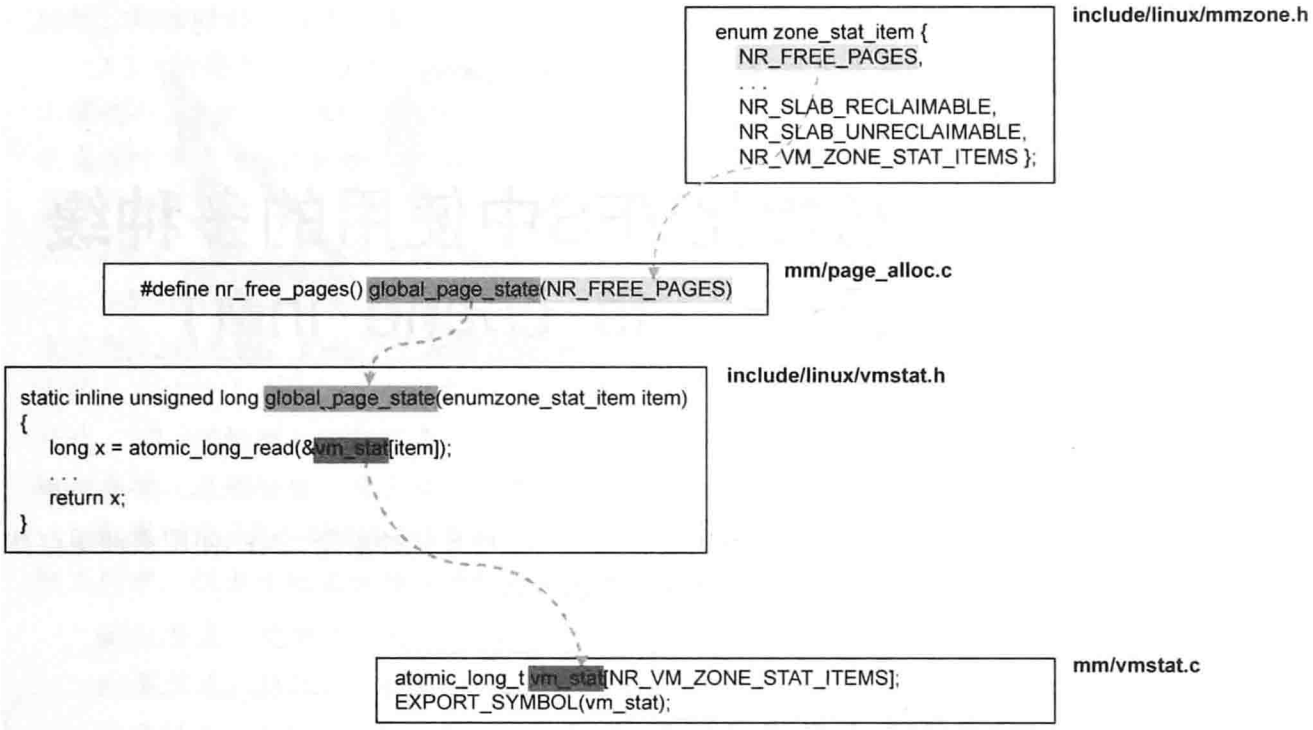


图24-1　nr_free_pages()

例如，通过图24-1的zone_stat_item的NR_SLAB_RECLAIMABLE和NR_SLAB_UNRECLAIMABLE可知，有多少页用于slab缓存。这种对页状态的统计信息不仅进行全局管理，也按各zone进行管理。

内核提供global_page_state() API以浏览vm_stat信息。nr_free_pages()将NR_FREE_PAGES标签传递给global_page_state()函数，返回空白页（free page）数。

代码段❷中生成names_cache slab缓存，它是VFS中为保存路径（path）信息而临时使用的内存空间缓存。通过__getname()、__putname()等函数使用names_cache slab缓存。

在代码段❸中调用dcache_init()。在dcache_init()中为支持对目录项的缓存做准备。生成目录项的slab缓存后，构建散列表进行管理。此外，还设置系统可用内存不足时回收目录项缓存的方式。

在代码段❹中调用inode_init()，顾名思义，它为了对索引节点执行缓存做准备。和初始化目录项一样，生成对索引节点的slab缓存，构建散列表后设置回收方式。

在代码段❺中调用files_init()函数。VFS使用file结构体显示系统内打开（open）的文件。为了管理file结构体，在file_init()中进行如下设置。

第一，系统内各进程每次打开文件时必须生成file结构体。内核为了提高频繁发生的file结构体的分配效率，生成file结构体的slab缓存filp。

第二，VFS通过fdtable（file descriptor table，文件描述符列表）列表管理打开的文件目录，各进程均具有自己的fdtable，通过fdtable_defer_list全局per-cpu列表进行管理。在files_init()中初始化per-cpu列表fdtable_defer_list。

第三，为有效管理内存，内核对内存文件对象（file结构体）的个数设置限定值，每次生成文件对象时记录其个数。与此相关的设置在files_init()函数中实现。

在代码段❻中调用mnt_init()函数。Linux系统中的所有文件系统通过VFS层向用户显示。无论何种文件系统类型，用户通过VFS均具有共同观点，可执行共同运算。

被挂载的文件系统通过vfsmount结构体显示，该结构体还显示VFS层。mm_init()生成vfsmount结构体的mnt_cache slab缓存，虚拟文件系统则初始化sysfs、rootfs。

在代码段❼中调用bdev_cache_init()函数。Linux将与系统连接的所有设备显示为文件，这些文件就是设备文件，和普通文件一样具有索引节点。

块设备相关设备文件通过bdev_inode特殊索引节点显示。在bdev_cache_init()中生成bdev_inode结构体的bdev_cache slab缓存，并初始化bdev虚拟文件系统。

最后，在代码段❽中调用chrdev_init()函数。内核保留着系统内可用字符设备和块设备的散列表，使用cdev_map散列表管理可用字符设备。在chrdev_init()中初始化cdev_map散列表。

以上就是vfs_caches_init()函数内执行的各项操作，下面深入剖析各函数。

首先观察dcache_init()函数，该函数制作VFS使用的页表项的slab缓存，并进行相关设置。

:: 代码24-2　fs/dcache.c的dcache_init()

```
static void __init dcache_init(void)
{
    int loop;

    dentry_cache = KMEM_CACHE(dentry,                                    ❶
        SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|SLAB_MEM_SPREAD);

    register_shrinker(&dcache_shrinker);                                 ❷

    if (!hashdist)                                                       ❸
        return;

    dentry_hashtable =                                                   ❹
    alloc_large_system_hash("Dentry cache",
                            sizeof(struct hlist_head),
                            dhash_entries,
                            13,
                            0,
                            &d_hash_shift,
                            &d_hash_mask,
                            0);

    for (loop = 0; loop < (1 << d_hash_shift); loop++)                   ❺
        INIT_HLIST_HEAD(&dentry_hashtable[loop]);
}
```

在代码段❶中生成dentry结构体的dentry_cache slab缓存。在代码段❷中通过register_shrinker()函数注册dentry_cache的shrinker函数。

内核通过slab分配器对常用对象执行缓存，通过这些slab缓存使频繁发生的内存分配请求快

速接受服务。因此，大部分内核子系统常维持着自己的slab缓存。

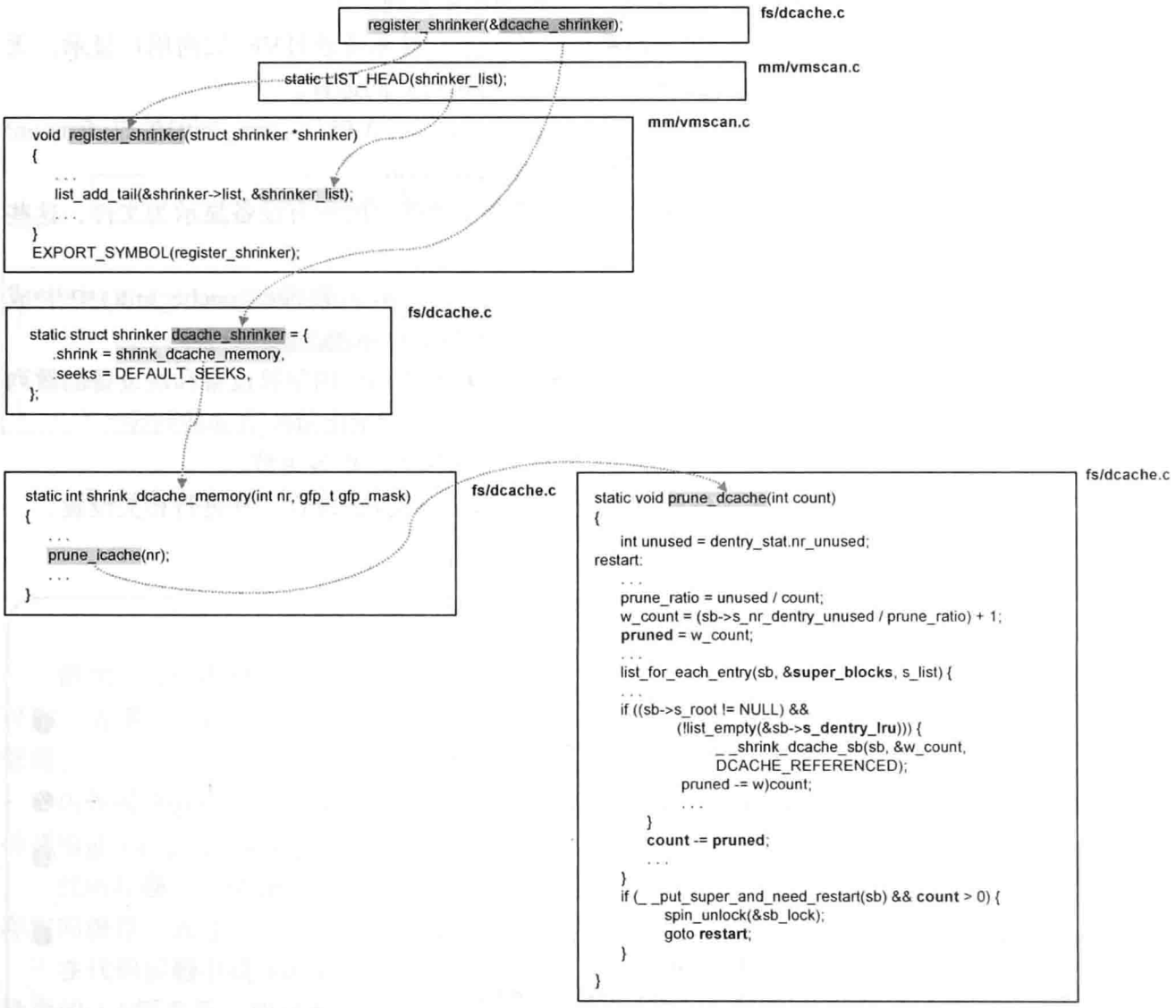


图24-2　register_shrinker(&dcache_shrinker)

而且，为了在内存不足时释放自己正在使用的内存，各子系统可注册shrinker函数。VFS将dcache_shrinker用作目录项的shrinker函数。dcache_shrinker通过register_shrinker()函数注册到全局列表shrink_list。shrink_list中注册了系统内的所有shrinker函数。

如图24-2所示，通过register_shrinker()函数注册的dcache_shrinker的实际执行者是shrink_dcache_memory()函数。该函数通过prune_dcache()函数对可正式回收的目录项缓存执行回收。

内存的dentry对象中，当前不使用的dentry对象（d_count==0）与各文件系统的超级块描述符superblock结构体的s_dentry_lru列表进行连接。一般通过活动列表（active list）和非活动列表（inactive list）这两个列表结构管理系统内所有dentry对象。

当前使用的dentry对象属于活动列表，这是dentry_hashtable全局变量；不使用的dentry对象属

于非活动列表，这是各文件系统superblock结构体中的s_dentry_lru列表。

内存不足时，s_dentry_lru列表中的dentry对象由内核运行的内存回收（memory reclaim）设备执行回收。prune_dcache()函数返回属于非活性列表s_dentry_lru的dentry对象，对所有文件系统的s_dentry_lru执行循环，同时返回一定比例的dentry对象。

内核中内存不足时，为了维持可用内存的平衡，内核启动kswapd内核守护进程。如图24-3所示，kswapd调用shrinker_list中注册的所有shrinker函数，以尝试确保内存。

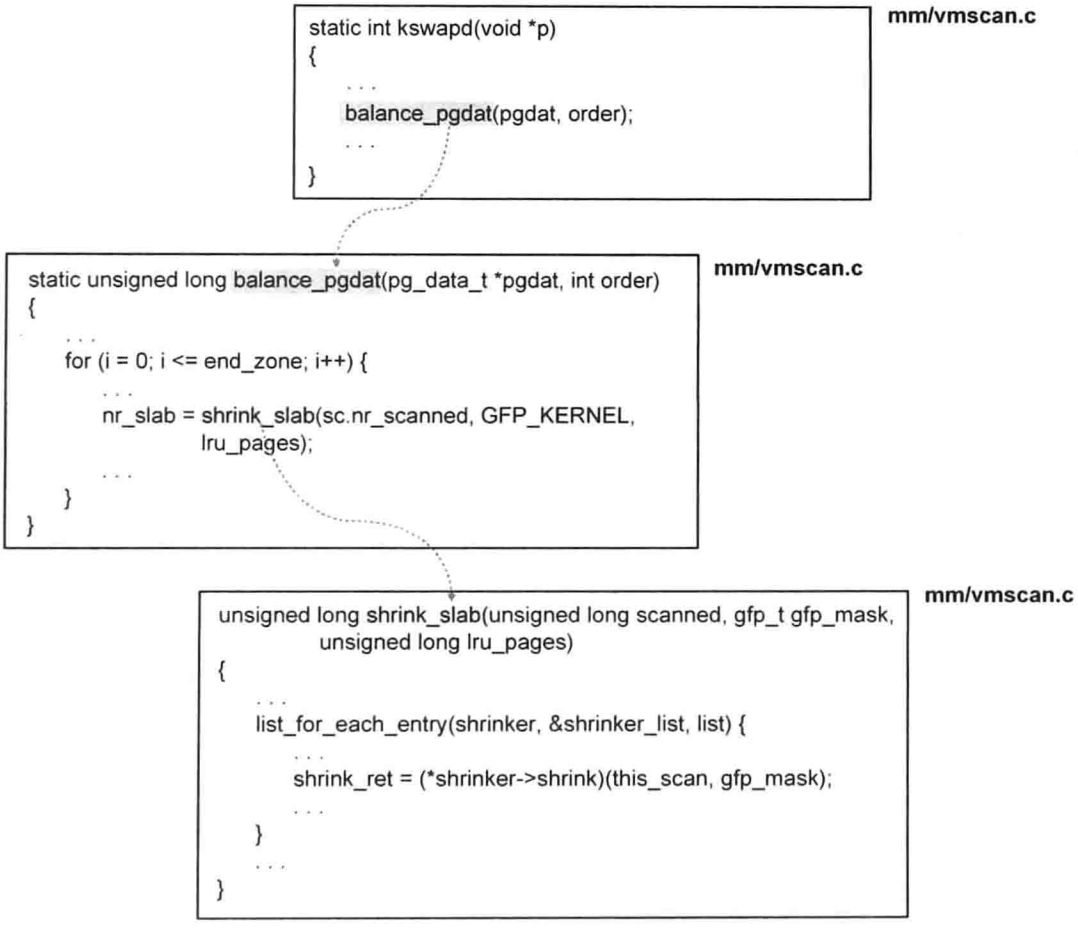


图24-3　kswapd

实际上，kwapd在内存不足时使用多种方法回收内存，shrinker_list的使用只是其中之一。内核正在使用的页不属于内存回收对象，但也有页在未使用状态下被内存回收排除。

大部分页都通过slab分配器分配，索引节点缓存和目录项缓存属于此类。即使不再使用这些缓存，为了以后的可能性，仍把它们留在内存中。内存不足还保留这些缓存真是太奢侈了，所以需要根据开发人员定义的方式回收。

通过前面介绍的register_shrinker()函数将开发人员所需shrinker函数注册到shrinker_list全局列表。各节点均有kswapd内核守护进程，非NUMA系统中只存在单一kswapd。

代码段❹、❺与前面介绍的代码24-2一样，生成散列表以管理目录项并执行初始化。通过代

码段❸决定目录项散列表的生成时点。

如果为非NUMA系统，则通过代码19-14的vfs_cache_init_early()生成目录项和索引节点散列表；如果是NUMA系统。就通过代码24-1 vfs_caches_init()生成。

通常用bootmem、vmalloc、get_free_pages()生成目录项和索引节点的散列表。尤其是NUMA系统中，散列表可能被物理分散，因此需要使用vmalloc。但执行vfs_caches_init_early()的时点上无法初始化vmalloc。

因此，在NUMA系统中，通过可用vmalloc的时点vfs_caches_init()生成用于目录项和索引节点的散列表。

接下来观察inode_init()函数。和目录项的初始化操作一样，生成索引节点的slab缓存并构建散列表，然后设置回收方式。

:: 代码24-3　fs/inode.c的inode_init()

```
void __init inode_init(void)
{
    int loop;

    inode_cachep = kmem_cache_create("inode_cache",                    ❶
                                     sizeof(struct inode),
                                     0,
                                     (SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|
                                     SLAB_MEM_SPREAD),
                                     init_once);

    register_shrinker(&icache_shrinker);                               ❷

    if (!hashdist)                                                     ❸
        return;
    inode_hashtable =                                                  ❹
        alloc_large_system_hash("Inode-cache",
                                sizeof(struct hlist_head),
                                ihash_entries,
                                14,
                                0,
                                &i_hash_shift,
                                &i_hash_mask,
                                0);

    for (loop = 0; loop < (1 << i_hash_shift); loop++)                 ❺
        INIT_HLIST_HEAD(&inode_hashtable[loop]);
}
```

在代码段❶中生成索引节点结构体的inode_cache slab缓存。

在代码段❷中注册对索引节点缓存的内存回收策略icache_shrinker。

观察图24-4。如前所述，和目录项缓存一样，索引节点缓存为内存回收注册额外的shrinker。此时，将shrink_icache_memory()函数用作shrinker函数，prune_icache()函数返回索引节点缓存。

和dentry对象一样，通过活动列表和非活动列表管理内存各索引节点缓存。

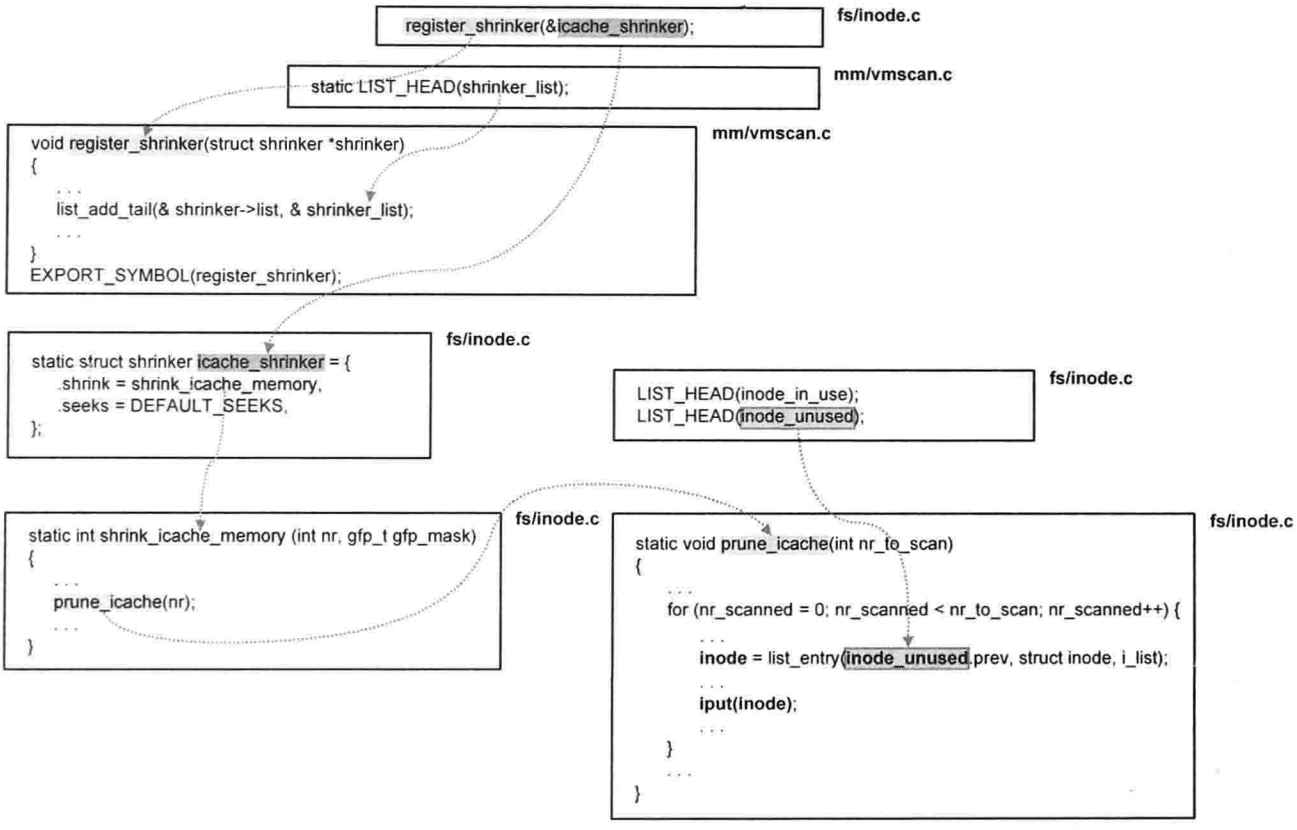


图24-4　register_shrinker(&icache_shrinker)

索引节点缓存的活动列表是inode_in_use全局列表，非活动列表则为inode_unused全局列表。通过prune_icache()返回inode_unused列表内的各索引节点缓存。

代码段❹、❺如代码24-3所述，生成散列表管理各索引节点对象并进行初始化。和目录项散列表的原因相同，在代码段❸中检查散列表是否需要分散生成。

在files_init()函数中设置文件对象slab缓存生成相关设置。

:: 代码24-4　fs/file_table.c的files_init()

```
void __init files_init(unsigned long mempages)
{
    int n;
    filp_cachep = kmem_cache_create("filp", sizeof(struct file), 0,                    ❶
        SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL);

    n = (mempages * (PAGE_SIZE / 1024)) / 10;                                          ❷
    files_stat.max_files = n;
    if (files_stat.max_files < NR_FILE)
        files_stat.max_files = NR_FILE;

    files_defer_init();                                                                ❸
    percpu_counter_init(&nr_files, 0);                                                 ❹
}
```

打开的文件在内核中显示为file结构体，每次打开文件时生成file结构体。在代码段❶中生成名为filp的文件对象slab缓存。

系统所需大部分数据以块设备的文件形态保存。因此，打开文件的操作频繁发生。通过文件对象slab缓存可提高生成文件对象的性能。

但为有效管理内存，内核对系统内可存在的文件对象个数加以限制。内核通过files_stat_struct结构体数据结构管理文件对象相关限制事项。

代码段❷的files_stat是数据类型为files_stat_struct结构体的全局变量。将系统内可分配的文件对象的最大数，即可同时访问的文件最大数保存到files_stat_struct结构体的max_files域。

代码段❷假定文件大小为1KB，将相当于系统可用内存中可存在文件的10%的数定为max_files值。

代码段❸调用files_defer_init()函数。

如前所述，各进程通过fdtable数据结构管理打开的文件，系统内所有fdtable通过名为fdtable_defer_list的per-cpu数据进行管理。

即，per-cpu数据fdtable_defer_list是可访问本地CPU内生成的所有fdtable的单向链表。如图24-5所示，通过fdtable_defer_list_init()函数将free_fdtable_work()函数注册到fdtable_defer_list的工作队列。

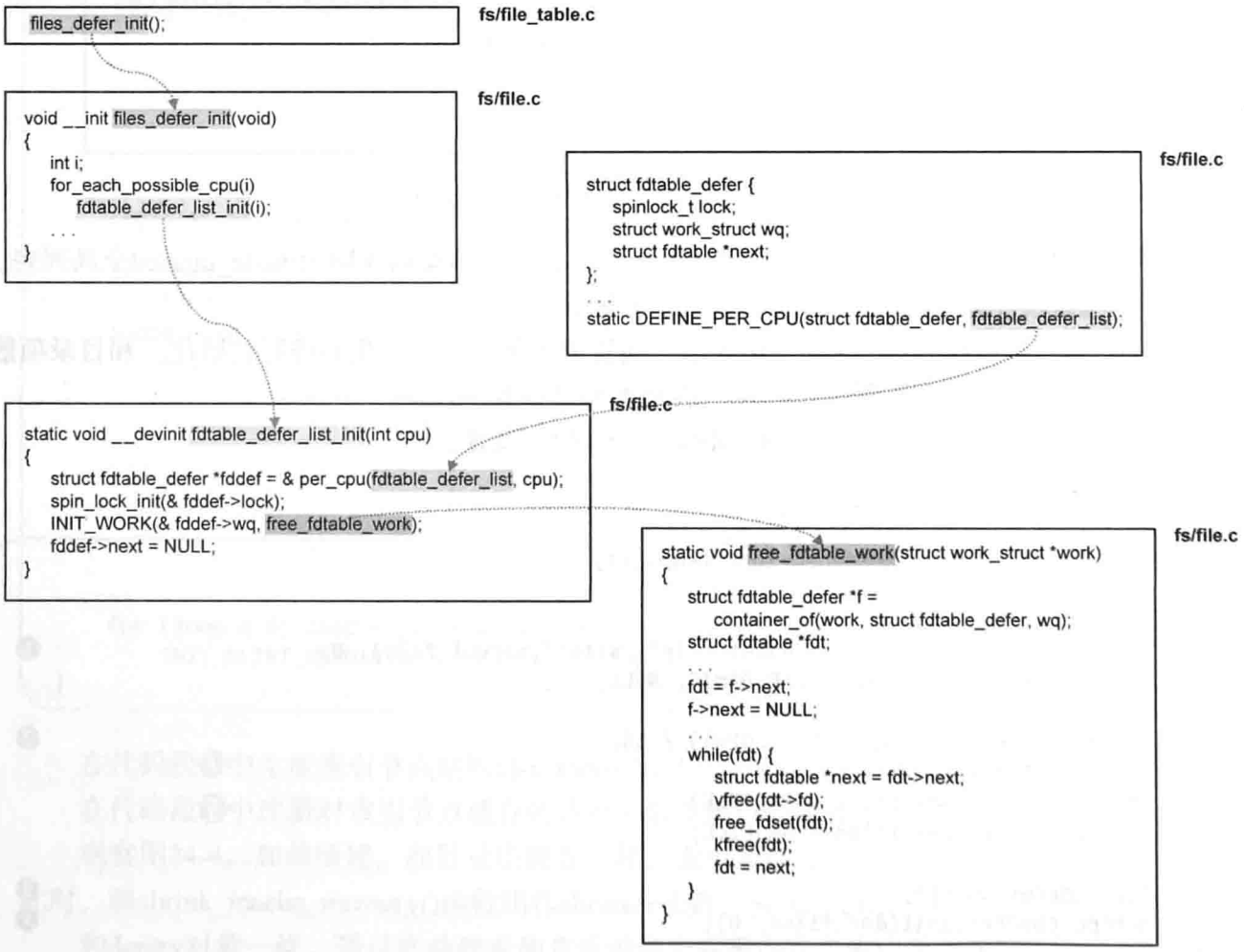


图24-5　files_defer_init函数

free_fdtable_work()函数只释放fdtable_defer_list内连接的所有fdtable的内存。fdtable_defer_list的存在自有原因。

各fdtable均生成工作队列以注册内存释放函数，这给系统性能带来很大负担。如前所述，各进程均具有fdtable对象，因此，每个进程都为返回fdtable对象内存而具有工作队列不得不说是一种负担。

鉴于这些原因，生成fdtable_defer_list的per-cpu列表以共享本地CPU内生成的对fdtable执行内存返回的工作队列，即可减少系统负担。

在代码段❹中调用percpu_counter_init()函数。函数参数nr_files是保存系统内所有文件对象数的全局计数器变量。

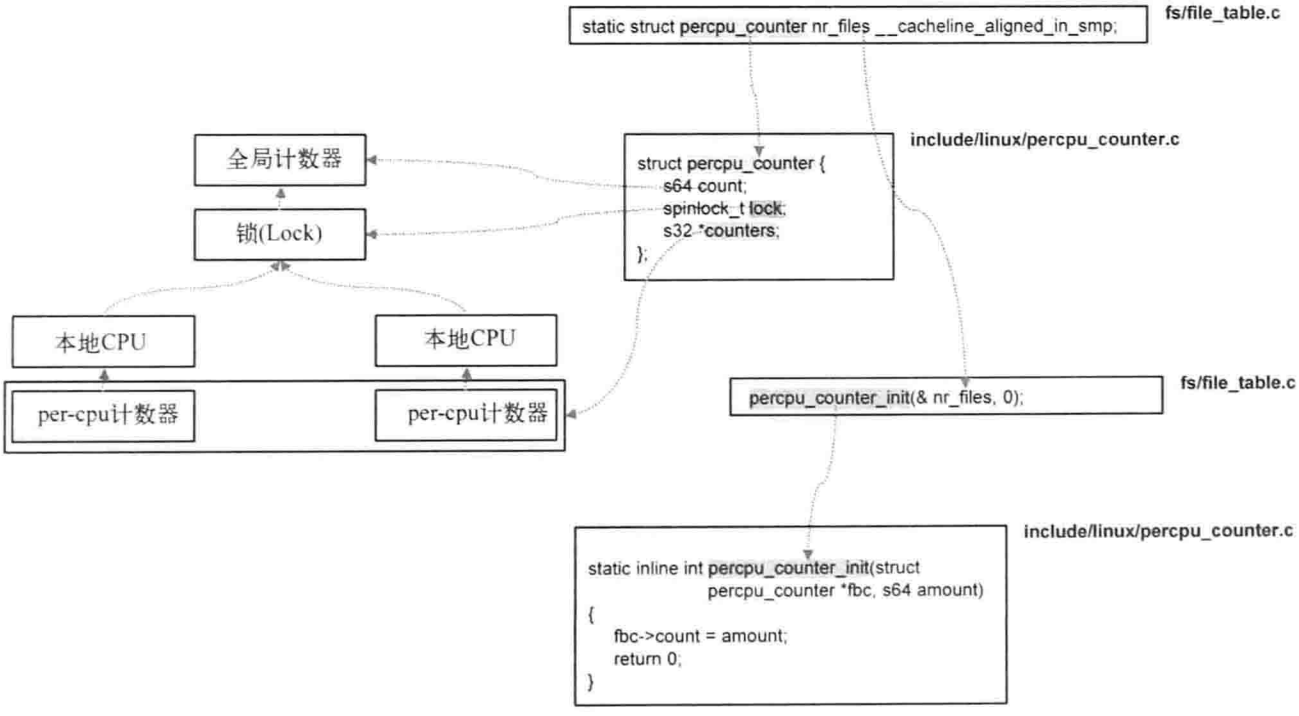


图24-6　nr_files

内核使用计数器管理多种内核资源的统计信息。计数器值是整数值，管理起来十分简单。但SMP环境中，不同核心可能引发相同时点更新计数器值的现象，必须考虑并发（concurrency）与同步（synchronization）问题。

在SMP环境中考虑并发与同步问题设计全局计数器时，最简单的方法就是借助atomic_t等支持原子运算的数据类型。

但是atomic_t在同步问题中具有很大缺陷——锁运算。如果SMP系统内的全局计数器具有频繁修正的特性，则需要执行很多锁运算。为保持同步，缓存线又被频繁移动，导致发生缓存反弹（cache bouncing）现象，从而降低系统性能。

SMP系统中，内核通过per-cpu计数器解决全局计数器面临的问题。使用局部计数器则无需为

同步执行锁运算，可减少并发所需的缓存线移动。

观察图24-6。由于上述原因，为了管理系统内所有文件对象的个数，nr_files全局变量被声明为per-cpu计数器，然后通过percpu_counter_init()函数初始化为0。

per-cpu计数器被记述为percpu_counter结构体。per-cpu计数器由局部计数器（.counters）、锁（.lock）和全局计数器（.count）组成。局部计数器有CPU个数大小，超过一定警戒值时，为全局计数器加值以进行处理。内核提供多种API以操作per-cpu计数器。

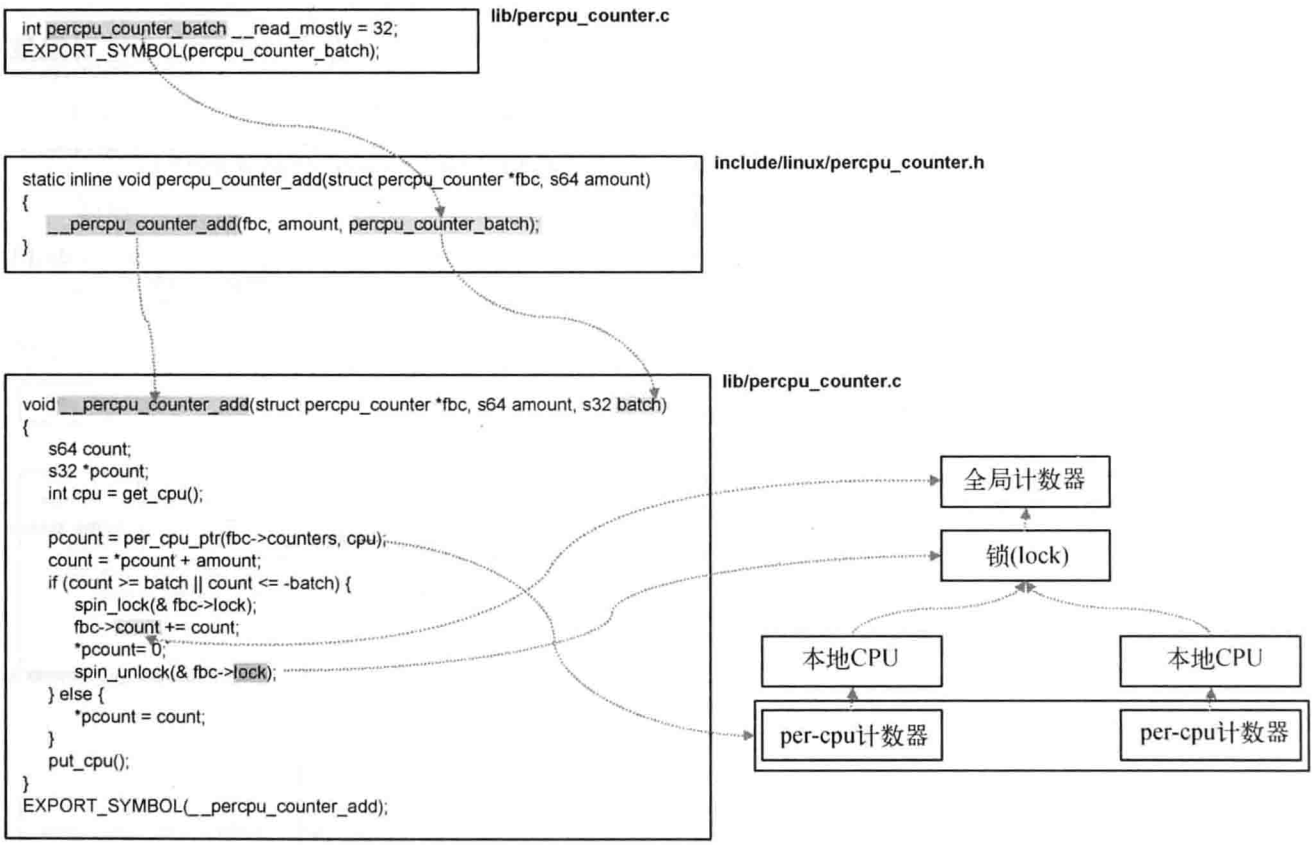


图24-7　percpu_counter_add

图24-7介绍的percpu_counter_add()是操作per-cpu的API之一，各局部计数器的值超过警戒值时会加到全局计数器。

警戒值由percpu_counter_batch全局变量定义，在全局计数器值更新时点为执行同步取锁。

mnt_init()函数为VFS层执行初始化，虚拟文件系统执行sysfs和rootfs的注册。

:: 代码24-5　fs/namespace.c的mnt_init()

```
void __init mnt_init(void)
{
    unsigned u;
    int err;
    ...
    mnt_cache = kmem_cache_create("mnt_cache", sizeof(struct vfsmount),    ❶
```

```
        0, SLAB_HWCACHE_ALIGN | SLAB_PANIC, NULL);

    mount_hashtable = (struct list_head *)__get_free_page(GFP_ATOMIC);          ❷
    ...
    for (u = 0; u < HASH_SIZE; u++)                                             ❸
        INIT_LIST_HEAD(&mount_hashtable[u]);

    err = sysfs_init();                                                         ❹
    ...
    fs_kobj = kobject_create_and_add("fs", NULL);                               ❺
    ...
    init_rootfs();                                                              ❻

    init_mount_tree();                                                          ❼
}
```

代码段❶中生成mnt_cache，它是vfsmount结构体的slab缓存。vfsmount结构体显示被挂载的文件系统位置及状态信息。

代码段❷为mount_hashtable散列表分配内存空间。Linux的文件系统通过挂载过程注册到系统。所有文件系统将根目录的下级目录作为挂载点执行注册。使用散列表快速访问注册到系统的所有文件系统。

该散列表就是mount_hashtable全局列表，将已注册的文件系统的vfsmount结构体地址值和该文件系统的挂载点目录的dentry结构体地址值用作给散列表编索引的散列键值。通过代码段❸的循环语句初始化散列表。

在代码段❹中通过sysfs_init()函数初始化并注册sysfs虚拟文件系统，详情参见代码24-6。

在代码段❺中，以fs的名称为sysfs虚拟文件系统的根目录生成文件系统类。

在代码段❻中调用init_rootfs()函数。

图 24-8 表示 init_rootfs() 执行的所有过程。首先，通过 bdi_init() 函数初始化名为ramfs_backing_dev_info的backing_dev_info类型的结构体。

backing_dev_info结构体是显示设备信息的描述符，具有设备和内核间I/O运算相关设置信息。例如，从块设备预读（read-ahead）的单位、是否适用回写（writeback）等。

ramfs_backing_dev_info结构体是具有ramfs虚拟文件系统相关块设备信息的backing_dev_info结构体。ramfs是虚拟文件系统，只在内存中构建块设备。

因此不存在实际块设备。所以，通过BDI_CAP_NO_ACCT_AND_WRITEBACK标签设置为无回写功能。然后，通过register_filesystem()内核API注册rootfs虚拟文件系统。rootfs注册完成于实质性的基于磁盘的根文件系统注册之前。rootfs是基于内存的虚拟文件系统，生成以后要注册的担任根文件系统挂载点的空目录。之后，实际的根文件系统挂载到通过rootfs制作的挂载点。

在代码段❼的init_mount_tree()中挂载已注册的rootfs虚拟文件系统。

sysfs虚拟文件系统提供系统内所有设备的拓扑（topology）。内核可通过sysfs与用户级互换值，提供设备间的拓扑也有助于内核的电源管理。

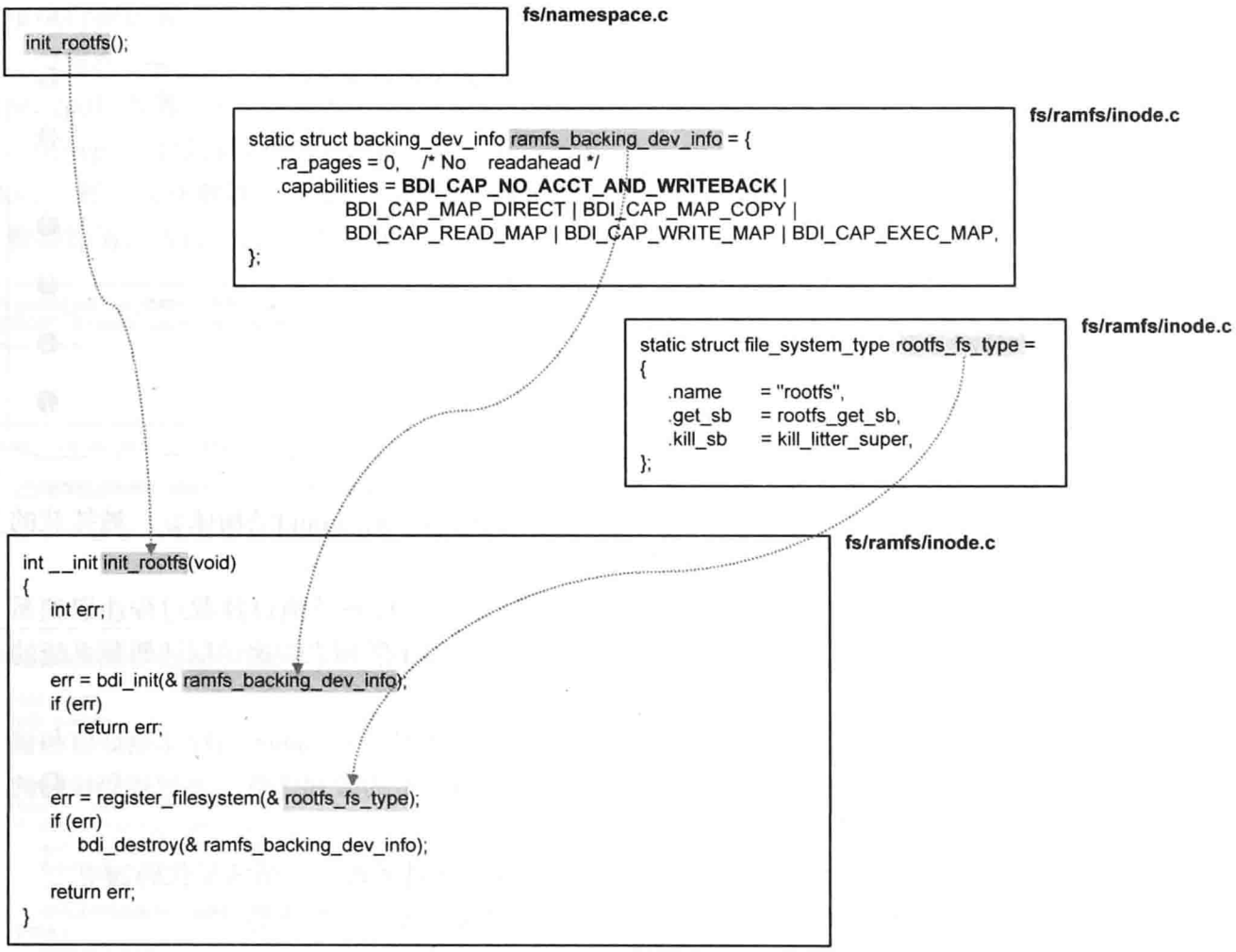


图24-8　init_rootfs

sysfs_init()注册并挂载sysfs虚拟文件系统。

:: 代码24-6　fs/sysfs/mount.c的sysfs_init()

```
int __init sysfs_init(void)
{
    ...

    sysfs_dir_cachep = kmem_cache_create("sysfs_dir_cache",                    ❶
                       sizeof(struct sysfs_dirent),
                       0, 0, NULL);
    ...

    err = sysfs_inode_init();                                                  ❷
    ...
    err = register_filesystem(&sysfs_fs_type);                                 ❸
    if (!err) {
        sysfs_mount = kern_mount(&sysfs_fs_type);                              ❹
        ...
```

```
    } else
        goto out_err;

    ...
}
```

在代码段❶中以sysfs_dir_cache命名生成的sysfs_dirent结构体的slab缓存。与一般文件系统不同，基于内存的虚拟文件系统sysfs使用sysfs_dirent结构体显示目录项，而非现有的dentry结构体。

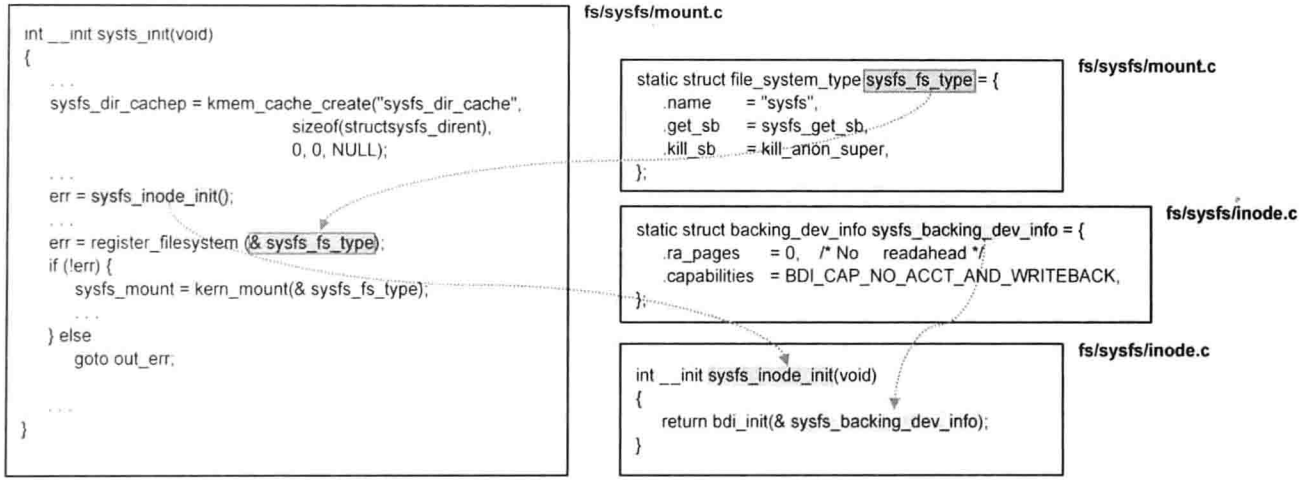


图24-9　sysfs_init()

