内存管理

by zenhumay

2012-04-06——2012-05-01

目录

1. 概述

本章中涉及的源码全部来自 linux 内核 2.6.34。

在【start_kernel 之物理内存管理】中介绍了物理内存管理的相关内容，那个是以启动过程为主线讨论物理内存管理的。在本章中，将讨论内存管理的大部分内容

- 物理内存管理
- 进程虚拟内存管理

内存管理是一个供需关系，物理内存管理是供应方，它提供实际的物理内存给进程虚拟地址空间（需求方），内存管理的功能就是实现供需平衡，保证系统高效、稳定的运行。

2. 物理内存管理

由于在之前的章节中，已经片段的讨论过物理内存管理，本节的重点是讨论物理内存管理的数据结构以及整体框架。

2.1 物理内存管理内容

内存管理是内核中最复杂同事也是最重要的一个模块，物理内存管理的实现涵盖了许多领域。

- 内存中的物理内存页的管理
- 分配大块内存的伙伴系统
- 分配小块内存的 slab 分配器
- 高端内存管理（非连续内存块、永久内核映射、临时内核映射）

Linux 内核将处理器的虚拟地址空间划分为两部分，高地址部分分配给内核使用，低地址部分分配给进程用户空间使用。当在两个进程中间切换时，低地址部分的地址映射关系会改变，但虚拟地址空间内核部分的映射总是保持不变。

在 IA-32 系统上，虚拟地址空间的用户进程和内核之间划分的典型比例是 3:1。

内核空间的虚拟地址和物理地址之间的关系是线性映射的，假设内核虚拟地址空间的起始地址为 PAGE_OFFSET（0xC0000000），则内核虚拟地址 kernel_vaddr 和其物理地址 kernel_paddr 对应的关系如下：

kernel_paddr = kernel_vaddr – PAGE_OFFSET

由于内核的虚拟地址空间范围是[0xC0000000,0xFFFFFFFF]，如果使用线性关系的映射，内核可以访问的物理地址空间范围为[0x00000000,0x40000000]也就是 0-1G 的物理地址空间。很显然，CPU 理论上支持的物理地址空间为 0-4G，如果内核仅仅使用线性地址的关系来映射物理内存的话，则内核无法使用物理内存 1G 以上的空间。内核借助于高端内存管理的方法来是内核可以访问所以的物理地址空间。

在 IA-32 体系中，内核可以直接使用的物理地址空间为 0-896M，也就是说，内核虚拟地址空间[3G，3G+896M]的部分使用线性地址映射，而[3G+896M，4G]的地址空间则使用高端内存管理的方式。

用两种管理物理内存的方法：

UMA（一致内存访问）：将可用内存以连续的方式组织起来。

NUMA（非一致内存访问）：系统的各个 CPU 都自己的本地内存，CPU 可以快速的访问本地内存。各个处理器之间通过总线连接起来，以支持对其他 CPU 的本地内存范围。

2.2 NUMA模型中的内存组织

LINUX 内核中使用三级的内存管理方式：节点-区域-页面。

内存划分为节点。每个节点关联到系统中的一个处理器，在内核中使用 pg_data_t 的一个实例表示节点。

每个节点又划分为内存区，是内存的进一步细分。

内核中使用下列的枚举变量来划分内存区域。

```
191 enum zone_type {
 192 #ifdef CONFIG_ZONE_DMA
 193         /*
 194          * ZONE_DMA is used when there are devices that are not able
 195          * to do DMA to all of addressable memory (ZONE_NORMAL).
Then we
```

```
196          * carve out the portion of memory that is needed for these devices.
197          * The range is arch specific.
198          *
199          * Some examples
200          *
201          * Architecture         Limit
202          * ---------------------------
203          * parisc, ia64, sparc    <4G
204          * s390                 <2G
205          * arm                  Various
206          * alpha                Unlimited or 0-16MB.
207          *
208          * i386, x86_64 and multiple other arches
209          *                      <16M.
210          */
211         ZONE_DMA,
212 #endif
213 #ifdef CONFIG_ZONE_DMA32
214         /*
215          * x86_64 needs two ZONE_DMAs because it supports devices that
are
216          * only able to do DMA to the lower 16M but also 32 bit devices that
217          * can only do DMA areas below 4G.
218          */
219         ZONE_DMA32,
220 #endif
221         /*
222          * Normal addressable memory is in ZONE_NORMAL. DMA
operations can be
223          * performed on pages in ZONE_NORMAL if the DMA devices
support
224          * transfers to all addressable memory.
```

```
225          */
226         ZONE_NORMAL,
227 #ifdef CONFIG_HIGHMEM
228         /*
229          * A memory area that is only addressable by the kernel through
230          * mapping portions into its own address space. This is for example
231         * used by i386 to allow the kernel to address the memory beyond
232          * 900MB. The kernel will set up special mappings (page
233          * table entries on i386) for each page that the kernel needs to
234          * access.
235          */
236         ZONE_HIGHMEM,
237 #endif
238         ZONE_MOVABLE,
239         __MAX_NR_ZONES
240 };
```

ZONE_DMA 标记适合 DMA 的内存域。该区域的长度依赖于处理器类型。在 IA-32 计算机上，一般的限制是 16MiB，这是古老的 ISA 设备强加的边界。但更现代的计算机也可能受这一限制的影响。

ZONE_DMA32 标记了使用 32 位地址字可寻址、适合 DAM 的内存域。显然，只有在 64 位系统上，两种 DAM 内存域才有差别。在 32 位计算机上，本内存与是空的，在 AMD64 系统上，该内存区域的长度可能从 0 到 4G。

ZONE_NORMAL 标记了可直接映射到内核段的普通内存域。这是在所有体系结构上都会存在的唯一内存域，但无法保证该地址范围对应了实际的物理内存。

ZONE_HIGHMEM 标记了超出内核段的物理内存。

ZONE_MOVABLE 内核定义的一个伪内存区域，在防止物理内存脆片中会用到该区域。

MAX_NR_ZONES 充当结束标记，在内核中想要迭代系统中所有区域时，都会用到该常量。

每个内存域都关联了一个数组，用来组织属于该内存区域的物理内存（内核中称为页帧）。对每个页帧，内核使用 struct page 实例以及所需要的管理数据。

2.2.1 数据结构

这里看看内核内存三级管理体系中涉及的相关数据结构。

2.2.1.1 节点管理

所在文件 include/linux/mmzone.h

```
588 /*
589  * The pg_data_t structure is used in machines with CONFIG_DISCONTIGMEM
590  * (mostly NUMA machines?) to denote a higher-level memory zone than the
591  * zone denotes.
592  *
593  * On NUMA machines, each NUMA node would have a pg_data_t to describe
594  * it's memory layout.
595  *
596  * Memory statistics and page replacement data structures are maintained on a
597  * per-zone basis.
598  */
599 struct bootmem_data;
600 typedef struct pglist_data {
601         struct zone node_zones[MAX_NR_ZONES];
602         struct zonelist node_zonelists[MAX_ZONELISTS];
603         int nr_zones;
604 #ifdef CONFIG_FLAT_NODE_MEM_MAP /* means !SPARSEMEM */
605         struct page *node_mem_map;
```

```
606 #ifdef CONFIG_CGROUP_MEM_RES_CTLR
607         struct page_cgroup *node_page_cgroup;
608 #endif
609 #endif
610 #ifndef CONFIG_NO_BOOTMEM
611         struct bootmem_data *bdata;
612 #endif
613 #ifdef CONFIG_MEMORY_HOTPLUG
614         /*
615          * Must be held any time you expect node_start_pfn,
node_present_pages
616          * or node_spanned_pages stay constant.    Holding this will also
617          * guarantee that any pfn_valid() stays that way.
618          *
619          * Nests above zone->lock and zone->size_seqlock.
620          */
621         spinlock_t node_size_lock;
622 #endif
623         unsigned long node_start_pfn;
624         unsigned long node_present_pages; /* total number of physical
pages */
625         unsigned long node_spanned_pages; /* total size of physical page
626                                              range, including holes */
627         int node_id;
628         wait_queue_head_t kswapd_wait;
629         struct task_struct *kswapd;
630         int kswapd_max_order;
631 } pg_data_t;
```

- node_zones 是一个数组，包含了节点内各内存区域的数据结构。
- node_zonelists 指定了备用节点及其内存域的列表，以遍在当前节点没有可用空间时，在备用节点分配内存。
- nr_zones 节点中内存区域的数目。

- node_mem_map 是指向 page 实例数组的指针，用于描述节点的所有物理内存页面。
- bdata 如果编译内核时没定义 CONFIG_NO_BOOTMEM，则使用 bootmem 分配器，2.6.34 之后的内核中，默认的是不使用 bootmem 的。
- node_start_pfn：该节点的第一个页帧的逻辑编号。
- node_present_pages 该节点中包含的实际页面数。
- node_spanned_pages 该节点内存区域涵盖的页面数，可能包含空洞
- node_id 是全局的节点号