代码段❷调用sysfs_inode_init()函数。如图24-9所示，sysfs_inode_init()通过sysfs_backing_dev_info对sysfs文件系统相关设置信息进行初始化。backing_dev_info结构体是显示设备信息的描述符。

backing_dev_info结构体的capabilities域显示是否使用内存页在块设备中执行回写。但由于sysfs是基于内存的虚拟文件系统，因此通过BDI_CAP_NO_ACCT_AND_WRITEBACK标签表明不使用回写。

通过代码段❸、❹注册并挂载sysfs。为了挂载sysfs，sysfs_fs_type记述内核要求的多种信息。file_system_type结构体标明内核挂载或未挂载文件系统所需多种信息。

例如，由于文件系统的超级锁保存在块设备中，因此，信息中包含了读取这些超级锁的方法、解锁方法、文件系统名等。

在bdev_cache_init()中生成bdev_inode结构体的bdev_cache slab缓存，并初始化bdev虚拟文件系统。

:: 代码24-7　fs/block_dev.c的bdev_cache_init()

```
void __init bdev_cache_init(void)
{
    int err;
    struct vfsmount *bd_mnt;
```

```
    bdev_cachep = kmem_cache_create("bdev_cache", sizeof(struct bdev_inode),          ❶
                0, (SLAB_HWCACHE_ALIGN|SLAB_RECLAIM_ACCOUNT|
                        SLAB_MEM_SPREAD|SLAB_PANIC),
                init_once);
    err = register_filesystem(&bd_type);                                              ❷
    ...
    bd_mnt = kern_mount(&bd_type);                                                    ❸
    ...
    blockdev_superblock = bd_mnt->mnt_sb;                                             ❹
}
```

在代码段❶中，对bdev_inode结构体生成bdev_cache slab缓存。图24-10表示bdev_cache_init()函数的整体流程。

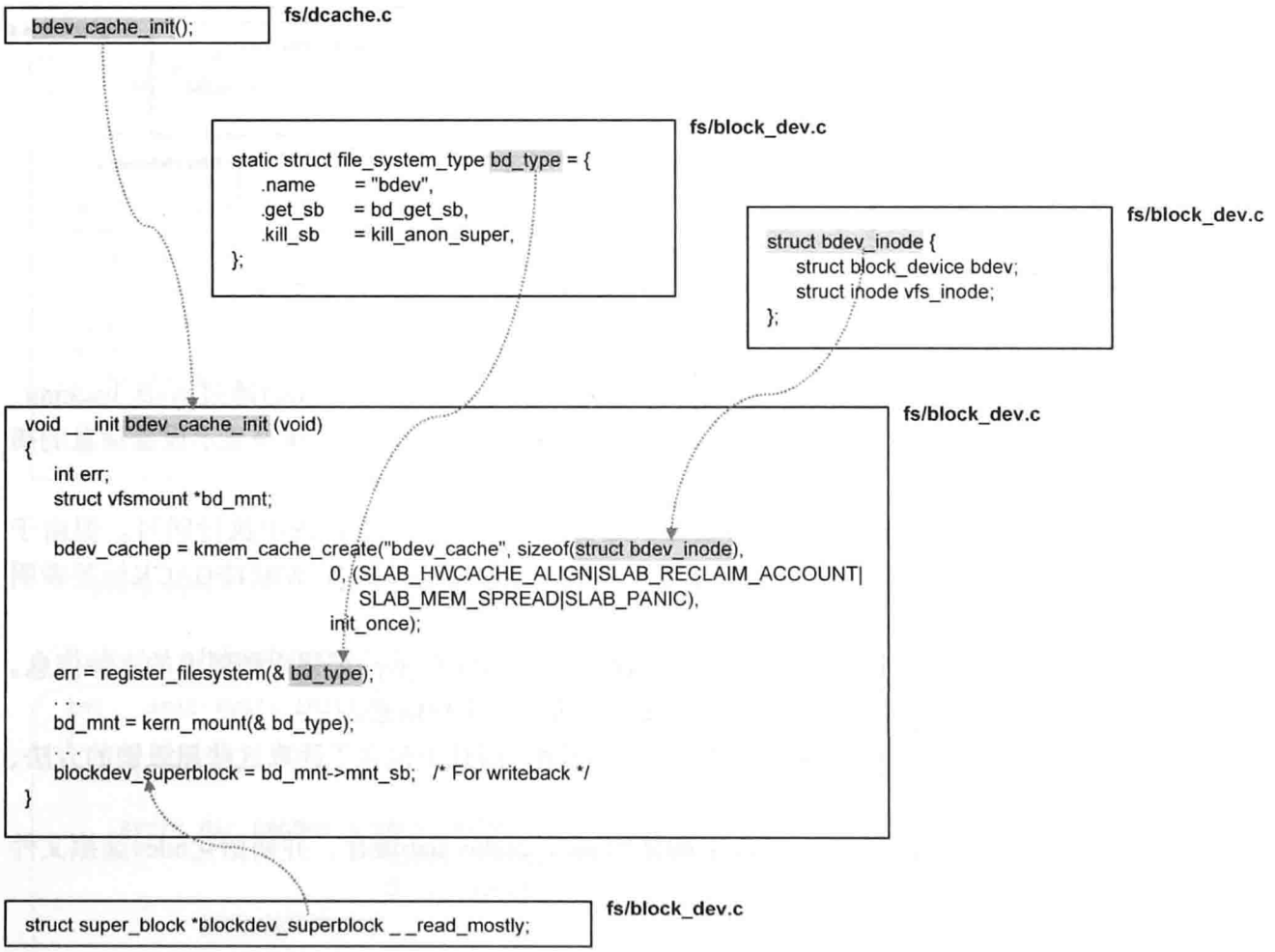


图24-10　bdev_cache_init()

通过block_device结构体显示原始块设备（raw block device）。如前所述，Linux中所有设备均显示为文件，块设备在/dev目录中显示为文件并具有索引节点对象。

bdev_inode结构体是显示原始块设备和设备文件索引节点关系的描述符。

在代码段❷、❸中注册并挂载bdev虚拟文件系统。内核使用bdev内核级虚拟文件系统管理显示块设备的设备文件索引节点。

然后，在代码段❹中让blockdev_superblock全局变量指向已挂载的bdev虚拟文件系统的超级锁。

第 25 章 radix_tree_init()~ftrace_init()

本章分析start_kernel()函数中调用的radix_tree_init()函数到ftrace_init()函数，调用顺序如图25-1所示。

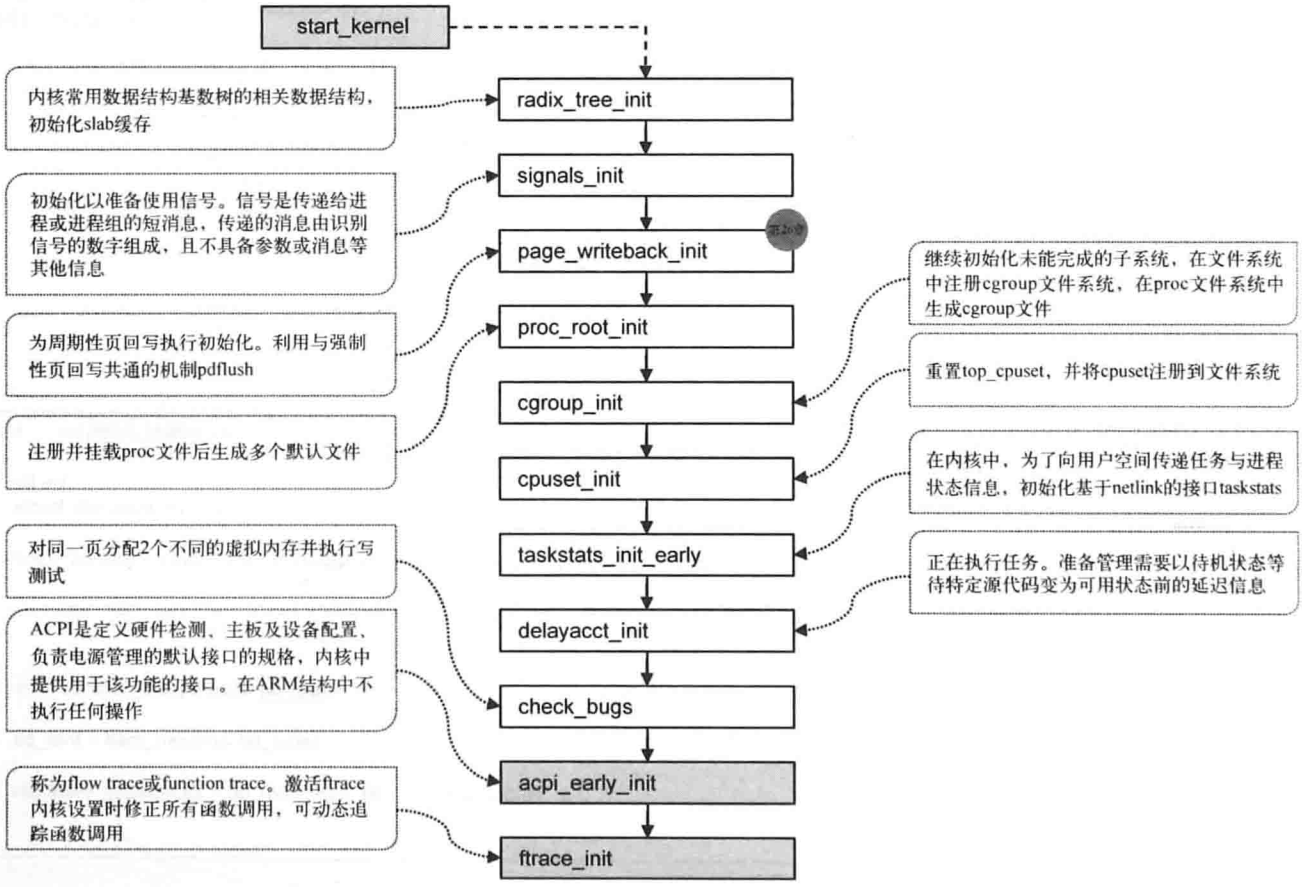


图25-1　从radix_tree_init()到ftrace_init()的调用关系

page_writeback_init()函数初始化对内核使用的内存和块设备等后备存储（backing store）执行同步的周期性页回写（periodic page writeback）机制，要想理解该函数，需要和内核使用的页回写机制共同讲解（第26章）。

acpi_early_init()函数不在ARM中使用，ftrace_init()函数用于函数调用追踪，此处省略。

25.1 基数树相关数据结构初始化——radix_tree_init()

该函数中进行与内核常用基数树相关的初始化，下面是详细内容。

:: 代码25-1 lib/radix-tree.c的radix_tree_init()

```
void __init radix_tree_init(void)
{
    radix_tree_node_cachep = kmem_cache_create("radix_tree_node",                    ❶
                                               sizeof(struct radix_tree_node), 0,
                                               SLAB_PANIC | SLAB_RECLAIM_ACCOUNT,
                                               radix_tree_node_ctor);
    radix_tree_init_maxindex();                                                      ❷
    hotcpu_notifier(radix_tree_callback, 0);                                         ❸
}
```

代码段❶生成内核中利用基数树结构管理数据结构时使用的节点radix_tree_node结构体的slab缓存。它是生成缓存时起到生成器作用的函数，设置负责初始化数据结构的radix_tree_ctor()。

在代码段❷中，初始化构建并使用基数树时可按节点高度保存的且保存最大键值的height_to_maxindex[]。该数据结构的值为实际可具有的高度+1大小的个数。

代码段❸是发生CPU热插拔事件时调用的函数，注册radix_tree_call_back()。发生事件的CPU的状态为CPU_DEAD、CPU_DEAD_FROZEN时，该函数释放相应CPU的radix_tree_node缓存。

25.2 准备使用信号——signals_init()

Unix系统使用信号使进程间相互作用。信号是传递给进程或进程组的短消息，传递的消息由识别信号的数字组成，不具备参数或消息等其他信息。

在运行start_kernel()的过程中执行初始化以使用信号，如代码25-2所示。

:: 代码25-2 kernel/signal.c的signals_init()

```
void __init signals_init(void)
{
    sigqueue_cachep = KMEM_CACHE(sigqueue, SLAB_PANIC);                              ❶
}
```

在代码段❶中为全局变量sigqueue_cachep分配slab缓存。sigqueue_cachep是kmem_cache结构体变量，保存信号位。

:: kernel/signal.c

```
static struct kmem_cache *sigqueue_cachep;
```

KMEM_CACHE()宏函数声明如下。这个宏调用分配slab缓存的kmem_cache_create()，slab缓

存名字是第一个参数__struct的字符串值，slab缓存的大小将成为__struct的大小。

:: include/linux/slab.h

```
#define KMEM_CACHE(__struct, __flags) kmem_cache_create(#__struct,\
            sizeof(struct __struct), __alignof__(struct __struct),\
            (__flags), NULL)
```

从代码段❶分配的slab缓存名为sigqueue，大小为sigqueue结构体的大小。可在/proc/slabinfo/中查看slab缓存名。

25.3　注册并挂载 proc 文件系统——proc_root_init()

proc文件系统提供内核内部数据结构的接口以获得系统信息，或在运行时变更内核参数。proc目录用子目录或文件保存系统中正在运行的进程的信息和重要的内核数据信息。通过自身PID生成的目录可获取进程信息，可通过特定文件查看内核数据信息。此时能看到的文件因内核设置不同而不同。

在proc_root_init()中注册并挂载proc文件系统后生成多个默认文件。

:: 代码25-3　fs/proc/root.c的proc_root_init()

```
void __init proc_root_init(void)
{
    int err;

    proc_init_inodecache();                                          ❶
    err = register_filesystem(&proc_fs_type);                        ❷
    if (err)
        return;

    proc_mnt = kern_mount_data(&proc_fs_type, &init_pid_ns);         ❸
    err = PTR_ERR(proc_mnt);
    if (IS_ERR(proc_mnt)) {
        unregister_filesystem(&proc_fs_type);
        return;
        }

    proc_symlink("mounts", NULL, "self/mounts");                     ❹

    proc_net_init();                                                 ❺

#ifdef CONFIG_SYSVIPC
    proc_mkdir("sysvipc", NULL);                                     ❻
#endif
    ...
    proc_sys_init();
}
```

首先，在代码段❶中生成proc_inode结构体的slab缓存。proc_inode结构体用于连接proc相关数据和VFS层的索引节点，用索引节点表示proc文件系统各项。

然后，在代码段❷中调用register_filesystem()注册proc文件系统。调用时传递的参数proc_fs_type是注册文件系统时使用的file_system_type结构体变量，为注册proc文件系统预先进行了初始化。

在代码段❸中挂载上面注册的proc文件系统。在代码段❹中，将mounts（self[①]/mounts的符号链接）文件生成到proc文件系统。此时使用的函数proc_symlink()执行的功能如下所示。

```
ln -s /proc/self/mounts /proc/mounts
```

在代码段❺中生成self/net的符号链接net，注册网络命名空间子系统。代码段❻在proc filesystem生成sysvips、fs、driver、fs/nfsd、openprom、tty/driver、tty/ldisc divice-tree、sys并注册相关数据结构。

25.4 注册未能初始化的子系统——cgroup_init()

在8.3节的cgroup_init_early()中，我们对init进程使用的cgroup相关数据结构及之前需要初始化的cgroup子系统执行了初始化。该函数中继续初始化未能初始化的子系统，并在文件系统中注册cgroups文件系统，在proc文件系统中生成cgroups文件。

:: 代码25-4 kernel/cgroup.c的cgroup_init()

```
int __init cgroup_init(void)
{
    int err;
    int i;
    struct hlist_head *hhead;

    err = bdi_init(&cgroup_backing_dev_info);                    ❶
    if (err)
        return err;

    for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {                  ❷
        struct cgroup_subsys *ss = subsys[i];
        if (!ss->early_init)
            cgroup_init_subsys(ss);


        ...
    }

    /* Add init_css_set to the hash table */
    hhead = css_set_hash(init_css_set.subsys);                   ❸
```

① 代码段❹、❺中浏览的self目录在启动中是/proc/0的符号链接，该链接指向正在访问/proc/self的进程的proc目录，因此经常变更。

```
    hlist_add_head(&init_css_set.hlist, hhead);

    err = register_filesystem(&cgroup_fs_type);                          ❹
    if (err < 0)
        goto out;

    proc_create("cgroups", 0, NULL, &proc_cgroupstats_operations);       ❺

out:
    if (err)
        bdi_destroy(&cgroup_backing_dev_info);

    return err;
}
```

在代码段❶中，初始化cgroup文件系统的backing_dev_info结构体变量cgroup_backing_dev_info。backing_dev_info结构体保存read-ahead[①]相关信息。

在代码段❷中初始化cgroup的所有子系统。然而，此时cgroup_early_init()中初始化的子系统已经完成了初始化，故此处不再进行。详细内容请回顾8.3.2节，本章只了解初始化子系统即可。

在代码段❸中，为有助于查找（cgroup中使用的）css_set数据结构的css_set散列表添加init_css_set。调用css_set_hash()返回散列表的特定套接字位置列表，连接对应此处的css_set以添加到散列表。

如cgroup_init_early()中所述，css_set是将进程的cgroup_subsys_state组合在一起的集合，css_set散列表是为了易于查找css_set而用散列进行管理的数据结构。init_css_set指向init进程使用的css_set。

在代码段❹中注册cgroup文件系统。最后，代码段❺将cgroups项生成到proc文件系统，并结束函数。读取该项则简单输出cgroup子系统相关信息，如下所示。

```
code@kernel:~/kernelsrc $ cat /proc/cgroups
subsys_name             hierarchy               num_cgroups             enabled
cpuset                  0                       1                       1
ns                      0                       1                       1
cpu                     0                       1                       1
cpuacct                 0                       1                       1
memory                  0                       1                       1
devices                 0                       1                       1
freezer                 0                       1                       1
net_cls                 0                       1                       1
```

25.5　重置 top_cpuset 并注册 cpuset 文件系统——cpuset_init()

之前调用的cpuset_init_early()对指向系统顶端cpuset的top_cpuset进行初始化。在cpuset_init()

① read-ahead是指，读取1个扇区时，预先读取相邻扇区并保存到页缓存。“读取特定扇区后，早晚也会读取其他相邻扇区”的概率较高，该技术就是从这种概率乐观论出发的。由于磁盘存储设备查找扇区的速度比读写扇区速度缓慢得多，该技术提高了不少性能。

中重置top_cpuset，将cpuset作为文件系统执行注册。仔细观察下列函数。

:: 代码25-5 kernel/cpuset.c的cpuset_init()

```
int __init cpuset_init(void)
{
    int err = 0;

    cpumask_setall(top_cpuset.cpus_allowed);                          ❶
    nodes_setall(top_cpuset.mems_allowed);

    fmeter_init(&top_cpuset.fmeter);
    top_cpuset.mems_generation = cpuset_mems_generation++;            ❷
    set_bit(CS_SCHED_LOAD_BALANCE, &top_cpuset.flags);
    top_cpuset.relax_domain_level = -1;

    err = register_filesystem(&cpuset_fs_type);                       ❸
    if (err < 0)
        return err;

    ...

    number_of_cpusets = 1;                                            ❹
    return 0;
}
```

代码段❶初始化cpuset结构体变量top_cpuset的成员变量cpus_allowed、mems_allowed。它们分别记录相应cpuset能用的CPU、内存节点，此处设置为使用所有CPU和节点。

在代码段❷中设置mems_generation，表示top_cpuset的内存设置已变更。设置CS_SCHED_LOAD_BALANCE标签以激活cpuset->cpus_allowed中设置的CPU间的负载均衡。cpuset_mems_generation变量用于对cpuset已变更的内存设置执行同步（19.11节）。

在代码段❸中注册cpuset文件系统，最后代码段❹将numbers_of_cpusets设置为1，从而将系统所有cpuset的个数初始化为1。

25.6 初始化任务统计信息接口——delayacct_init()

本节分析taskstats_init_early()函数，先了解什么是taskstats。

taskstats是可通过用户空间共享任务或进程的内核内部统计信息的基于netlink的接口。内核提供taskstats的优点如下所示。

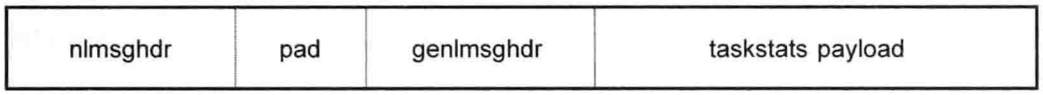
(1) 任务执行期间（lifetime）和结束时，有效提供任务在内核内部的统计信息。

(2) 为多种审计子系统（accounting subsystem）提供统一接口。

(3) 保证以后提供的审计补丁（accounting patch）的扩展性。

通过taskstats接口获取统计信息的方法十分简单，打开unicast netlink套接字（NETLINK_GENERIC family）传送指定PID或TGID的命令即可。内核指定PID时将相应任务的内核内统计信

息传送到用户空间，指定TGID时将进程的所有任务的统计信息进行整合并传送到用户空间。内核空间和用户空间的数据传送将成为netlink消息，消息结构如图25-2所示。



- nlmsghdr : netlink message header
- genlmsghdr : generic netlink message header

图25-2　netlink消息结构

图25-2的taskstats有效载荷积攒以下3种消息之一进行传送。

(1) 从用户空间向内核空间传递的命令（指定TASKSTAS_CMD_ATTR_PID或TASKSTATS_CMD_ATTR_TGID属性）

(2) 从内核空间向用户空间提供的统计信息

(3)完成任务时内核向用户空间传递的消息

※参考文献：内核文件Documentation/taskstats.txt

既然已经了解了taskstats的基本概念，下面观察对其进行初始化的taskstas_init_early()函数，其定义如下所示。

:: kernel/taskstats.c

```
void __init taskstats_init_early(void)
{
    unsigned int i;

    taskstats_cache = KMEM_CACHE(taskstats, SLAB_PANIC);

    for_each_possible_cpu(i) {
        INIT_LIST_HEAD(&(per_cpu(listener_array, i).list));
        init_rwsem(&(per_cpu(listener_array, i).sem));
    }
}
```

该函数的作用十分简单。先生成taskstats结构体的slab缓存，初始化per-cpu数据listener_array的列表及信号量。

25.7　为管理延迟信息做准备——delayacct_init()

本节讲解管理延迟信息的准备阶段——delayacct_init()函数。

25.7.1　延迟审计

执行任务或进程的过程中，有时需要处于待机状态等待特定源代码变为可用状态。

例如，为执行任务等待CPU分配、完成任务开始的块I/O的同步前需要等待、页置换、为可用内存执行内存回收时任务要处于待机状态。这种待机状态可看做用于（非处理时间的）待机的延迟时间（delay time）或待机时间（waiting time），这些延迟时间与任务的优先顺序有密切关系。

因此，延迟时间的信息可用于决定任务的优先顺序。利用延迟信息可以防止任务饥饿（starvation）、调整I/O优先顺序。此外，如果延迟的是十分重要的任务，可以向内核请求提高自身优先顺序。

内核可以管理延迟相关信息，并与用户空间共享。内核提供的延迟统计信息称为延迟审计（delay accounting）。负责对延迟审计执行初始化的函数就是delayacct_init()函数。

25.7.2 delayacct_init

具有任务或进程的延迟统计信息的延迟审计通过taskstats接口向用户空间提供内核空间的信息，编译时需要指定以下2个环境设置器以访问这些信息。

```
CONFIG_TASKSTATS=y
CONFIG_TASK_DELAY_ACCT=y
```

其中，CONFIG_TASK_DELAY_ACCT在Kconfig文件中有如下记述。

:: init/Kconfig

```
config TASK_DELAY_ACCT
    bool "Enable per-task delay accounting (EXPERIMENTAL)"
    depends on TASKSTATS
    help
        Collect information on time spent by a task waiting for system
        resources like cpu, synchronous block I/O completion and swapping
        in pages. Such statistics can help in setting a task's priorities
        relative to other tasks for cpu, io, rss limits etc.
    Say N if unsure.
```

通过代码25-6了解初始化延迟审计的delayacct_init()函数。

:: 代码25-6 kernel/delayacct.c的delayacct_init()

```
struct kmem_cache *delayacct_cache;
...

void delayacct_init(void)
{
    delayacct_cache = KMEM_CACHE(task_delay_info, SLAB_PANIC);          ❶
    delayacct_tsk_init(&init_task);                                     ❷
}
```

代码结构十分简单。代码段❶通过KMEM_CACHE()生成delayacct_cache cache，它构建task_delay_info结构体的slab。代码段❷从slab分配器分配task_delay_info结构体的内存，并与

init_task的init_task.delays连接。这样就完成了保存init_task的延迟及待机信息的准备工作。

25.7.3　task_delay_info结构体和delayacct_tsk_init()

下面分析task_delay_info结构体和delayacct_tsk_init()函数。首先，task_delay_info结构体声明如下。

:: include/linux/sched.h

```
#ifdef CONFIG_TASK_DELAY_ACCT
struct task_delay_info {
    spinlock_t  lock;
    unsigned int    flags;  /* Private per-task flags */

    struct timespec blkio_start, blkio_end; /*块I/O起始时间、结束时间*/
    u64 blkio_delay;    /*块I/O同步完成前待机时间*/
    u64 swapin_delay;   /*用于置换的块I/O完成前待机时间*/
    u32 blkio_count;    /*用于块I/O同步的总I/O数*/
    u32 swapin_count;   /*用于置换的块I/O的总数*/

    struct timespec freepages_start, freepages_end;
    u64 freepages_delay;    /*回收内存前待机时间*/
    u32 freepages_count;    /*回收内存前的总执行数*/
};
#endif  /* CONFIG_TASK_DELAY_ACCT */
```

延迟时间在task_delay_info结构体中以纳米秒积攒并保存到blkio_delay、swapin_delay、freepages_delay成员变量。

接下来分析delayacct_fsk_init()函数，其定义如下所示。

:: include/linux/delayacct.h

```
static inline void delayacct_tsk_init(struct task_struct *tsk)
{
    tsk->delays = NULL;
    if (delayacct_on)
        __delayacct_tsk_init(tsk);
}
```

该函数只将接收的任务结构体的delays成员变量参数初始化为NULL，若已指定delayacct_on则调用__delayacct_tsk_init()函数。①

__delayacct_tsk_init()函数的定义如下所示。

---

① 指定激活及关闭（turn on / turn off）延迟审计的变量，声明如下

kernel/delayacct.c

```
int delayacct_on __read_mostly = 1;
```

:: kernel/delayacct.c

```
void __delayacct_tsk_init(struct task_struct *tsk)
{
    tsk->delays = kmem_cache_zalloc(delayacct_cache, GFP_KERNEL);
    if (tsk->delays)
        spin_lock_init(&tsk->delays->lock);
}
```

调用kmem_cache_zalloc()函数，从delayacct_cache到task_delay_info结构体执行分配后，使任务的delays成员变量指向它。当前任务的延迟信息会保存到delays成员变量指向的task_delay_info结构体。



补充知识点 内核提供的taskstats接口示例

如前所述，保存并管理任务的延迟信息称为延迟审计，从内核空间向用户空间传送信息时使用的接口是taskstats接口。taskstats接口将对应于pid、tgid的统计信息传送到用户空间。内核十分亲切，它为用户提供了活用taskstats接口的应用程序getdelay.c文件。

内核提供的应用程序的位置

```
code@kernel:~/src/liunx-2.6.30.4/Documentation/accounting$ ls
cgroupstats.txt       getdelays    Makefile             taskstats.txt
delay-accounting.txt  getdelays.c  taskstats-struct.txt
```

编译

```
code@kernel:~/src/liunx-2.6.30.4/Documentation/accounting$ gcc -I/usr/src/linux/
include \
 getdelays.c -o getdelays
```

运行

```
code@kernel:~/src/liunx-2.6.30.4/Documentation/accounting$ ./getdelays -d -p 1
print delayacct stats ON
PID     1

CPU             count     real total  virtual total    delay total
                 1650      880000000      854832856      135643679
IO              count    delay total
                  112      489398389
SWAP            count    delay total
                    0              0
RECLAIM         count    delay total
                    0              0
```

输出PID=1的延迟审计信息。

通过本节的delayacct_init()函数，我们了解了任务如何管理执行过程中发生的待机时间或延

迟时间的信息。保存并管理延迟时间称为延迟审计，利用该信息可调整任务的优先顺序，并通过taskstas接口向用户空间提供信息。

25.8　检查写缓冲一致性——check_bugs()

设置CONFIG_MMU时将check_bugs定义为check_writebuffer_bugs，该函数对同一页分配2个不同虚拟内存并执行写测试。

:: 代码25-7　arch/arm/mm/fault-armv.c的check_writebuffer_bugs()

```
void __init check_writebuffer_bugs(void)
{
    struct page *page;
    const char *reason;
    unsigned long v = 1;

    printk(KERN_INFO "CPU: Testing write buffer coherency: ");

    page = alloc_page(GFP_KERNEL);                                    ❶
    if (page) {
        unsigned long *p1, *p2;

        pgprot_t prot = __pgprot(L_PTE_PRESENT|L_PTE_YOUNG|
                                 L_PTE_DIRTY|L_PTE_WRITE|
                                 L_PTE_MT_BUFFERABLE);

        p1 = vmap(&page, 1, VM_IOREMAP, prot);                        ❷
        p2 = vmap(&page, 1, VM_IOREMAP, prot);                        ❸

        if (p1 && p2) {
            v = check_writebuffer(p1, p2);                            ❹
            reason = "enabling work-around";
        } else {
            reason = "unable to map memory\n";
        }

        vunmap(p1);
        vunmap(p2);
        put_page(page);
    } else {
        reason = "unable to grab page\n";
    }

    if (v) {                                                          ❺
        printk("failed, %s\n", reason);
        shared_pte_mask = L_PTE_MT_UNCACHED;
    } else {
        printk("ok\n");
    }
}
```

在代码段❶中分配1页。将分配的页在代码段❷和❸中通过vmap()映射为VMALLOC_START和VMALLOC_END之间的虚拟地址。p1和p2是指向同一页的具有不同地址的虚拟地址。

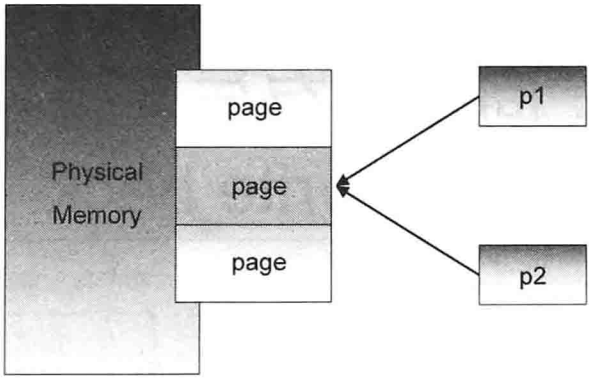


图25-3 执行vmap()后

执行vmap()后如图25-3所示。在代码段❹中向p1和p2写入值，若写入p1的值覆写了p2，则在代码段❺中视为测试成功。

第 26 章

同步内存与后备存储——page write back

页回写将内存中的脏页（dirty page，已修正的页）在块设备等后备存储中清除。内核中执行页回写的方法大致有两种。

① 周期性页回写（periodic page writeback）

设置计时器，每个特定时间周期用后备存储清除内存脏页。

② 强制性页回写（forced page writeback）

sync等系统调用或脏页数超出临界值（limit）时强制清除脏页。

周期性页回写方式和强制性页回写方式均使用共同方法，只是最后调用的函数不同。本章先了解这些内容，然后分析初始化周期性页回写的page_writrback_init()函数。

26.1 页回写机制

图26-1是内核中使用的页回写机制。

首先，在pdflush()函数中调用MIN_PDFLUSH_THREADS大小的start_one_pdflush_thread()函数。start_one_pdflush_thread()函数调用kthread_run()函数生成pdflush线程，并调用pdflush()函数。用pdflush_list管理生成的pdflush线程。

pdflush()函数在内部调用__pdflush()函数，传递pdflush_work结构体参数。在pdflush_work结构体中可指定pdflush线程为了执行页回写要执行的回调函数。向__pdflush()函数传递pdflush_work结构体时，回调函数首先被设置为NULL。pdflush线程调用schedule()让出处理器，并等待wake up信号。

唤醒pdflush线程的函数是pdflush_operation()，该函数接收函数指针参数。pdflush_operation()函数在pdflush_list中查看是否存在可用的pdflush线程，若存在，则获得pdflush的pdflush_work结构体。然后，在该结构体的回调函数中指定接收pdflush_operation()函数参数的函数，并唤醒pdflush线程以执行要页回写的回调函数。

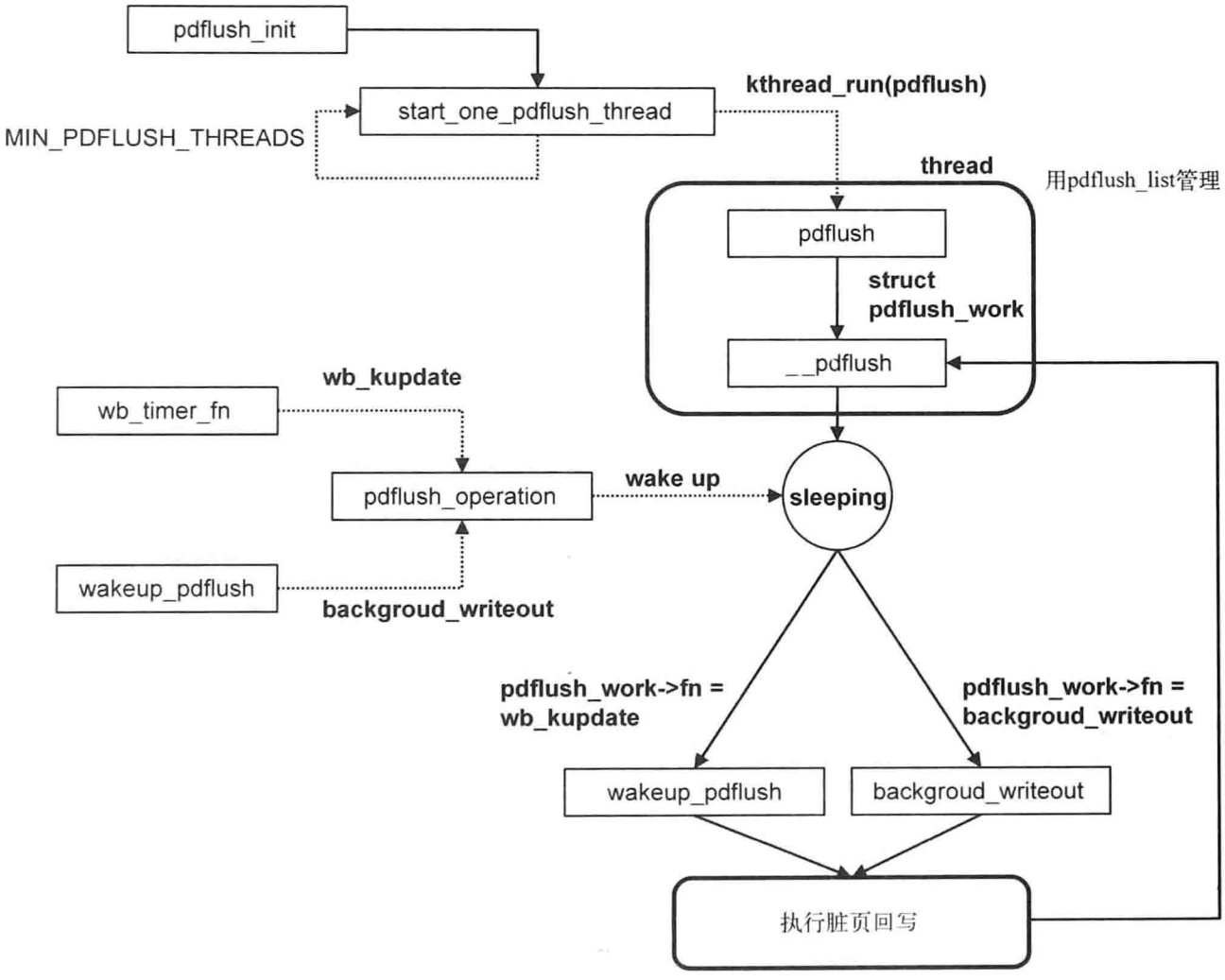


图26-1 页回写机制

内核中执行页回写的方法有两种：周期性页回写的函数是wb_timer_fn()函数，强制性页回写的函数是wakeup_pdflush()函数。这两个函数在调用pdflush_operation()函数时，分别传递wb_kupdata()、background_writeout()函数指针参数以执行脏页回写。因此，在内部周期性页回写的函数是wb_kupdate()函数，强制性页回写的函数是background_writeout()函数。

26.2 激活页回写——pdflush_init()

我们在上层级别中了解了页回写机制，本节通过源代码分析如何对其进行初始化。如图26-1所示，最初激活页回写机制的是pdflush_init()函数，通过代码26-1分析该函数。

:: 代码26-1　mm/pdflush.c的pdflush_init()

```
static int __init pdflush_init(void)
{
    int i;

    nr_pdflush_threads = MIN_PDFLUSH_THREADS;

    for (i = 0; i < MIN_PDFLUSH_THREADS; i++)
        start_one_pdflush_thread();                                    ❶
    return 0;
}


static void start_one_pdflush_thread(void)
{
    struct task_struct *k;

    k = kthread_run(pdflush, NULL, "pdflush");                         ❷
    ...
}
```

在代码段❶中调用MIN_PDFLUSH_THREADS大小的start_one_pdflush_thread()函数，生成pdflush线程，MIN_PDFLUSH_THREADS的定义如下所示。

:: mm/pdflush.c

```
#define MIN_PDFLUSH_THREADS 2
```

生成两个pdflush线程。

在代码段❷中，start_one_pdflush_thread()函数从内部调用kthread_run()函数，生成pdflush线程。kthread_run()函数作为kthread_create()函数的包装（wrapper）定义如下。

:: include/linux/kthread.h

```
#define kthread_run(threadfn, data, namefmt, ...)                      \
({                                                                     \
    struct task_struct *__k                                            \
        = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__); \
    if (!IS_ERR(__k))                                                  \
        wake_up_process(__k);                                          \
    __k;                                                               \
})
```

通过调用kthread_run()函数生成pdflush线程，使能够调用pdflush()函数（26.3节）。

26.3 pdflush 线程

pdflush线程执行脏页回写，该线程调用__pdflush()函数。__pdflush()函数的定义如代码26-2所示。

:: 代码26-2 mm/pdflush.c的pdflush()

```
static int pdflush(void *dummy)
{
    struct pdflush_work my_work;                                    ❶

    ...

    return __pdflush(&my_work);                                     ❷
}
```

代码段❶中声明了pdflush_work结构体，通过该结构体可注册pdflush线程要执行的回调函数。pdflush_work结构体的声明如下所示。

:: mm/pdflush.c

```
struct pdflush_work {
    struct task_struct *who;                    /*pdflush线程*/
    void (*fn)(unsigned long);                  /*pdflush线程要执行的回调函数*/
    unsigned long arg0;                         /*向回调函数传递的参数*/
    struct list_head list;                      /*注册到全局变量pdflush_list*/
    unsigned long when_i_went_to_sleep;         /*进入睡眠时的jiffies值*/
};
```

pdflush_work结构体中要讲的成员变量是list，它会注册到全局变量pdflush_list，pdflush_list用于检查有无要执行页回写的可用的pdflush线程。若有，则在pdflush_list中选择pdflush_work结构体，然后为pdflush_work->fn指定要执行页回写的函数，并唤醒pdflush_work->who任务，使该任务能够执行页回写函数。

代码段❷调用内部函数__pdflush()函数，该函数调用pdflush_work结构体中指定的回调函数执行页回写。通过代码26-3分析__pdflush()函数。

:: 代码26-3 mm/pdflush.c的__pdflush()

```
static int __pdflush(struct pdflush_work *my_work)
{
    ...

    my_work->fn = NULL;                                             ❶
    my_work->who = current;
    INIT_LIST_HEAD(&my_work->list);

    ...
```

```
    for ( ; ; ) {

        struct pdflush_work *pdf;                                          ❷
        set_current_state(TASK_INTERRUPTIBLE);
        list_move(&my_work->list, &pdflush_list);
        my_work->when_i_went_to_sleep = jiffies;
        ...

        schedule();                                                        ❸

        ...

        (*my_work->fn)(my_work->arg0);                                     ❹

        ...
}
```

在代码段❶中，初始化传递来的参数pdflush_work结构体变量my_work。

在代码段❷中，将初始化的my_work添加到用pdflush_work结构体链表进行管理的全局变量pdflush_list。

在代码段❸中，调用schedule()向其他任务让出当前处理器，进入睡眠状态。

代码段❹中，在当前任务由wake up信号唤醒的状态下执行my_work的回调函数。该回调函数就是实际执行页回写的函数。

内核中，pdflush_operation()函数使回调函数在pdflush_work结构体中注册回调函数，并唤醒执行该回调函数的任务以执行函数（26.4节）。

26.4　指定页回写函数

pdflush_operation()函数调用要执行页回写的函数。pdflush_operation()函数的作用是唤醒pdflush线程，使pdflush线程能够执行pdflush_work结构体中注册的页回写函数。通过代码26-4进行分析。

:: 代码26-4　mm/pdflush.c的pdflush_operation()

```
int pdflush_operation(void (*fn)(unsigned long), unsigned long arg0)
{
    ...

    if (list_empty(&pdflush_list)) {                                       ❶
        ret = -1;
    } else {
        struct pdflush_work *pdf;

        pdf = list_entry(pdflush_list.next, struct pdflush_work, list);    ❷
        list_del_init(&pdf->list);
        if (list_empty(&pdflush_list))
```

```
            last_empty_jifs = jiffies;
        pdf->fn = fn;
        pdf->arg0 = arg0;

        wake_up_process(pdf->who);                                    ❸
    }

    ...
}
```

代码段❶确认全局变量pdflush_list中有无列表。pdflush_list为空则表明不存在当前可用的pdflush线程，无法执行页回写。

代码段❷在pdflush_list中用pdf获得pdflush_work结构体元素，将作为参数接收的函数注册为pdf->fn中的回调函数。

代码段❸唤醒pdf->who指向的pdflush线程，在__pdflush()函数中执行schedule()后的(*my_work->fn)(my_work->ago)行。最终，pdf->fn由__pdflush()函数执行。

还未确认调用pdflush_operation()函数时传递的是页回写函数指针参数，那么，需要确认哪个函数中调用pdflush_operation()函数（26.5节）。

26.5 周期性页回写和强制性页回写回调函数调用方法

内核中执行页回写的方法有两种：将传递给pdflush_operation()函数参数的函数指针注册为pdflush_work结构体的回调函数，pdflush线程通过回调函数执行页回写。

本节讲解调用pdflush_operation()函数的两种方法：注册周期性页回写回调函数和注册强制性页回写回调函数。

26.5.1 周期性页回写函数——wb_kupdate()

周期性页回写由wb_timer_fn()函数调用，wb_timer_fn()函数是wb_timer计时器经过特定间隔（interval）后执行的函数，其定义如下所示。

:: mm/page-writeback.c

```
static void wb_timer_fn(unsigned long unused)
{
    if (pdflush_operation(wb_kupdate, 0) < 0)
        mod_timer(&wb_timer, jiffies + HZ);
}
```

调用pdflush_operation()函数，使pdflush线程调用wb_kupdate()函数执行页回写。若pdflush_operation()函数的结果为-1，则意味着不存在要执行页回写的pdflush线程。因此，为周期性调用wb_timer_fn()函数的wb_timer计时器给予1秒延迟，使其1秒后调用wb_timer_fn()函数。请

确认HZ相当于1秒。

周期性执行页回写的wb_kupdate()函数的定义如代码26-5所示。

:: 代码26-5　mm/page-writeback.c的wb_kupdate()

```
static void wb_kupdate(unsigned long arg)
{
    ...

    struct writeback_control wbc = {                                    ❶
        .bdi              = NULL,
        .sync_mode        = WB_SYNC_NONE,
        .older_than_this  = &oldest_jif,
        .nr_to_write      = 0,
       .nonblocking       = 1,
        .for_kupdate      = 1,
        .range_cyclic     = 1,
    };

    sync_supers();                                                      ❷

    ...

    nr_to_write = global_page_state(NR_FILE_DIRTY)                      ❸
        global_page_state(NR_UNSTABLE_NFS) +
        (inodes_stat.nr_inodes - inodes_stat.nr_unused);

    while (nr_to_write > 0) {

    ...

    writeback_inodes(&wbc);                                             ❹

    ...

    nr_to_write -= MAX_WRITEBACK_PAGES - wbc.nr_to_write;
    }

    ...
}
```

代码段❶声明struct writeback_control结构体变量wbc并设置值。writeback_control结构体的信息表示回写代码该如何写入后备存储块设备。

代码段❷对超级锁执行同步。

在代码段❸中求出要回写的页数。通过global_page_state()函数，vm_stat将NR_FILE_DIRTY、NR_UNSTABLE_NFS的页数和正在使用的所有节点页数作为对象。

代码段❹利用writeback_control结构体信息实际执行回写。

补充知识点 保存页状态的全局计数器数组——vm_stat

vm_stat是内存相关全局计数器数组，其声明如下所示。

:: mm/vmstat.c

```
atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS];
EXPORT_SYMBOL(vm_stat);
```

具有NR_VM_ZONE_STAT_ITEMS大小的元素，各元素与enum类型的zone_stat_item执行映射，zone_stat_item的定义如下所示。

:: include/linux/mmzone.h

```
enum zone_stat_item {
    NR_FREE_PAGES,
    NR_LRU_BASE,
    NR_INACTIVE_ANON = NR_LRU_BASE,
    NR_ACTIVE_ANON,
    NR_INACTIVE_FILE,
    NR_ACTIVE_FILE,

    ...

    NR_FILE_DIRTY,
    NR_WRITEBACK,

    ...

    NR_UNSTABLE_NFS,

    ...

    NR_VM_ZONE_STAT_ITEMS };
```

因此，vm_stat[NR_FILE_DIRTY]中，加载到内存的文件内容会保存已修正的内存上的页数，vm_stat[NR_UNSTABLE_NFS]中保存不稳定的NFS页数。

26.5.2 强制性页回写函数——background_writeout()

下面了解需要通过sync()等系统调用强制执行页回写的情况。强制性页回写由wakeup_pdflush()函数调用，其定义如代码26-6所示。

:: 代码26-6　mm/page-writeback.c的wakeup_pdflush()

```
int wakeup_pdflush(long nr_pages)
{
    if (nr_pages == 0)                                                    ❶
        nr_pages = global_page_state(NR_FILE_DIRTY) +
            global_page_state(NR_UNSTABLE_NFS);

    return pdflush_operation(background_writeout, nr_pages);              ❷
}
```

在代码段❶中求出要回写的页数，并求出vm_stat为NR_FILE_DIRTY的页数和NR_UNSTABLE_NFS的页数之和。

在代码段❷中调用pdflush_operation()函数，使pdflush线程执行回调函数background_writeout()函数。这样，background_writeout()函数将成为强制执行页回写的函数。

通过代码26-7分析background_writeout()函数。

:: 代码26-7　mm/page-writeback.c的background_writeout()

```
static void background_writeout(unsigned long _min_pages)
{
    long min_pages = _min_pages;                                          ❶

    struct writeback_control wbc = {                                      ❷
        .bdi        = NULL,
        .sync_mode  = WB_SYNC_NONE,
        .older_than_this = NULL,
        .nr_to_write    = 0,
        .nonblocking    = 1,
        .range_cyclic   = 1,
    };

    for ( ; ; ) {
        unsigned long background_thresh;
        unsigned long dirty_thresh;
        get_dirty_limits(&background_thresh, &dirty_thresh, NULL, NULL);  ❸
        if (global_page_state(NR_FILE_DIRTY) +
            global_page_state(NR_UNSTABLE_NFS) < background_thresh
                && min_pages <= 0)
            break;

        ...

        writeback_inodes(&wbc);                                           ❹

        ...
    }
}
```

在代码段❶中设置要执行页回写的最小页数，至少要执行_min_pages大小的页回写。

在代码段❷中声明writeback_control结构体变量，设置执行回写时使用的信息。

在代码段❸中获得内存内部脏页的临界值，该临界值会保存到background_thresh，执行页回写直到小于background_thresh值。

在代码段❹中，以writeback_control结构体值为基准执行回写。

26.6 初始化周期性页回写

执行周期性页回写的函数是wb_kupdate()函数，由wb_timer计时器周期性调用的wb_timer_fn()函数调用pdflush_operation()函数，并由pdflush线程执行。

那么，在何处生成wb_timer计时器，又如何设置wb_timer_fn()函数呢？page_writeback_init()函数负责此项操作，其定义如下所示。

:: mm/page-writeback.c

```
void __init page_writeback_init(void)
{
    ...
    mod_timer(&wb_timer,
        jiffies + msecs_to_jiffies(dirty_writeback_interval * 10));

    ...
}
```

page_writeback_init()函数调用mod_timer()函数，在wb_timer中设置期满时间。dirty_writeback_interval的声明如下所示。

:: mm/page-writeback.c

```
unsigned int dirty_writeback_interval = 5*100;
```

根据dirty_writeback_interval×10，每次以5秒为间隔调用wb_timer。wb_timer定义如下。

:: mm/page-writeback.c

```
static void wb_timer_fn(unsigned long unused);
static DEFINE_TIMER(wb_timer, wb_timer_fn, 0, 0);
```

计时器期满时调用wb_timer_fn()函数。DEFINE_TIMER的定义如下所示，计时器期满时注册时间和调用的函数。

:: include/linux/timer.h

```
#define TIMER_INITIALIZER(_function, _expires, _data) {     \
        .entry = { .prev = TIMER_ENTRY_STATIC },    \
```

```
        .function = (_function),            \
        .expires = (_expires),              \
        .data = (_data),                \
        .base = &boot_tvec_bases,           \
        __TIMER_LOCKDEP_MAP_INITIALIZER(        \
        __FILE__ ":" __stringify(__LINE__)) \
    }

#define DEFINE_TIMER(_name, _function, _expires, _data)     \
    struct timer_list _name =               \
        TIMER_INITIALIZER(_function, _expires, _data)
```

page_writeback_init()函数的作用可大致整理如下：在wb_timer计时器中注册wb_timer_fn()函数并对环境进行初始化，使其每隔5秒调用wb_timer_fn()函数，以执行wb_kupdate()函数进行周期性页回写。

第27章 查看启动内核的最终函数结构——rest_init()

本章先分析rest_init()函数执行的大致工作。rest_init()函数生成init进程和内核线程，其他章节将讲解分析量相对较多的函数。图27-1是rest_init()函数中调用的函数。

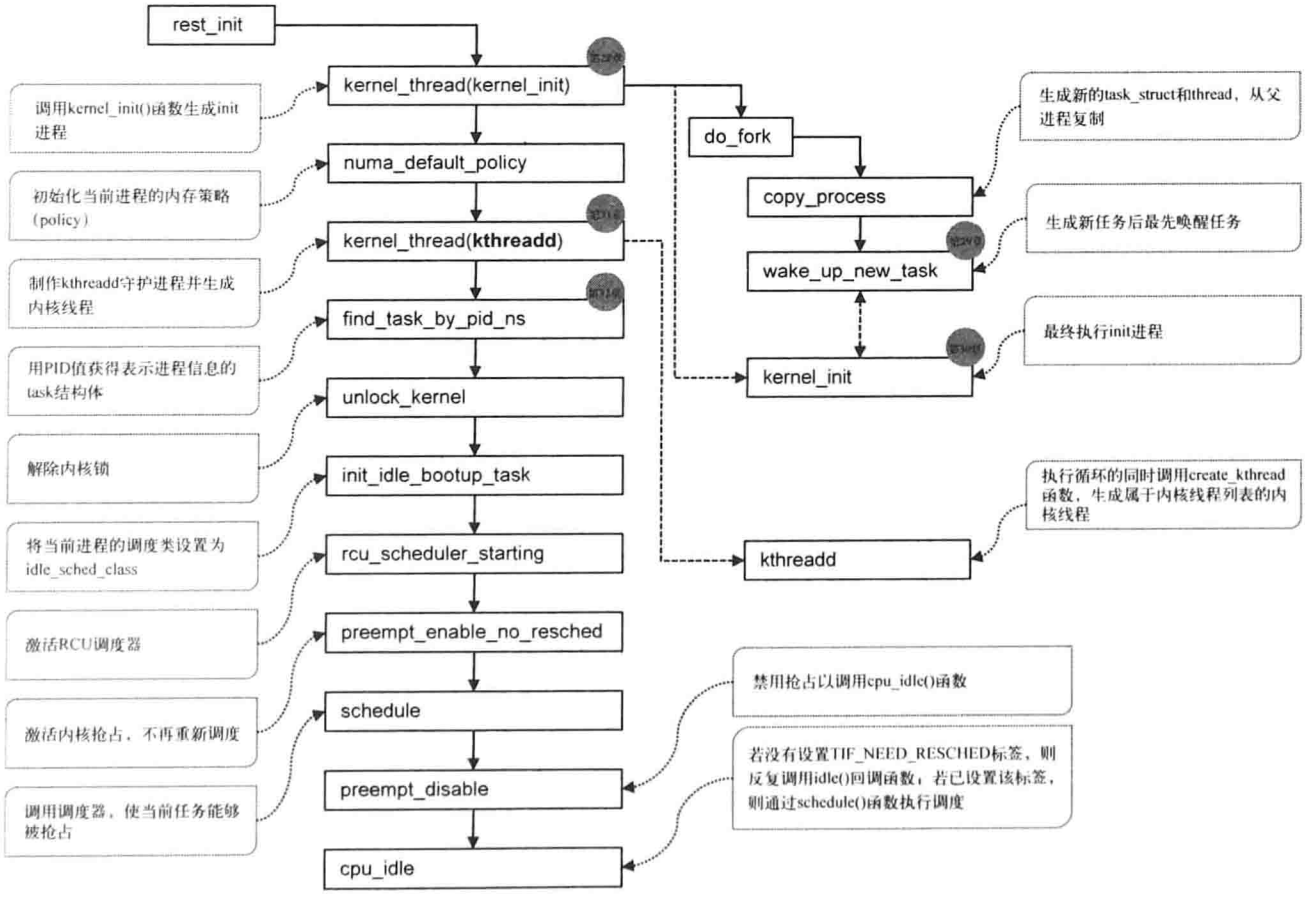


图27-1　rest_init()函数的调用关系

内核启动阶段的最后函数rest_init()函数在内部再次调用多个函数以执行该阶段，通过代码27-1简单了解其结构。

:: 代码27-1　init/main.c的rest_init()

```
static noinline void __init_refok rest_init(void)
    __releases(kernel_lock)
{
    int pid;

    kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);          ❶

    numa_default_policy();                                               ❷

    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);         ❸

    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);              ❹
    unlock_kernel();
    init_idle_bootup_task(current);
    rcu_scheduler_starting();
    preempt_enable_no_resched();
    schedule();
    preempt_disable();
    cpu_idle();
}
```

在代码段❶中调用kernel_thread()函数，生成要执行kernel_init()函数（第30章）的任务。[①] kernel_thread()函数（第28章）从内部调用do_fork()函数以生成进程。

在代码段❷中初始化当前进程的内存策略，实际代码如下所示。

:: mm/mempolicy.c

```
void numa_default_policy(void)
{
    do_set_mempolicy(MPOL_DEFAULT, 0, NULL);
}
```

该函数只调用do_set_mempolicy()以重新设置当前进程的内存策略。

代码段❸和代码段❶相同，但生成的任务生成内核线程守护进程，这些守护进程生成内核线程。[②]（第31章）

代码段❹使当前任务能够执行抢占，而自身成为idle进程（第32章）。

① Linux各发布版略有不同，但一般生成的进程为/sbin/init，PID=1（可通过Linux指令ps – ef确认）。

② 内核线程守护进程的PID=2（可通过Linux命令 ps – ef确认）。

第 28 章

生成执行函数的内核线程——kernel_thread()

28.1 查看 kernel_thread()结构

Linux中用额外进程代替偶尔执行的重要服务，内核线程在后台执行仅在内核模式下运行的服务。内核线程在kernel_thread()中生成，该函数制作新的内核线程执行作为参数传递的函数。

:: 代码28-1 arch/arm/kernel/process.c的kernel_thread()

```
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
    struct pt_regs regs;

    memset(&regs, 0, sizeof(regs));

    regs.ARM_r1 = (unsigned long)arg;                                          ❶
    regs.ARM_r2 = (unsigned long)fn;
    regs.ARM_r3 = (unsigned long)do_exit;
    regs.ARM_pc = (unsigned long)kernel_thread_helper;
    regs.ARM_cpsr = SVC_MODE;

    return do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, &regs, 0, NULL, NULL);    ❷
}
EXPORT_SYMBOL(kernel_thread);
```

在代码段❶中，由以后执行的copy_thread()设置要复制到子进程的寄存器值。首次执行子进程时调用的ret_from_fork()执行上面ARM_pc中设置的kernel_thread_helper()，在CPU的寄存器组置中如下设置该值。

```
r0    <- arg
lr    <- do_exit        (fn结束后调用的函数)
pc    <- fn
```

设置子进程的寄存器组后，在代码段❷中调用do_fork()正式生成进程。

28.2　生成处理器的网关——do_fork()

讲解操作系统时，一般会提到进程和线程。那么，Linux中如何生成进程和线程呢？通常会调用fork()或vfork()，或者调用pthread_create()函数。下图表示用户级应用程序和系统调用间的关系。

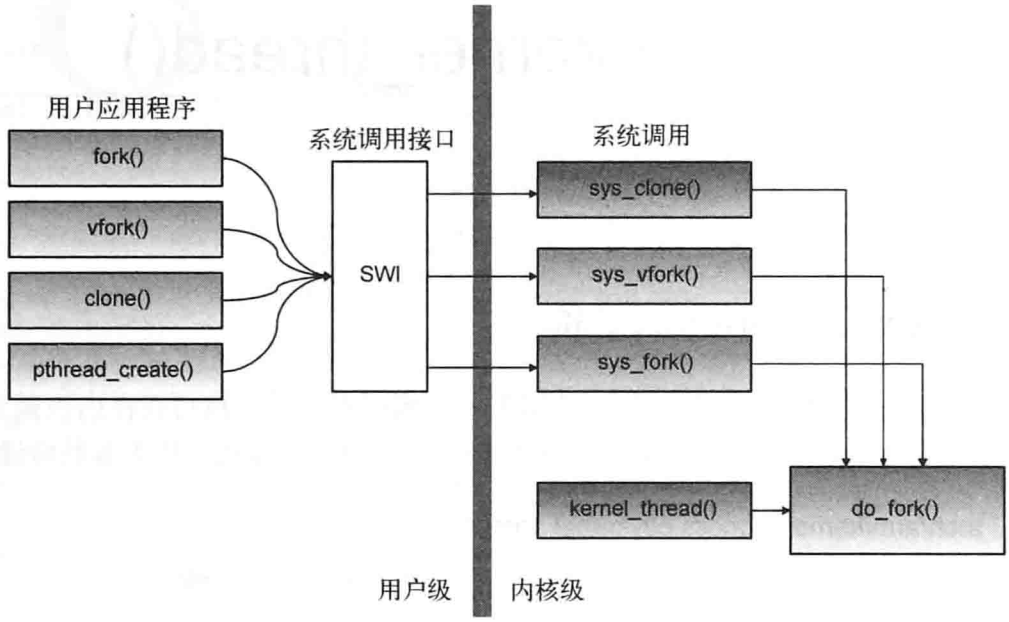


图28-1　生成用户级任务

用户级中可调用fork()、vfork()、clone()、pthread_create()等多种函数，最终会发生3种系统调用，在该系统调用内均调用do_fork()函数。甚至在内核内部生成内核线程的kernel_thread()函数也会调用do_fork()。也就是说，与生成任务相关的所有工作均是do_fork()掌管。

无论要生成不共享内存空间的进程，还是要生成共享内存空间的线程，均通过do_fork()的第一个参数clone_flags实现，该标签如表28-1所示。

表28-1　CLONE_FLAGS

标　签	说　明
CLONE_FILES	共享父进程文件描述符列表（files_struct）
CLONE_FS	共享文件系统信息（fs_struct）
CLONE_IDLETASK	将PID设置为0（仅使用idle任务）
CLONE_NEWNS	制作新的命名空间
CLONE_PARENT	复制父任务
CLONE_PTRACE	指定和父进程相同的ptrace状态
CLONE_SETTID	向用户空间通知TID值
CLONE_SIGHAND	共享信号处理器（signal_struct、sighand_struct）

（续）

标　签	说　明
CLONE_SYSVSEM	对System V SEM_UNDO继承父进程的运行
CLONE_THREAD	与父进程属于相同线程组
CLONE_VFORK	子任务释放内存空间（mm_struct）前待机
CLONE_UNTRACED	强制解除父进程的ptrace状态
CLONE_STOPPED	让任务状态从TASK_STOPPED开始
CLONE_SETTLS	为子进程生成新的TLS
CLONE_CHILD_CLEARTID	删除子进程的TID值
CLONE_CHILD_SETTID	设置子进程的TID值
CLONE_PARENT_SETTID	设置父进程的TID值
CLONE_VM	共享父进程的地址空间
CLONE_NEWUSER	生成新的用户命名空间

接下来观察do_fork()是如何运行的。

:: 代码28-2　kernel/fork.c的do_fork()

```
long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
             struct pt_regs *regs,
             unsigned long stack_size,
             int __user *parent_tidptr,
             int __user *child_tidptr)
{
    struct task_struct *p;
    int trace = 0;
    long nr;
    if (clone_flags & CLONE_NEWUSER) {                                    ❶
        if (clone_flags & CLONE_THREAD)
            return -EINVAL;
        if (!capable(CAP_SYS_ADMIN) || !capable(CAP_SETUID) ||
                !capable(CAP_SETGID))
            return -EPERM;
    }

    if (unlikely(clone_flags & CLONE_STOPPED)) {                          ❷
        static int __read_mostly count = 100;
        if (count > 0 && printk_ratelimit()) {
            char comm[TASK_COMM_LEN];
            count--;
            printk(KERN_INFO "fork(): process `%s' used deprecated "
                  "clone flags 0x%lx\n",
                  get_task_comm(comm, current),
                  clone_flags & CLONE_STOPPED);
        }
```

```
    }
    if (likely(user_mode(regs)))
        trace = tracehook_prepare_clone(clone_flags);

    p = copy_process(clone_flags, stack_start, regs, stack_size,
                     child_tidptr, NULL, trace);                          ❸

    if (!IS_ERR(p)) {
        struct completion vfork;

        trace_sched_process_fork(current, p);
        nr = task_pid_vnr(p);                                             ❹

        if (clone_flags & CLONE_PARENT_SETTID)
            put_user(nr, parent_tidptr);                                  ❺

        if (clone_flags & CLONE_VFORK) {                                  ❻
            p->vfork_done = &vfork;
            init_completion(&vfork);
        }

        audit_finish_fork(p);
                                                                          ❼

        tracehook_report_clone(regs, clone_flags, nr, p);                 ❽
        p->flags &= ~PF_STARTING;

        if (unlikely(clone_flags & CLONE_STOPPED)) {
            sigaddset(&p->pending.signal, SIGSTOP);
            set_tsk_thread_flag(p, TIF_SIGPENDING);
            __set_task_state(p, TASK_STOPPED);
        } else {
            wake_up_new_task(p, clone_flags);                             ❾
        }
        tracehook_report_clone_complete(trace, regs,
                                        clone_flags, nr, p);              ❿

        if (clone_flags & CLONE_VFORK) {                                  ⓫
            freezer_do_not_count();
            wait_for_completion(&vfork);
            freezer_count();
            tracehook_report_vfork_done(p, nr);
        }
    } else {
        nr = PTR_ERR(p);
    }
    return nr;
}
```

使用新的用户命名空间时设置代码段❶的CLONE_NEWUSER标签，若已设置该标签，则无法同时设置CLONE_THREAD。因为如果用户命名空间不同，则无法和父进程属于相同线程组。

在进程状态为STOPPED时执行启动的情形设置CLONE_STOPPED标签。引入printk_ratelimit()

函数是为了防止过度输出内核消息对系统运行产生的影响。该函数使5秒之内最多输出10个内核消息。

在代码段❸中调用copy_process()，根据接收的多种标签值参数复制父任务的多种信息（28.3节）。

在代码段❹中，如果生成进程没有问题，则从current任务的PID命名空间获取PID。

在代码段❺中，若已设置CLONE_PARENT_SETTID，则在传递来的参数parent_tidptr地址中保存nr值。

在代码段❻中，若已设置CLONE_VFORK，则将进程的vfork_done completion变量设置为vfork并进行初始化。设置好的子进程在mm_release时唤醒父进程。

在代码段❼中初始化子进程p的audit_context。如果子进程p中有ptrace，则向pending.signal添加SIGSTOP，在thread_info->flag中设置SIGPENDING，并清除PF_STARTING标签。

在代码段❽中，若已设置CLONE_STOPPED，则为pending.signal添加SIGSTOP，在thread_info->flag中设置TIF_SIGPENDING，在任务state中设置TASK_STOPPED。

在代码段❾中，若已设置CLONE_STOPPED，则唤醒任务。在代码段❿中检查当前任务的ptrace值，向父处理器传递SIGTRAP。

在代码段⓫中，若已设置CLONE_VFORK标签，则设置PF_FREEZER_SKIP标签，一直等到用vfork生成的子进程结束。标记为“任务可再次被freeze”后，通知“vfork已完成”。

以上就是do_fork()中执行的工作，实际do_fork()函数的核心是copy_process()（28.3节）。

28.3 复制父进程——copy_process()

Linux生成进程时调用的函数中，该函数处理最核心的工作。将父进程的数据结构task_struct结构体和thread_info结构体进行复制，生成新进程。传递的参数标签决定与父进程共享数据还是重新生成。实际通过fork()、vfork()、clone()生成进程时，调用该函数处理很多部分。接下来观察该函数。

:: 代码28-3 kernel/fork.c的copy_process()-1/8

```
static struct task_struct *copy_process(unsigned long clone_flags,
                                        unsigned long stack_start,
                                        struct pt_regs *regs,
                                        unsigned long stack_size,
                                        int __user *child_tidptr,
                                        struct pid *pid,
                                        int trace)
{
    ...
    int cgroup_callbacks_done = 0;

    if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))          ❶
        return ERR_PTR(-EINVAL);
    if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))
```

```
        return ERR_PTR(-EINVAL);
    if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
        return ERR_PTR(-EINVAL);

    retval = security_task_create(clone_flags);                          ❷
    if (retval)
        goto fork_out;

    retval = -ENOMEM;
    p = dup_task_struct(current);                                        ❸
    if (!p)
        goto fork_out;
```

调用copy_process()生成进程时，有几种不被允许的标签组。代码段❶中有如下标签组时，返回错误信息并中止生成进程。

同时设置CLONE_NEWNS**和**CLONE_FS**时**共享文件系统时，现有的命名空间也被共享，因此，生成命名空间的CLONE_NEWNS和共享文件系统的CLONE_FS不可用。

已设置CLONE_THREAD**但未设置**CLONE_SIGHAND**时，已设置**CLONE_SIGHAND**但未设置**CLONE_VM**时**是线程时，组之间的地址空间和信号被共享，因此必须同时设置3个标签。

在代码段❷中，检查权限以确认设置的标签是否被允许（grant）。完成简单的标签检查后，在代码段❸中调用dup_task_struct()复制当前进程的task_struct、thread_info结构体，然后重置值并返回task_struct。

此处复制task_struct结构体时会调用arch_dup_task_struct()，该函数如果有可按结构复制的最优命令，则使用通过该命令实现的函数；否则使用默认实现的函数。向已复制进程的thread_info末尾地址后写入检测溢出的魔数。

复制的task_struct结构体与父进程的task_struct不同的部分只是重新分配的内核模式栈。task_struct结构体的stak指针指向该栈，并保存CPU寄存器值等低级信息。

:: 代码28-4　kernel/fork.c的copy_process()-2/8

```
    ...
    if (atomic_read(&p->real_cred->user->processes) >=                   ❹
        p->signal->rlim[RLIMIT_NPROC].rlim_cur) {
        if (!capable(CAP_SYS_ADMIN) && !capable(CAP_SYS_RESOURCE) &&
            p->real_cred->user != INIT_USER)
            goto bad_fork_free;
    }
    ...
    if (nr_threads >= max_threads)                                       ❺
        goto bad_fork_cleanup_count;

    if (!try_module_get(task_thread_info(p)->exec_domain->module))       ❻
        goto bad_fork_cleanup_count;
    if (p->binfmt && !try_module_get(p->binfmt->module))
        goto bad_fork_cleanup_put_domain;
```

在代码段❹中，检查是否超过当前用户所能拥有的进程最大个数。超过时，若不具有CAP_SYS_ADMIN、CAP_SYS_RESOURCE功能（capability）[①]，则不允许fork。即使不是INIT_USER也执行相同处理，由于目前正在启动，故没有相应事项。

在代码段❺中，检查是否超过系统所能生成的最大线程数max_threads。该值在fork_init()中初始化如下。

:: kernel/fork.c

```
max_threads = mempages / (8 * THREAD_SIZE / PAGE_SIZE);
if(max_threads < 20) max_threads = 20;
```

若无法再生成进程，则返回错误信息并中止生成。

在代码段❻中检查是否已设置exec_domain和binfmt的module，未设置时返回错误信息。exec_domain描述符具有系统调用的调用原理、信号如何编号等信息。binfmt（binary fomat，二进制格式）表示执行文件组成形式。

:: 代码28-5 kernel/fork.c的copy_process()-3/8

```
    p->did_exe = 0;                                          ❼

    copy_flags(clone_flags, p);                              ❽
    ...
    clear_tsk_thread_flag(p, TIF_SIGPENDING);                ❾
    init_sigpending(&p->pending);
    posix_cpu_timers_init(p);                                ❿
    ...
    cgroup_fork(p);                                          ⓫
```

did_exe成员变量保存的信息显示fork时加载的程序是否依然执行，还是通过execve()加载新程序执行。代码段❼未执行execve()，故设置为0。

代码段❽中，在传递来的参数clone_flags中清除PF_SUPERPRIV，添加设置FORKNOEXEC、PF_STARTING标签。各标签的作用如下所示。

- PF_SUPERPRIV：表示正在使用超级用户权限。
- PF_FORKNOEXEC：表示当前进程fork后尚未执行。
- PF_STARTING：表示正在生成当前进程。

子进程不会继承父进程的信号待处理信息和待处理信号。因此，在代码段❾中清除TIP_SIGPENDING，初始化待处理信号相关信息。

在代码段❿中，对保存任务使用的CPU时间、用户模式下的执行时间、内核模式下的执行时

① 即使没有另外进行与security相关的内核设置，仍然运行默认的capability，因此也能进行目前的这种检查。启动过程中赋予了除修正capability以外的所有权限，启动结束后，不是root用户或有必要的情况下限制权限（示例：ntp守护进程限制与网络无关的权限，即使遭到攻击也能将系统受到的影响降到最小）。

间等CPU占用时间信息的成员变量进行初始化。将单调时间转换成timespec格式，设置到start_time、real_start_time。另外，以开始启动时为基准，为real_start_time添加经过时间。

在代码段⓫中，设置为共享父进程的css_set——cgroup，使其能够包含到与父进程相同的cgroup层次图。

:: 代码28-6　kernel/fork.c的copy_process()-4/8

```
#ifdef CONFIG_NUMA
    p->mempolicy = mpol_dup(p->mempolicy);                          ⓬
    if (IS_ERR(p->mempolicy)) {
        retval = PTR_ERR(p->mempolicy);
        p->mempolicy = NULL;
        goto bad_fork_cleanup_cgroup;
    }
    mpol_fix_fork_child_flag(p);
#endif
    ...
#ifdef __ARCH_WANT_INTERRUPTS_ON_CTXSW                              ⓭
    p->hardirqs_enabled = 1;
#else
    p->hardirqs_enabled = 0;
#endif
    ...
    sched_fork(p, clone_flags);                                     ⓮
    if ((retval = audit_alloc(p)))                                  ⓯
        goto bad_fork_cleanup_policy;
    ...
    if ((retval = copy_io(clone_flags, p)))
        goto bad_fork_cleanup_namespaces;
```

在代码28-3的代码段❸中调用dup_task_struct()函数，现有情况下只共享了父进程的mempolicy结构体地址。在代码段⓬中，将具有父进程内存策略信息的结构体变量进行整体复制。

在代码段⓭中，如果已定义__ARCH_WANT_INTERRUPTS_ON_CTXSW，那么，为了在上下文切换时激活中断，将hardirqs_enabled设置为1，这样可避免较高的潜伏期。

代码段⓮调用sched_fork()执行以下内容。

- 将进程状态设置为TASK_RUNNING[①]。
- SMP时找到负荷最小的CPU，设置为要执行进程的CPU。
- 若不具有实时优先顺序（real-time priority），则将进程的调度器类设置为fair_sched_class。
- 禁用进程的内核抢占。

在代码段⓯中，将SEM_UNDO list、文件描述符、文件系统、信号处理器、信号、地址空间、命名空间、I/O上下文等从父进程复制到子进程。

- SEM_UNDO意味着父进程的System V信号量。

① 将无需立即执行的任务设置为TASK_RUNNING是为了防止内核的其他部分执行未完成设置的任务。

- 文件描述符若已设置CLONE_FILES标签，则与父进程共享；若未设置，则生成files_struct。这是进程对打开（open）文件的结构体。
- 文件描述符若已设置CLONE_FS标签，则共享父进程的文件上下文。用fs_struct结构体记述文件上下文，进程具有当前执行的目录或顶级目录等信息。
- 信号处理器若已设置CLONE_SIGHAND、CLONE_THREAD标签，则共享父进程的信号处理器。用sighand_struct结构体记述。
- 信号若已设置CLONE_THREAD标签，则与父进程共享除信号处理器外的信号处理部分。
- 地址空间若已设置CLONE_VM标签，则共享父进程的地址空间，mm_struct结构体记述地址空间；若未设置，则复制地址空间。此时，虽然制作页表但不会复制页的数据，因为有COW机制。

:: 代码28-7 kernel/fork.c的copy_process()-5/8

```
    retval = copy_thread(clone_flags, stack_start, stack_size, p, regs);          ⑯
    if (retval)
        goto bad_fork_cleanup_io;

    if (pid != &init_struct_pid) {                                                ⑰
        retval = -ENOMEM;
        pid = alloc_pid(p->nsproxy->pid_ns);
        if (!pid)
            goto bad_fork_cleanup_io;
        if (clone_flags & CLONE_NEWPID) {
            retval = pid_ns_prepare_proc(p->nsproxy->pid_ns);
        if (retval < 0)
            goto bad_fork_free_pid;
        }
    }
    ...
    p->pid = pid_nr(pid);                                                         ⑱
    p->tgid = p->pid;
    if (clone_flags & CLONE_THREAD)
        p->tgid = current->tgid;
    if (current->nsproxy != p->nsproxy) {
        retval = ns_cgroup_clone(p, pid);
    if (retval)
        goto bad_fork_free_graph;
    }
```

代码段⑯调用的copy_thread()中，复制上下文切换时另行保管的寄存器信息等thread_info结构体相关信息。将传递来的寄存器值参数复制到进程的thread_info，重置r0、sp后保存到进程的cpu_context.sp。r0、sp寄存器的返回值与起始位置会和父进程不同，因此需要重置。

在cpu_context.pc中重置ret_from_fork()的地址，使fork后能够跳转到该函数。在ret_from_fork()中处理fork后恢复内核栈中的寄存器值并执行调度等剩余任务。

在代码段⑰中查看pid和init_struct_pid是否一致，若不一致，则分配pid。clone_flags中若已设

置CLONE_NEWPID，则将原先分配pid的PID命名空间挂载到proc文件系统。只有制作idle进程时，传递的pid和init_struct_pid参数不一样。

在代码段⓲中重置pid和tgid。若使用CLONE_THREAD标签调用fork()，则将tgid重置为父进程的tgid。此时属于用线程执行生成的情况，因此，制造“我”的进程将成为“我”的兄弟进程，兄弟进程的父进程将成为“我”的真实父进程real_parent。最后，如果父进程的nsproxy不同，就复制1个命名空间cgroup，移动新生成的进程。

:: 代码28-8　kernel/fork.c的copy_process()-6/8

```
    if ((clone_flags & (CLONE_VM|CLONE_VFORK)) == CLONE_VM)                    ⓳
        p->sas_ss_sp = p->sas_ss_size = 0;
    p->group_leader = p;                                                       ⓴
    INIT_LIST_HEAD(&p->thread_group);

    p->cpus_allowed = current->cpus_allowed;                                   ㉑
    p->rt.nr_cpus_allowed = current->rt.nr_cpus_allowed;


    if (unlikely(!cpu_isset(task_cpu(p), p->cpus_allowed)
                                        || !cpu_online(task_cpu(p))))          ㉒
    set_task_cpu(p, smp_processor_id());
```

若已设置CLONE_VM标签，则需要初始化signalstack指针和栈的大小。在代码段⓳中执行该操作。

在代码段⓴中将自身设置为线程组组长。若已设置CLONE_THREAD标签，则在代码段㉕中设置线程组组长。

若已设置CLONE_THREAD标签，线程组内会再生成1个线程，此时父进程将成为线程组的组长；若未设置标签，则属于生成新进程的情况，因此自身将成为组长。

代码段㉑中，父进程的cpu_allowed位图可能已被变更，因此再次复制父进程的cpus_allowed、rt.nr_cpus_allowed。然后在代码段㉒中，如果满足下列条件之一，则为任务p分配当前运行的进程。

- 为已复制的进程分配的CPU不是自身cpus_allowed中设置的CPU时。
- 设置的CPU非联机状态。

set_task_cpu()中更换分配给任务p的CPU。此处，进程利用目前的cfs_rq和新放置的cfs_rq间的min_vruntime值的差，更新进程的se.vruntime。

最后，将记录执行进程的CPU号的thread_info结构体的cpu变量重置为新分配的CPU号。本段使用的函数概括如下。

task_cpu(p)：返回为任务p分配的CPU号。

cpu_isset(cupid,cpumap)：返回第cpuid个处理器是否设置在cpumap中。

cpu_online(cpuid)：返回相应第CPU个处理器是否处于联机状态。

:: 代码28-9 kernel/fork.c的copy_process()-7/8

```
    if (clone_flags & (CLONE_PARENT|CLONE_THREAD)) {                    ㉓
        p->real_parent = current->real_parent;
        p->parent_exec_id = current->parent_exec_id;
    } else {
        p->real_parent = current;
        p->parent_exec_id = current->self_exec_id;
    }
    ...
    recalc_sigpending();                                                ㉔
    if (signal_pending(current)) {
        spin_unlock(&current->sighand->siglock);
        write_unlock_irq(&tasklist_lock);
        retval = -ERESTARTNOINTR;
        goto bad_fork_free_graph;
    }

    if (clone_flags & CLONE_THREAD) {                                   ㉕
        p->group_leader = current->group_leader;
        list_add_tail_rcu(&p->thread_group, &p->group_leader->thread_group);
    }
```

在代码段㉓中，若已设置CLONE_PARENT、CLONE_THREAD之一，则用已复制进程的父进程指定复制进程（cloner、currrent）的父进程（current->real_parent）；否则，用已复制进程的父进程设置current。

在代码段㉔调用的recalc_sigpending()中，如果正在执行进程复制的current无延迟信号，且不处于freezing状态，则清除TIF_SIGPENDING并继续执行fork过程；如果有延迟信号，则中止fork过程。

在代码段㉕中，若已设置CLONE_THREAD标签，则意味着已复制线程属于和current相同的线程组，因此对group_leader和current进行相同设置。thread_group成员变量具有属于线程组的线程列表。最后，为Group leader的thread_group列表添加当前线程。

:: 代码28-10 kernel/fork.c的copy_process()-8/8

```
    if (likely(p->pid)) {                                               ㉖
        list_add_tail(&p->sibling, &p->real_parent->children);          ㉗
        ...
        if (thread_group_leader(p)) {                                   ㉘
            if (clone_flags & CLONE_NEWPID)                             ㉙
                p->nsproxy->pid_ns->child_reaper = p;
            ...
            attach_pid(p, PIDTYPE_PGID, task_pgrp(current));            ㉚
            attach_pid(p, PIDTYPE_SID, task_session(current));
            list_add_tail_rcu(&p->tasks, &init_task.tasks);             ㉛
            __get_cpu_var(process_counts)++;


        }
```

```
        attach_pid(p, PIDTYPE_PID, pid);                                    ㉜
        nr_threads++;
    }
    total_forks++;                                                          ㉝
    ...
    proc_fork_connector(p);                                                 ㉞
    return p;
}
```

在多线程系统中，task_struct结构体的real_parent成员变量指向生成自身的进程。parent成员变量指向自身所属的线程组的父进程。在代码段㉗中，为父进程的子列表添加p。

如果新生成的进程p是自身所属组长(包括生成新进程的情况)，则需要另外执行几项任务(代码段㉘)。若已设置CLONE_NEWPID标签（代码段㉙），就会生成新的命名空间，因此需要将命名空间的child reaper设置为自身。属于新生成的命名空间的进程已经结束并成为僵尸进程时，没有自身父进程的情况由chiled reaper执行后续处理。

然后，在代码段㉚中将生成的进程p添加到current的任务组和节区。在代码段㉛中，向init_task任务列表添加p，增加进程计数器记录当前运行的CPU已完成1次fork。process_counts是per-cpu变量，执行fork()时增加，执行exit()时减少。

最终，在代码段㉜中将生成的PID连接到PID网络，并增加保存已生成的全部线程数的nr_threads，完成p为任务组组长时的额外处理。

现在执行结束copy_process()前的几项剩余任务。在代码段㉝中，增加系统启动后保存fork次数的total_forks。在代码段㉞中，向通过套接字连接的客户端传递被fork的进程信息事件。

第 29 章

唤醒新生成的任务

本章讲解生成新任务后最先唤醒任务的wake_up_new_task()函数。

29.1 查看 wake_up_new_task 结构

生成新任务后，需要唤醒任务并执行，wake_up_new_task()函数负责这项工作。wake_up_new_task()函数适当设置生成后首次执行的任务的调度器统计信息，并在就绪队列中添加任务以唤醒生成的任务。通过下列代码了解wake_up_new_task()函数的作用。

:: 代码29-1 kernel/sched.c的wake_up_new_task()

```
void wake_up_new_task(struct task_struct *p, unsigned long clone_flags)
{
    unsigned long flags;
    struct rq *rq;

    rq = task_rq_lock(p, &flags);                                    ❶
    BUG_ON(p->state != TASK_RUNNING);
    update_rq_clock(rq);                                             ❷

    p->prio = effective_prio(p);                                     ❸

    if (!p->sched_class->task_new || !current->se.on_rq) {           ❹
        activate_task(rq, p, 0);
    } else {
        p->sched_class->task_new(rq, p);
        inc_nr_running(rq);
    }
    trace_sched_wakeup_new(rq, p, 1);

    check_preempt_curr(rq, p, 0);                                    ❺
#ifdef CONFIG_SMP
    if (p->sched_class->task_wake_up)                                ❻
        p->sched_class->task_wake_up(rq, p);
#endif
    task_rq_unlock(rq, &flags);
}
```

先整体把握wake_up_new_task()函数执行的工作，以后再分析各代码段中的函数。

代码段❶调用task_rq_lock()函数获取任务p的就绪队列。就绪队列先找到任务p要执行的CPU，然后获得该CPU的就绪队列（29.2节）。

在代码段❷中，将任务的就绪队列的时钟（clock）变量（rq->clock）按如下所示更新为当前时间。

:: kernel/sched.c

```
static inline void update_rq_clock(struct rq *rq)
{
    rq->clock = sched_clock_cpu(cpu_of(rq));
}
```

sched_clock_cpu()函数接收对应于就绪队列的CPU和通过cpu_of()函数求出的CPU参数，将相应CPUU的调度器时钟（scheduler clock）以纳米秒单位返回。[①]

在代码段❸中计算任务的优先顺序，设置任务的优先顺序变量p->prio值。实际上，该代码内也设置了另一个优先顺序变量p->normal_prio值（29.3节）。

在代码段❹中检查以下2个条件。

(1) 生成新任务时，任务要使用的调度器中是否存在task_new()函数?

(2) 当前就绪队列中是否存在已生成任务的父任务current? [②]

若以上2个条件中任意一个都不满足，则调用activate_task()函数，将生成的任务插入就绪队列。代码29-2是activate_task()函数的定义。

:: 代码29-2　kernel/sched.c的activate_task()

```
static void activate_task(struct rq *rq, struct task_struct *p, int wakeup)
{
    if (task_contributes_to_load(p))
        rq->nr_uninterruptible--;

    enqueue_task(rq, p, wakeup);
    inc_nr_running(rq);
}
```

调用enqueue_task()函数将任务p插入就绪队列（rp），并调用inc_nr_running()函数增加1个就绪队列正在执行的任务数，在内部只执行rq->nr_running++代码。

在代码段❹中，若2个条件均满足，则存在任务的调度器类的task_new()函数。因此启动（startup）新生成的任务，在连接任务与调度器的task_new()函数中执行，调用inc_nr_running()函数增加1个就绪队列中正在执行的任务数。

---

① sched_clock_cpu()函数返回的struct sched_clock_date结构体相关内容请参考19.6节。

② 注册到就绪队列时，current->se.on_rq==1。

在代码段❺中调用check_preempt_curr()函数，检查新生成的任务p能否抢占当前执行的任务，按如下所示调用调度器类的check_preemp_curr()函数。

:: kernel/sched.c

```
static inline void check_preempt_curr(struct rq *rq, struct task_struct *p, int sync)
{
    rq->curr->sched_class->check_preempt_curr(rq, p, sync);
}
```

代码段❻是SMP环境时，若存在调度器类的task_wake_up()函数，则进行调用。

29.2 获取任务的就绪队列——task_rq_lock()

task_rq_lock()函数挂上给定任务的就绪队列的锁并获得就绪队列，获取就绪队列时将禁用中断。代码29-3是task_rq_lock()函数的定义。

:: 代码29-3 kernel/sched.c的task_rq_lock()

```
static struct rq *task_rq_lock(struct task_struct *p, unsigned long *flags)
    __acquires(rq->lock)
{
    struct rq *rq;

    for (;;) {
        local_irq_save(*flags);
        rq = task_rq(p);                                    ❶
        spin_lock(&rq->lock);
        if (likely(rq == task_rq(p)))
            return rq;
        spin_unlock_irqrestore(&rq->lock, *flags);
    }
}
```

实际上，就绪队列是调用task_rq()函数获得的，task_rq()函数按如下所示定义为cpu_rq()函数和task_cpu()函数的组合。

:: kernel/sched.c

```
#define task_rq(p)    cpu_rq(task_cpu(p))
```

顾名思义，task_cpu()函数通过task_thread_info宏获得要执行的任务的CPU号，如下所示。

:: include/linux/sched.h

```
#define task_thread_info(task)  ((struct thread_info *)(task)->stack)

static inline unsigned int task_cpu(const struct task_struct *p)
```

```
{
    return task_thread_info(p)->cpu;
}
```

cpu_rq()函数获取对应于参数CPU的就绪队列，其定义如下所示。

:: kernel/sched.c

```
#define cpu_rq(cpu)   (&per_cpu(runqueues,(cpu)))
```

per_cpu()宏利用CPU号对per-cpu数据，即各就绪队列获取相应CPU的就绪队列地址。per_cpu()宏及其内部使用的SHIFT_PERCPU、RERLOC_HIDE宏的定义如下。

:: include/asm-generic/percpu.h

```
#define per_cpu(var, cpu) \
    (*SHIFT_PERCPU_PTR(&per_cpu_var(var), per_cpu_offset(cpu)))

#ifndef SHIFT_PERCPU_PTR
#define SHIFT_PERCPU_PTR(__p, __offset) RELOC_HIDE((__p), (__offset))
#endif
```

:: include/linux/compiler-gcc.h

```
#define RELOC_HIDE(ptr, off)                                    \
    ({ unsigned long __ptr;                                     \
        __asm__ ("" : "=r"(__ptr) : "0"(ptr));                  \
        (typeof(ptr)) (__ptr + (off)); })
```

per_cpu()宏的核心是在RERLOC_HIDE宏中利用ptr、off将ptr+off的值转换成ptr类型。以就绪队列类型（rq结构体）为单位，从就绪队列数据的起始位置（runqueues）将指针移动CPU号的偏移量大小，然后将该位置转换成就绪队列类型（rq结构体），这样即可获得相应CPU的就绪队列。①

29.3　改善任务的优先顺序——effective_prio()

我们用effective_prio()函数填充了具有新生成任务的优先顺序信息的prio变量值。通过下列代码了解effective_prio()函数结构。

---

① per-cpu数据runqueues如下静态生成。

kernel/shed.c

```
static DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
```

:: 代码29-4 kernel/sched.c的effective_prio()

```
static int effective_prio(struct task_struct *p)
{
    p->normal_prio = normal_prio(p);                    ❶

    if (!rt_prio(p->prio))                              ❷
        return p->normal_prio;
    return p->prio;
}
```

在代码段❶中调用normal_prio()函数，设置任务的normal_prio变量的值。normal_prio变量可能成为静态优先顺序值，即任务的static_prio值，按调度策略设置。normal_prio()函数的定义如代码29-5所示。

:: 代码29-5 kernel/sched.c的normal_prio()

```
static inline int normal_prio(struct task_struct *p)
{
    int prio;

    if (task_has_rt_policy(p))
        prio = MAX_RT_PRIO-1 - p->rt_priority;
    else
        prio = __normal_prio(p);
    return prio;
}
```

代码29-5中先调用task_has_rt_policy()函数，检查当前任务是否具有实时任务策略。如果具有实时策略，则在normal_prio中设置从MAX_RT_PRIO中减掉任务优先顺序值的值。如果当前任务不使用实时调度，则调用__normal_prio()函数，该函数只返回p->static_prio，故p->normal_prio值和p->static_prio值相同。

task_has_rt_policy()函数检查任务是否具有实时策略，__normal_prio()函数返回任务的static_prio值，二者定义如下所示。

:: kernel/sched.c

```
static inline int task_has_rt_policy(struct task_struct *p)
{
    return rt_policy(p->policy);
}

static inline int __normal_prio(struct task_struct *p)
{
    return p->static_prio;
}
```

在代码段❷中调用rt_prio()函数，决定返回p->normal_prio还是p->prio值。如果rt_prio()的返回值为0，则返回p->normal_prio值；如果返回值为1，则返回p->prio值。由于在代码29-1的代码段❸中按如下所示调用effective_prio()函数，所以由rt_prio()函数的结果值决定，可能成为p->prio = p->normal_prio，也可能是p->prio = p->prio。

```
p->prio = effective_prio(p)
```

rt_prio()函数的定义如下所示。

∷ include/linux/sched.h

```
#define MAX_USER_RT_PRIO    100
#define MAX_RT_PRIO     MAX_USER_RT_PRIO

#define MAX_PRIO        (MAX_RT_PRIO + 40)
#define DEFAULT_PRIO    (MAX_RT_PRIO + 20)

static inline int rt_prio(int prio)
{
    if (unlikely(prio < MAX_RT_PRIO))
        return 1;
    return 0;
}
```

如果输入的任务的优先顺序小于MAX_RT_PRIO，说明这是实时任务，故返回1；若大于该值，则返回0。一般任务的优先顺序值越小，任务越重要；实时任务值越大，任务越重要。任务的优先顺序值的范围具有[0...MAX_PRIO-1]之前的值，任务是实时任务（SCHED_RT、SCHED_FIFO）是，具有[0...MAX_RT_PRIO-1]范围的值；为一般任务（SCHED_NORMAL、SCHED_BATCH）是，则具有[MAX_RT_PRIO...MAX_PRIO-1]范围的优先顺序值。

最后，如果是实时任务（如果任务的优先顺序相当于实时优先顺序范围），则设置为p->prio = p->prio；如果是一般任务（具有超出实时优先顺序范围的值），就设置为p->prio = p->normal_prio。



补充知识点　优先顺序相关任务结构体成员变量

使用4种优先顺序值决定进程（任务）优先顺序，task_struct结构体中优先顺序相关成员变量如下所示。

```
struct task_struct {
    ...

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;

    ...
}
```

进程为一般进程或实时进程时使用的优先顺序不一样。任务结构体的3个成员变量prio、static_prio、normal_prio值决定一般进程优先顺序，概括如下。

变　量	说　明
static_prio	进程启动时分配的优先顺序，在Linux中可通过nice命令或sched_setscheduler()函数修正。但进程运行时无法变更，所以将static_prio称为静态优先顺序值
normal_prio	根据static_prio值和进程的调度策略计算的优先顺序，受进程为一般进程或实时进程的影响较大。这种可变性使normal_prio又称动态优先顺序值。若某个进程被fork，其子进程会直接继承normal_prio值
prio	调度器实际考虑的优先顺序值，也有normal_prio优先顺序。引入新优先顺序值是为了临时启动（boost）进程优先顺序，此时的优先顺序值的变化并不是永久性的，而是临时性的。static_prio、normal_prio不受prio值的影响。由于prio可动态变更，故又称动态优先顺序值

rt_priority是实时任务的优先顺序值，与之前介绍的static_prio、normal_prio、prio无任何关系。rt_priority值具有[0...MAX_RT_PRIO-1]范围的值，值越高，具有的优先顺序越高。反之，一般任务的优先顺序越高，其重要度越低。

第30章 准备使用内核

第28章中提到的内核线程执行的函数是kernel_init()，先仔细观察下列代码。

:: 代码30-1 init/main.c的kernel_init()

```
static int __init kernel_init(void * unused)
{
    lock_kernel();

    set_cpus_allowed_ptr(current, cpu_all_mask);                    ❶

    init_pid_ns.child_reaper = current;                             ❷

    cad_pid = task_pid(current);                                    ❸

    smp_prepare_cpus(setup_max_cpus);                               ❹

    do_pre_smp_initcalls();                                         ❺
    start_boot_trace();

    smp_init();                                                     ❻
    sched_init_smp();                                               ❼

    do_basic_setup();                                               ❽

    if (!ramdisk_execute_command)                                   ❾
        ramdisk_execute_command = "/init";

    if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
        ramdisk_execute_command = NULL;
        prepare_namespace();
    }

    init_post();                                                    ❿
    return 0;
}
```

在代码段❶中找到要执行当前任务的CPU，并注册任务。该初始化任务在任何CPU上均可执行。current变量具有当前任务的信息（31.4节）。

进程结束后，由父进程执行剩余操作。进程没有父进程时，由child_reaper特定进程完成这项任务。在代码段❷中，将全局pid命名空间init_pid_ns的child_reaper设置为current。每次生成新的命名空间时，都要设置child_reaper。

用户同时按下Ctrl+Alt+Del调用ctrl_alt_del()函数，该函数为全局工作队列keventd_wq调度1个任务，使deferred_cad()得以执行。这是最终重启系统的函数。在代码段❸中，为了让执行该功能的进程与执行启动的current属于同一个pid，设置为让cad_pid指向current的pid。

在代码段❹中，准备激活并使用用于启动的CPU以外的CPU。这一过程是通过与结构密切相关的启动过程处理的。但是，由于还没有与调度器联动，因此无法使用。

在代码段❺中，初始化CPU间移动进程时使用的队列。调用该函数后，可使用smp_prepare_cpus()中初始化的所有CPU。

在代码段❻中，调用initcall区域内所有需要early init的初始化函数以初始化相关子系统。在代码段❼中，将当前进程移动到未指定为内核参数isocpu（isolated CPU）的CPU之一，生成并初始化用于负载均衡的数据结构——调度域树。之后，为了防止发生CPU相关热插拔事件，将几个函数注册为回调函数。

代码段❽调用许多函数以初始化RCU，并为使用工作队列、current以及设备驱动进行初始化。此外，生成可查看中断信息的proc文件系统。执行该函数后运行CPU子系统（subsystem），并良好运转内存和进程管理器。

在代码段❾中，如果ramdisk_execute_command为空，则设置为“/init”。之后，检查该变量保存的用户级init进程是否存在。

完成所有必要的初始化过程后，在kernel_init()的最后代码段❿中打开控制设备，制作对应于标准输入、标准输出、标准错误的文件描述符，并执行init进程。下面逐一分析。

30.1 将当前进程转移到其他 CPU——sched_init_smp()

该函数将当前进程移动到未指定为内核参数isocpu的CPU之一，生成并初始化用于负载均衡的数据结构——调度域树。之后，为了防止发生CPU相关热插拔事件，将几个函数注册为回调函数。详细事项参见下列代码。

:: 代码30-2 kernel/sched.c的sched_init_smp()

```
void __init sched_init_smp(void)
{
    cpumask_var_t non_isolated_cpus;
    alloc_cpumask_var(&non_isolated_cpus, GFP_KERNEL);                    ❶

#if defined(CONFIG_NUMA)
    sched_group_nodes_bycpu =
        kzalloc(nr_cpu_ids * sizeof(void **),GFP_KERNEL);                 ❷
    BUG_ON(sched_group_nodes_bycpu == NULL);
#endif
    get_online_cpus();
```

```
    mutex_lock(&sched_domains_mutex);

    arch_init_sched_domains(cpu_online_mask);                                    ❸

    cpumask_andnot(non_isolated_cpus, cpu_possible_mask, cpu_isolated_map);      ❹
    if (cpumask_empty(non_isolated_cpus))
        cpumask_set_cpu(smp_processor_id(), non_isolated_cpus);

    mutex_unlock(&sched_domains_mutex);
    put_online_cpus();

#ifndef CONFIG_CPUSETS
    /* XXX: Theoretical race here - CPU may be hotplugged now */
    hotcpu_notifier(update_sched_domains, 0);                                    ❺
#endif

    /* RT runtime code needs to handle some hotplug events */
    hotcpu_notifier(update_runtime, 0);                                          ❻

    init_hrtick();                                                               ❼

    /* Move init over to a non-isolated CPU */
    if (set_cpus_allowed_ptr(current, non_isolated_cpus) < 0)                    ❽
        BUG();

}
```

代码段❶的non_isolated_cpus用于记录（代码段❹）系统内安装的CPU（cpu_possible_map）中未设置于cpu_isolated_map[①]的CPU。

使用CONFIG_NUMA设置时，在代码段❷中分配系统内所有CPU数大小的指针，使sched_group_nodes_bycpu指向它们。sched_group_nodes_bycpu是NUMA中每个CPU及内存单位——节点具有的变量，它将指向sched_group结构体列表的指针用作数组。

该变量的声明如下所示。

:: kernel/sched.c

```
static struct sched_group ***sched_group_nodes_bycpu;
```



补充知识点　sched_group、sched_domain将用于何处？

调度组（sched_group）和调度域（sched_domain）是2.6版内核起开始支持的概念，它们在SMT、SMP、NUMA等多种结构的系统中有效支持CPU间的负载均衡。

① 此处提到的CPU isolation为了在SMP系统中调度系统进程或执行负载均衡时，使其只能在特定CPU中执行，从而提高缓存性能，或用于对时间敏感的进程分配CPU。

调度域是共享属于自身的各CPU和负载均衡策略、属性的CPU集合。调度域包含多个CPU，由各调度组记述该CPU。负载均衡形成于特定调度域内的调度组之间。具有这些特性的调度域将系统的CPU资源构建为几个阶段的树结构。

调度域包含的CPU集合（sched_domain.span）相当于属于域的调度组显示的CPU（sched_group.cpumask）合集。调度域内的调度组使用sched_domain.groups连接到单向链表。

调度域定期或不定期发生负载均衡。

周期性负载均衡

- 发生SCHED_SOFTIRQ时调用run_rebalance_domain()，若可执行负载均衡，则在域内的调度组间执行负载均衡。[①]

非周期性负载均衡

- 逻辑性CPU处于idle状态时，域中设置SD_BALANCE_NEWIDLE标签，之后立即执行负载均衡。
- 执行fork()、exec()后，域中设置SD_BALANCE_EXEC标签，之后立即负载均衡。此外，唤醒任务时，执行负载均衡可以省电。

在代码段❸中，生成适合系统的调度域和调度组并执行初始化，从而构筑调度域层次图。然后，将调度域的信息注册到sysctl层次图，使其能够通过/proc/sys/kernel/sched_domain进行确认。

在代码段❹中初始化non_isolated_cpus失败时（没有已设置的cpu时），将执行当前线程的CPU设置到non_isolated_cpus位图。代码段❽调用set_cpus_allowed_ptr()以变更执行当前进程的CPU（migrate），用设置好的位图在其中进行选择。

在代码段❺中，未使用CONFIG_CPUSETS设置时，将update_sched_domains()注册为发生CPU热插拔事件时调用的函数。系统添加或清除CPU时调用该函数，并将相应CPU反映到调度器域层次图。使用CPUSET时该任务由cpuset负责，因此不注册该函数。

在代码段❻中，将update_runtime()注册为发生CPU热插拔事件时调用的函数。发生CPU热插拔事件时，该函数重置实时进程使用的就绪队列时间相关域。

在代码段❼中调用init_hrtick()，将hotplug_hrtick()注册为发生CPU热插拔事件时调用的函数。已清除的CPU中还有激活的hrtimer时，该函数取消相应hrtimer。

30.2 结束系统整体初始化——do_basic_setup()

下面分析do_basic_setup()。在do_basic_setup()函数中初始化RCU，为使用工作队列、cpuset以及设备驱动进行初始化。并且，生成可查看中断信息的proc文件系统。执行do_basic_setup()内的所有函数后运行CPU子系统，并良好运行内存和进程管理器。图30-1就是do_basic_setup()函数中调用的函数流程。

① sched_init()中发生SCHED_SOFTIRQ时，将run_rebalance_domains()注册为要调用的函数。

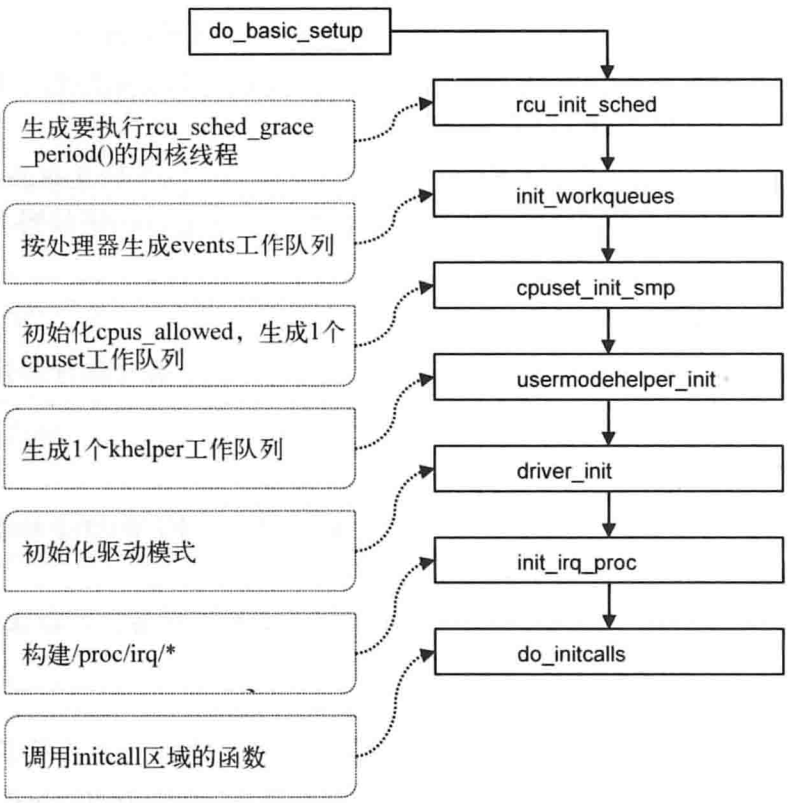


图30-1　do_basic_setup的函数流程图

下面依次讲解各函数。

30.2.1　生成执行rcu_sched_grace_period()的线程——rcu_init_sched()

:: 代码30-3　kernel/rcupreempt.c的rcu_init_sched()

```
void __init rcu_init_sched(void)
{
    rcu_sched_grace_period_task =                                          ❶
        kthread_run(rcu_sched_grace_period,  NULL, "rcu_sched_grace_period");
    WARN_ON(IS_ERR(rcu_sched_grace_period_task));                          ❷
}
```

代码段❶生成并唤醒要执行rcu_sched_grace_period()的线程。kernel_run是kernel_create()的包装（wrapper）函数，用于返回已生成的线程的task_struct地址，在代码段❷中检查该地址是否有效。

30.2.2　生成events工作队列——init_workqueues

工作队列是延迟执行中断未临界任务的底半部之一。工作队列将延迟的任务移交给内核线程，基于内核线程实现，在内核上下文中执行，具有可调度可睡眠的特征。底半部相关内容请参考21.2节“补充知识点”。下面讲解init_workqueues()中如何执行初始化。

:: 代码30-4 kernel/workqueue.c的init_workqueues()

```
void __init init_workqueues(void)
{
    alloc_cpumask_var(&cpu_populated_map, GFP_KERNEL);                    ❶

    cpumask_copy(cpu_populated_map, cpu_online_mask);
    singlethread_cpu = cpumask_first(cpu_possible_mask);
    cpu_singlethread_map = cpumask_of(singlethread_cpu);                  ❷
    hotcpu_notifier(workqueue_cpu_callback, 0);                           ❸
    keventd_wq = create_workqueue("events");                              ❹
    BUG_ON(!keventd_wq);
}
```

在代码段❶中，将cpumask结构体分配到cpu_populated_map，并在位图中标注联机状态的CPU。然后求出对应于第一个CPU的CPU号，并保存到singlethread_cpu。

在代码段❷中，求出该CPU所属的cpumask值，并保存到cpu_singlethread_map。

之前初始化的singlethread_cpu用于使用create_singlethread_workqueue()时运行的CPU设置。通过create_singlethread_work()函数生成的情况下，在对应于singlethread_cpu号的CPU中运行。变量cpu_singlethread_map只用于使用共同接口函数for_each_cpu()，没有特别的意义。

设置CPU位图后，在代码段❸中注册发生CPU热插拔时执行的workqueue_cpu_callback。

代码段❹调用create_workqueue()函数，为使用工作队列而生成最重要的events工作队列。create_workqueue()如图30-2所示。

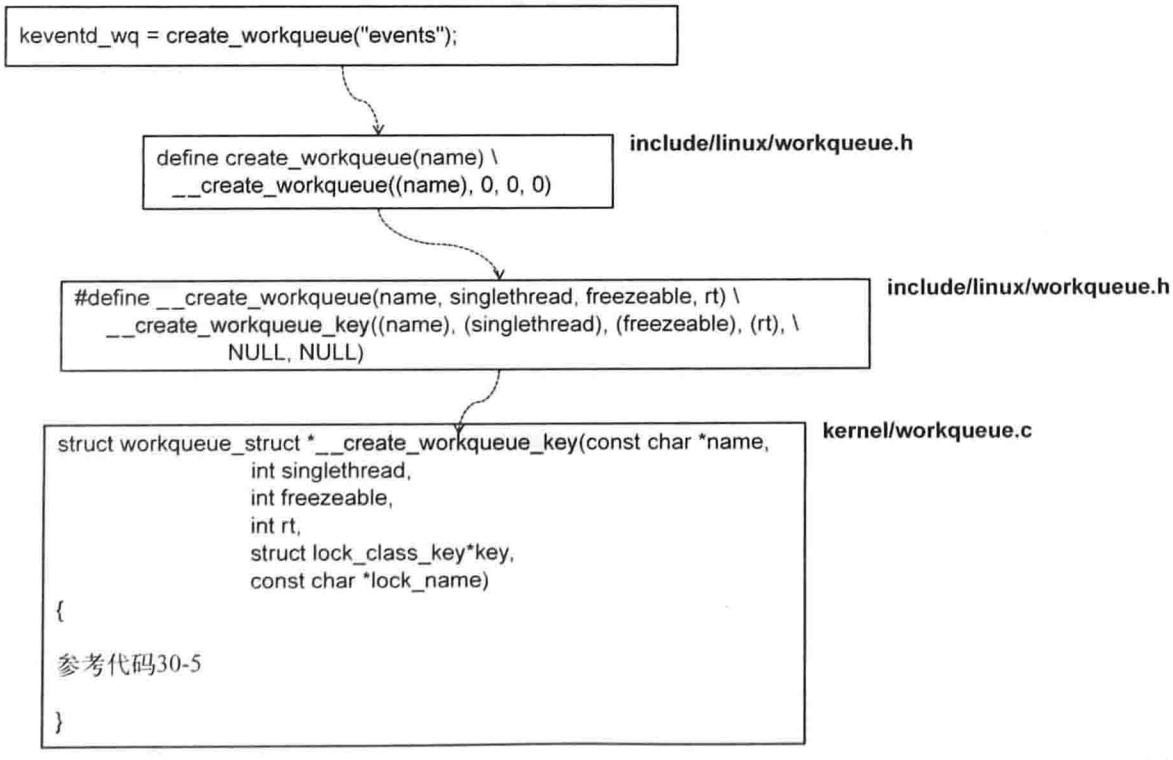


图30-2 create_workqueue的定义

通过多个宏调用函数__create_workqueue_key()，下面分析代码。

:: 代码30-5　kernel/workqueue.c的__create_workqueue_key()

```
struct workqueue_struct *__create_workqueue_key(const char *name,
                                                int singlethread,
                                                int freezeable,
                                                int rt,
                                                struct lock_class_key *key,
                                                const char *lock_name)
{
    struct workqueue_struct *wq;
    struct cpu_workqueue_struct *cwq;
    int err = 0, cpu;

    wq = kzalloc(sizeof(*wq), GFP_KERNEL);                              ❶
    if (!wq)
        return NULL;

    wq->cpu_wq = alloc_percpu(struct cpu_workqueue_struct);             ❷
    if (!wq->cpu_wq) {
        kfree(wq);
        return NULL;
    }
                ...

    if (singlethread) {                                                 ❸
        cwq = init_cpu_workqueue(wq, singlethread_cpu);
        err = create_workqueue_thread(cwq, singlethread_cpu);
        start_workqueue_thread(cwq, -1);
    } else {
                ...
        cpu_maps_update_begin();

        spin_lock(&workqueue_lock);
        list_add(&wq->list, &workqueues);
        spin_unlock(&workqueue_lock);

        for_each_possible_cpu(cpu) {                                    ❹
            cwq = init_cpu_workqueue(wq, cpu);
            if (err || !cpu_online(cpu))
                err = create_workqueue_thread(cwq, cpu);
            start_workqueue_thread(cwq, cpu);
        }
        cpu_maps_update_done();
    }

    if (err) {
        destroy_workqueue(wq);
        wq = NULL;
    }
    return wq;
}
EXPORT_SYMBOL_GPL(__create_workqueue_key);
```

在代码段❶中分配workqueue_struct结构体，在代码段❷中按不同CPU分配cpu_workqueue_struct。正如代码段❸所示，singlethread设置不同会有不同调用。若已设置singlethread值，则依次调用init_cpu_workqueue()、create_workqueue_thread()、start_workqueue_thread()。

若未设置singlethread值，则如代码段❹所示，按不同CPU执行循环，同时调用init_cpu_workqueue()、create_workqueue_thread()、start_workqueue_thread()。

下面依次讲解init_cpu_workqueue()、create_workqueue_thread()、start_workqueue_thread()函数。

:: 代码30-6 kernel/workqueue.c的init_cpu_workqueue()

```
static struct cpu_workqueue_struct *
init_cpu_workqueue(struct workqueue_struct *wq, int cpu)
{
    struct cpu_workqueue_struct *cwq = per_cpu_ptr(wq->cpu_wq, cpu);          ❶

    cwq->wq = wq;
    spin_lock_init(&cwq->lock);
    INIT_LIST_HEAD(&cwq->worklist);
    init_waitqueue_head(&cwq->more_work);                                     ❷

    return cwq;
}
```

在代码段❶中获取相应CPU的cpu_workqueue_struct结构体，并初始化worklist。在代码段❷中初始化名为more_work的waitqueue。执行当前函数后的情形如下图所示。

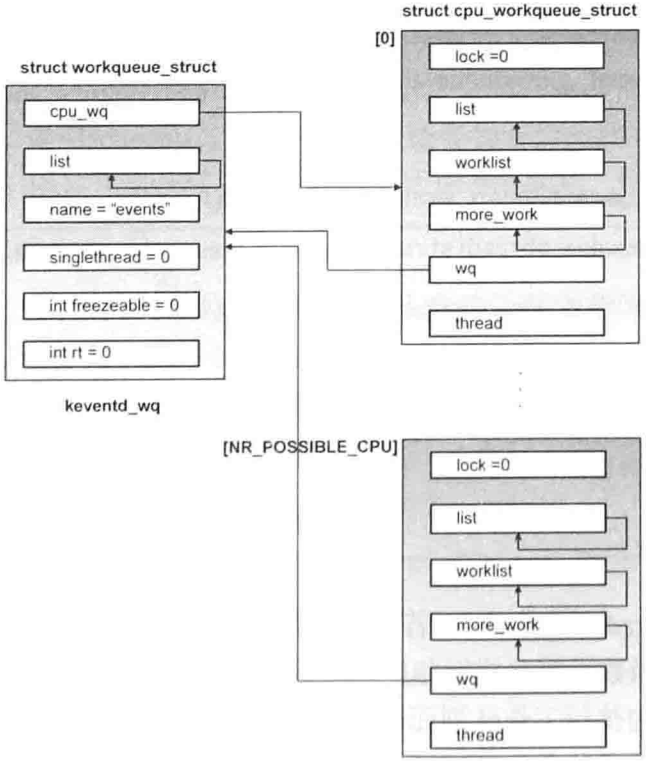


图30-3 执行init_cpu_workqueue()后

完成初始化并返回cpu_workqueue_struct后，调用create_workqueue_thread()，该函数代码如下所示。

:: 代码30-7　kernel/workqueue.c的create_workqueue_thread()

```
static int create_workqueue_thread(struct cpu_workqueue_struct *cwq, int cpu)
{
    struct sched_param param = { .sched_priority = MAX_RT_PRIO-1 };
    struct workqueue_struct *wq = cwq->wq;
    const char *fmt = is_wq_single_threaded(wq) ? "%s" : "%s/%d";
    struct task_struct *p;

    p = kthread_create(worker_thread, cwq, fmt, wq->name, cpu);                    ❶

    if (IS_ERR(p))
        return PTR_ERR(p);
    if (cwq->wq->rt)
        sched_setscheduler_nocheck(p, SCHED_FIFO, &param);                         ❷
    cwq->thread = p;

    trace_workqueue_creation(cwq->thread, cpu);

    return 0;
}
```

代码段❶生成要执行实际操作的worker_thread，若已设置工作队列的rt值，则调用代码段❷的sched_setscheduler_nocheck()函数，变更已生成的工作者线程的调度策略和优先顺序。这样，生成的工作者线程就执行start_workqueue_thread()函数进行运转。下列代码就是start_workqueue_thread()。

:: 代码30-8　kernel/workqueue.c的start_workqueue_thread()

```
static void start_workqueue_thread(struct cpu_workqueue_struct *cwq, int cpu)
{
    struct task_struct *p = cwq->thread;

    if (p != NULL) {
        if (cpu >= 0)
            kthread_bind(p, cpu);                                                  ❶
        wake_up_process(p);                                                        ❷
    }
}
```

start_workqueue_thread()函数中，若接收的参数cpu值大于0，则在代码段❶中调用kthread_bind()函数，该函数设置要执行相应线程的CPU。最后，在代码段❷中唤醒线程，使其处于可执行状态。这就是为执行工作队列而进行初始化的整个过程。

实际执行工作队列的函数是worker_thread()，如代码30-9所示。

:: 代码30-9 kernel/workqueue.c的worker_thread()

```
static int worker_thread(void *__cwq)
{
    struct cpu_workqueue_struct *cwq = __cwq;
    DEFINE_WAIT(wait);

    if (cwq->wq->freezeable)
        set_freezable();

    set_user_nice(current, -5);

    for (;;) {
        prepare_to_wait(&cwq->more_work, &wait, TASK_INTERRUPTIBLE);          ❶
        if (!freezing(current) && !kthread_should_stop() &&
            list_empty(&cwq->worklist))
        schedule();                                                           ❷
        finish_wait(&cwq->more_work, &wait);                                  ❸

        try_to_freeze();

        if (kthread_should_stop())
            break;

        run_workqueue(cwq);                                                   ❹
    }

    return 0;
}
```

在代码段❶中，为more_work列表插入定义为DEFINE_WAIT(wait)的待机队列。检查worklist，若为空，则在代码段❷中调用schedule()函数，使当前任务处于待机状态。通过queue_work_on()函数或schedule_work_on()函数注册新任务后，在内部调用下列函数，这样就可唤醒工作者线程。

```
wake_up(&cwp->more_wok);
```

唤醒工作者线程后，调用代码段❸的函数将其从待机队列中删除，并将当前任务设置为RUNNING状态。实际执行操作的是代码段❹的run_workqueue()函数。调用该函数对worklist中注册的所有work结构体执行循环，并调用需要执行的函数。

events工作队列如图30-4所示。

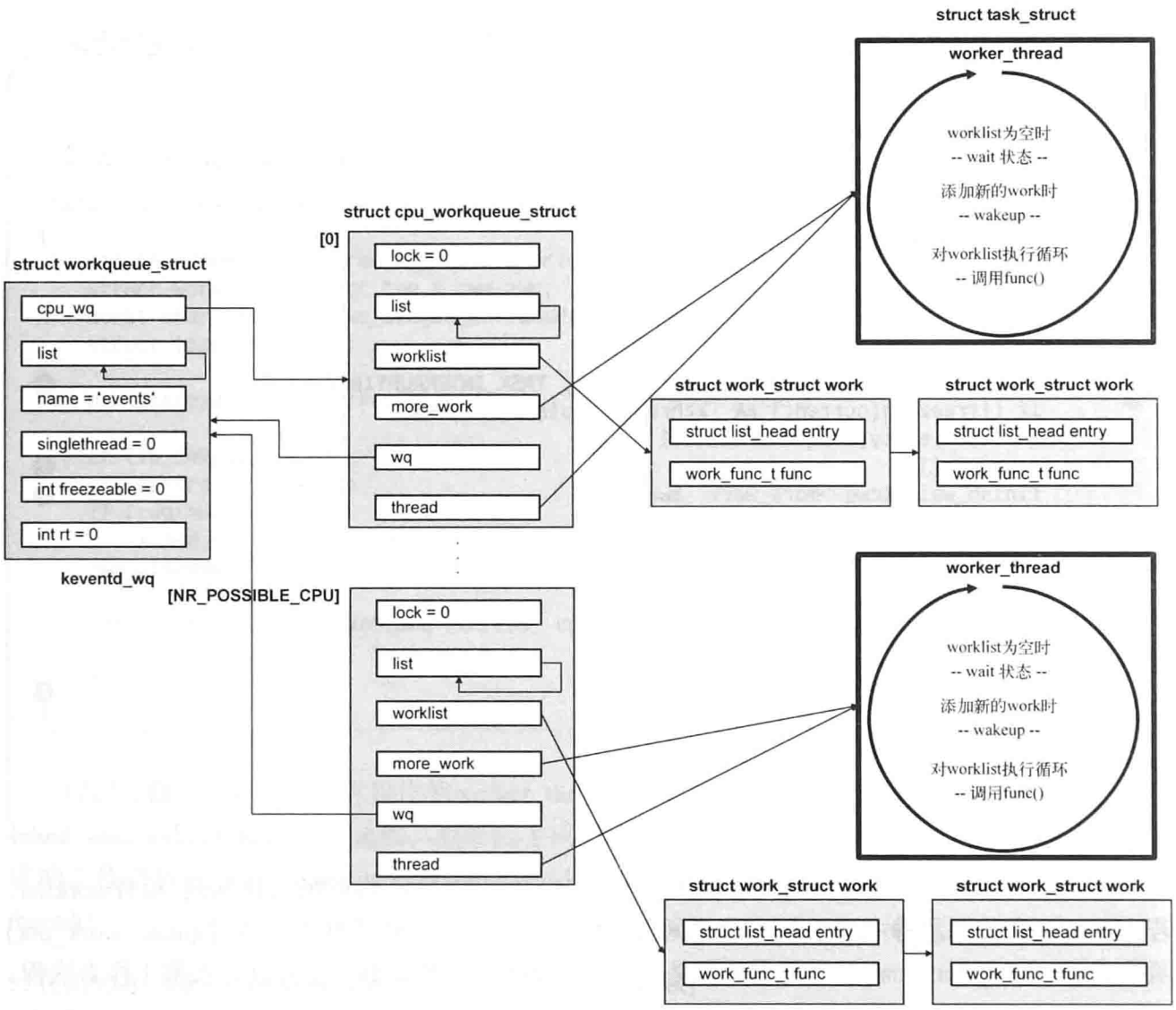


图30-4　events工作队列

这样就完成了使用工作队列前的初始化过程。通过下列函数定义新任务并注册到工作者线程，从而使用工作队列。

■ 定义新任务的宏

```
DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

■ 在工作队列中注册任务的函数

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work,
                       unsigned long delay);
```

30.2.3 初始化cpuset子系统的top_cpuset——cpuset_init_smp()

该函数额外设置cpuset子系统的顶层cpuset，即top_cpuset。然后，为具有发生CPU热插拔事件时调用的回调函数列表的CPU链（chain）添加函数，使CPU、内存状态发生变更时能够更新相关cpuset。最后，为执行cpuset相关处理的进程生成1个工作队列线程，请参考下列代码。

:: 代码30-10 kernel/cpuset.c的cpuset_init_smp()

```
void __init cpuset_init_smp(void)
{
    cpumask_copy(top_cpuset.cpus_allowed, cpu_online_mask);                    ❶
    top_cpuset.mems_allowed = node_states[N_HIGH_MEMORY];                      ❷

    hotcpu_notifier(cpuset_track_online_cpus, 0);                              ❸
    hotplug_memory_notifier(cpuset_track_online_nodes, 10);                    ❹

    cpuset_wq = create_singlethread_workqueue("cpuset");                       ❺
    BUG_ON(!cpuset_wq);
}
```

在代码段❶中，参考cpu_online_mask更新top_cpuset的cpus_allowed。top_cpuset必须包含系统中的所有CPU，因此，始终与最新的cpu_online_mask同步。在代码段❷中，参考已记录包含N_HIGH_MEMORY区域的节点位图，更新top_cpuset的mems_allowed。top_cpuset->mems_allowed始终与node_states[N_HIGH_MEMORY]同步。

在代码段❸中，将cpuset_track_online_cpus()注册为发生CPU热插拔事件时调用的函数。每次发生CPU热插拔事件时调用该函数，并更新cpuset相关信息。该函数执行的工作可整理如下。

(1) cpu_online_map和top_cpuset的成员变量cpus_allowed间执行同步。

(2) 清除只使用脱机CPU的cpuset，将进程移动到父cpuset。

(3) 更新调度器域（sched domain）。

在代码段❹中，将cpuset_track_online_nodes()注册为发生内存热插拔事件时调用的函数。该函数为使top_cpuset的成员变量mems_allowed始终具有有效的N_HIGH_MEMORY节点而执行同步。此外，内存脱机时，如果有只包含相应节点的cpuset，则将属于该cpuset的任务移动到父cpuset。

在代码段❺中调用create_singlethread_workqueue()，为cpu_possible_mask中第一个被设置的CPU生成cpuset工作队列线程。特定cgroup的负载均衡相关设置变更时，使用该工作队列线程执行rebuild_sched_domains_work()调用以进行重置。

30.2.4 生成khelper工作队列——usermodehelper_init()

khelper工作队列能帮助用户态（user mode application）在核心态（kernel mode）中执行。该工作队列在设备驱动中也常用。例如，有用户区域设备驱动模块时插入模块，内核区域就会调用相应函数——requst_module()函数。按如下顺序注册到实际的工作队列，工作队列中插入的函数由工作者线程执行。

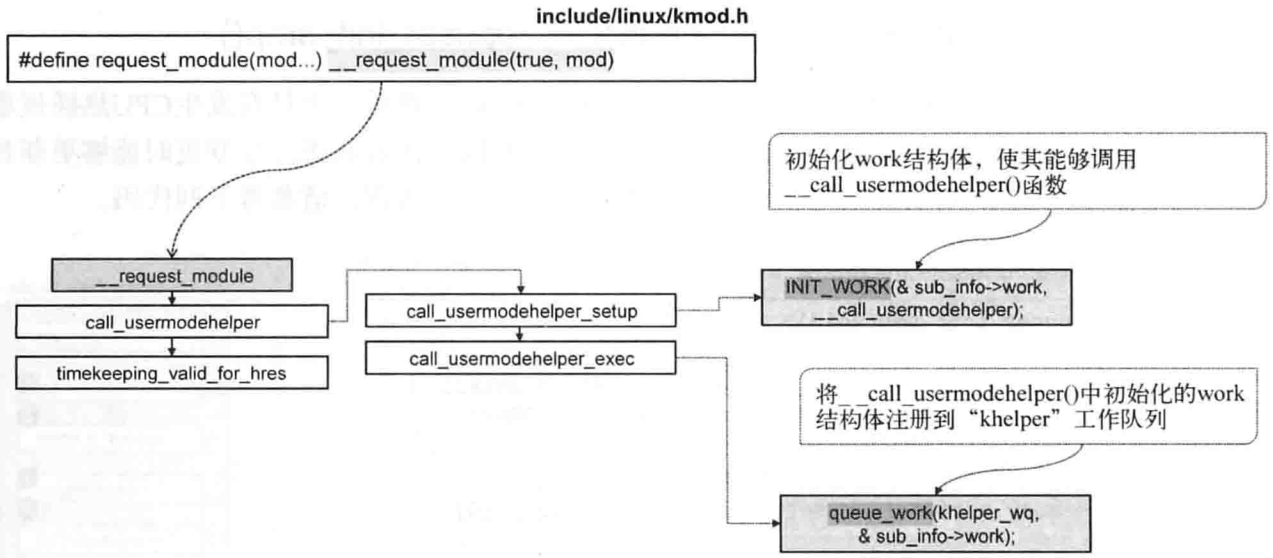


图30-5　request_module的执行过程

观察实际代码可以发现，只生成“khelper”工作队列，继续观察下列代码。

:: 代码30-11　kernel/kmod.c的usermodehelper_init()

```
void __init usermodehelper_init(void)
{
    khelper_wq = create_singlethread_workqueue("khelper");          ❶
    BUG_ON(!khelper_wq);
}
```

代码段❶只调用create_singlethread_workqueue()函数，该函数的定义如下图所示。

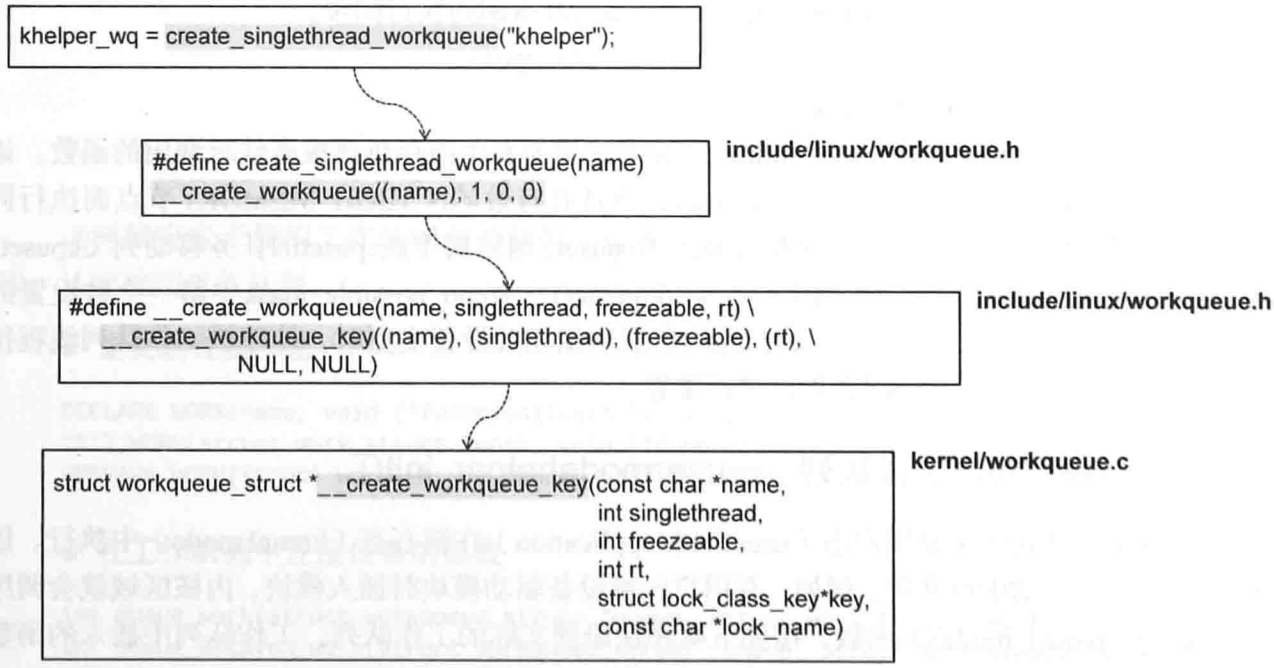


图30-6　create_singlethread_workqueue的定义

和create_workqueue()函数一样调用__create_workqueue_key()函数，不同的是，先将对应于singlethread的参数设置为1。生成的工作队列在cpu_possible_mask中标注的第一个CPU内运行（30.2.2节）。

30.2.5 初始化Linux的设备模型——driver_init()

2.6版内核的设备驱动模式是单一的，它将原本单独使用的设备模式综合在一起。该模式用共同的功能实现了诸多对象的浏览计数，并对系统的所有设备分层，构建树状结构以轻松查看设备信息（状态、连接BUS信息、种类）。最重要的是把握设备顺序，支持能够按正确顺序控制电源的智能电源管理功能。

曾以2.6版内核驱动模式为目标的统一连续机制造就了sysfs这个出色成果。sysfs文件系统是虚拟文件系统，它向用户空间提供kobject构成的层次图的视图。此外，kobject包含在特定对象中，与能将包含自己的对象显示在层次图的bound作用类似。此处的sysfs是机制名，kobject是构成该机制的数据结构。

本节讲解kobject和相关数据结构的关系，并了解这些数据结构最终如何显示在sysfs中。然后以这些kobject知识为基础，了解driver_init()的实际的工作内容。

30.2.5.1 kobject和sysfs

如前所述，kobject的作用像bound一样，在2.6版引入的设备驱动模式中，为系统的所有设备层次图的节点添加设备。

kobject是设备模式的核心数据结构，但常包含在其他结构体中使用

:: include/linux/kobject.h

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

为制作各对象的层次图，kobject（kernel object，内核对象）具有所需父指针、名称、对象类型等成员变量。kobject通常并不直接写入，而是包含在其他结构体中，由相应结构体使用kobject功能。kobject通过这种方法提供管理内核内多个对象（数据结构）的通用方法（general method）。即，kobject由管理内核内多个数据结构时所需共同属性构成。

可集成相关kobject的kset

kset提供的工具能够集成许多kobject中相互关联的kobject。例如，制作集成所有PCI设备的kset，使设备的kobject->ktype指向该kset，kset通过链表具有属于自己的设备（当然是kobject连接的链表)。

:: include/linux/ kobject.h

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

用于记述kobject的运行及特性的ktype

ktype（kernel object type，内核对象类型）定义kobject能够执行的运算，而且由多个kobject共享，并将具有共同运算的kobject集成在一起。基于这种共同运算的组合与kset属于不同概念。共享共同运算的各kobject属于1个ktype，但有可能属于不同的kset。由kobject->ktype指向，并表现为kobj_type结构体。

:: include/linux/ kobject.h

```
struct kobj_type {
    void (*release)(struct kobject *kobj);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

release指定kobject的浏览计数为0时调用的函数，sysfs_ops记述sysfs上kobject显示为文件时的读写操作。

sysfs只是用目录结构显示kobject层次图的文件系统

sysfs是只存在于内存的虚拟文件系统。如前所述，kobject通过parent指针成员构建层次结构。通过kobject构建的层次图可在sysfs中显示为目录结构，该操作由kobject的dentry指针完成。sysfs显示构建当前系统的硬件拓扑，提供内核空间和用户空间之间传输数据的更便捷的接口。

我们了解了内核中的通用设备模式，还有内核对象、内核对象类型、内核设置等概念，及其与sysfs虚拟内存系统有何关系。

图30-7以块设备为例，表现了各种概念间的关系。系统内的硬盘等块设备显示为内核对象sda，是sysfs内的目录。使用内核对象类型显示各内核对象属性，在sysfs内名为dev、size、stat文件，并向用户空间提供信息。此外，所有块设备均通过内核设置block集成。需要注意的是，名为block的kset内包含了kobject对象。由此，sysfs具有block目录，并成为所有下级kobject的父对象。这种关系体现了sysfs内的层次结构。

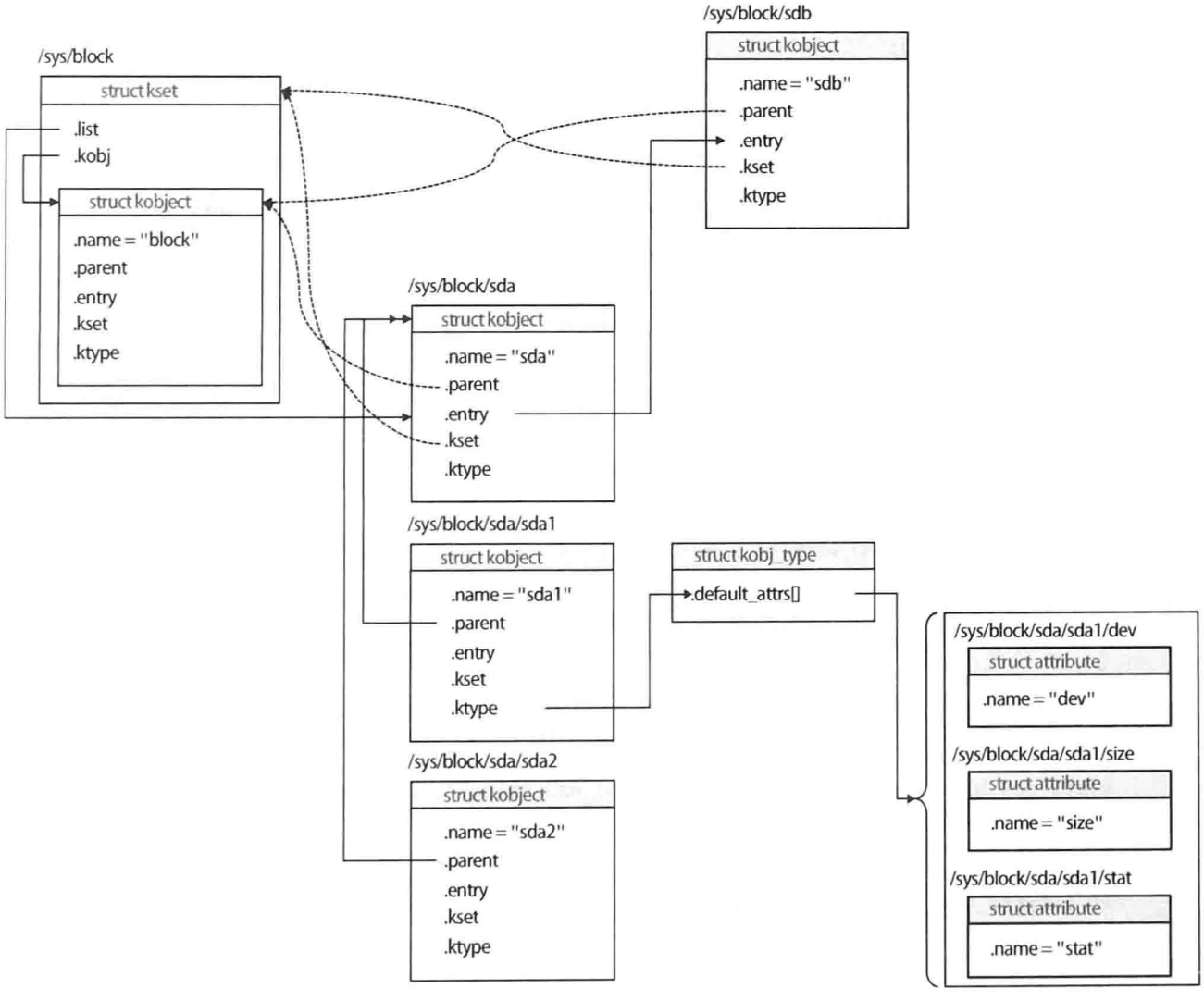


图30-7 sysfs中块设备的连接结构

30.2.5.2 driver_init()

在driver_init()中构筑sysfs的骨架，使sysfs能够显示以后调用的do_initcalls()中初始化的设备驱动。

:: 代码30-12 drivers/base/init.c的driver_init()

```
void __init driver_init(void)
{
    devices_init();                                    ❶
    buses_init();
    classes_init();
    firmware_init();
    hypervisor_init();
```

```
    platform_bus_init();                                                    ❷
    system_bus_init();
    cpu_dev_init();
    memory_dev_init();
}
```

该函数的作用大致有2点：第一，生成设备驱动模式中最核心的kobject，即设备、BUS、类等，并注册到sysfs；第二，在前面注册的子系统中注册platform_bus、system_bus、cpu、memory等其余核心子系统。

第一部分调用的大部分函数执行的工作都是类似的，先分析devices_init()，然后简单讲解其余函数。

:: 代码30-13　drivers/base/core.c的devices_init()

```
int __init devices_init(void)
{

    devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL);    ❶
    if (!devices_kset)
        return -ENOMEM;

    dev_kobj = kobject_create_and_add("dev", NULL);                             ❷
    if (!dev_kobj)
        goto dev_kobj_err;
    sysfs_dev_block_kobj = kobject_create_and_add("block", dev_kobj);           ❸
    if (!sysfs_dev_block_kobj)
        goto block_kobj_err;
    sysfs_dev_char_kobj = kobject_create_and_add("char", dev_kobj);             ❹
    if (!sysfs_dev_char_kobj)
        goto char_kobj_err;

    return 0;
...
}
```

代码段❶调用kset_create_and_add()生成kset结构体变量。该变量的名称是devices，将device_uevent_ops注册为发生uevent时需要的运算。kset_create_and_add()经过如下过程初始化kset，在sysfs中生成devices目录。

在代码段❷中调用kobject_create_and_add()函数，生成1个kobject结构体变量。经过如下过程生成并初始化kobject，在sysfs的顶层目录中生成dev目录。

代码段❸、❹也经过如上过程生成并初始化kobject。

devices_init()之后调用的buses_init()、classes_init()、firmware_init()、hypervisor_init()分别生成kset变量BUS、class、以及kobject变量firmware、hypervisor，没有什么不同。

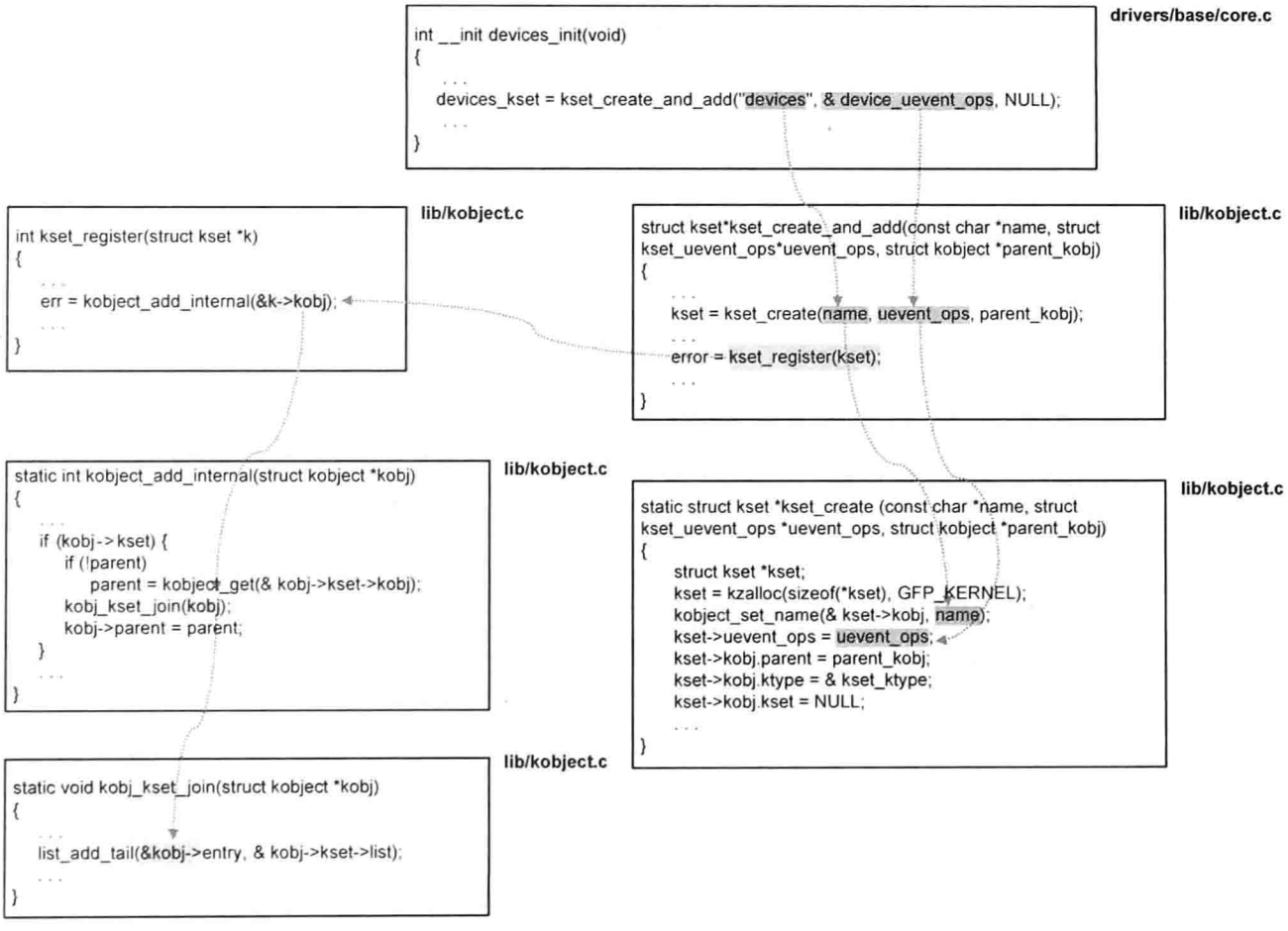


图30-8 调用结构

下面依次观察第二段中调用的函数。

:: 代码30-14 drivers/base/platform.c的platform_bus_init()

```
int __init platform_bus_init(void)
{
    int error;

    early_platform_cleanup();                                    ❶

    error = device_register(&platform_bus);                      ❷
    if (error)
        return error;
    error =  bus_register(&platform_bus_type);                   ❸
    if (error)
        device_unregister(&platform_bus);
    return error;
}
```

在代码段❶中，清除以前构筑的列表early_platform_add_devices中注册的所有设备信息。该

设备清除初始启动时（early boot time）进行初始化的、与平台密切相关的设备，如序列控制台。

在代码段❷中调用device_register()，将platform_bus注册为设备。如下所示，platform_bus只有名称被初始化，其父域和BUS线域被初始化为NULL。此时该设备注册为顶层BUS，在/sys/devices中生成platform目录。

:: drivers/base/platform.c

```
struct device platform_bus = {
    .init_name      = "platform",
};
```

代码段❸将platform_bus也注册到BUS。之前注册为设备，那注册到BUS的原因是什么呢？很简单，因为物理上存在的BUS也是一种设备。

:: drivers/base/platform.c

```
struct bus_type platform_bus_type = {
    .name           = "platform",
    .dev_attrs      = platform_dev_attrs,
    .match          = platform_match,
    .uevent         = platform_uevent,
    .pm             = PLATFORM_PM_OPS_PTR,
};
```

调用函数后，在/sys/bus中生成platform目录，内部又生成devices、drivers目录。

platform_init()以后调用的函数system_bus_init()做的工作十分简单：生成名为system的kset变量并注册到sysfs。调用函数时，向父进程传递devices_kset->kobj参数，并指定为系统的父进程。最终，在/sys/devices下生成system目录。

接下来调用cpu_dev_init()，生成并注册系统设备cpu。

:: 代码30-15　drivers/base/cpu.c的cpu_dev_init()

```
int __init cpu_dev_init(void)
{
    int err;

    err = sysdev_class_register(&cpu_sysdev_class);                    ❶
    if (!err)
        err = cpu_states_init();                                       ❷

#if defined(CONFIG_SCHED_MC) || defined(CONFIG_SCHED_SMT)
    if (!err)                                                          ❸
        err = sched_create_sysfs_power_savings_entries(&cpu_sysdev_class);
#endif

    return err;
}
```

在代码段❶中，将cpu_sysdev_class用作参数调用函数，以初始化cpu对象，在/sys/devices/system中生成为目录。观察sysdev_class_register()可知，将system_kset设置为cpu的父进程，注册到system下。

代码段❷在/sys/devices/system/cpu中生成显示CPU状态（topology information）的如下几种文件。

- kernel_max 内核设置允许的最大CPU数。
- online 显示联机状态且可用于调度的CPU。
- offline 显示因内核热插拔事件被off，因此不再处于联机状态的CPU，或由于当前系统超过了kernel_max允许的CPU数而无法使用的CPU。
- present 显示已确认安装在系统中的CPU。
- possible 显示可安装在系统中的CPU。

在代码段❸中，内核支持多核心调度器或超线程调度器时，额外为sys/devices/system/cpu生成文件。

最后调用memory_dev_init()，生成系统设备memory的kobject，并注册到sysfs。该函数结束后可通过sysfs删除内存并使用内存热插拔功能。

:: 代码30-16 drivers/base/memory.c的memory_dev_init()

```
int __init memory_dev_init(void)
{
    unsigned int i;
    int ret;
    int err;

    memory_sysdev_class.kset.uevent_ops = &memory_uevent_ops;                    ❶
    ret = sysdev_class_register(&memory_sysdev_class);
    if (ret)
        goto out;


    for (i = 0; i < NR_MEM_SECTIONS; i++) {                                      ❷
        if (!present_section_nr(i))                                              ❸
            continue;
        err = add_memory_block(0, __nr_to_section(i), MEM_ONLINE,                ❹
                               0, BOOT);
    if (!ret)
        ret = err;
    }

    err = memory_probe_init();                                                   ❺
    if (!ret)
        ret = err;
    err = block_size_init();                                                     ❻
    if (!ret)
        ret = err;
out:
```

```
    if (ret)
        printk(KERN_ERR "%s() failed: %d\n", __func__, ret);
    return ret;
}
```

代码段❶注册memory_uevent_ops以处理系统设备，即内存的uevent，并将memory注册到sysfs，生成/sys/devices/system/memory。

在代码段❷、❸、❹中，将启动过程中初始化的显示内存节区信息的项生成为/sys/devices/system/memory/中的目录。目录名以memory开始，节区ID紧随其后（如：memoryxxx）。此时，不通过代码段❸处理不存在的节区。

代码段❹生成内存节区项，并在相应目录内生成phys_index、state、phys_device、removable文件。这4个文件的作用如下。

- phys_index　保存内存节区的ID。
- state　可读写内存节区状态（联机、脱机）。
- phys_device　保存内存节区的物理内存设备名。
- removable　用数字保存能否删除内存节区（1：可删除，0：不可删除）

某些系统（虚拟环境）中，固件无法向内核通知内存热插拔事件。这种环境支持“probe”接口。在代码段❺中，内核支持该接口时生成/sys/devices/system/memory/probe文件。可通过下列命令将新的内存地址传递给内核。

```
$ echo 内存地址 > /sys/devices/system/memory/probe
```

目前只有PowerPC支持CONFIG_ARCH_MEMORY_PROBE。

30.2.6　在proc文件系统注册irq信息——init_irq_proc()

使用proc文件系统可在/proc/irq/下级目录看到对应于irq号的目录，该目录内有如下文件。

```
affinity_hint node smp_affinity spurious
```

init_irq_proc()中，为在Linux启动后通过proc文件系统获取irq信息，生成并注册上述文件及目录。图30-9显示了init_irq_proc()函数的工作流程。

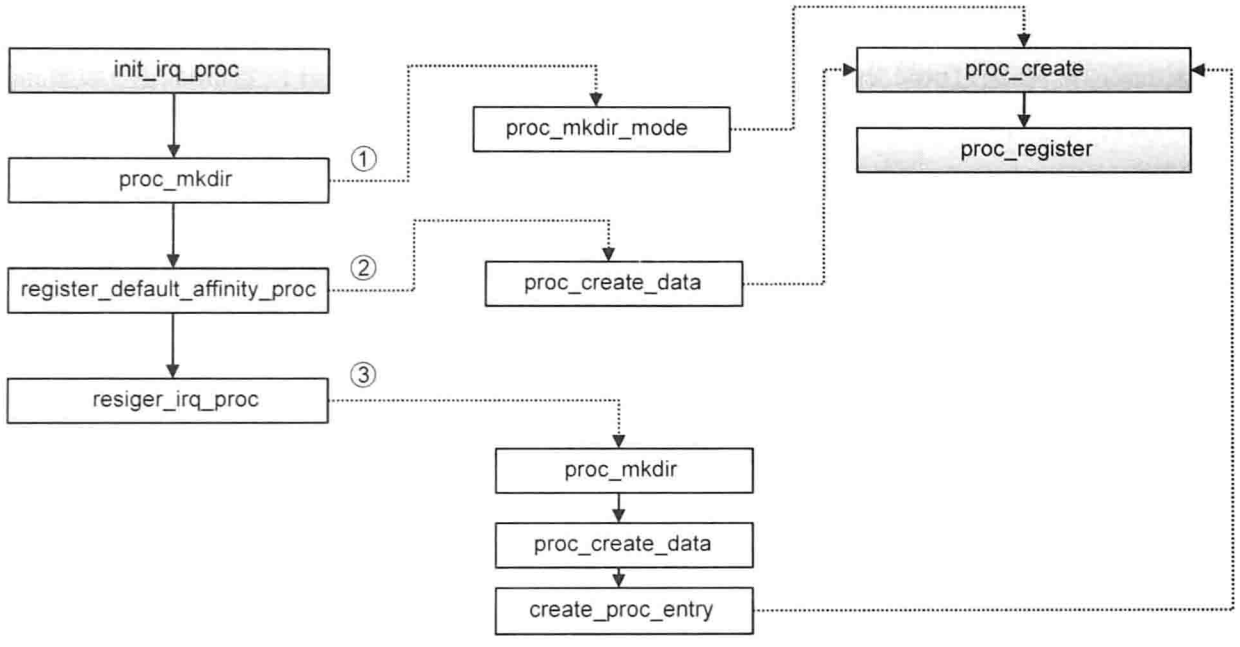


图30-9 init_irq_proc()函数的工作流

仔细观察下列代码。

:: 代码30-17 kernel/irq/proc.c的init_irq_proc()

```
void init_irq_proc(void)
{
    unsigned int irq;
    struct irq_desc *desc;

    root_irq_dir = proc_mkdir("irq", NULL);                    ❶
    if (!root_irq_dir)
        return;

    register_default_affinity_proc();                          ❷

    for_each_irq_desc(irq, desc) {
        if (!desc)
            continue;

        register_irq_proc(irq, desc);                          ❸
    }
}
```

代码30-17的代码段❶调用proc_mkdir()生成/proc/irq目录，然后在代码段❷中生成irq/default_smp_affinity文件，最后在代码段❸中生成对应于各irq号的目录，并生成affinity_hint、node、smp_affinity、spurious文件。如图30-9所示，init_irq_proc()函数的核心是调用proc_create()和proc_register()以制作文件和目录。

proc_create()函数的功能是动态分配proc_dir_entry结构体，并初始化该结构体的成员变量name或mode等。在通过proc_create()分配的proc_dir_entry结构体中求出并设置inode值，设置inode操作结构体proc_iops和file操作结构体proc_fops。

30.2.7　调用内核未知子系统——do_initcalls()

do_initcalls()函数十分简单，从__early_initcall_end节区到__initcall_end节区执行循环，同时调用保存的函数。这样做具有以下优点。

(1) 将子系统的初始化函数保存到特定区域（节区），无需了解内核中的所有子系统。

(2) 无需为添加并初始化某个新的子系统而修改主函数。

接下来讲解如何使其由do_initcalls()调用并位于initcalls节区。在内核的主目录中执行下列命令。

```
grep -rn "[a-z]*_init_call"
```

执行该命令会搜索定义多种子系统初始化的宏，对其中定义在arch/arm/mach-s3c6410/cpu.c的core_initcall(s3c6410_core_init)进行讲解。下面是各宏的定义。

:: include/linux/init.h

```
#define __define_initcall(level,fn,id)                \
    static initcall_t __initcall_##fn##id __used            \
    __attribute__((__section__(".initcall" level ".init"))) = fn

#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)
```

执行宏置换后定义如下。

```
static initcall_t __initcall_s3c6410_core_init_1 __used
__attribute__((__section__(.initcall1.init))) = s3c6410_core_init;
```

这样定义的值将位于.initcall1.init节区。相当于level的数字用于指定各函数调用顺序。core_initcall宏在其他函数前执行，然后执行通过postcore_initcall宏定义的函数，最后执行通过

late_initcall宏定义的函数。有可定义7个阶段顺序的宏。

利用该执行顺序可决定何时执行初始化程序。如果有先初始化特定子系统才能使用的子系统，那么可利用该阶段值决定顺序。而且，内核2.6版引入initcall系列的宏，之前的内核版本中只使用__initcall宏。设备驱动中可以看到用device_initcall定义的多种函数。

理解了initcall的讲解，那就逐一分析do_initcalls()。

:: 代码30-18 init/main.c的do_initcalls()

```
static void __init do_initcalls(void)
{
    initcall_t *call;

    for (call = __early_initcall_end; call < __initcall_end; call++)          ❶
        do_one_initcall(*call);                                               ❷

    flush_scheduled_work();                                                   ❸
}
```

在代码段❶中，从__early_initcall_end到__initcall_end区域执行循环，获得保存的函数指针值。vmlinux.lds.S和INITCALLS的定义如下所示。

:: arch/arm/kernel/vmlinux.lds.S

```
.init : {
    INIT_TEXT
    ...
    __initcall_start = .;
    INITCALLS
    __initcall_end = .;
    __con_initcall_start = .;
    *(.con_initcall.init)
    __con_initcall_end = .;
    ...
}
```

:: include/asm-generic/vmlinux.lds.h

```
#define INICALLS                                                \
 *(.initcallearly.init)                                         \
  VMLINUX_SYMBOL(__early_initcall_end) = .;                     \
  *(.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)
```

由链接器描述符定义在initcallN中的各初始化函数位于__early_initcall_end和__initcall_end之间。值得注意的是，也将位于.init节区内。这也意味着仅在启动的初始阶段使用，内核启动后不会占用内存。

重新观察代码段❶，最终获得初始化函数并调用代码段❷的do_one_initcall()。do_one_initcall()函数如下所示。

:: 代码30-19　init/main.c的do_one_initcall()

```
int do_one_initcall(initcall_t fn)
{
    int count = preempt_count();
    ktime_t calltime, delta, rettime;
    char msgbuf[64];
    struct boot_trace_call call;
    struct boot_trace_ret ret;

    if (initcall_debug) {                                          ❶
        call.caller = task_pid_nr(current);
        printk("calling  %pF @ %i\n", fn, call.caller);
        calltime = ktime_get();
        trace_boot_call(&call, fn);
        enable_boot_trace();
    }

    ret.result = fn();                                             ❷

    if (initcall_debug) {                                          ❸
        disable_boot_trace();
        rettime = ktime_get();
        delta = ktime_sub(rettime, calltime);
        ret.duration = (unsigned long long) ktime_to_ns(delta) >> 10;
        trace_boot_ret(&ret, fn);
        printk("initcall %pF returned %d after %Ld usecs\n", fn,
                ret.result, ret.duration);
    }

    msgbuf[0] = 0;

    ...

    return ret.result;
}
```

在代码段❶中，根据initcall_debug设置与否决定是否输出调试信息。在内核命令中赋予initcall_debug就会设置该值。若已经设置initcall_debug，则输出调用哪个函数以及从哪个进程调用等信息。然后求出函数调用前的时间。实际调用初始化函数的是代码段❷。

执行函数后，在代码段❸中重新输出调试信息。输出的信息有函数返回值、函数执行时间等。下面是输出的调试信息示例。

```
Calling initcall 0xc001dadc: net_ns_init+0x0/0x114()
net_namespace: 64 bytes
initcall 0xc001dadc: net_ns_init+0x0/0x114() returned 0.
initcall 0xc001dadc ran for 2 msecs: net_ns_init+0x0/0x114()
Calling initcall 0xc000ec0c: ptrace_break_init+0x0/0x2c()
initcall 0xc000ec0c: ptrace_break_init+0x0/0x2c() returned 0.
initcall 0xc000ec0c ran for 0 msecs: ptrace_break_init+0x0/0x2c()
Calling initcall 0xc000f79c: consistent_init+0x0/0x100()
initcall 0xc000f79c: consistent_init+0x0/0x100() returned 0.
initcall 0xc000f79c ran for 0 msecs: consistent_init+0x0/0x100()
Calling initcall 0xc0012ac8: sysctl_init+0x0/0x30()
initcall 0xc0012ac8: sysctl_init+0x0/0x30() returned 0.
initcall 0xc0012ac8 ran for 5 msecs: sysctl_init+0x0/0x30()
Calling initcall 0xc00139a4: init_jiffies_clocksource+0x0/0x1c()
initcall 0xc00139a4: init_jiffies_clocksource+0x0/0x1c() returned 0.
initcall 0xc00139a4 ran for 0 msecs: init_jiffies_clocksource+0x0/0x1c()
```

若需要更快启动，则根据该信息删除执行时间较长的或尚未使用的函数。这就是do_one_initcall()的执行过程，但do_initcalls()函数还未结束。最后，回到代码30-18的代码段❸调用flush_scheduled_work()函数。该函数将等待工作队列中的剩余函数全部执行结束，然后清空工作队列。此时才算结束do_initcalls()函数。

30.3 为初始化之后的操作做准备——init_post()

我们对start_kernel()中Linux内核所需全部数据结构进行了初始化，并生成了允许中断且用于kernel_init()的内核线程。此处的init_post()函数完成内核初始化并生成init进程，准确地说，正在执行的内核线程会转换成具有专属于自己的内核数据结构的一般进程。

init进程是启动过程中内核生成的第一个进程，其PID为1。它生成所有用户进程并监视其运行，在系统结束前始终保持执行状态。通过代码30-20详细了解init进程生成过程。

:: 代码30-20 init/main.c的init_post()

```
static noinline int init_post(void)
    __releases(kernel_lock)
{
    async_synchronize_full();                                    ❶
    free_initmem();
    unlock_kernel();
    ...

    if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)   ❷
```

```
    ...

    (void) sys_dup(0);
    (void) sys_dup(0);

    ...

    if (ramdisk_execute_command) {                                          ❸
        run_init_process(ramdisk_execute_command);
        printk(KERN_WARNING "Failed to execute %s\n",
                        ramdisk_execute_command);
    }

    if (execute_command) {                                                  ❹
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s.  Attempting "
                            "defaults...\n", execute_command);
    }
    run_init_process("/sbin/init");                                         ❺
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found.  Try passing init= option to kernel.");
}
```

首先，init_post()并不是__init函数。为防止和编译器声明为__init的kernel_init()初始化运行函数一起位于内存的.init.text，声明为noinline。若没有noinline声明，编译器就向调用init_post()的kernel_init()内部执行内联处理，和kernel_init()函数一起位于init.text。

代码段❶结束内核初始化。在asyne_synchronize_full()中结束所有非同步操作，准备释放内存，在free_initmem()中释放所有初始化函数和由数据使用的.init.data内存区域。unlock_kernel()减少lock_depth值解锁。

在代码段❷中生成用于标准输入、标准输出、标准错误的文件描述符（fd）。在sys_open()中打开系统中使用的控制设备“/dev/console”。初始运行时，生成标准输入文件描述符0，复制该文件描述符，在sys_dup()中生成标准输出文件描述符1和标准错误文件描述符2。

在代码段❸、❹、❺中，通过run_init_process()企图生成最初的用户区域进程init进程。run_init_process()接收文件名参数以完成此项操作。使用execve系统调用[①]，若执行成功就不返回，内核线程将变成init进程。

因此，从代码段❸的ramdisk_execute_command变量值“/init”（在kernel_init()中设置）的位置、代码段❹的execute_command变量值、代码段❺中的“/sbin/init”依次执行init进程，若所有执行均失败，则执行最后的panic()函数，发生内核恐慌。

① exec系统调用使用新执行的进程替代调用exec的进程内存。

第31章

内核线程守护进程

Linux内核的启动临近结束时，执行内核线程守护进程生成内核线程。本章讲解生成内核线程守护进程的kthreadd()函数。

31.1 内核线程守护进程——kthreadd()

内核线程守护进程是执行内核线程的守护进程，在内核启动时生成注册到kthread_create_list的所有内核线程。下面是kthreadd()函数。

:: 代码31-1 kernel/kthread.c的kthreadd()

```
int kthreadd(void *unused)
{
    struct task_struct *tsk = current;                                    ❶

    /* Setup a clean context for our children to inherit. */
    set_task_comm(tsk, "kthreadd");                                       ❷
    ignore_signals(tsk);                                                  ❸
    set_user_nice(tsk, KTHREAD_NICE_LEVEL);                               ❹
    set_cpus_allowed_ptr(tsk, cpu_all_mask);                              ❺

    current->flags |= PF_NOFREEZE | PF_FREEZER_NOSIG;

    for (;;) {
        set_current_state(TASK_INTERRUPTIBLE);                            ❻
        if (list_empty(&kthread_create_list))                             ❼
            schedule();
        __set_current_state(TASK_RUNNING);                                ❽

        spin_lock(&kthread_create_lock);

        while (!list_empty(&kthread_create_list)) {
            struct kthread_create_info *create;

            create = list_entry(kthread_create_list.next,                 ❾
                                struct kthread_create_info, list);

            list_del_init(&create->list);                                 ❿
            spin_unlock(&kthread_create_lock);
```

```
            create_kthread(create);

            spin_lock(&kthread_create_lock);
        }
        spin_unlock(&kthread_create_lock);
    }

    return 0;
}
```

在代码段❶中获取当前任务，current的定义如下所示。

:: arch/arm/include/asm/current.h

```
static inline struct task_struct *get_current(void)
{
    return current_thread_info()->task;
}

#define current (get_current())
```

调用get_current()函数，通过current_thread_info()函数返回thread_info结构体的任务。

在代码段❷中，将当前任务的comm成员变量值设置为“kthreadd”。task->comm是执行任务时使用的名称。[①]set_task_comm()函数只复制字符串，其定义如下所示。

:: fs/exec.c

```
void set_task_comm(struct task_struct *tsk, char *buf)
{
    task_lock(tsk);
    strlcpy(tsk->comm, buf, sizeof(tsk->comm));
    task_unlock(tsk);
}
```

在代码段❸中，将任务的信号处理器的操作（action）设置为忽略所有信号（31.2节）。

在代码段❹中，将任务的nice值设置为KTHREAD_NICE_LEVEL。KTHREAD_NICE_LEVEL声明如下。

:: kernel/kthread.c

```
#define KTHREAD_NICE_LEVEL (-5)
```

set_user_nice()函数相关内容请参考31.3节。

① 名称中不包含目录路径。

代码段❺的set_cpus_allowed_ptr()函数将变更可以执行给定任务的CPU值，即CPU亲和性（affinity），并移动到适当的CPU（31.4节）。

在代码段❻中，将当前任务设置为可执行中断（TASK_INTERRUPTIBLE）。

在代码段❼中，检查kthreadd要生成的具有内核线程列表的kthread_create_list是否为空，如果是空的，则意味着不再具有要生成的内核线程，因此调用schedule()函数让出处理器资源，使其他任务能够执行。schedule()函数相关内容请参考32.6节。

在代码段❽中，将当前任务的状态设置为TASK_RUNNING。需要注意的是，代码段❻中设置任务状态时使用了set_current_state()函数，但代码段❽中则使用了__set_current_state()函数。二者定义如下所示。

:: include/linux/sched.h

```
#define __set_current_state(state_value)                \
    do { current->state = (state_value); } while (0)
#define set_current_state(state_value)          \
    set_mb(current->state, (state_value))
```

二者最大区别在于set_current_state()函数使用内存屏障。需要对调用set_current_state()函数的调用者执行序列化（serialization）时使用，不需要序列化时，即不受顺序影响时使用__set_current_state()函数。

代码段❽到代码段⓫是kthreadd()函数的核心部分。

在代码段❾中，调用list_entry()函数使kthread_create_info结构体类型的create指向kthread_create_list的第一个成员。list_entry()函数返回Linux内核中具有list_head结构体成员的数据结构的起始位置（31.5节）。

kthread_create_info结构体具有要生成的内核线程的信息，其定义如下所示。

:: kernel/kthread.c

```
struct kthread_create_info
{
    /* Information passed to kthread() from kthreadd. */
    int (*threadfn)(void *data);                                    /*要执行的函数*/
    void *data;                                                     /*传递给函数的数据*/
    struct completion started;                                      /*通知“已开始”*/

    /* Result passed back to kthread_create() from kthreadd. */     /*from kthreadd.*/
    struct task_struct *result;                                     /*生成内核线程后的任务*/
    struct completion done;                                         /*通知“已结束”*/

    struct list_head list;                                  /*连接到kthread_create_list的成员*/
};
```

利用kthread_create_info结构体中设置的值生成内核线程。

由于代码段❾已获得要在kthread_create_list中生成的任务成员的信息，因此在代码段❿中将该成员从kthread_create_list中删除。

代码段⓫调用create_kthread()函数，利用代码段❾中获得的任务信息生成内核线程（31.6节）。

31.2　忽略信号——ignore_signals()

所有任务都要通过信号处理器（signal handler）处理传递给自己的信号，信号处理器成员是处理各信号操作处理器（action handler）数组，与传递给任务的信号数量相同。代码31-2是为忽略传递给任务的所有信号而设置的ignore_signals()函数。

:: 代码31-2　kernel/signal.c的ignore_signals()

```
void ignore_signals(struct task_struct *t)
{
    int i;

    for (i = 0; i < _NSIG; ++i)
        t->sighand->action[i].sa.sa_handler = SIG_IGN;                    ❶

    flush_signals(t);                                                     ❷
}
```

在代码段❶中，对所有信号（信号数_NSIG大小）设置操作处理器，也就是将用于处理任务相应信号的处理器设置为SIG_IGN。跟踪SIG_IGN可知其定义如下。

:: include/asm-generic/signal.h

```
#define SIG_IGN ((__force __sighandler_t)1) /* ignore signal */
```

:: include/linux/compiler.h

```
#define __force   __attribute__((force))
```

:: include/asm-generic/signal.h

```
typedef void __signalfn_t(int);
typedef __signalfn_t __user *__sighandler_t;
```

综上所述，SIG_IGN将成为用编译器属性（compiler attribute）指定force的void__signalfn_t(int)类型函数。

代码段❷调用flush_signals()函数，处理当前任务中处于待机状态（pending）的信号。下面是flush_signals()函数。

:: 代码31-3 kernel/signal.c的flush_signals()

```
void flush_signals(struct task_struct *t)
{
    unsigned long flags;

    spin_lock_irqsave(&t->sighand->siglock, flags);
    clear_tsk_thread_flag(t, TIF_SIGPENDING);                    ❶
    flush_sigqueue(&t->pending);                                 ❷
    flush_sigqueue(&t->signal->shared_pending);                  ❸
    spin_unlock_irqrestore(&t->sighand->siglock, flags);
}
```

代码段❶从任务获得thread_info结构体，在线程信息标签（thread information flag）中清除TIF_SIGPENDING标签位。①

代码段❷调用flush_sigqueue()函数全部释放待处理信号。flush_sigqueue()函数的定义如下所示。

:: kernel/signal.c

```
void flush_sigqueue(struct sigpending *queue)
{
    struct sigqueue *q;

    sigemptyset(&queue->signal);
    while (!list_empty(&queue->list)) {
        q = list_entry(queue->list.next, struct sigqueue , list);
        list_del_init(&q->list);
        __sigqueue_free(q);
    }
}
```

对具有任务待处理信号列表的t->pending列表按顺序执行循环，同时将所有信号从t->pending列表中删除，之后将分配的内存返回slab分配器。②

代码段❸也将任务共享的待处理信号（t->signal->shared_pending）从所有列表中删除，使slab分配器能够重新使用内存。图31-1是task_struct结构体中各信号相关成员变量的结构图，表示__signqueue_free()函数中释放的内存区域。

① 从任务获得thread_info结构体的方法请参考第8章“补充知识点”。

② __sigqueue_free()函数从内部调用kmem_cache_free()，生成可分配struct_sigqueue结构体的sigqueue_cachep，通过slab分配器分配或回收struct sigqueue对象。

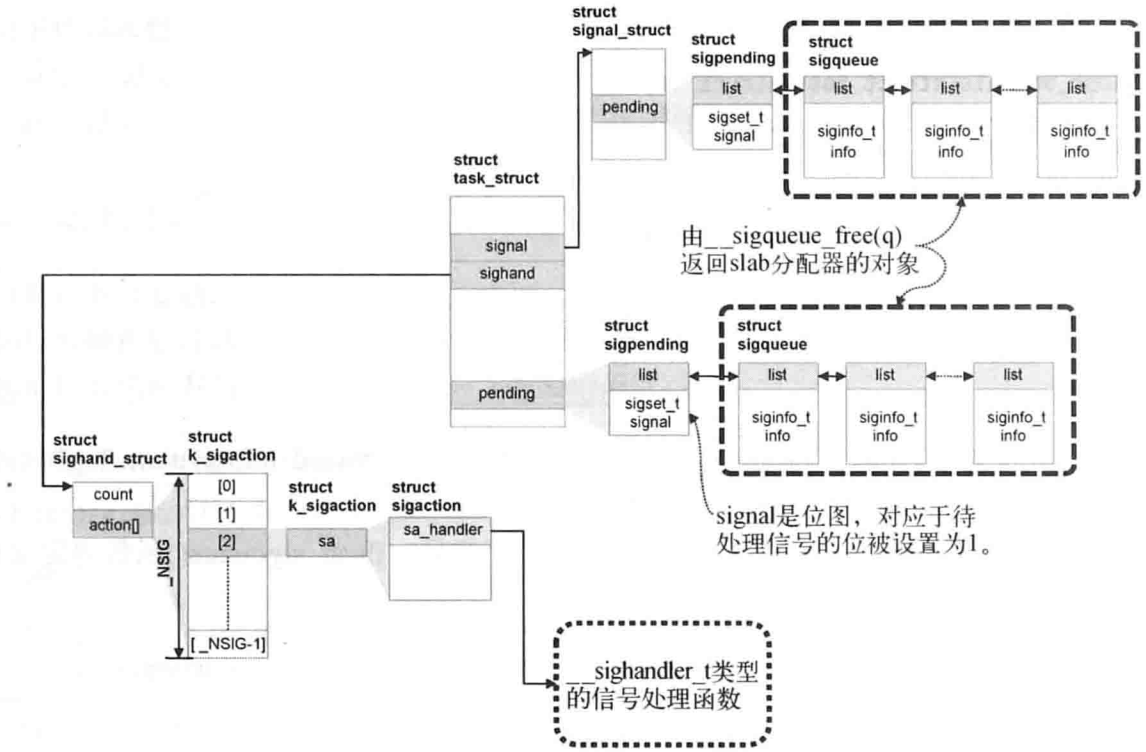


图31-1　task_struct结构体的信号相关结构

31.3　设置 nice 值——set_user_nice()

任务可具有的nice值范围是[–20…19]共40个阶段的值。nice值越高，优先顺序就会移到其他任务；nice值越低，任务就越重要。[①]通过set_user_nice()函数设置任务nice值，接收任务和nice值参数。下面通过代码31-4进行分析。

:: 代码31-4　kernel/sched.c的set_user_nice()

```
void set_user_nice(struct task_struct *p, long nice)
{
    int old_prio, delta, on_rq;
    unsigned long flags;
    struct rq *rq;

    if (TASK_NICE(p) == nice || nice < -20 || nice > 19)                    ❶
        return;

    rq = task_rq_lock(p, &flags);                                           ❷
    update_rq_clock(rq);
```

① nice在英语里是“亲切、好”的意思，任务“nice”是指由于自己经常做出让步，因此对其他的任务来说显得十分亲切。因此nice值越高，其优先顺序越低。

```
    if (task_has_rt_policy(p)) {                                    ❸
        p->static_prio = NICE_TO_PRIO(nice);
        goto out_unlock;
    }
    on_rq = p->se.on_rq;
    if (on_rq)                                                      ❹
        dequeue_task(rq, p, 0);

    p->static_prio = NICE_TO_PRIO(nice);                            ❺
    set_load_weight(p);
    old_prio = p->prio;
    p->prio = effective_prio(p);                                    ❻
    delta = p->prio - old_prio;

    if (on_rq) {                                                    ❼
        enqueue_task(rq, p, 0);
        if (delta < 0 || (delta > 0 && task_running(rq, p)))
            resched_task(rq->curr);
    }
out_unlock:
    task_rq_unlock(rq, &flags);
}
EXPORT_SYMBOL(set_user_nice);
```

代码段❶检查输入的nice值是否在[–20…19]范围内，并比较输入的nice值和当前任务的nice值是否一致。TASK_NICE宏将任务的static_prio值转换成nice值，其定义如下所示。

:: kernel/sched.c

```
#define NICE_TO_PRIO(nice)  (MAX_RT_PRIO + (nice) + 20)
#define PRIO_TO_NICE(prio)  ((prio) - MAX_RT_PRIO - 20)
#define TASK_NICE(p)        PRIO_TO_NICE((p)->static_prio)
```

nice值具有[–20…19]的值，而static_prio值在[0…MAX_RT_PRIO+19+20]，即MAX_RT_PRIO==100时，具有[0…139]范围的值。

在代码段❷中获得任务要执行的CPU的就绪队列。task_rq_lock()函数相关分析请参考29.2节。

代码段❸检查任务是否具有实时任务策略（SCHED_FIFO或SCHED_RR）。如果是实时任务，则利用NICE_TO_PRIO宏将输入的nice值转换成优先顺序值，然后填充static_prio变量值并结束。

任务从代码段❹起不是实时任务，如果当前就绪队列中存在任务，则调用dequeue_task()函数将任务从中删除。如下所示，dequeue_task()函数从内部调用并执行任务调度器类中定义的dequeue_task()函数。

:: kernel/sched.c

```
p->sched_class->dequeue_task(rq, p, sleep);
p->se.on_rq = 0;
```

代码段❺调用NICE_TO_PRIO宏，将输入的nice值转换成优先顺序值，调用set_load_weight()函数指定p->se->load值。sched_entity结构体的成员变量load是用于负载均衡的值，该值表示可调度实体（schedulable entity），即任务或任务组占据的队列中的负载。该值以nice值为基准，在内核中预先计算并保存到静态数组，各数组值和nice值相关内容请参考“补充知识点”。

代码段❻调用29.3节的effective_prio()函数设置任务的prio值。此外，将以前的prio值和调用effective_prio()函数变更的prio值之差保存到delta，以delta值为基准，在代码段❼中决定是否重新调度任务。

代码段❼将任务再次插入就绪队列，要记住，代码段❹曾从就绪队列中删除过任务。由于on_rq变量具有任务未删除时的值，因此，如果删除则调整优先顺序值，之后重新插入就绪队列。插入任务后，根据任务的优先顺序值和任务执行与否调用resched_task()函数重新执行调度。有一点需要确认，若代码段❹中由于任务不在就绪队列而未删除，则不执行代码段❼。



补充知识点　指定负载权重（load weight）并使用倒数权重（inverse weight）值的示例

内核使用负载权重值保持任务负载均衡。负载权重在load_weight结构体中保存值，作为任务结构体（task_struct）的成员变量即sched_entity结构体的成员变量，其定义如下所示。

:: include/linux/sched.h

```
struct load_weight {
    unsigned long weight, inv_weight;
};
```

weight变量是任务在就绪队列中占据的负载，inv_weight是对应于任务nice值的weight的倒数值。讲解set_load_weight()函数时再阐述二者关系。

指定负载权重的函数是set_load_weight()，其定义如下所示。

:: kernel/sched.c

```
static void set_load_weight(struct task_struct *p)
{
    /*为实时任务时*/
    if (task_has_rt_policy(p)) {
        p->se.load.weight = prio_to_weight[0] * 2;
        p->se.load.inv_weight = prio_to_wmult[0] >> 1;
        return;
    }

    /*idle任务*/
    if (p->policy == SCHED_IDLE) {
        p->se.load.weight = WEIGHT_IDLEPRIO;
        p->se.load.inv_weight = WMULT_IDLEPRIO;
        return;
    }
```

```
    /*一般任务、SCHED_NORMAL、SCHED_BATCH*/
    p->se.load.weight = prio_to_weight[p->static_prio - MAX_RT_PRIO];
    p->se.load.inv_weight = prio_to_wmult[p->static_prio - MAX_RT_PRIO];
}
```

以实时任务、idle任务、一般任务为基准填充负载权重值，如代码所示，从prio_to_weight[]数组和prio_to_wmult[]数组取值。这2个数组是以nice值为基准在内核中预先计算的值。prio_to_weight[]数组和prio_to_wmult[]数组的定义如下所示。

:: kernel/sched.c

```
#define WEIGHT_IDLEPRIO                3    /*idle任务的weight*/
#define WMULT_IDLEPRIO         1431655765    /* inverse (WEIGHT_IDLEPRIO) */

static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};

static const u32 prio_to_wmult[40] = {
 /* -20 */     48388,     59856,     76040,     92818,    118348,
 /* -15 */    147320,    184698,    229616,    287308,    360437,
 /* -10 */    449829,    563644,    704093,    875809,   1099582,
 /*  -5 */   1376151,   1717300,   2157191,   2708050,   3363326,
 /*   0 */   4194304,   5237765,   6557202,   8165337,  10153587,
 /*   5 */  12820798,  15790321,  19976592,  24970740,  31350126,
 /*  10 */  39045157,  49367440,  61356676,  76695844,  95443717,
 /*  15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};
```

prio_to_weight[]数组为相当于[-20...19]范围内的nice值静态保存权重值。nice值为0时，weight为1024。nice值每增加1，可使用的CPU时间增加10%，weight值大约增加到1.25倍。

那么为什么需要倒数权重值？这是为了提高运算速度。prio_to_wmult[]数组保存着2^32/weight值，以将除法运算变更为乘法运算。

下面看看使用倒数权重值的示例。calc_delta_mine()函数以nice值0为基准，计算自己还有多少权重，其定义如下所示。

:: kernel/sched.c

```
#if BITS_PER_LONG == 32
# define WMULT_CONST    (~0UL)
```

```
#else
# define WMULT_CONST    (1UL << 32)
#endif

#define WMULT_SHIFT 32
#define SRR(x, y) (((x) + (1UL << ((y) - 1))) >> (y))

static unsigned long
calc_delta_mine(unsigned long delta_exec, unsigned long weight,
                struct load_weight *lw)
{
    u64 tmp;

    /*inv_weight值不存在时，求出倒数值。已定义时跳转*/
    if (!lw->inv_weight) {
        if (BITS_PER_LONG > 32 && unlikely(lw->weight >= WMULT_CONST))
            lw->inv_weight = 1;
        else
            lw->inv_weight = 1 + (WMULT_CONST-lw->weight/2) / (lw->weight+1);
    }

    tmp = (u64)delta_exec * weight;

    if (unlikely(tmp > WMULT_CONST))
        tmp = SRR(SRR(tmp, WMULT_SHIFT/2) * lw->inv_weight, WMULT_SHIFT/2);
    else
         /*和delta_exec * weight * lw->inv_weight / WMULT_CONST)相同*/
        tmp = SRR(tmp * lw->inv_weight, WMULT_SHIFT);

    return (unsigned long)min(tmp, (u64)(unsigned long)LONG_MAX);
}
```

该函数最终执行如下运算并返回结果值delta。

```
delta = delta_exec * weight / lw->weight
```

delta_exec和weight值是输入的参数值。weight值将成为利用特定nice值的权重值评价自己的负载权重的基准值。lw->weight就是相应任务的负载权重值。运算时需进行除法运算，这会耗费很多CPU周期。因此，将除法运算变更为乘法运算即可减少CPU周期。

对很多任务逐一执行实时负载权重计算后执行除法运算会造成CPU浪费。下面介绍使用lw->weight的倒数权重值lw->inv_weight值删除除法运算的方法。

weight的倒数值lw->inv_weight的定义如下所示。

```
lw->inv_weight    = (2^32) / lw->weight
                  = WMULT_CONST / lw->weight
```

将该公式整理为关于lw->weight的公式如下。

```
lw->weight        = WMULT_CONST / lw->inv_weight
```

再将lw->weight代入求delta的公式，如下所示。

```
delta   = delta_exec * weight / lw->weight
        = delta_exec * weight * lw->inv_weight / WMULT_CONST
```

由于WMULT_CONST被定义为1<<32，因此，上面的公式和将delta_exec*weight*lw->inv_weight值向右移动（WMULT_SHIFT）32位的值一样。最终，除法运算将变更为乘法运算和移位运算以提高运算速度。请各位确认注意，calc_delta_mine()函数中定义并使用了SRR宏以执行该运算。

※参考文献：http://studyfoss.egloos.com/5326671

31.4 搜索执行任务的 CPU——set_cpus_allowed_prt()

单一CPU生成的任务始终在单一CPU中执行，但在SMP环境下需要指定执行任务的CPU。此外，指定的CPU处于不可用状态（脱机状态）时，需要将其移动到其他CPU。

因此，SMP环境中有能执行任务的CPU集合，比集合之外的CPU具有更高的亲和性。通过下列代码分析set_cpus_allowed_ptr()函数，它生成新任务并指定执行该任务的CPU。

:: 代码31-5 kernel/sched.c的set_cpus_allowed_ptr()

```
int set_cpus_allowed_ptr(struct task_struct *p, const struct cpumask *new_mask)
{
    struct migration_req req;
    unsigned long flags;
    struct rq *rq;
    int ret = 0;

    rq = task_rq_lock(p, &flags);

    ...

    if (p->sched_class->set_cpus_allowed)                                    ❶
        p->sched_class->set_cpus_allowed(p, new_mask);
    else {
        cpumask_copy(&p->cpus_allowed, new_mask);
        p->rt.nr_cpus_allowed = cpumask_weight(new_mask);
    }

    if (cpumask_test_cpu(task_cpu(p), new_mask))                             ❷
        goto out;

    if (migrate_task(p,                                                      ❸
                     cpumask_any_and(cpu_online_mask, new_mask), &req)) {
        /* Need help from migration thread: drop lock and wait. */
        task_rq_unlock(rq, &flags);
        wake_up_process(rq->migration_thread);
        wait_for_completion(&req.done);
        tlb_migrate_finish(p->mm);
```

```
        return 0;
    }
out:
    task_rq_unlock(rq, &flags);

    return ret;
}
EXPORT_SYMBOL_GPL(set_cpus_allowed_ptr);
```

代码段❶如果有任务的调度器类的set_cpus_allowed()函数，则调用set_cpus_allowed()函数将任务的cpus_allowed成员变更为接收的new_mask。[①]

若不存在任务的调度器类的set_cpus_allowed()函数，则调用cpumask_copy()函数，直接将p->cpus_allowed域的值复制到new_mask。求出new_mask中设置的位数后，将其指定为p->rt.nr_cpus_allowed值，即指定可执行任务的CPU个数。

在代码段❷中获得执行任务p的CPU号，并检查对应于CPU号的位是否设置到new_mask位图。如果未设置，就无需将该CPU移动到其他CPU，因此结束set_cpus_allowed_ptr()函数。task_cpu()函数利用任务的thread_info结构体获得任务p的CPU号。

:: include/linux/sched.h

```
static inline unsigned int task_cpu(const struct task_struct *p)
{
    return task_thread_info(p)->cpu;
}
```

代码段❸表示要执行任务p的CPU未设置到new_mask位图，因此将任务p移动到new_mask位图中设置的CPU之一。调用就绪队列的migration_thread()函数，它作为wake_up_process()函数的参数传递而来。将任务移动到其他CPU，并调用wait_for_completion()函数，等待至移动完成。最后调用tlb_migrate_finish()函数，它在内部不执行任何操作。

:: arch/arm/include/asm/tlb.h

```
#define tlb_migrate_finish(mm)    do { } while (0)
```

31.5　搜索包含列表的实际结构体位置——list_entry()

Linux内核中有很多包含list_head结构体的结构体，使用list_head结构体将结构体连接到链表。内核访问链表列表后，如何找到包含list_head结构体成员的结构体起始位置呢？内核为此定义了container_of()宏。

① struct task_struct结构体的cpus_allowed成员变量是long类型组成的位图，它用于指定多处理器系统中相应任务要执行的CPU。

container_of()宏在内部利用（7.1节“补充知识点 offsetof的定义”）offsetof()宏求出list_head结构体成员的偏移量，从而找到包含list_head结构体的结构体起始位置。下面分析list_entry()宏以了解内核查找结构体起始位置的方法。

list_entry()宏的定义如下所示。

∷ include/linux/list.h

```
/**
 * 返回嵌入（包含）list_entry - 项的结构体
 * @ptr：list_head结构体指针
 * @type：要返回的结构体类型
 * @member：结构体的list成员变量名
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

接下来讲解代码31-1代码段❾的使用示例。

```
create = list_entry(kthread_create_list.next,
                    struct kthread_create_info, list);
```

从示例中可以看出，ptr、type、member分别按如下所示执行映射。

```
ptr    = kthread_create_list.next     (list_head结构体类型)
type   = struct kthread_create_info
member= list
```

请确认kthread_create_info结构体的list_head结构体类型成员的名称确为list。list_entry()的核心container_of()宏的定义如下。

∷ include/linux/kernel.h

```
/**
 * 将类型转换成包含container_of - list的结构体
 * @ptr：对member的指针
 * @type：用于类型转换的结构体类型
 * @member：成员变量名
 *
 */

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

将__mptr变量声明为结构体（如：kthread_create_info结构体）的成员变量（list）类型，从__mptr中减掉通过offsetof()宏获得的偏移量，即结构体内的成员变量（list）的位置，即可获得结构体（如：kthread_create_info结构体）的起始位置。

将该起始位置转换成结构体类型（kthread_create_info结构体）就能找到包含list_head结构体

的结构体起始位置，该过程如图31-2所示。

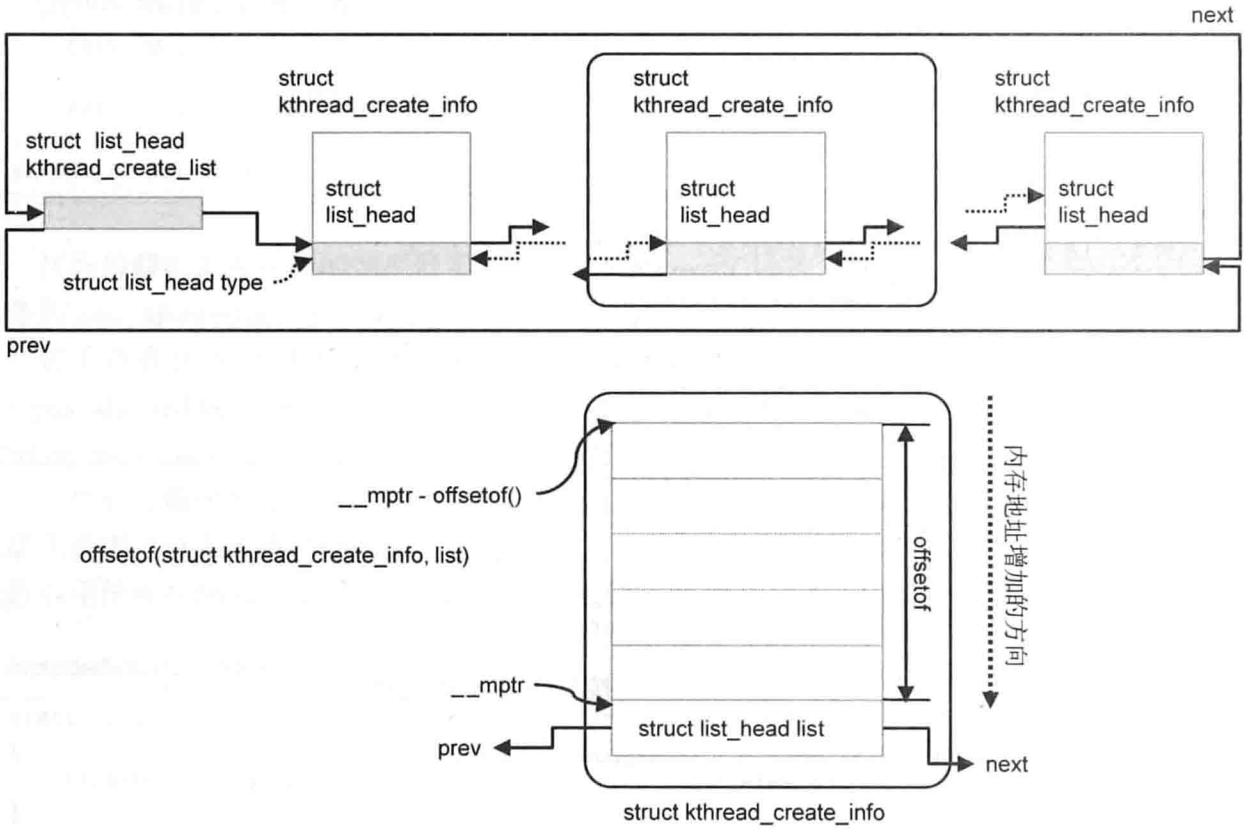


图31-2　从列表元素取得结构体

上述示例中，list_head结构体是kthread_create_info结构体的最后成员，但一般情况下，list_head无需成为结构体的最后成员。按如下所示，无需考虑list_head的位置也能找到结构体的起始位置。

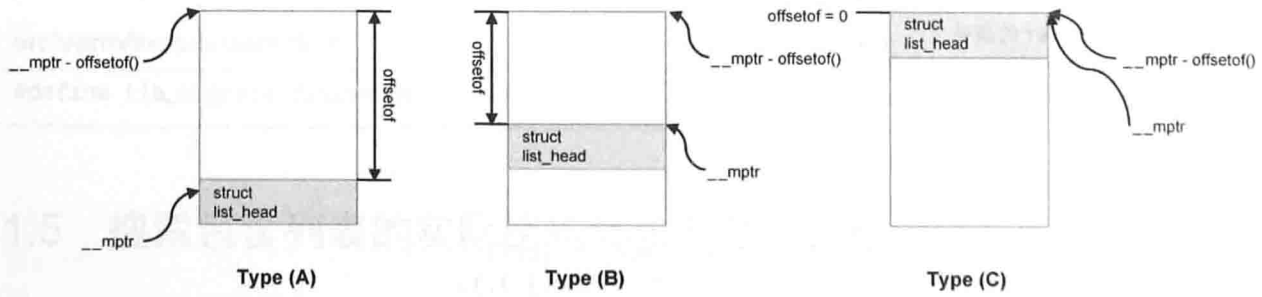


图31-3　结构体内list_head结构体的位置

31.6　生成内核线程——create_kthread()

内核线程守护进程kthreadd对具有内核线程信息的kthread_create_list执行循环，同时调用

create_kthread()函数生成内核线程。create_kthread()函数的定义如下所示。

:: 代码31-6 kernel/kthread.c的create_kthread()

```
static void create_kthread(struct kthread_create_info *create)
{
    int pid;

    /* We want our own signal handler (we take no signals by default). */
    pid = kernel_thread(kthread, create, CLONE_FS | CLONE_FILES | SIGCHLD);          ❶
    if (pid < 0)
        create->result = ERR_PTR(pid);
    else
        wait_for_completion(&create->started);                                      ❷
    complete(&create->done);                                                        ❸
}
```

如代码段❶所示，调用kernel_thread()函数生成对应于内核线程的任务。初次运行任务时执行kthread()函数。执行kthread()函数时，接收内核线程的信息kthread_create_info结构体类型参数create以执行内核线程。

代码段❷调用wait_for_completion()函数，等待create->started值变更。至于create->started值，内核线程成功生成时，create->started->done值加1。

代码段❸向create->done标注内核线程成功生成。通过代码31-7了解运行内核线程任务时执行的kthread()函数。

:: 代码31-7 kernel/kthread.c的kthread()

```
static int kthread(void *_create)
{
    struct kthread_create_info *create = _create;
    int (*threadfn)(void *data);
    void *data;
    int ret = -EINTR;

    /* Copy data: it's on kthread's stack */
    threadfn = create->threadfn;                                                    ❶
    data = create->data;

    /* OK, tell user we're spawned, wait for stop or wakeup */
    __set_current_state(TASK_UNINTERRUPTIBLE);                                      ❷
    create->result = current;
    complete(&create->started);
    schedule();

    if (!kthread_should_stop())                                                     ❸
        ret = threadfn(data);

    /* It might have exited on its own, w/o kthread_stop.  Check. */
    if (kthread_should_stop()) {
        kthread_stop_info.err = ret;
```

```
            complete(&kthread_stop_info.done);
        }
        return 0;
    }
```

在代码段❶中，将首次执行任务的函数和数据保存到局部变量。

在代码段❷中，将当前任务的状态设置为TASK_UNINTERRUPTIBLE，将当前任务current指定为接收返回结果的任务。complete()函数向create->started表示线程已成功生成，调用schedule()函数进入睡眠状态。

需要确认当前任务current正是内核线程。然后，调用kernel_thread()函数的（代码31-6）create_kthread()函数的任务就是通知内核线程已经生成的位置。

代码段❸调用kthread_should_stop()函数，接收结束当前内核线程的信号或唤醒任务的信号时执行。该函数检查内核线程是否停止执行。kthread_should_stop()函数定义如下，只按如下所示检查kthread_stop_info.k==current。

:: kernel/kthread.c

```
int kthread_should_stop(void)
{
    return (kthread_stop_info.k == current);
}
EXPORT_SYMBOL(kthread_should_stop);
```

kthread_should_stop()函数从其他任务调用kthread_stop()函数，需要停止内核线程时得到调用。kthread_stop_info结构体的定义如下所示，设置该结构体的值中止内核线程。

:: kernel/kthread.c

```
struct kthread_stop_info
{
     struct task_struct *k;
     int err;
     struct completion done;
};
```

如果无需中止内核线程，即kthread_should_stop()函数的返回值为0时，将执行接收的kthread_create_info结构体中指定的threadfn(data)函数参数。

第32章 find_task_by_pid_ns()~cpu_idle()

本章分析rest_init()函数中调用的find_task_by_pid_ns()函数到最后的cpu_idle()函数，调用顺序如图32-1所示。

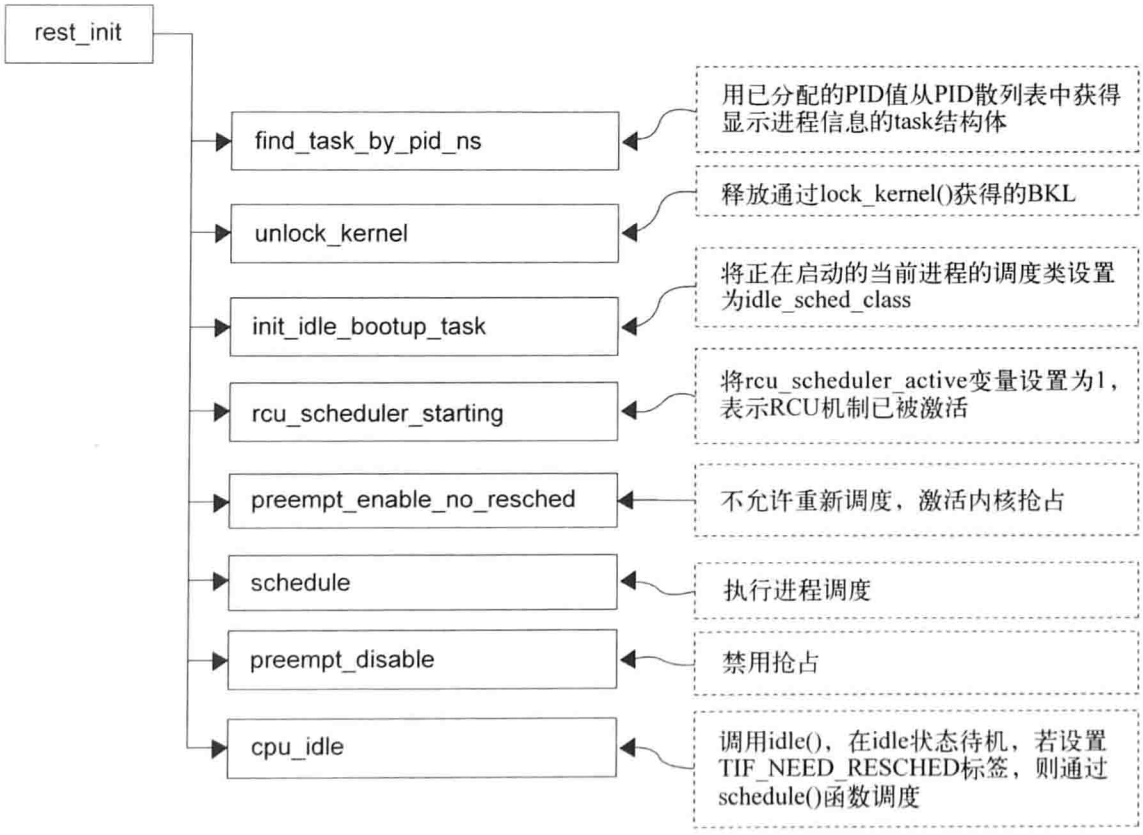


图32-1 find_task_by_pid_ns() ~ cpu_idle()

为使内核正式开始服务，解除大内核锁。此前执行启动的任务为了让其他进程抢占CPU而采取了一些措施，自己则作为idle任务留下来。

32.1　用 PID 搜索任务——find_task_by_pid_ns()

23.6节初始化了用于PID分配的pidmap，18.9节也预先构建了PID散列表以进行快速搜索。find_task_by_pid_ns()函数用分配到的PID值获得显示进程信息的task结构体。Linux的下列结构体管理PID。

pid结构体、upid结构体、namespace结构体、hlist_head(pid_hash)结构体

本节讲解上述这些结构体间的关系，以及如何用PID查找task结构体。为防止出现混淆，我们把struct pid称为pid结构体，将已知的PID（Process ID）称为PID号。接下来分析代码。

图32-2表示find_task_by_pid_ns()函数的调用流程。

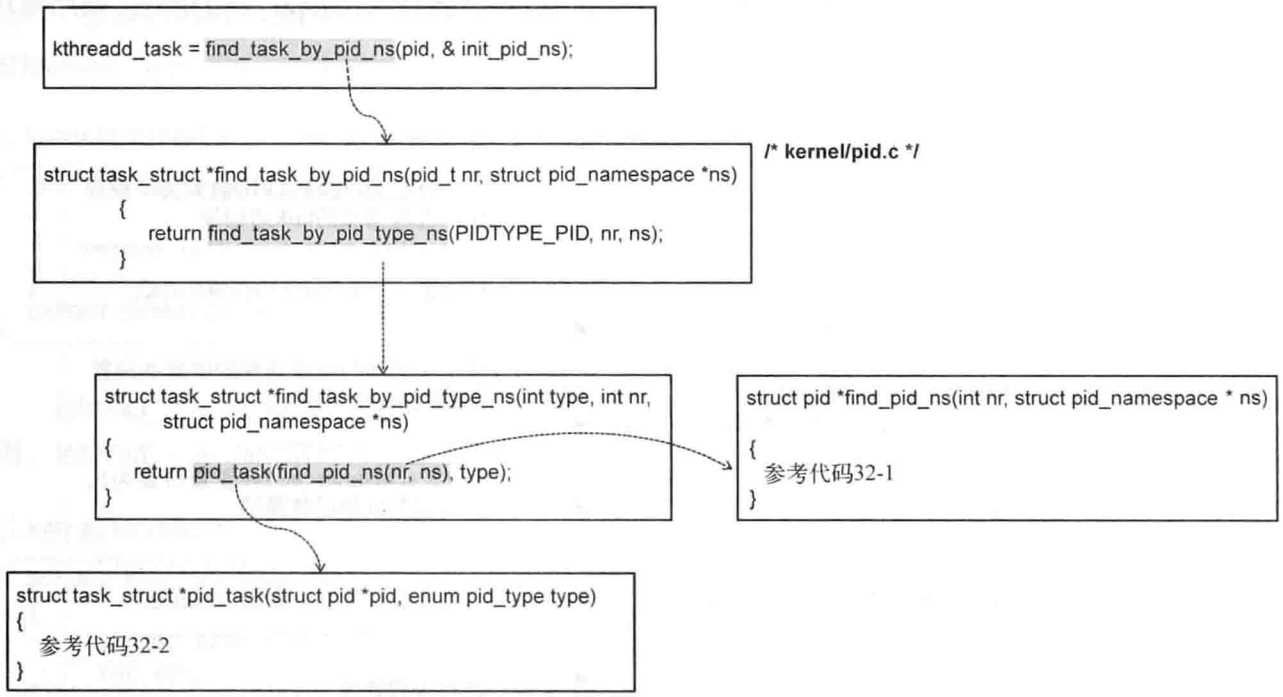


图32-2　find_task_by_pid_ns函数调用流程

如图所示，find_task_by_pid_ns()函数添加1个PIDTYPE_PID类型值以调用find_task_by_pid_type_ns()。find_task_by_pid_type_ns()函数也调用pid_task()函数获得task结构体。pid_task()函数调用find_pid_ns()函数，从相应的pid命名空间求出pid结构体，并将其作为pid_task()的参数传递。

下面通过find_pid_ns()代码了解用PID号从相应命名空间求出pid结构体的方法。

:: 代码32-1　kernel/pid.c的find_pid_ns()

```
struct pid *find_pid_ns(int nr, struct pid_namespace *ns)
{
    struct hlist_node *elem;
```

```
    struct upid *pnr;

    hlist_for_each_entry_rcu(pnr, elem,
            &pid_hash[pid_hashfn(nr, ns)], pid_chain)                    ❶
        if (pnr->nr == nr && pnr->ns == ns)
            return container_of(pnr, struct pid, numbers[ns->level]);    ❷

    return NULL;
}
```

pid结构体在内核内部识别进程，该结构体在各任务或进程组、节区中被浏览。此外，进程存活期间留在pid_hash列表中。因此，浏览该结构体的进程能用pid值快速查找task_struct结构体。

在代码段❶中，利用pid_hash中保存的upid结构体检查nr值。此处保存的nr就是已知PID号。搜索PID值之后，在代码段❷中找到PID并返回。为便于理解，图32-3显示了查找对应于PID号2的pid结构体的过程。

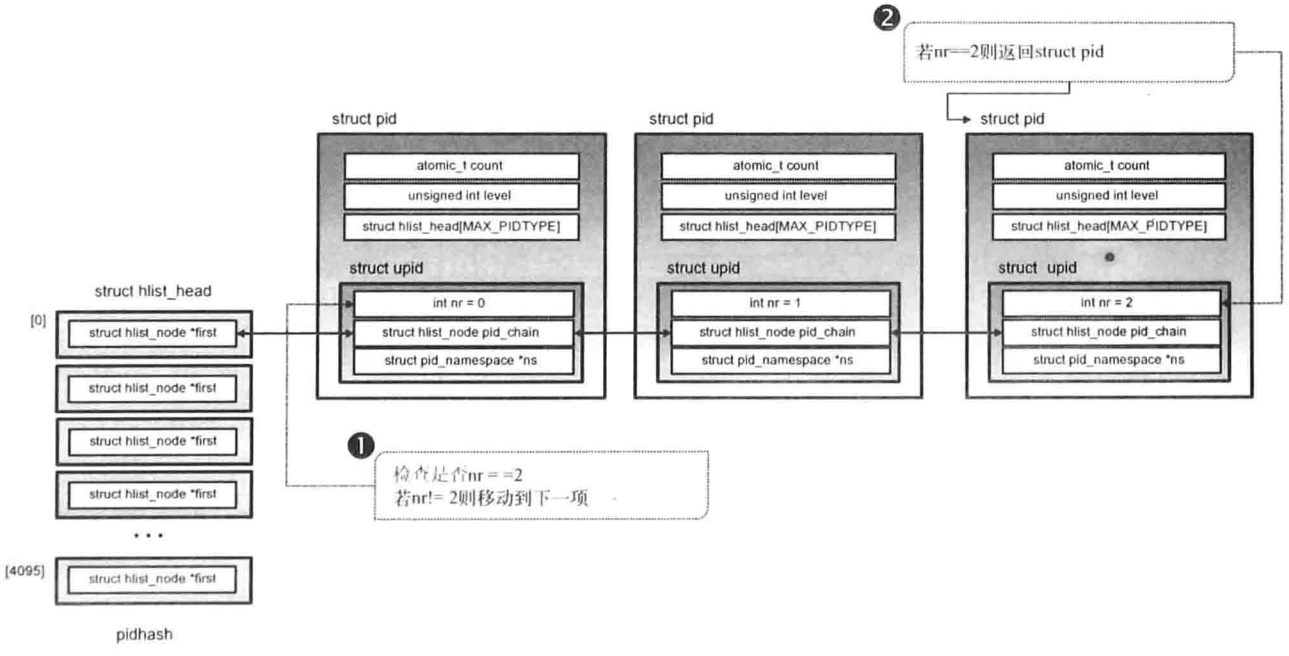


图32-3 查找对应于PID 2号的pid结构体

接下来讲解用找到的struct pid调用pid_task以查找task结构体的过程。

:: 代码32-2 kernel/pid.c的pid_task()

```
struct task_struct *pid_task(struct pid *pid, enum pid_type type)
{
    struct task_struct *result = NULL;
    if (pid) {
        struct hlist_node *first;
        first = rcu_dereference(pid->tasks[type].first);
```

```
        if (first)
            result = hlist_entry(first, struct task_struct, pids[(type)].node);          ❶
    }
    return result;
}
EXPORT_SYMBOL(pid_task);
```

task结构体的成员变量pids中，对应于各type（PIDTYPE_PID、PTDTYPE_PGID、PIDTYPE_SID）的pid结构体相互连接。因此，在代码段❶中找到与传递的pid结构体参数相连接的pids，返回task结构体。图32-4表示task结构体和pid结构体的连接关系。

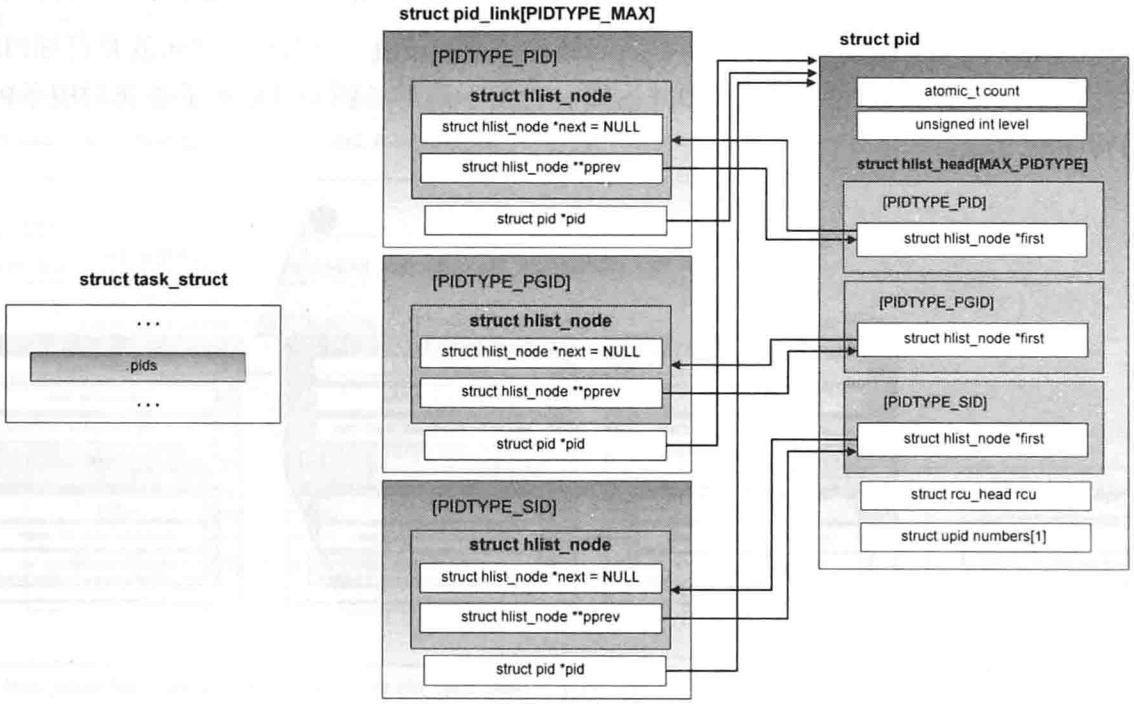


图32-4　pid结构体和task结构体的连接

这就是通过find_task_by_pid_ns()函数利用pid值求出相应task结构体的过程。本节虽然没有详细说明，但在clone系统调用中设置CLONE_NEWNS | CLONE_NEWPID标签可生成新的命名空间。

默认分配1个pid结构体的upid结构体，若生成新的命名空间，则pid结构体的level值加1，再生成1个upid结构体成员。以此找到按不同命名空间定义的PID号，从而找到task结构体。

32.2　解除 BKL——unlock_kernel()

unlock_kernel()是通过lock_kernel()获得的函数，用于释放BKL。

:: 代码32-3 lib/kernel_lock.c的unlock_kernel()

```
void __lockfunc unlock_kernel(void)
{
    BUG_ON(current->lock_depth < 0);
    if (likely(--current->lock_depth < 0))                    ❶
        __unlock_kernel();
}
```

BKL具有递归属性，因此可多次获得。但是，为解除通过lock_kernel()重复获得的锁，需要执行相同次数的unlock_kernel()。通过不同进程具有的lock_depth信息记录已获得的锁次数。如代码段❶所示，若调用unlock_kernel()将减少lock_depth计数，成为负值时才能解除BKL。

32.3 将调度类变更为 idle——init_idle_bootup_task()

如下所示，在该函数中将正在启动的当前进程的调度类设置为idle_sched_class。idle_sched_class是为idle进程而存在的具有特殊用途的调度类。

:: kernel/sched.c

```
idle->sched_class = &idle_sched_class
```

32.4 RCU 机制激活完成通知——rcu_scheduler_starting()

:: 代码32-4 kernel/rcupdate.c的rcu_scheduler_starting()

```
void rcu_scheduler_starting(void)
{
    WARN_ON(num_online_cpus() != 1);                          ❶
    WARN_ON(nr_context_switches() > 0);
    rcu_scheduler_active = 1;                                 ❷
}
```

代码段❶中，若当前处于联机状态的CPU数不是1个，或所有可能的（possible）CPU的上下文切换次数大于0，则调用__WARN()。

最后，在代码段❷中将rcu_scheduler_active变量设置为1，通知RCU机制已激活。

32.5 激活内核抢占——preempt_enable_no_resched()

:: 代码32-5 include/linux/preempt.h的preempt_enable_no_resched()

```
#define preempt_enable_no_resched() \
do { \
```

```
    barrier(); \                                                    ❶
    dec_preempt_count(); \                                          ❷
} while (0)


#define barrier() __asm__ __volatile__("": : :"memory")            ❸
#define dec_preempt_count() sub_preempt_count(1)
#define sub_preempt_count(val) do { preempt_count() -= (val); } while (0)
#define preempt_count() (current_thread_info()->preempt_count)
```

preempt_enable_no_resched()不设置重新调度标签，直接激活内核抢占。该宏的定义如上述代码所示，在代码段❶、❷中依次调用barrier()和dec_preempt_count()。

代码段❶的定义如代码段❸所示，该宏添加优化屏障，以该屏障为基准，可以保证前面的代码未完成执行前不执行后面的代码。[①]添加屏障的原因在于，执行编译器的代码优化后，抢占机制可能出现问题。

在代码段❷中调用dec_preempt_count()，最终将当前线程的preempt_count减少1。



补充知识点　激活/禁用抢占相关API

- preempt_enable()
  激活内核抢占。调用preempt_check_resched()检查是否需要重新调度，然后执行。
- preempt_enable_no_resched()
  激活内核抢占。
- preempt_disable()
  禁用内核抢占。

32.6　执行进程调度表——schedule()

在内核多处调用schedule()将CPU资源从当前运行的进程分配给其他进程。回到系统调用时，内核会检查是否设置当前进程的TIF_NEED_RESCHED标签。若已设置，则调用schedule()开始向其他进程转换。函数如下所示。

:: 代码32-6　kernel/sched.c的schedule()

```
asmlinkage void __sched schedule(void)
{
need_resched:                                                       ❶
    preempt_disable();                                              ❷

    __schedule();                                                   ❸
```

① 准确地说，不是所有代码，而是只对load_store指令给予保障。

```
    preempt_enable_no_resched();                                    ❹
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))               ❺
        goto need_resched;
}
EXPORT_SYMBOL(schedule);
```

函数头使用的__sched属性用于可能调用schedule()的函数。使用该属性的函数会进入.sched.text特殊节区，显示栈跳转等特殊信息时，激活这些函数就能忽略进入相应节区的所有函数调用。

在代码段❷中，通过preempt_disable()设置为调度时不执行内核抢占，然后代码段❸中止当前执行启动的进程，并调度为执行下一进程（32.6.1节）。

代码段❹不检查是否重新调度，直接激活内核抢占。代码段❺检查调度过程中need_resched标签变更后是否需要重新调度。此时，移动到代码段❶的标签以重新执行调度。

schedule()的真正内核——__schedule()

:: 代码32-7 kernel/sched.c的__schedule()

```
asmlinkage void __sched __schedule(void)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();                                       ❶
    rq = cpu_rq(cpu);
    . . .
    prev = rq->curr;
    switch_count = &prev->nivcsw;
    . . .
need_resched_nonpreemptible:
    . . .
    update_rq_clock(rq);
    clear_tsk_need_resched(prev);

    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {       ❷
        if (unlikely(signal_pending_state(prev->state, prev)))
            prev->state = TASK_RUNNING;
        else
            deactivate_task(rq, prev, 1);
        switch_count = &prev->nvcsw;
    }

    . . .

    put_prev_task(rq, prev);                                        ❸
```

```
    next = pick_next_task(rq);

    if (likely(prev != next)) {                                    ❹
        . . .
        context_switch(rq, prev, next);

        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
    } else
        spin_unlock_irq(&rq->lock);
    . . .
}
```

在代码段❶中，将正在执行的进程的task_struct结构体变量另行保存到prev，初始化need_resched标签并更新就绪队列的时间。

代码段❷中，如果正在运行的进程状态不是TASK_INTERRUPTIBLE，则可接收信号，因此检查有无延迟信号。如果有待处理的信号，则将进程状态转换为TASK_RUNNING，否则关闭进程。

在代码段❸中，调用put_prev_task()通知调度器类："当前运行的进程被替换为其他进程"。该函数不会在就绪队列中删除进程，而是使相应进程的统计信息等能够得到更新。然后，由pick_next_task()选择下一个要执行的进程。

代码段❹在显示正在执行的进程的rq->curr中设置下一个进程，并执行上下文切换。context_switch是与结构密切相关的函数，执行的是替换当前执行的mm_struct和寄存器信息等低级上下文切换。

32.7　Linux启动万里长征的终点——cpu_idle()

本节的cpu_idle()函数是start_kernel()函数内调用的最后函数，该函数的默认功能是：若未设置TIF_NEED_RESCHED标签，则反复调用idle()回调函数；若已设置，则通过schedule()函数执行调度。如代码32-8所示。

:: 代码32-8　arch/arm/kernel/process.c的cpu_idle()

```
void cpu_idle(void)
{
    local_fiq_enable();

    /* endless idle loop with no priority at all */
    while (1) {

        void (*idle)(void) = pm_idle;                              ❶

#ifdef CONFIG_HOTPLUG_CPU

        if (cpu_is_offline(smp_processor_id())) {
```

```
            leds_event(led_idle_start);
            cpu_die();
        }
#endif

        if (!idle)
            idle = default_idle;
        leds_event(led_idle_start);
        tick_nohz_stop_sched_tick(1);

        while (!need_resched())
            idle();                                                    ❷
        leds_event(led_idle_end);
        tick_nohz_restart_sched_tick();
        preempt_enable_no_resched();
        schedule();
        preempt_disable();
    }
}
```

代码段❶中，如果提供为电力管理执行特殊处理的回调函数pm_idle()，则该函数被注册为idle()函数指针；若不存在pm_idle()回调函数，则default_idle()函数被注册为idle()函数指针，在代码段❷中调用。

若已激活CONFIG_HOTPLUG_CPU，则通过相应处理器ID确认是否处于脱机状态。如果是，则调用cpu_die()宏。

通过下列代码观察未注册pm_idle()而调用default_idle()函数的情形。

:: 代码32-9 arch/arm/kernel/process.c的default_idle()

```
static volatile int hlt_counter;

__setup("nohlt", nohlt_setup);
__setup("hlt", hlt_setup);

static int __init nohlt_setup(char *__unused)
{
    hlt_counter = 1;
    return 1;
}

static int __init hlt_setup(char *__unused)
{
    hlt_counter = 0;
    return 1;
}

static void default_idle(void)
{
    if (hlt_counter)
```

```
            cpu_relax();                                          ❶
        else {
            local_irq_disable();
            if (!need_resched())
                arch_idle();                                      ❷
            local_irq_enable();
        }
    }
```

__setup()宏在内核命令行中注册对应于已定义参数的回调函数（18.2节）。如果已定义nohlt，则hlt_counter值为1；如果已定义hlt，则hlt_counter值为0。由于默认值为0，所以如果未定义任何参数，即使在代码段❷中调用arch_idle()函数，也可通过disable_hlt()、enable_hlt()内核API变更hlt_counter值。

内核命令行中已定义nohlt，如果hlt_counter值非0，则调用代码32-9代码段❶的cpu_relax()宏，其定义如下所示。

:: arch/arm/include/asm/processor.h

```
#define cpu_relax()                     barrier()
```

:: include/linux/compiler-gcc.h

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

cpu_relax()宏调用barrier()宏并制作内存屏障，内存屏障为了提高CPU性能而阻止对指令执行重排序（re-ordering）。嵌入式汇编代码中标记为“memory”的部分指内存变更，由于内存可变更，所以不通过编译器重排序执行优化。

附 录

附录 A　汇编语言、gas 关键词总结

附录 B　内核分析常见 API

附录 C　浅谈 ext2 文件系统

附录 D　Linux 线程模型

附录 E　链接器脚本文件结构

附录 A

汇编语言、gas关键词总结

A.1 ARM基本指令

分支指令

指　令	功　能	示　例
B label{f,b}	移动到label。 label后面的选项f、b是指位于下、上侧的label	b 1f b call_kernel
BL label{f,b}	和B相同，但lr中记录bl下一条指令的地址	bl cache_on

算数运算指令

指　令	功　能	示　例

ADD X Y Z	X中保存Y与Z之和	add r12, r12, #4*5
		add r2, sp, #0×10000
		add r5, r5, r0

SUB X Y Z	X中保存Y减Z之差	sub pc, lr, r0 sub r3, r4, #16384

逻辑运算指令

指　令	功　能	示　例
BIC X Y Z	在Y中解除Z中设置的位并保存到X X = Y&~Z	bic r3, r3, #0x3f00 bic r3, r3, #0xff
EOR X Y Z	将Y和Z执行exclusive or运算并保存到X X = Y^Z	eor r1, r1, r6
ORR X Y Z	Rd = Rn | N	orr r0, r0, #0x002d orr r2, r2, #0xc0
AND X Y Z	将Y和Z执行and运算的结果值保存到X	and r0, r1, r2

比较运算指令

指　令	功　能	示　例
CMP X Y	比较X和Y并更新状态标签	cmp r1, r2 cmp r6, ip
TEQ X Y	检查X和Y的值是否一致	teq r0, r2
TST X Y	检查Y中设置的位是否全部设置到X	tst r1, r1

数据传输指令

指　令	功　能	示　例

LDR X Y Z	将Y+Z的地址中保存的值加载到X	ldr r1, [r6, #0]
		ldr r13, __switch_data
		ldr r2, [r12, #4]
		ldr r6, =CONFIG_PROCESSOR_ID

STR X Y Z	将X指向的地址值保存到Y+Z	str r0, [r2], #4 str r1, [r6], #4
LDM X{!} Rx-Ry	从Rd中保存的地址依次取值并读入rx~ry的寄存器。每次都更新地址。后面有!时x的值会更新	ldmfd sp! , {r4 – r7, r9, pc} @r0中保存的值按顺序@放入值， r0不会更新 ldmia r0, {r1, r2, r3, r4, r5, r6} ldmia r3!, {r4, r5, r6, r7}
STM X{!} Rx-RY	将x指向的地址的值依次写入寄存器列表Rx-Ry。后面有!时x的值不会更新	stmfd sp!, {r4 – r7, r9, lr}

状态寄存器指令

指　令	功　能	示　例
MRS X CPSR	将CPSR的值读入X	mrs r2, cpsr
MSR CPSR X	在CPSR中设置X的值	mrs cpsr_c, r2

协处理器指令

指　令	功　能	示　例
MCR p15,0,Rx,Cx,Cy,x	执行相关协处理器指令	mcr p15, 0, r0, c1, c0, 0 @ r0写入控制寄存器 mcr p15, 0, r0, c2, c0, 0 @ D-Cache on mcr p15, 0, r0, c2, c0, 1 @ I-Cache on

（续）

指　令	功　能	示　例
MCR p15,0,Rx,Cx,Cy,x	执行相关协处理器指令	mcr p15, 0, r0, c3, c0, 0 @ write-buffer on mcr p15, 0, r0, c5, c0, 0 @ invalidate whole TLB v3 mcr p15, 0, r0, c6, c7, 1 mcr p15, 0, r0, c7, c0, 0 @ invalidate whole cache v3 mcr p15, 0, r0, c7, c10, 4 @ drain write buffer mcr p15, 0, r0, c7, c5, 0 @ flush(inval) I-Cache mcr p15, 0, r0, c7, c6, 0 @ flush(inval) D-Cache mcr p15, 0, r0, c7, c7 @ invalidate whole cache v4 mcr p15, 0, r0, c8, c7 @ invalidate whole TLB v4 mcr p15, 0, r0, c8, c7, 0 @清除flush I、D TLBs、I、D cache、TLB mcr p15, 0, r1, c3, c0, 0 @ load domain access control mcr p15, 0, r1, c7, c10, 4 @将drain WB、缓冲内容适用于内存 mcr p15, 0, r1, c7, c14, 0 @ clean+invalidate D mcr p15, 0, r1, c7, c15, 0 @ clean+invalidate unified mcr p15, 0, r1, c7, c5, 0 @ I-Cache、BTB无效 mcr p15, 0, r4, c2, c0, 0 @ load page table pointer mcr p15, 0, r5, c3, c0, 0 @ load domain access register
MRC p15,0,X,Cx,Cy,0	将对应于cx、cy的控制寄存器加载到Rd	mrc p15, 0, r0, c1, c0 @control reg复制到r0 mrc p15, 0, r0, c1, c0, 0 @ read control reg to r0 mrc p15, 0, r3, c0, c0, 0 @ read id reg mrc p15, 0, r6, c0, c0 @ get processor ID mrc p15, 0, r9, c0, c0 @ get processor id

其他指令

指　令	功　能	示　例
ADR X label	将label的地址保存到X	adr r12, proc_types adr r0, adr lr, __enable_mmu
MOV X Y	将Y的值保存到X	mov r0, r1

A.2　执行条件运算符

执行指令前检查状态标签，决定执行与否。

后 缀	标 签	含 义	cond code
EQ	**Z == 1**	相同	0000
NE	**Z == 0**	不同	0001
CS	**C == 1**	carry	0010
CC	**C == 0**	非carry	0011
MI	**N == 1**	负数	0100
PL	**N == 0**	正数、0	0101
VS	**V == 1**	溢出	0110
VC	**V == 0**	非溢出	0111
HI	**C == 1 and Z == 0**	无符号 大于	1000
HS	**C == 1**	无符号 大于等于	和CS相同
LO	**C == 0**	无符号 小于	和CC相同
LS	**C == 0 or Z == 1**	无符号 小于等于	1001
GE	**N == V**	有符号 大于等于	1010
LT	**N != V**	有符号 小于	1011
GT	**Z == 0 and N == V**	有符号 大于	1100
LE	**Z == 1 or N != V**	有符号 小于等于	1101

A.3 gas指令

指 令	含 义	示 例
.macro name arg1... expr **.endm**	像C语言的宏一样定义，调用name使用。和#define func(arg1, arg2) arg1 + arg2属于一个脉络	@ 执行mrc 指令 .macro writeb ch,rb mcr p14, 0, \ch, c0, c5, 0 .endm writeb r2,r3
.section	定义节区	@定义 __ex_table节区。 根据a标签allocatable， 标签是选项.section __ex_table,"a"
.align expr1	用no-op命令填充，以按照相应参数的2的倍数排列	@ 以4字节排列地址。 .align 2
.type name type	声明特定类型的符号 类型值有function、object等。类型值前可以有%,#,@,""，但每个结构都有不同用法。ARM中@表示注释开始，因此使用#	@ 声明data object类型的__secondary_data符号 .type __secondary_data, %object
.rept count **.endr**	将.rept和endr之间的命令反复记录count大小	@ 将mov r0,r0重复8次。 .rept 8 mov r0, r0 .endr

（续）

指　令	含　义	示　例
.word value or name	以WORD大小保存相应常数或符号标签的地址	@保存cpu_v6_proc_init的地址 .word cpu_v6_proc_init
.size name expr	将expr的值设置于符号name中	@在LC0 符号设置从当前地址（.）减掉LC0的地址的值 .size LC0, . - LC0
.global name **.globl** name	定义全局符号name	
.ltorg	在使用命令处制作当前节区的数据缓冲池。标签结束后主要进入的是填满空间的.word或.align的话，临时空间也会自动向后推移。若脱离偏移量则无法使用数据缓冲池，因此使用该命令制作数据缓冲池。数据缓冲池可视为用于读入32位常数值的列表	
.space size	用0填充size大小的空间	
.equ symbol value	在相应符号中设置值，.set是相同指令	
.long value	在指令所在处保存相应机器的整数型，.int是相同指令	
.if code **.endif**	和C preprocessor的#if、#endif相似。其用法完全相同。支持.ifdef .ifeq .ifge .ifle等	
.asciz	制作结尾有null的字符串	@制作“ARMv6-compatible processor”字符串。 .asciz “ARMv6-compatible processor”
name .req reg_name	制作register reg_name的别名name	@制作寄存器r0的别名foo。 foo .req r0
.text	使该指令下排列的指令均位于text节区	
.data	使该指令下排列的指令均位于data节区	
.previous	使该指令下排列的指令位于使用过的节区	

附录 B

内核分析常见API

B

双向链表

API名称	作　用
list_add	添加新项
list_add_tail	将新项添加到列表结尾
__list_del_entry	从列表删除相应项
list_replace	用新项替代旧项
list_del_init	从列表删除相应项并重新进行初始化
list_move	从列表删除相应项并添加到其他项
list_move_tail	从列表删除相应项并添加到其他列表末尾
list_is_last	确认是否为列表结束项
list_empty	确认列表是否为空
list_empty_careful	确认列表是否为空以及是否已修正
list_is_singular	确认列表是否具有1项
list_splice	将2个列表合在一起
list_splice_tail	2个列表相加，将新的列表添加到最后
list_splice_init	2个列表相加，重新初始化空的列表
list_splice_tail_init	2个列表相加，将新列表添加到最后，并重新初始化空的列表
list_entry	获得对应于项的结构体
list_first_entry	获得列表的第一个元素
list_for_each	搜索所有列表项
__list_for_each	搜索所有列表项
list_for_each_prev	逆向搜索所有列表项
list_for_each_safe	为删除项而安全搜索所有列表项

位运算API

API名称	作　用
set_bit()	将内存内的相应位设置为1
clear_bit()	将内存内的相应位设置为0
change_bit()	对内存内的相应位执行切换
test_and_set_bit()	将相应位设置为1并返回前值
test_and_clear_bit()	将相应位设置为0并返回前值
test_and_change_bit()	对内存内的相应位执行切换并返回前值
find_first_zero_bit()	返回最初设置为0的位的位置
find_first_next_zero_bit()	返回相应位置中设置为下一个0的位的位置
find_first_bit()	返回最初设置为1的位的位置
find_next_bit()	返回相应位置中设置为下一个1的位的位置

※参考文献：http://www.kernel.org/doc/htmldocs/kernel-api/

附录 C

浅谈ext2文件系统

各位一定听说过Linux将所有一切都作为文件（file）进行管理。因此，文件这一概念是Linux核心中的核心。文件系统（filesystem）管理保存媒介中的文件。本章不会详细分析文件系统内部，而是以磁盘文件系统中的ext2文件系统为例讲解文件系统概念。

C.1 显示文件的结构：索引节点

编译Linux源代码会生成vmlinux文件。那么，该文件在Linux中是如何管理的呢？文件被映射成1个索引节点。ext2文件系统中，ext2_inode结构体具有所有文件信息，该结构体定义如下。

:: include/linux/ext2_fs.h

```
struct ext2_inode {
    __le16  i_mode;                         /*文件模式*/
    __le16  i_uid;                          /*拥有者的uid*/
    __le32  i_size;                         /*文字大小（字节）*/
    __le32  i_atime;                        /*访问时间*/
    __le32  i_ctime;                        /*生成时间*/
    __le32  i_mtime;                        /*修正时间*/
    __le32  i_dtime;                        /*删除时间*/
    __le16  i_gid;                          /*组gid*/
    __le16  i_links_count;                  /*链接数*/
    __le32  i_blocks;                       /*块数*/
    __le32  i_flags;                        /*文件标签*/

    ...

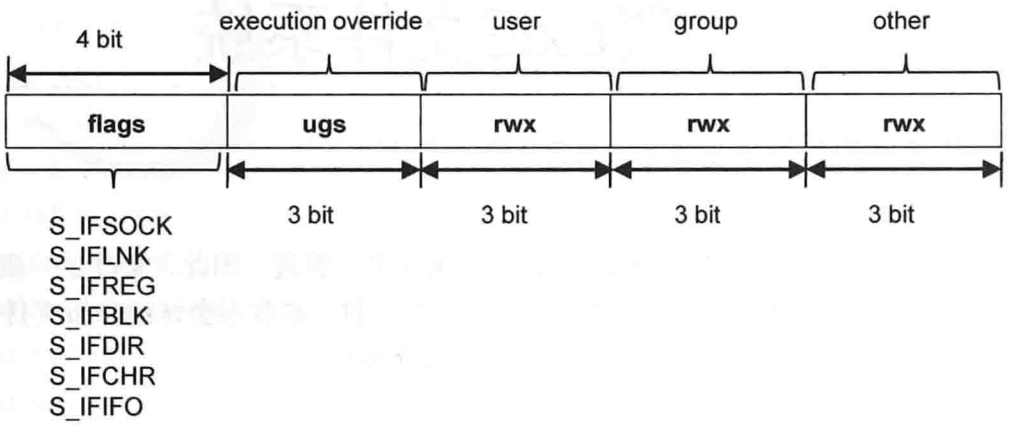
    __le32  i_block[EXT2_N_BLOCKS];         /*数据块指针*/

   ...
};
```

和用字节单位显示文件大小的i_size、显示访问时间的i_atime一样，可通过域名猜测各域所指含义。下节将详细讲解主要域中的i_mode域和i_block[]域。

C.2　实现i_mode域的位结构

i_mode域是16位类型，各位代表含义如图C-1所示。



图C-1　i_mode域各位含义

i_mode的顶层4位是表示文件类型的标签，其声明如下所示。

:: include/linux/stat.h

```
#define S_IFSOCK 0140000                    /*套接字文件*/
#define S_IFLNK  0120000                    /*符号链接文件*/
#define S_IFREG  0100000                    /*一般文件*/
#define S_IFBLK  0060000                    /*特殊块文件、块设备，用b标记*/
#define S_IFDIR  0040000                    /*目录*/
#define S_IFCHR  0020000                    /*特殊字符文件、文字设备，用c标记*/
#define S_IFIFO  0010000                    /*管道，用p标记*/
```

execution override区域是3位，声明如下。

:: include/linux/stat.h

```
#define S_ISUID  0004000                    /*指定setuid位*/
#define S_ISGID  0002000                    /*指定setgid位*/
#define S_ISVTX  0001000                    /*sticky位*/
```

S_ISUID是setuid（Set User ID，设置用户ID）位，也是执行Linux指令“chmod u+s〈filename〉”时设置的位。使可用相应文件的拥有者权限执行文件。同理，指定为可通过相应文件的组权限执行。主要用于需要比一般用户具有更高权限的情况。

例如，假设更换自己的登录密码。此时，用户需要打开并修正密码文件。用于更换密码的passwd命令可由一般用户执行，但必须有root权限。按如下所示用ls – al将passwd文件编成列表时，文件的属性为“–rwsr–xr–x”，其中显示用户访问权限区域的“s”通知相应文件中已设置setuid位。

```
code@kernel:~$ ls -al /usr/bin/passwd
-rwsr-xr-x 1 root root 42856 2012-07-15 07:11 /usr/bin/passwd
code@kernel:~$
```

最后，S_ISVTX是sticky位，文件是一般文件或目录时，其作用不一样。为一般文件时，使swap区域中无法删除相应文件，这样做是为了让vi或emacs等常用程序能够更快加载。但在现在的一些较快的磁盘或较大的内存中，这种方法并不十分有用。

但sticky位用作目录时，从安全角度讲是有利的。例如，/tmp目录等不特定多数共享的目录中设置了sticky位，就可以只让相应文件的拥有者删除文件。如下所示，/tmp目录的属性类似“drwxrwxrwt”，最后的“t”通知相应目录被设置为sticky位。

```
code@kernel:~$ ls -ald /tmp
drwxrwxrwt 18 root root 12288 2012-07-15 16:07 /tmp
code@kernel:~$
```

各用3位表示用户（user）、组（group）、其他（other）用户的访问权限。“rwx”的各位分别是读取（read）、写入（write）、执行（execute）权限的位。

C.3　带有文件数据的域——i_block[]

文件数据实际保存在哪里呢？i_block[]数组，它在ext2文件系统的inode结构体中声明如下。

:: include/linux/ext2_fs.h

```
#define EXT2_NDIR_BLOCKS                /*直接映射*/
#define EXT2_IND_BLOCK                  (EXT2_IND_BLOCK + 1)          /*间接映射，1阶段*/
#define EXT2_TIND_BLOCK                 (EXT2_DIND_BLOCK + 1)         /*间接映射，2阶段*/
#define EXT2_N_BLOCKS                   (EXT2_TIND_BLOCK + 1)         /*间接映射，3阶段*/

struct ext2_inode {
    ...

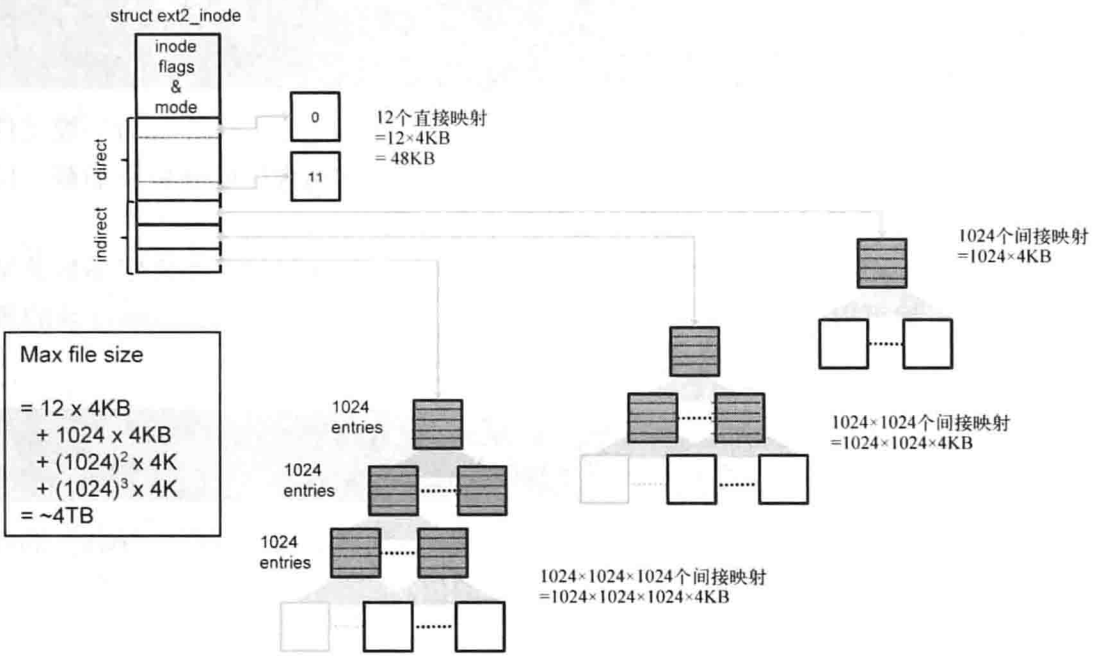
    __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */

    ...
};
```

i_block[]是__le32（32位）类型的数组。数组的元素（element）共EXT2_N_BLOCKS个，和EXT2_NDIR_BLOCKS + 3相同。EXT2_NDIR_BLOCKS表示直接映射的块数，12个数据块被直接映射。数据块的大小为文件系统中给定的大小，一般可保存4KB的数据。[①]

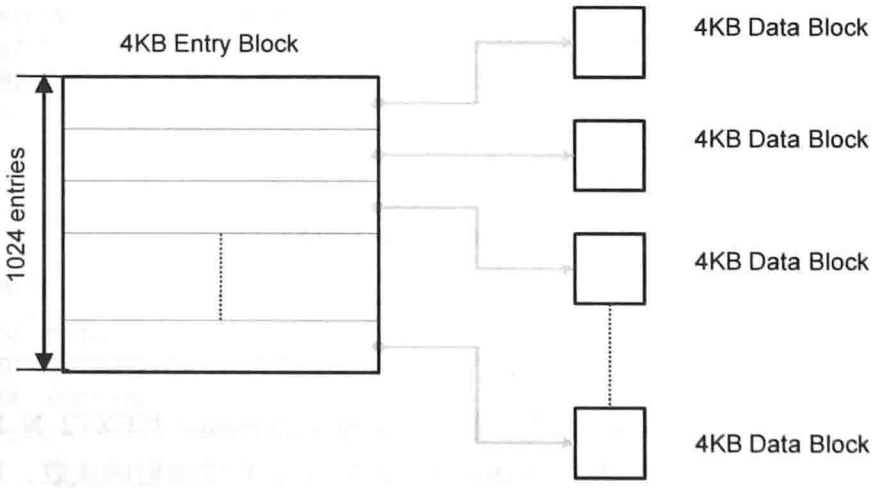
因此，直接映射时可保存12×4KB = 48KB数据。换言之，如果文件大小不足48KB，那仅用直接映射的12个块也可保存数据。超过48KB时需要通过间接映射块保存数据。图C-2表示如何通过直接及间接映射保存文件数据。

① 可指定1KB、2KB、4KB大小的数据块。



图C-2　直接/间接数据块映射

共可通过3个间接映射保存数据，为访问实际数据块，i_block[EXT2_DIND_BLOCK-3]需要1阶段、i_block[EXT2_TIND_BLOCK-2]需要2阶段、i_block[EXT2_N_BLOCK-1]需要3阶段的间接映射。



图C-3　按不同项映射数据块

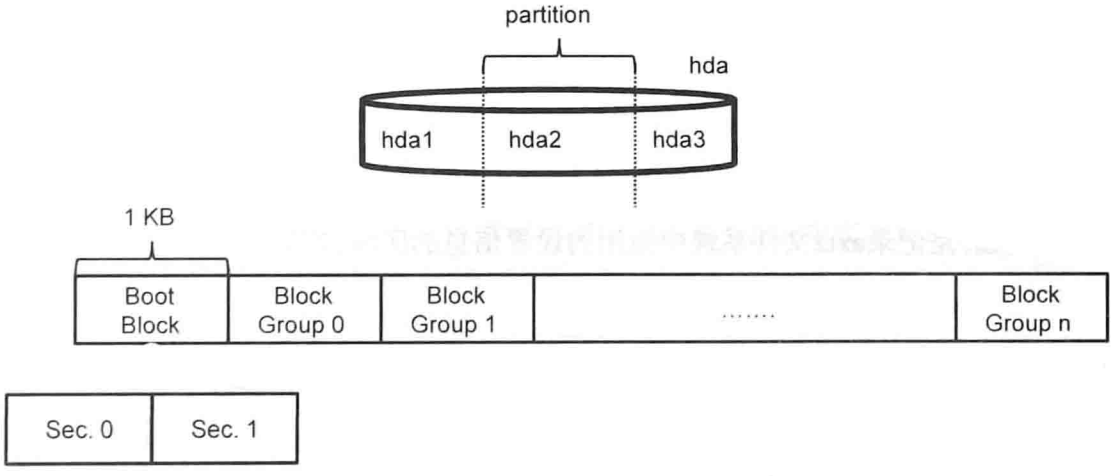
间接映射中，位于中间的4KB块由指向1024个块的地址组成，如图C-3所示。因此，1阶段间接映射时可保存1024×4KB大小、2阶段时可保存1024×1024×4KB大小、3阶段时可保存1024×1024×1024×4KB大小的数据。

C.4　磁盘中保存的结构

上节讲述了如何通过inode结构体表示保存文件数据的方法，通过inode表示的文件必须保存到磁盘才能访问并读取。本节对ext2文件系统如何实际管理磁盘进行讲解。

C.4.1　分区具有的结构

假设有个硬盘/dev/had。将该硬盘分成3个分区使用时，各分区会生成/dev/hda1、/dev/hda2、/dev/hda3等设备文件。文件系统按不同分区进行设置，在Linux中通过mke2fs / dev/hda2等命令将ext2文件系统设置到/dev/hda2。设置文件系统后，/dev/hda2由1个引导块（Boot Block）和多个块组（Block Group）构成，如图C-4所示。

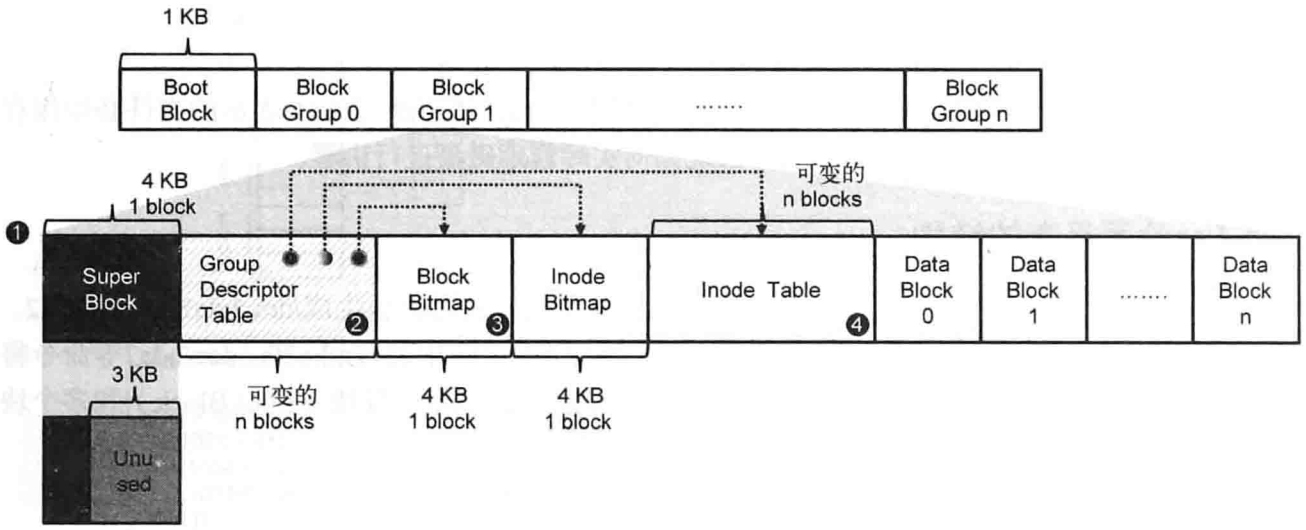


图C-4　文件系统的顶层抽象化（引导块和块组）

引导块大小为1KB，由2个512字节的节区构成。顾名思义，引导块记录启动映像的位置以执行启动。块组中有构成文件系统的信息和实际数据块（C.4.2节）。

C.4.2　块组结构

如图C-5所示，块组内部由超级块、组描述符列表、块位图、索引节点位图、索引节点列表、数据块构成。数据块中保存实际文件数据，是由索引节点的i_block[]数组指向的区域。



图C-5　块组结构

下面讲解块组包含的元数据。

1. 超级块：

由1个块构成，是记录ext2文件系统中使用的设置信息的区域，实际在1KB区域中保存元数据。保存的信息如下所示。

:: include/linux/ext2_fs.h

```
struct ext2_super_block {
    __le32   s_inodes_count;                    /*文件系统全部索引节点数*/
    __le32   s_blocks_count;                    /*文件系统全部块数*/
    ...
    __le32   s_free_blocks_count;               /*可用块数*/
    __le32   s_free_inodes_count;               /*可用索引节点数*/
    __le32   s_first_data_block;                /*块组0号开始的第一个数据块*/
    __le32   s_log_block_size;                  /*块大小（1KB、2KB、4KB）*/
    __le32   s_blocks_per_group;                /*每个组的块数*/
    __le32   s_inodes_per_group;                /*每个组的索引节点数*/
    __le32   s_mtime;                           /*挂载时间*/
    __le16   s_state;                           /*文件系统状态*/
    __le16   s_inode_size;                      /*inode结构体大小*/
    __le16   s_block_group_nr;                  /*保存超级块的块组号*/
    __u8     s_uuid[16];                        /*128位UUID卷识别符*/
    char     s_volume_name[16];                 /*卷名*/
    ...
};
```

2. 组描述符列表

和超级块一样，文件系统内的所有块组中保存相同信息。由于保存的是各块组的信息，因此称为列表（Table），由多个块组成。描述符的大小为32字节，其定义如下所示。

:: include/linux/ext2_fs.h

```
struct ext2_group_desc
{
    __le32  bg_block_bitmap;                /*块位图开始的块地址*/
    __le32  bg_inode_bitmap;                /*索引节点位图开始的块地址*/
    __le32  bg_inode_table;                 /*索引节点列表开始的地址*/
    __le16  bg_free_blocks_count;           /*可用块数*/
    __le16  bg_free_inodes_count;           /*可用索引节点数*/
    __le16  bg_used_dirs_count;             /*块组内生成的目录数*/
    __le16  bg_pad;                         /*待机以按4字节排列*/
    __le32  bg_reserved[3];                 /*预约区域*/
};
```

组描述符具有块组内指向块位图、索引节点位图、索引节点列表的地址信息，因为组描述符列表本身具有可变大小。各组的组描述符均保存在组描述符列表中，所有块组具有相同信息。

3. 块位图&索引节点位图

- 块位图用位图管理块组中的块可用与否，由1个块构成。因此，块的大小为4KB时，块位图可管理1024 × 4 × 8位个块。
- 索引节点位图用位图管理块组中的索引节点个数，由1个块构成。块的大小为4KB时，索引节点位图可管理1024 × 4 × 8个索引节点。

4. 索引节点列表

索引节点列表中保存ext2_inode结构体，由多个连续的块构成。在超级块中利用每组的索引节点个数信息，可求出索引节点列表所占块数和索引节点大小。此外，可通过索引节点位图确认索引节点是否可用。从索引节点号可知块组号，索引节点列表项由128字节构成。

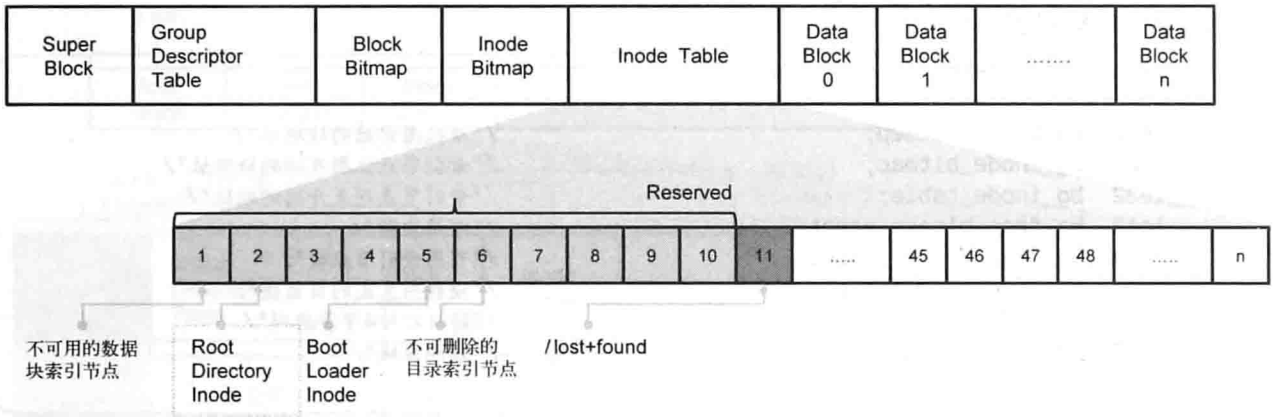
INODES_PER_BLOCK = 4KB / 128 字节

Block Group # = (inode # - 1) / INODES_PER_GROUP

用于索引节点列表的块数 = NODES_PER_GROUP / INODES_PER_BLOCK

C.4.3　索引节点列表结构

索引节点列表由保存索引节点的连续的块构成，索引节点号从1号开始。图C-6表示索引节点列表结构。



图C-6　索引节点列表结构

1号到10号是被预约的索引节点，其定义如下所示。

:: include/linux/ext2_fs.h

```
#define EXT2_BAD_INO            1      /*不可用的数据块索引节点*/
#define EXT2_ROOT_INO           2      /*根目录索引节点*/
#define EXT2_BOOT_LOADER_INO    5      /*启动加载项索引节点*/
#define EXT2_UNDEL_DIR_INO      6      /*不可删除的目录索引节点*/
#define EXT2_GOOD_OLD_FIRST_INO 11     /*未预约的第一个索引节点*/
```

索引节点1号将无法保存数据的数据块集合在一起进行管理。若坏扇区（bad sector）引发坏块（bad block），则添加到索引节点1号管理的数据块，即i_block[]数组。

索引节点2号是根目录的索引节点，文件系统会挂载到该根目录。此外，它还是访问所有文件时能的基准索引节点。假设访问/home/code/src/linux-2.6.30.4/README文件时，先访问索引节点2号，之后搜索各目录以查找对应于文件的索引节点（C.4.4节）。

索引节点5号和6号存在于源代码中但不使用。从11号起可用，索引节点11号实际用作/lost+found目录的索引节点。①

C.4.4　目录、文件、索引节点列表、数据块之间的关系

索引节点列表保存着索引节点，数据块保存着实际文件数据。下面了解一下一般使用的文件和目录与它们有怎样的关系。

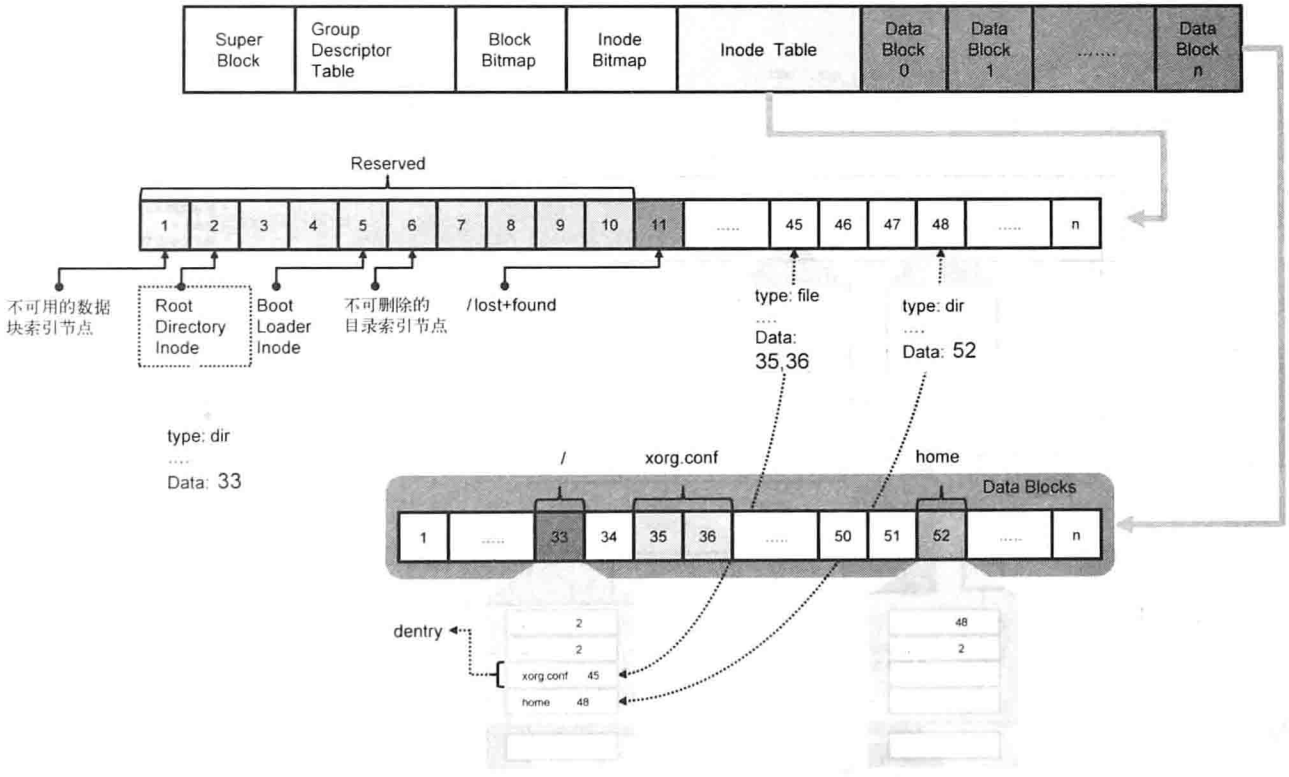
假设根目录（/）中有home目录和xorg.conf文件，通过下列代码了解如何访问xorg.conf文件。

```
code@kernel:/$ ls -al
total 132
drwxr-xr-x  22 root root  4096 2012-07-01 14:24 .
drwxr-xr-x  22 root root  4096 2012-07-01 14:24 ..
drwxr-xr-x   6 root root  4096 2012-07-29 13:11 home
```

① /lost+found目录保存异常中断的目录或文件。

```
-rw-r--r--   1 root root  2699 2012-06-25 17:19 xorg.conf
...
code@kernel:/$
```

以根目录为起点逐一访问所有文件和目录。由于目录也是文件，因此通过索引节点管理文件。根目录信息保存在索引节点2中。图C-7展示了从根目录的索引节点访问xorg.conf文件和home目录的过程。



图C-7　通过索引节点访问文件

由图C-7可知，只有访问根目录的索引节点才能先访问xorg.conf文件。索引节点2号是根目录的索引节点，读取它即可知其为目录。由索引节点2号可知该目录的数据保存在数据块33号。

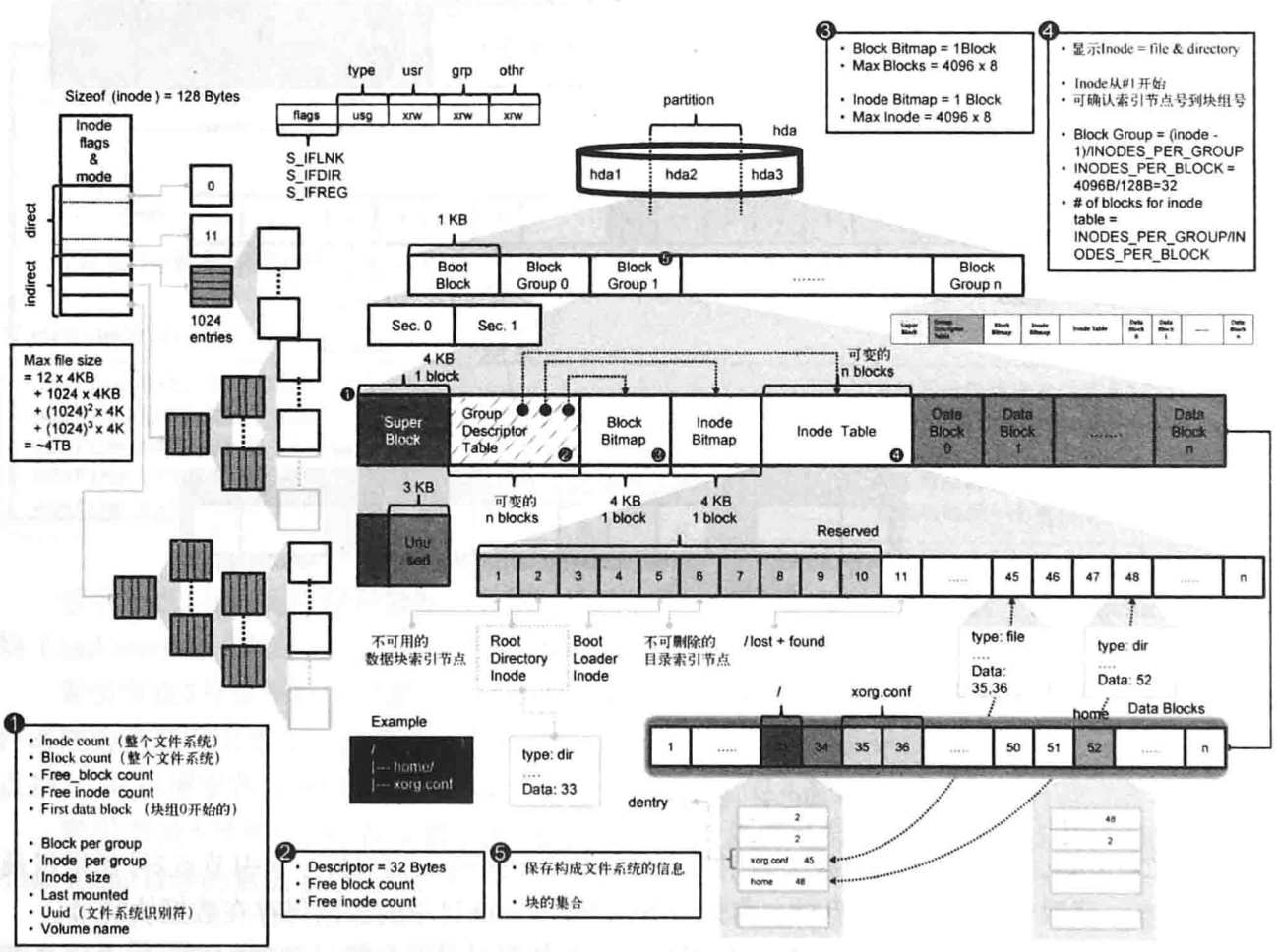
由于索引节点2号显示目录，因此，数据块33号由具有目录信息的目录项构成。目录项具有目录和文件的索引节点号，其定义如下所示。

:: include/linux/ext2_fs.h

```
#define EXT2_NAME_LEN 255
struct ext2_dir_entry {
    __le32  inode;          /*索引节点号*/
    __le16  rec_len;        /*目录项长度*/
    __le16  name_len;       /*名称长度*/
    char    name[EXT2_NAME_LEN];    /*名称*/
};
```

通过目录项中记录的索引节点号访问索引节点列表。要访问的xorg.conf文件的目录项是图C-7的45号索引节点。接着访问索引节点列表中的索引节点45号，它的类型是文件，文件数据存在于数据块35号和36号。可用相同方法访问/home目录。需要注意，索引节点显示目录时，实际数据块由目录项组成。

ext2文件系统的一系列构造综合如图C-8所示。



图C-8　ext2文件系统整体结构

本章简单介绍了ext2文件系统。ext2的升级版有ext3、ext4等。据说Fedora发布版预计包含btrfs（btree file system）。可以看出，Linux支持ext2文件系统以及多种文件系统。分析自己感兴趣的文件系统的源代码是最有效的方法。希望本章对ext2文件系统的简介能为大家提供帮助。

附录 D

Linux线程模型

本章讲解Linux中使用的线程模式。分析Linux源代码时会经常接触到进程、任务、线程等术语，因此，了解线程模式前先进行简单介绍。 进程、任务、线程既相似又有微妙差异。讲解操作系统时，这些术语的定义十分多样。本书一直分析的是Linux内核，所以从Linux观点进行说明。

首先，进程和任务经常混用，进程用于在用户空间指向正在执行的程序示例，任务是内核管理的进程。用户程序在内核中由任务实现，在用户空间称为进程。Linux中的线程就是进程，和进程不同的是，与生成线程的进程共享代码。因此，线程又称为轻量级进程（lightweighted process）。但线程在内核中又是任务，无论用户程序是进程还是线程，均由task_struct结构体管理。

D.1 线程模型

实现线程时需要某项基准，此处的基准就是模型。线程模型共分为3种，如表D-1所示。

表D-1 线程模型（Thread Model）[①]

模 型	说 明
1:1	• 内核级线程（kernel-level threading）
	• 将用户生成的线程与内核中调度的实体1:1映射
	• 简化线程实现
	• 在Solaris、NetBSD、FreeBSD中使用1:1模型
N:1	• 用户级线程（user-level threading）
	• 所有应用程序级线程映射为1个内核级调度实体
	• 内核完全不具有应用程序线程信息
	• 可快速执行上下文切换，但无法获得多线程处理器（multi-threaded processor）或多处理器计算机（multi-process computer）等硬件上的优点
	• 无法同时调度多个线程
N:M	• 混合线程（hybrid threading）
	• 将N个应用程序线程映射到可执行M个调度的内核级线程
	• 线程库（thread library）负责调度用户线程，不执行系统调用，故可快速执行上下文切换

① 参考文献：http://en.wikipedia.org/wiki/Thread_(computer_science)

（续）

模　型	说　明
N:M	• 难以实现，且优先顺序可能变更 • 需要管理用户空间调度和内核空间调度

D.2　实现Linux线程

由于具有3种线程模型，所以实现方法也有3种。N:1模型用于GNU便携式线程（Portable Thread），1:1模型实现于LinuxThread和之后的NPTL（Native POSIX Thread Library，本地POSIX线程库），N:M模型实现于NGPT（Next Generation POSIX Threads，下一代POSIX线程）。

实现N:1模型的是GNU Pth（Portable Thread），它是基于POSIX/ANSI-C的用户空间线程库。对各线程的调度均通过GNU Pth库实现，内核无法认知用户空间线程。该特性使GNU Pth中无法活用SMP。

Linux初期使用基于1:1模型的LinuxThreads，由于LinuxThreads存在几个问题，所以启动NPTL项目和NGPT项目进行解决。NPTL基于1:1模型实现，NGPT基于N:M模型实现。

基于1:1模型的LinuxThreads使用克隆系统调用（clone system call）生成新进程，生成的子进程共享父进程的地址空间。虽然LinuxThreads项目在用户区域使用这些系统调用以支持线程，但只实现了部分POSIX线程。

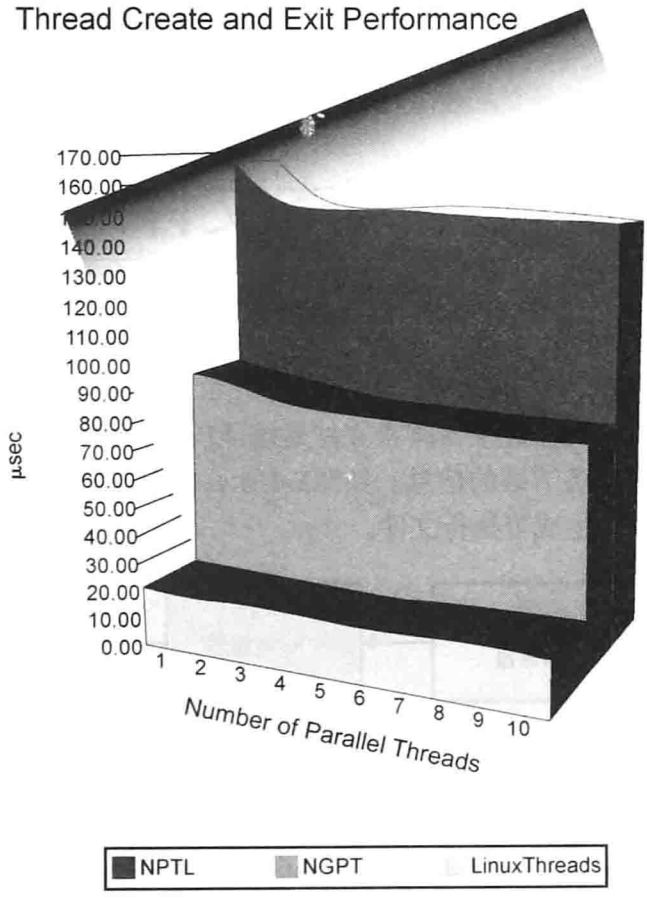
LinuxThreads的最大问题在于，由于各线程具有自己的固定编号（district identifier），因此会发生信号处理（signal handling）问题。LinuxThreads中使用SIGUSR1和SIGUSR2管理线程，这也意味着程序中无法使用这2个信号。

Linux内核2.6版以前未能支持真正的线程，而是利用克隆系统调用支持的。因此，该时期的pthread都形成于用户空间，内部包含信号处理、调度、进程间同步的问题。

NPTL是“红帽（RedHat）“主持的项目，用于解决基于1:1模型实现的LinuxThreads的问题，它也基于1:1模型。NPTL遵循POSIX规定，因此以进程为单位处理信号。getpid()函数在所有线程中返回相同PID。

例如，发出信号SIGSTOP时，LinuxThreads中只有接收该信号的线程停止，但NPTL中所有进程均会停止。这种特性可以在基于NPTL的应用中强化GDB等调试支持。

IBM主持的NGPT项目也能解决LinuxThreads模型缺点。NGPT基于N:M模型，2003年已停止开发。因此，虽然NPTL和NGPT都能解决LinuxThreads的问题，但Linux目前使用NPTL。图D-1展现了“红帽”测定的3种线程实现方法性能。



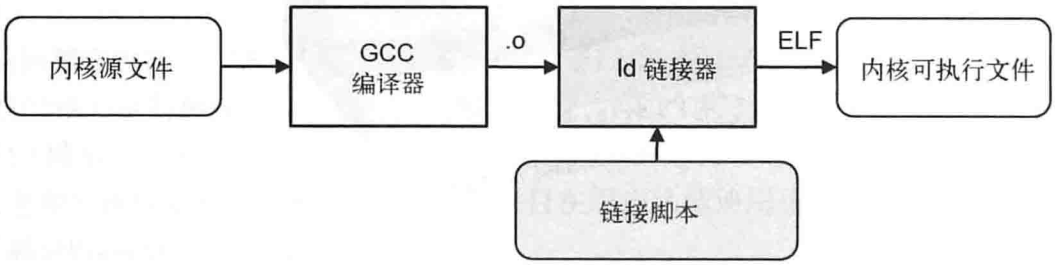
图D-1　LinuxThreads、NPTL、NGPT性能比较

※参考文献：http://people.redhat.com/drepper/nptl-design.pdf

附录 E

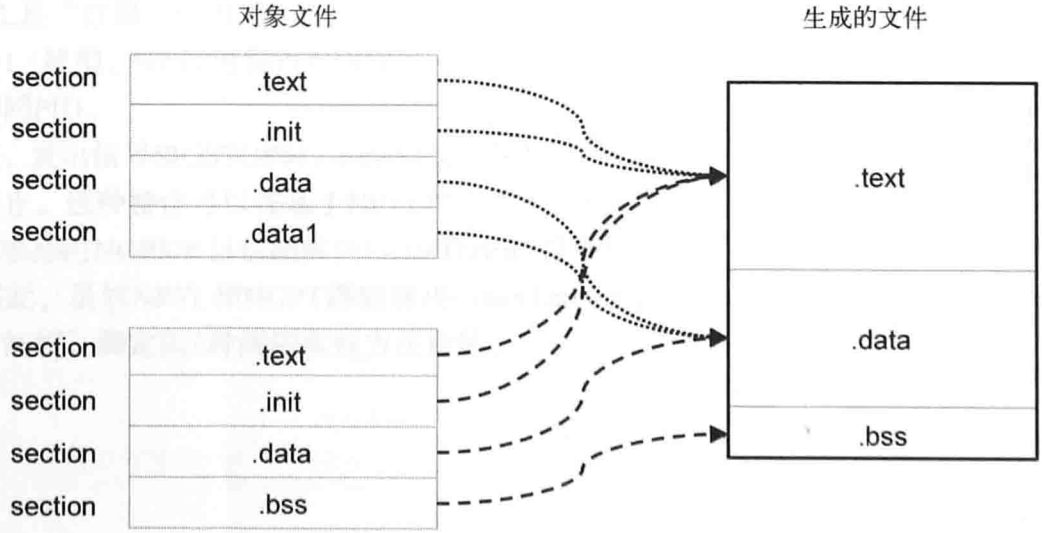
链接器脚本文件结构

本章讲解链接器脚本文件结构。链接器脚本文件是编译Linux源代码并生成内核映像时由链接器浏览的文件，具有如何配置各节区的信息。如图E-1所示，编译源文件并生成对象文件后，链接器将对象文件整合在一起，生成可执行文件。



图E-1　编译过程中的链接器和链接器脚本文件

内核源文件由许多文件组成，文件被编译后会生成各自的对象文件。对象文件由具有特殊属性的节区组成，链接器以脚本文件中记述的内容为基础，对这些具有相同属性的节区进行构建。图E-2表示对象文件的节区和通过链接器生成的最终结果之间的关系。



图E-2　节区构成的对象文件和链接器生成的文件

E.1　ARM链接器脚本文件结构

Linux的链接器脚本文件在ARM中利用vmlinux.lds.S文件和vmlinux.lds.h文件构建，Linux的链接器脚本文件vmlinux.lds.S的定义如下。

:: arch/arm/kernel/vmlinux.lds.S

```
#include <asm-generic/vmlinux.lds.h>

...

OUTPUT_ARCH(arm)
ENTRY(stext)

#ifndef __ARMEB__
jiffies = jiffies_64;
#else
jiffies = jiffies_64 + 4;
#endif

SECTIONS
{
#ifdef CONFIG_XIP_KERNEL
    . = XIP_VIRT_ADDR(CONFIG_XIP_PHYS_ADDR);
#else
    . = PAGE_OFFSET + TEXT_OFFSET;
#endif
    .text.head : {
        _stext = .;
        _sinittext = .;
        *(.text.head)
    }

    .init : {                       /* Init code and data */                ❶
        INIT_TEXT
        _einittext = .;
        __proc_info_begin = .;
            *(.proc.info.init)
    ...

    }

    ...

    .text : {                   /* Real text segment */                     ❷
        _text = .;              /* Text and read-only data */
            __exception_text_start = .;
            *(.exception.text)
            __exception_text_end = .;

    ...

    RODATA                                                                  ❸
    _etext = .;                     /* End of text and rodata section */
```

```
    ...

    .data: AT(__data_loc) {                                          ❹
        _data = .;                  /* address in memory */

    ...

        _edata = .;
    }
    _edata_loc = __data_loc + SIZEOF(.data);

    .bss : {                                                         ❺
        __bss_start = .; /* BSS                          */
        *(.bss)
        *(COMMON)
        _end = .;
    }

    ...
}

    ...
```

代码段❶指定init节区在内核映像中的位置，代码段❷指定要放置data节区的位置，代码段❸指定放置读取专用区域rodata节区的位置，代码段❹和代码段❺指定放置data节区和bss节区的位置。指定rodata节区位置的RODATA在vmlinux.lds.h文件中定义如下。

:: asm-generic/vmlinux.lds.h

```
...

#define RO_DATA(align)                                                \
    . = ALIGN((align));                                               \
    .rodata           : AT(ADDR(.rodata) - LOAD_OFFSET) {             \
        VMLINUX_SYMBOL(__start_rodata) = .;                           \
        *(.rodata) *(.rodata.*)                                       \
        *(__vermagic)         /* Kernel version magic */              \
        *(__markers_strings)  /* Markers: strings */                  \
        *(__tracepoints_strings)/* Tracepoints: strings */            \
    }                                                                 \
                                                                      \
    .rodata1          : AT(ADDR(.rodata1) - LOAD_OFFSET) {            \
        *(.rodata1)                                                   \
    }                                                                 \
                                                                      \
    BUG_TABLE                                                         \
                                                                      \
    /* PCI quirks */                                                  \
    .pci_fixup        : AT(ADDR(.pci_fixup) - LOAD_OFFSET) {          \
        VMLINUX_SYMBOL(__start_pci_fixups_early) = .;                 \
```

```
    ...

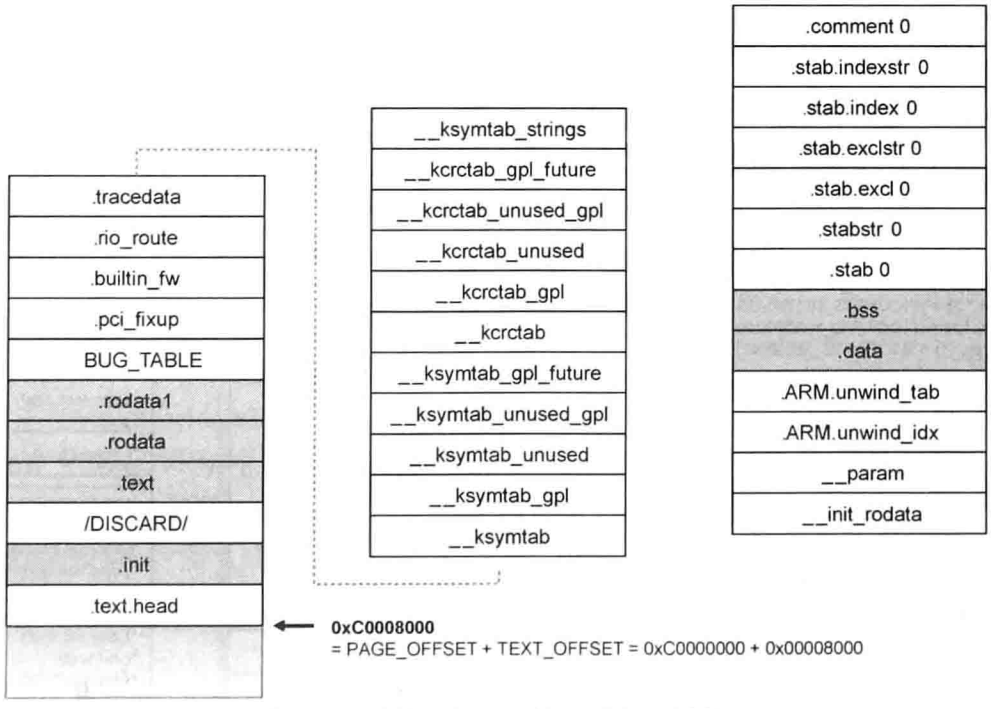
    }                                                                    \
    . = ALIGN((align));

#define RODATA RO_DATA(4096)

    ...
```

E.2　链接器脚本结构图

上一节讲解了vmlinux.lds.S文件和vmlinux.lds.h的大致结构。本节分析这2个文件，构建结构图以了解Linux的链接器脚本实际是如何构成的。结构图中的节区都是分析vmlinux.lds.S文件和vmlinux.lds.h文件获得的结果，由此可知内核映像实际由哪些节区组成。



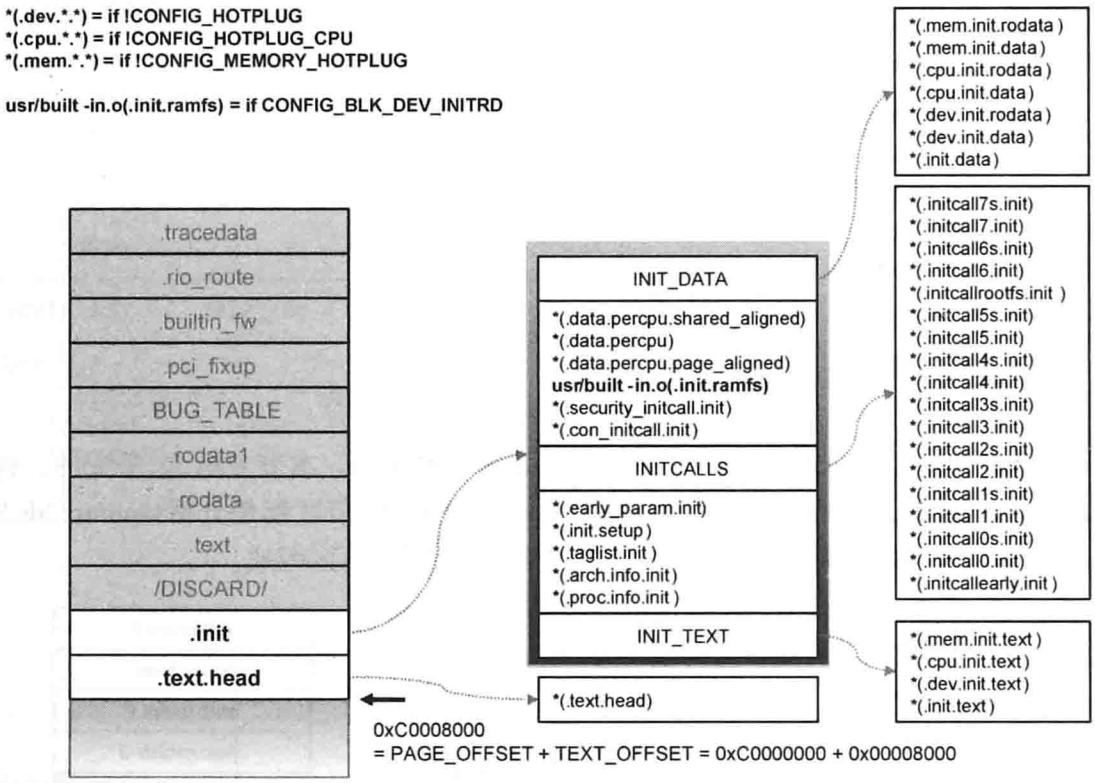
图E-3　链接器脚本文件内的节区结构

图E-3是以链接器脚本文件vmlinux.lds.S文件为基准重新构建的节区位置。

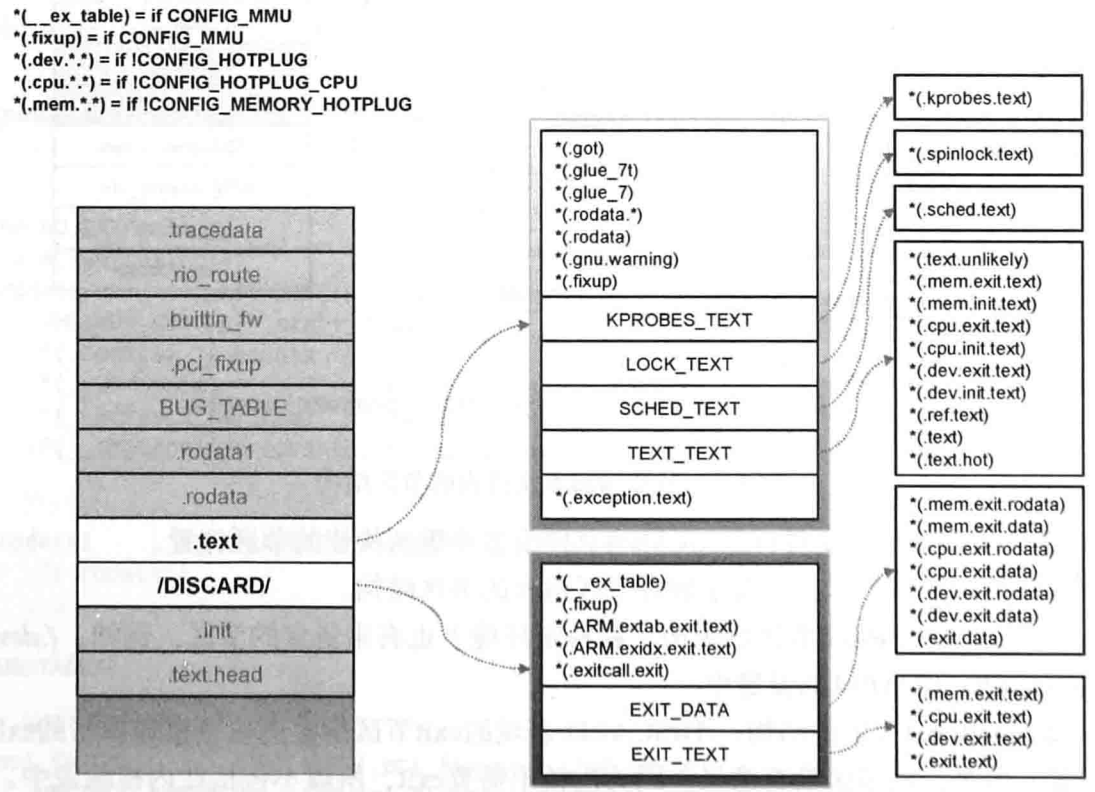
顶级节区包含下级节区，下面了解各节区包含的节区结构。

图E-4是.text_.head和init节区的结构。在特定环境下也有未包含的节区。例如，*(.dev..*)*节区不包括在CONFIG_HOTPLUG设置中。

图E-5是exit和text节区的结构。/DISCARD/表现的exit节区指定内核结束时执行的exit代码和数据的位置。但是，该节区没有意义。因为内核不需要exit，所以不包括在内核映像中。也就是说，该节区被DISCARD了。

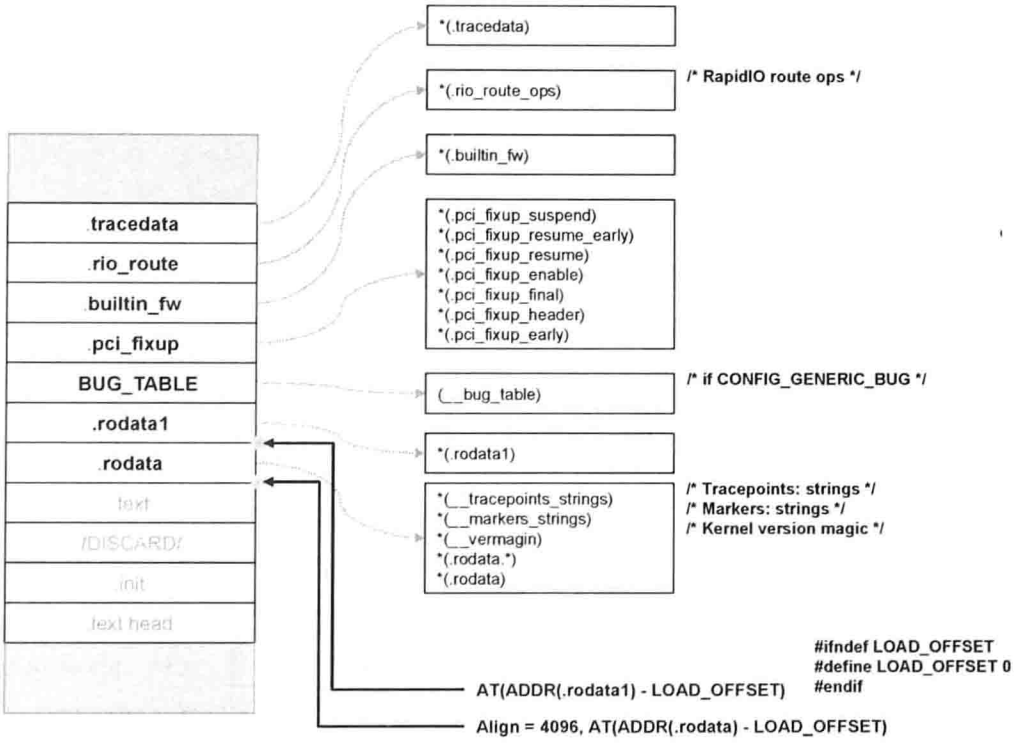


图E-4　text_head和init节区



图E-5　exit和text节区

在text节区中放置内核代码。这2个节区均包含着只包含于特定设置的代码。例如，CONFIG_MMU环境中包含*(__ex_table)和*(.fixup)节区属性指定的代码。



图E-6　RODATA节区

图E-6表示读取专用数据的节区结构。链接器脚本文件vmlinux.lds.S中标记为RODATA，RODATA记述在vmlinux.lds.h文件中。读取专用数据用4096字节排列并包含于内核映像。

该读取专用节区保存注册为内核版本magic、Markers、Tracepoints的函数名。Marker和Tracepoint是调用内核函数时为获得统计信息而插入内核代码的代码，调用指定为Marker或Tracepoint的函数时执行插入代码的代码。DEFINE_MARKER和DEFINE_TRACE宏分别指定内核中用Marker和Tracepoint监控的函数。DEFINE_MARKER和DEFINE_TRACE的定义如下所示。

:: nclude/linux/marker.h

```
#define _DEFINE_MARKER(name, tp_name_str, tp_cb, format)                \
        static const char __mstrtab_##name[]                            \
        __attribute__((section("__markers_strings")))                   \
        = #name "\0" format;                                            \
        static struct marker __mark_##name                              \
        __attribute__((section("__markers"), aligned(8))) =             \
        { __mstrtab_##name, &__mstrtab_##name[sizeof(#name)],           \
          0, 0, marker_probe_cb, { __mark_empty_function, NULL},        \
          NULL, tp_name_str, tp_cb }

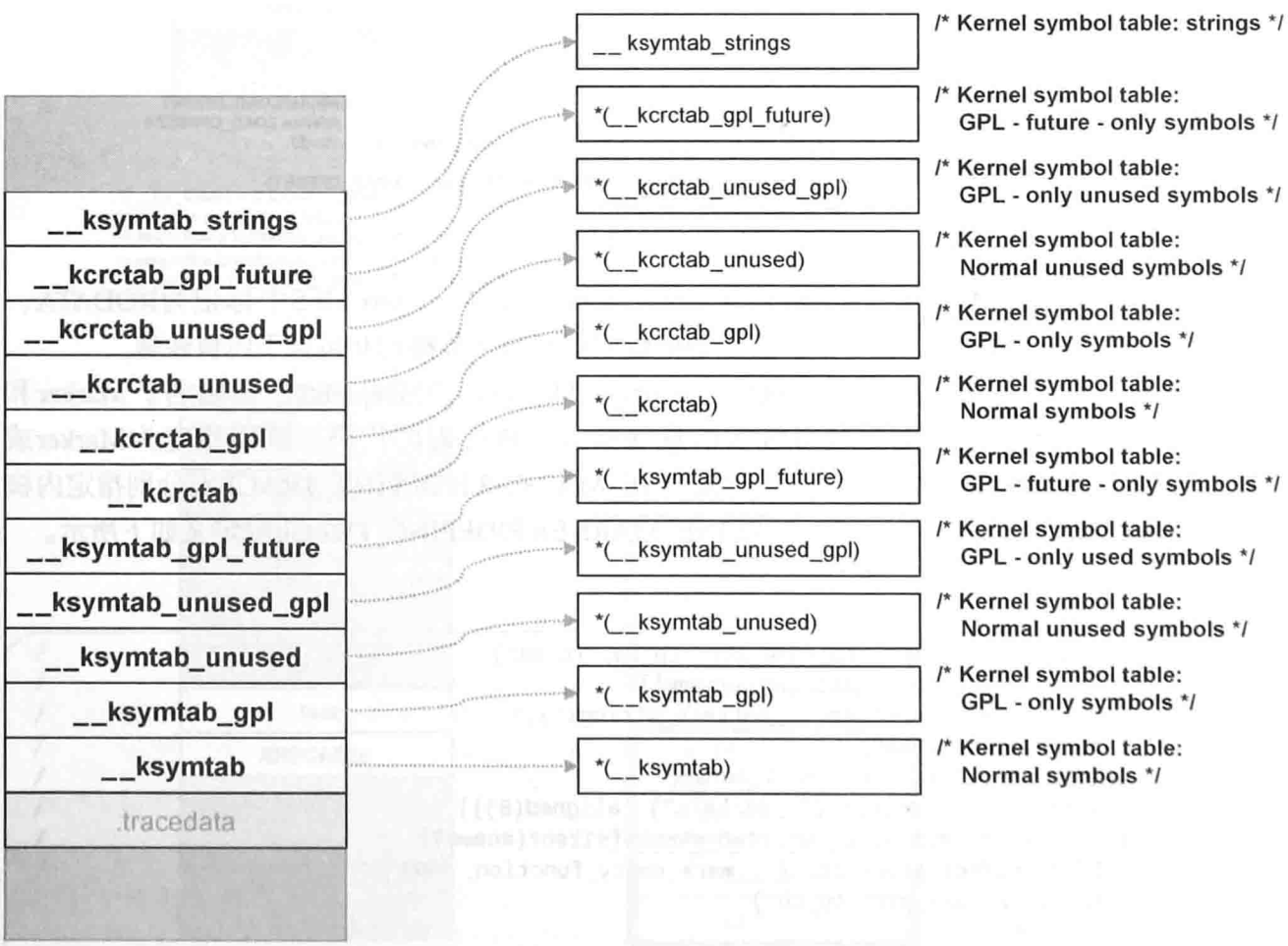
#define DEFINE_MARKER(name, format)                                     \
       _DEFINE_MARKER(name, NULL, NULL, format)
```

:: include/linux/tracepoint.h

```
#define DEFINE_TRACE(name)                                                    \
    static const char __tpstrtab_##name[]                                     \
    __attribute__((section("__tracepoints_strings"))) = #name;                \
    struct tracepoint __tracepoint_##name                                     \
    __attribute__((section("__tracepoints"), aligned(32))) =                  \
        { __tpstrtab_##name, 0, NULL }
```

通过DEFINE_TRACE宏注册的函数如下所示。

```
DEFINE_TRACE(block_bio_bounce);
DEFINE_TRACE(kmalloc);
DEFINE_TRACE(kmem_cache_alloc);
DEFINE_TRACE(sched_process_fork);
DEFINE_TRACE(workqueue_insertion);
DEFINE_TRACE(softirq_entry);
DEFINE_TRACE(irq_handler_entry);
DEFINE_TRACE(block_rq_insert);
```



图E-7　RODATA：ksymtab节区

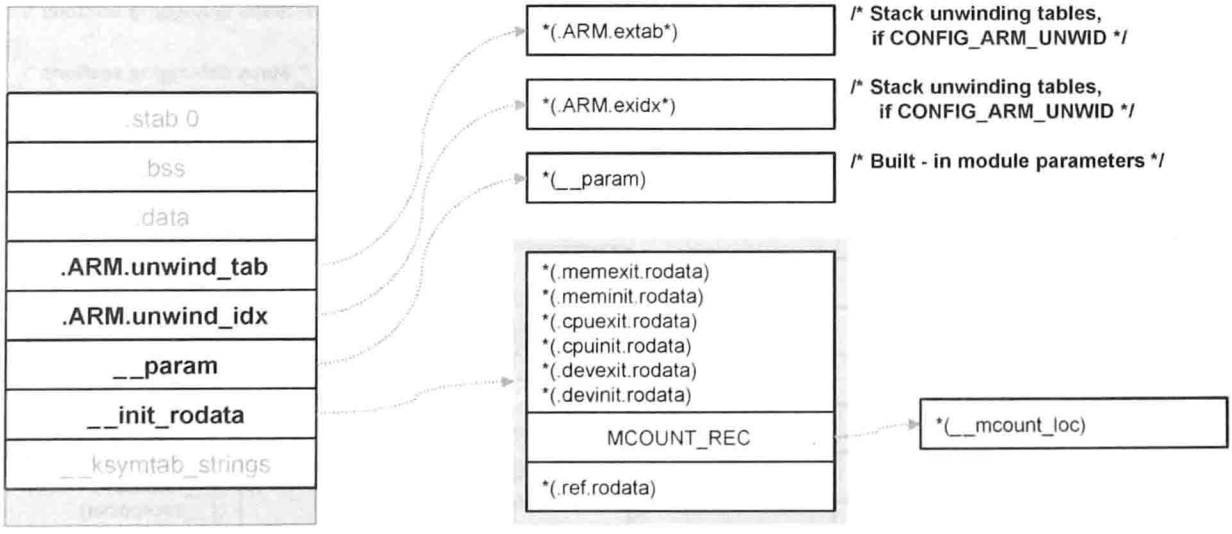
图E-7表示内核中使用的符号列表所在节区结构。内核符号分为几种进行管理，例如，一般符号（normal symbol）列表，管理声明为EXPORT_SYMBOL_GPL()的符号的GPL ONLY符号列表。①

各内核符号被分成列表，可通过/proc/kallsyms进行搜索。下面是__ksymtab列表中的内核符号。

```
code@kernel:~$ cat /proc/kallsym | grep __ksymtab
ffffffff8174fcb0 r __ksymtab_init_task
ffffffff8174fcb0 R __start__ _ksymtab
ffffffff8174fcc0 r __ksymtab_nr_cpu_ids
                    ...
code@kernel:~$
```

图E-8的init_rodata节区将初始化时使用的读取专用数据集合在一起，param节区将built-in模块参数（built-in module parameter）集合在一起，还有stack unwinding列表节区。

```
*(__mcount_loc) = if CONFIG_FTRACE_MCOUNT_RECORD
*(.dev.*.*) = if !CONFIG_HOTPLUG
*(.cpu.*.*) = if !CONFIG_HOTPLUG_CPU
*(.mem.*.*) = if !CONFIG_MEMORY_HOTPLUG
```



图E-8　__init_rodata、__param、.ARM.unwind节区

__init_rodata节区中保存的数据在内核中集合已定义的所有数据。

```
#define __cpuexitconst    __section(.cpuexit.rodata)  /* ./include/linux/init.h */
#define __cpuinitconst    __section(.cpuinit.rodata)  /* ./include/linux/init.h */
#define __memexitconst    __section(.memexit.rodata)  /* ./include/linux/init.h */
#define __meminitconst    __section(.meminit.rodata)  /* ./include/linux/init.h */
```

① 内核中利用EXPORT_SYMBOL、EXPORT_SYMBOL_GPL使符号能够在其他模块中使用，主要用于制作设备驱动模块。尤其是通过EXPORT_SYMBOL_GPL导入（export）的符号，在其他模块中浏览这些符号时，相应模块必须遵循GPL许可。

```
#define __devexitconst    __section(.devexit.rodata)  /* ./include/linux/init.h */
#define __devinitconst    __section(.devinit.rodata)  /* ./include/linux/init.h */
```

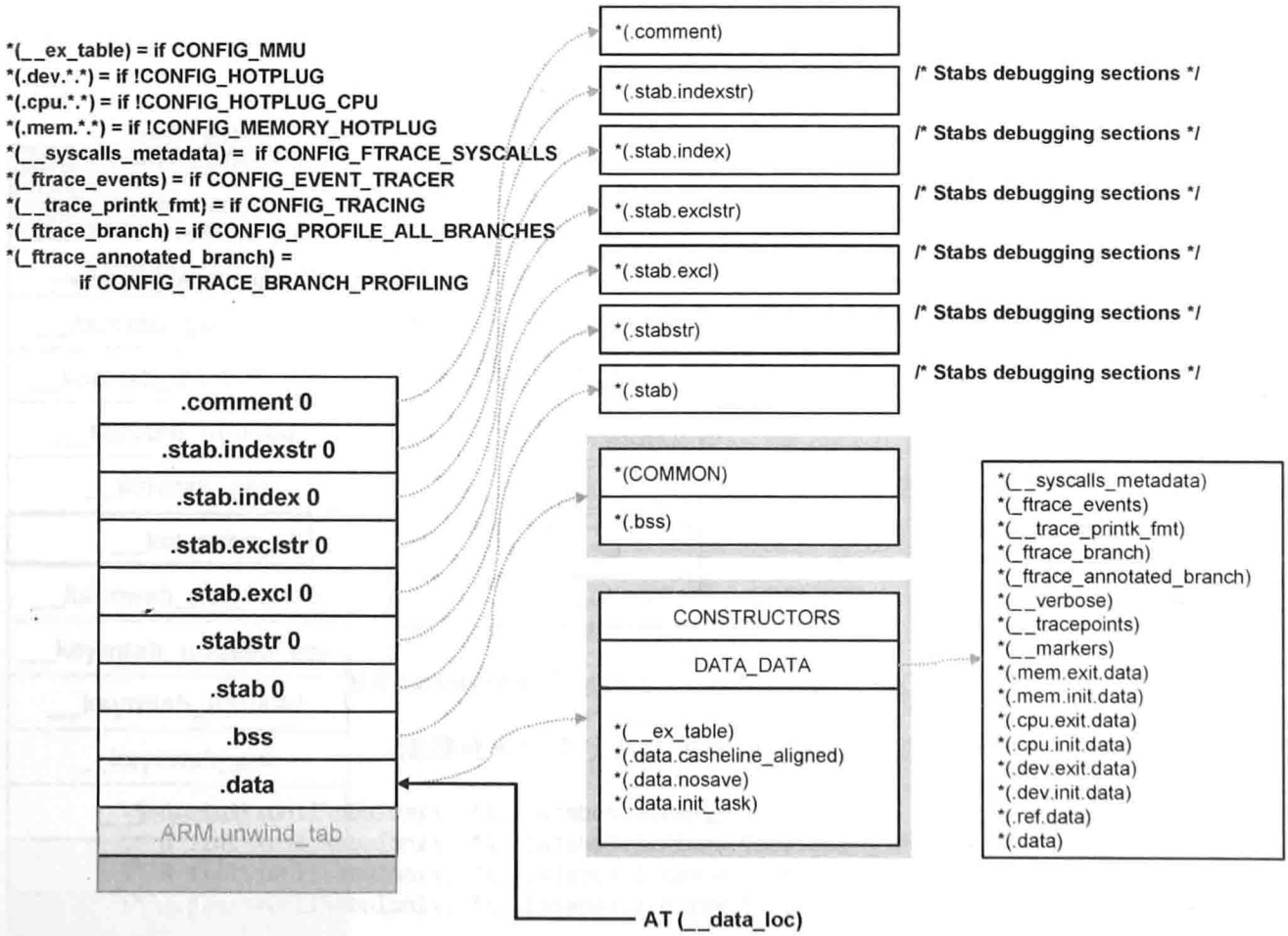
图E-9表示数据节区、未初始化的数据节区bss节区、stabs调试节区的结构。将8092字节排列的.data.init_task节区置于数据节区，该节区中会同时放置init任务的thread_info和内核栈（参考图8-7）。内核中指定.data.init_task节区之处如下所示。

:: arch/arm/kernel/init_task.c

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

struct task_struct init_task = INIT_TASK(init_task);

EXPORT_SYMBOL(init_task);
```



图E-9　data、bss、stab调试节区

数据节区中会放置排列缓存线的数据、异常指令表（exception fixup table）等。

stabs调试节区为调试器提供内核信息，stabs是symbol tables strings（符号列表字符串）的缩写。调试信息会保存到.stab和.stabstr节区，包括线号、源文件的pass、变量及类型声明的信息。

※参考文献：Introduction to the DWARF Debugging Format（http://www.dwarfstd.org/doc/ Debugging%20using%20DWARF.pdf）

后　记

卢瑞燦

Linux内核是凝聚了众多工程师共同努力和贡献的宝贵结晶。人类何曾有过像Linux这样由大范围同业协作成就的伟大财富呢？虽然我们分析并学习了Linux内核源代码，但学到更多的是Linux本身具有的精神内涵。我在执笔过程中感受到的是接踵而来的喜悦和冲动，但现在写后记却突然感到些许遗憾：应该有更加深入的分析、更加细致的说明，即使没有作家的那种华丽文笔，也应该突出重点、简洁明了。反过来想，正如编写团队在前言当中提到的那样，这些部分说不定将成为读者们要补充的内容。希望广大读者能够完成我们未竟的部分。最后，感谢为本书面世而努力的全体作者、鼓励我们出书的Nusco白昌佑代表、不嫌弃原稿语句不通而使本书能够顺利出版的Jpub张成斗（音译）室长，以及始终支持只能单休的我那亲爱的妻子、小霞、小松。

尹锡训

我本以为内核分析是一项艰苦的劳动，但和志趣相投的同事们一起工作，就变成了一场十分快乐的游戏。令人惊讶的是，我竟然坚持学习了如此之久，并最终将自己的名字变成铅字。如果让平时胆小、处处小心的我独立完成，我能做到吗？这是一项只有团结才能完成的工作，我将全部荣耀献给给予我多重灵感的所有同事。

我希望阅读本书的读者朋友能找到心有灵犀的同事，以教材为友，一起研究内核。因为我体会到，与一个人埋头苦读相比，大家共同讨论学习会事半功倍。就像我之前连vi都写不好，可现在不也出了书嘛。

朋友们！我现在有勇气能做出更好的成绩！

姜镇成

3年前开始的旅程终于结束了，或许可以说这又是新旅程的开始。回想走过的这段路，好像一场旅行。我们乘着vi这趟车，每周末支付“一天”的费用，在与伙伴们共同抵达的终点站看到这本书的那一瞬间，内心深处的激动令我无法入睡。我希望满载着方法和努力的这本书能够在他人的脑海中呼吸，也能够为被Linux内核困住的人们带来一道希望的曙光。最后，由衷感谢我的导师白昌佑、给予我帮助的JCDSOFT郑俊浩（音译）先生、权洪才（音译）先生。

宋元僔

3年前近30人聚在一起，就为了学习Linux内核。我们每周都要挤出宝贵时间，利用周六的8个小时边看内核代码边讨论。不知不觉2年过去了，只留下了现在的几位作者。能够和这些坚韧又充满热情的队员一起整理并推出1年的成果，我心里无比高兴。在白昌佑老师的鼓励下决定出书时，我还有些恍惚，不过起初确实充满了期待与梦想。可随着时间的流逝，我越发感到自己的不足，并为之深感羞愧。

我边写后记边回想当初，能够想到的都是些遗憾。我认为，对于专门学习计算机工程的人来讲，内核是一项值得学习的软件。即使和自己所学的领域没有紧密联系，可我确信它总会以某种方式提供极大的帮助。希望通过这份经历感受到的各种遗憾能成为以后继续学习Linux内核的强大动力。无论如何，我得感谢坚韧而充满热情的所有团队成员。用不知疲惫的热情成为集体精神支柱的卢瑞烘博士、用工程师气质创造不知疲倦的学习氛围的李闉载大哥、用特有的诚实和沉默增加责任感和意志的林润斋大哥、能够作为团队先锋察觉被其他成员忽略的部分并积极推进的尹锡训以及始终充满斗志的姜镇成，我感谢和所有成员的缘分，能够与大家共度这3年，我感到非常幸福。最后，感谢父母一直以来对我的帮助与支持，是他们让我能够在良好的环境下学习、工作。

李闉载

首先，本书的出版得益于全体成员和白昌佑老师的帮助，以及照顾我们、使我们能够在公司里继续学习的所有人。我仅仅是一个知道“ls-al”命令就喜欢上Linux的年轻人，能够遇到一些优秀的成员并共同学习，甚至出版书籍，这令我自己都难以置信。

其实，我们写的书只是在Linux内核的大海里将无数个前人努力获得的知识海豚捆绑在一起而已，但对于真正想要跳入这个大海的、或需要这方面知识的人来说，我希望这本书能够给他们带来帮助。并且希望这本书能够超越内核的范围，为各位的生活尽绵薄之力。

我在大家的帮助下完成了这本书，在此想对父母以及终有一天会出现的女友说声“谢谢”！

林润斋

分析Linux内核代码带给我们的不仅仅是Linux的相关知识，还包括分析代码的能力、通过观察内核代码中适用的多种数据结构和算法而取得的知识，以及能够将其应用到其他编程的能力等。

我在公司的工作本来就很繁忙，还要耗费时间精力去仔细分析代码并制作成书，的确不容易。由于自身能力欠缺，有多处都未能详细说明，这令我感到遗憾。因此，出书时有一种把孩子丢在河边，或在众人前袒露身体的那种羞愧。虽然本书尚有很多不足之处，但对于想要了解内核或通过分析内核提高实力的人还是很有帮助的。



作者合影

索　引

符号及数字

1模型 530, 531, 532
2-3-4树 386
2-3树 386

A

AAPCS 87, 89
Address Aliasing 172
ALU 24, 25
ARM 12, 21, 35
Atomic Notifier Chains 125
AXI 149
安全密钥 385
安全模式 365

B

BANK 141, 147, 181
Big Endian 170
Block Group 524, 526
Blocking Notifier Chains 125
Boot Block 524
bottom half 132, 299
Branch Target Cache 70
BSD 375
BTC 70, 71, 72
built-in模块参数 541
包装 416, 451
备份 11, 78, 195
编译器属性 477
标签 35, 54, 58
标准错误 448, 473
标准输出 448, 473
别名页帧 199
并发 395, 396
并行计算系统 363

C

C API 78, 84
Cache 58, 60, 80
CISC 21
coloring 379
CR_位 98
参数 56, 68, 78
操作 2, 4, 10
操作处理器 477
策略模式 370
超级块 390, 524
超级用户 434
程序计数器 26, 28, 57
初始启动时 370, 465
处理器结构信息 165, 167
处理器模式 26, 27
处理器信息 73, 75, 77
处理器异常 28
处理器族 23, 24
处理时间 409
磁盘缓存 270

D

D-Cache 62, 70, 73
DSP 22, 23
大端序 231
大内核锁 118, 127, 490
待机队列 456
待机时间 409
待机状态 408, 456, 477
单调时间 302, 306, 434
单内核 2, 3, 4
单一缓存 30, 31

倒数权重 481, 482, 483
底半部 132, 299, 335
地址寄存器 24
调度类 494
调度器 305, 441, 446
调度器类 435, 442, 480
调度器时钟 441
调度器统计信息 440
调度域 448, 449, 450
调度组 449, 450
调试 105, 108, 116
顶半部 336
冻结 335
读/写锁 118
堆 376
堆索引 376
对象 11, 13, 18
对象文件 11, 16, 18
多处理器计算机 94, 530
多级页表 65
多线程处理器 530
多线程系统 439

E

EINVAL 288, 430, 432
ELF 16, 18, 19
exception handling table 138

F

FIFO 65, 246, 445
FIQ Handler 157
flush 61, 69, 75
范围HR 297, 298
非活动列表 390, 391, 392
非抢占 249
分割 45, 65, 330
分配请求 268, 313, 339
分析 2, 8, 9
分页结构 184, 185, 189
分页请求 194, 202
冯·诺依曼体系结构 375
符号 13, 14, 23
符号列表 78, 138, 167
辅助器 137, 151, 152
父节点 146, 386
父进程 241, 247, 386
负载存储 21, 25
负载均衡 248, 407, 458
负载权重 481, 482, 483
覆盖 283

G

GCC 55, 56, 119
GDB 532
GNU 2, 58, 531
GPL 2, 128, 453
GPL ONLY 540
grace period 283
高端内存 133, 178, 182
根目录 397, 527, 528
根文件系统 143, 307, 308, 397
工作队列 336, 394, 448
工作者线程 455, 456, 457
共享对象 90
共享内存段 377
固定编号 531
挂载点 397
关闭缓存 61, 69, 72

H

哈佛体系结构 25, 30, 94
核心 7, 16, 21
黑节点 386
红黑树 187, 244, 298
红节点 386
红帽 532
宏 4, 35, 38
后备存储 402, 414, 420
互斥体 273, 295, 335
坏块 527
坏扇区 527
缓冲缓存 382, 383
缓存 7, 23, 29
缓存别名 199, 369
缓存策略 66, 179, 180
缓存激活 62, 66
缓存交替策略 62, 66
缓存清除 70
缓存清理 70, 175
缓存失败 181

缓存线 71, 100, 173
缓存线弹跳 236
缓存一致性 179, 198, 236
唤醒任务 301, 432, 440
回写 61, 198, 199
混合缓存 30, 31
混合线程 531
活动列表 390, 392, 393
伙伴 224, 226, 312
伙伴系统 223, 226, 316

I

I/O优先顺序 409
I-Cache 62, 93, 506
IDR 365
ISA 21, 22, 183

J

Jazelle 22, 23
JTAG 23
饥饿 409
基数树 365, 403
集合 111, 202, 222
计时控制器 296
计时器 16, 132, 248
计时器初始化 373
计时器中断 290, 296, 299
计数器 81, 187, 324
寄存器组 24, 25, 427
间接映射 522, 523
僵尸 439
接口 4, 29, 89
节点 110, 128, 130
节点号 141, 181, 205
节点列表 524, 526, 529
节点描述符 201, 208, 214
节点顺序 255, 258, 261
节区项 467
解引用 129
进程调度 246, 495
进程管理 3, 448, 450
进程集成 109
进程描述符 118, 119, 290
进程上下文 125, 336, 337
进程识别符 374
禁用中断 55, 155, 279
经过时间 307, 435
静态分析工具 79, 130
就绪队列 241, 243, 246
就绪队列类型 443
局部计数器 395, 396
巨页 269, 270
绝对时间 302, 303

K

可用块 330, 525, 526
可执行文件 533
客户端 439
空白列表 273, 274, 300
空白页 317, 323, 325
空间局部性 309
控制设备 448, 473
块设备 6, 110, 229
宽限期 283, 284
扩展性 7, 246, 283
扩展域 27

L

L1 63, 65, 98
lightweighted process 530
Little Endian 170
Lr 26, 57, 62
冷页 365, 366, 369
李纳斯 2, 5, 11
联机节点 254, 257, 313
链接寄存器 26
链接器 16, 58, 60
链接器脚本 14, 533, 536
链接器脚本文件 533, 536, 538
列表项 180, 309, 518
临界值 414, 423
零页 178, 179, 194, 197

M

MCR 30, 101, 506
MINIX 2
MMU 3, 23, 29
monotonic time 306, 307
M模型 531, 532
密码 521

密钥保留服务 385
密钥环 384, 385
命令行 136, 141, 235
命名空间 374, 405, 429
目录项 33, 92, 209
目录项缓存 309, 388, 390

N

NetBSD 530
netlink 407, 408
Newton-Raphson 339
NGPT 531, 532
node order 255
noinline 426, 472, 473
NPTL 531, 532
NTP 301, 302, 303
内存overcommit 382
内存策略 313, 370, 426
内存分配 3, 136, 145
内存管理 3, 10, 29
内存局部性 254
内存类型列表 179, 210
内存屏障 129, 250, 476
内存热插拔 458, 466, 467
内存碎片 270, 271, 331
内存条 145, 146, 182
内核安装 10, 19
内核版本magic 538
内核初始化 10, 67, 105
内核构建 10, 13, 18
内核级线程 530, 531
内核空间 130, 178, 183
内核恐慌 314, 365, 473
内核模式 280, 427, 433
内核内存 18, 270, 291
内核抢占 249, 435, 494
内核设置 62, 98, 141
内核线程 250, 337, 375
内核线程守护进程 426, 474, 487
内核消息 431
内联汇编 120, 151

O

off slab 339, 349
on slab 350, 351
ONLINE 208, 213, 219
OOM 7, 223, 254

P

page frame number 190
page global directory 184
page middle directory 184
page mobility 254, 268
page swapping 375
per-cpu列表 309, 339, 388
PID 290, 294, 374
PIPT 172, 173
PIVT 172, 174
PMD 92, 180, 190
Portable Thread 531
POSIX 12, 365, 531
POSIX/ANSI-C 531
priority 124, 128, 225
profiling cpu mask map 293
PST 376
PTE 175, 180, 192
平衡度 386
平衡树 386
凭证 377, 378

Q

期满时间 293, 297, 423
启动 2, 8, 10
启动参数 141, 274, 276
启动加载 55, 88, 140
启动加载项 55, 73, 86
启发式 382
迁移类型 330, 331
潜伏期 173, 435
嵌入式系统 13, 18, 89
强制性页回写 414, 415, 419
抢占 159, 249, 254
轻量级进程 530
清除 11, 14, 18
全局计数器 395, 396, 421

R

Raw Notifier Chains 125
raw notifier head 295
RCU 7, 118, 129

RISC 21
RODATA 534, 538, 540
Round Robin 62
热插拔事件 282, 312, 403
热页 365, 366
任务 3, 8, 25
任务统计信息 407
软时间 297, 298
闰秒 303

S

SPSR 26, 27, 160
SRCU 125
SRR 483, 484
SWI 26, 29, 152
散列表slot 291
上下文切换 435, 494, 497
上下文值 161
设备区域映射 189
设备驱动 3, 8, 16
设备文件 389, 401, 524
深度优先搜寻 298
审计补丁 407
审计子系统 407
时间局部性 309
时钟事件 123, 124, 130
实时任务策略 444, 480
数据副本 236, 282
数据块 520, 522, 527
数据输入输出寄存器 24
水印 366, 368, 382
睡眠状态 418, 489
碎片问题 379
索引节点 309, 388, 392
索引节点缓存 309, 391, 392
锁 106, 120, 125
锁运算 395, 396

T

TGID 407, 408
time vector base 293
topology 239, 397, 466
touched command line 235, 236
Tracepoint 538
套接字 4, 114, 377
特权模式 26, 27
同步 61, 94, 118
同步机制 118, 313
脱机 131, 334, 458
拓扑 239, 397, 461

U

Undefined Handler 156
Undefined Instruction 29, 137
unicast netlink套接字 407
unknown IRQ 288, 289
user mode application 458
UTC 303

V

VIC 289, 290
Virtual File System 387

W

wait queue 216, 322
wall time 307
workqueue 337, 452, 539
WRITE BACK 181
write buffer 65, 412, 507
WRITE THROUGH 180, 181
网络 4, 10, 16
微内核 4
未挂载 399
位置计数器 79, 81, 84
文件 4, 10, 14
文件访问 309, 378
文件描述符 388, 435, 448
文件系统 4, 6, 8
无效 116, 152, 507
物理地址 30, 63, 78

X

XIP 12, 102, 183
系统调用 4, 6, 8
线程 102, 132, 240
线程库 531
线程模型 530, 531
线程信息标签 478
线程组 429, 431, 437
相对时间 303

向量表 28, 96, 116
小端序 170, 231
校正LR 156
协处理器 29, 30, 62
协调世界时 303
写缓冲 62, 65, 67
信号 24, 156, 163
信号处理 436, 477, 531
信号处理器 429, 435, 475
虚拟地址 30, 65, 78
虚拟内存 6, 30, 89
虚拟内存盘映像 202
虚拟文件系统 4, 387, 389
序列号 384
序列化 476

Y

延迟任务 299, 336
延迟审计 408, 411
延迟时间 409, 410, 412
延迟信息 408, 409, 411
页 33, 39, 42
页管理位图 219, 318, 322
页缓存 8, 133, 229
页回收 268, 376
页回写 414, 417, 420
页列表 328, 365, 366
页目录 54, 60, 63
页数组 215, 225, 228
页顺序 322, 349, 355
页移动性 254, 268, 270
页帧号 190, 198, 202
一般内存区域 319, 333
一般任务 445, 446, 482
一般文件 399, 521, 522
一致性 94, 149, 172
移动类型 269, 330
异常处理 22, 98, 138
异常处理列表 138
异常列表项 281
异常模式 150, 155
异常向量 137, 151, 154
异常指令表 542
溢出 109, 223, 254
引用计数 322
硬件缓存排列 379
硬件缓存线 378, 385
硬件计时器 157, 161, 304
硬时间 297, 298
用户ID 384, 521
用户程序 530
用户级线程 530
用户空间 4, 130, 133
用户模式 27, 158, 434
用户模式寄存器 162
用户内存 270
优先顺序 28, 162, 246
有效位 71
有效载荷 384, 408
域 5, 21, 36
域访问权限 99
元 25, 94, 108
元数据 382, 525
元信息 320, 321, 322
原子运算 395
源代码树 11, 19, 143

Z

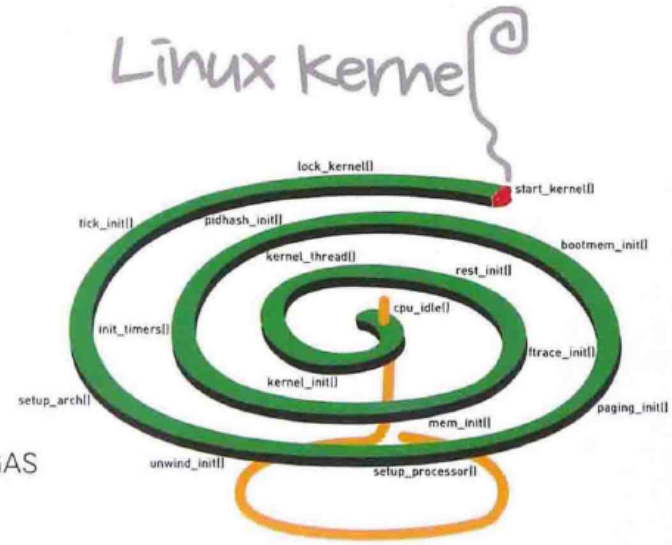
脏页 414, 417, 423
栈空间 137, 150
栈溢出 109
栈展开 138
栈指针 26, 57, 101
正规化 302
帧 57, 199, 202
直接映射 522
指令解码器 23, 24
中断处理器 29, 116, 157
重排序 129, 271, 499
重装 356
周期 21, 156, 246
周期性负载均衡 450
周期性页回写 402, 414, 419
状态寄存器 26, 155, 280
状态域 27
子节点 146, 148, 386
子进程 427, 429, 432
自旋锁 118, 125, 129
组ID 384
组长 437, 439
最佳节点 258, 260

ARM Linux内核源码剖析

Linux内核代码分析全新方法 + 实用指南!

本书主要内容

- 内核源代码构建系统
- ARM处理器结构
- 构建高效分析环境
- 汇编级启动过程
- 内核分析常用API、ARM指令、GAS



本书是多位作者在3年Liunx内核分析经验和庞大资料基础上写成的，收录了其他同类书未曾讲解的内容并进行逐行分析，一扫当前市场中其他理论书带给读者的郁闷。书中详细的代码分析与大量插图能够使读者对Linux内核及ARM获得正确认识，自然而然习得如何有效分析定期发布的Linux内核。

本书适合想从Linux内核启动开始透彻分析全部启动过程的读者。因Linux代码量庞大而束手无策的人、想要了解Linux实际运行过程的人、渴求OS实操理论的人，本书也必将成为他们不可或缺的参考书。

Jpub 제이펍

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/操作系统

人民邮电出版社网址：www.ptpress.com.cn

TURING 1519704 www.ituring.com.cn 来社区 领银子

ISBN 978-7-115-35910-0

9 787115 359100

ISBN 978-7-115-35910-0

定价：99.00元