2.2.1.2 内存区域

```
280 struct zone {
 281        /* Fields commonly accessed by the page allocator */
 282
 283        /* zone watermarks, access with *_wmark_pages(zone) macros */
 284        unsigned long watermark[NR_WMARK];
 285
294        unsigned long          lowmem_reserve[MAX_NR_ZONES];
 295
 296 #ifdef CONFIG_NUMA
 297        int node;
 298        /*
 299         * zone reclaim becomes active if more unmapped pages exist.
 300         */
 301        unsigned long          min_unmapped_pages;
 302        unsigned long          min_slab_pages;
 303 #endif
 304        struct per_cpu_pageset __percpu *pageset;
 305        /*
 306         * free areas of different sizes
 307         */
 308        spinlock_t         lock;
```

```
309         int                             all_unreclaimable; /* All pages pinned */
310 #ifdef CONFIG_MEMORY_HOTPLUG
311         /* see spanned/present_pages for more description */
312         seqlock_t           span_seqlock;
313 #endif
314         struct free_area       free_area[MAX_ORDER];
315
316 #ifndef CONFIG_SPARSEMEM
317         /*
318          * Flags for a pageblock_nr_pages block. See pageblock-flags.h.
319          * In SPARSEMEM, this map is stored in struct mem_section
320          */
321         unsigned long           *pageblock_flags;
322 #endif /* CONFIG_SPARSEMEM */
323
324
325         ZONE_PADDING(_pad1_)
326
327         /* Fields commonly accessed by the page reclaim scanner */
328         spinlock_t          lru_lock;
329         struct zone_lru {
330                 struct list_head list;
331         } lru[NR_LRU_LISTS];
332
333         struct zone_reclaim_stat reclaim_stat;
334
335         unsigned long           pages_scanned;        /* since last reclaim */
336         unsigned long           flags;              /* zone flags, see below */
337
338         /* Zone statistics */
339         atomic_long_t           vm_stat[NR_VM_ZONE_STAT_ITEMS];
```

```
 340
 354        int prev_priority;
 355
 360        unsigned int inactive_ratio;
 361
 362
 363        ZONE_PADDING(_pad2_)
 390        wait_queue_head_t     * wait_table;
 391        unsigned long         wait_table_hash_nr_entries;
 392        unsigned long         wait_table_bits;
 393
 394        /*
 395         * Discontig memory support fields.
 396         */
 397        struct pglist_data    *zone_pgdat;
 398        /* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */
 399        unsigned long         zone_start_pfn;
 400
 411        unsigned long         spanned_pages;    /* total size, including holes */
 412        unsigned long         present_pages;    /* amount of memory (excluding holes) */
 413
 414        /*
 415         * rarely used fields:
 416         */
 417        const char            *name;
 418 } ____cacheline_internodealigned_in_smp;
```

- watermark 内存水位标记，在分配内存时会用到。
- lowmem_reserve 数组分别为各种内存域指定了若干页，用于一些无论如何都不能失败的关键性内存分配。

- pageset 是一个数组，用于实现每个 CPU 的热/冷页面帧列表。内核使用这些列表来保存用于可用于满足实现的“新鲜”页。
- free_area 是一个数组，用于实现伙伴系统。
- lru 用于内存回收扫描时的链表。
- page_scanned 最新一次扫描扫描的页面数
- flags 内存域标志，标记内存的状态。
- vm_stat 维护了大量有关该内存域的统计信息。
- prev_priority 存储了上一次扫描操作扫描该内存域的优先级。
- wait_table，wait_table_bits，wait_table_hash_nr_entries 实现了一个等待队列，可用于等待某一页变为进程可用。
- zone_pgdat 内存域与父节点之间的关联
- zone_start_pfn 内存域第一个页帧的索引
- spanned_pages 内存域中页面数目，包含空洞
- present_pages 内存域中可用页面数
- name 内存域名字

2.2.1.3 页帧

页帧代表系统内存的最小单位，对内存中的每个页都会创建 struct page 的一个实例。

内核设计需要尽量保证该结构尽量小。

由于 page 结构会由于很多地方，所以其中有很多字段在某种情况下有用，而在另外的情况下无用，为了保证 page 字段尽量小，内核使用联合的方式将同一内存出的内容做不同的解释。

```
27 /*
28  * Each physical page in the system has a struct page associated with
29  * it to keep track of whatever it is we are using the page for at the
30  * moment. Note that we have no way to track which tasks are using
31  * a page, though if it is a pagecache page, rmap structures can tell us
32  * who is mapping it.
33  */
34 struct page {
35      unsigned long flags;          /* Atomic flags, some possibly
36                                    * updated asynchronously */
```

```
37      atomic_t _count;            /* Usage count, see below. */
38      union {
39          atomic_t _mapcount; /* Count of ptes mapped in mms,
40                               * to show when page is mapped
41                               * & limit reverse map searches.
42                               */
43          struct {            /* SLUB */
44              u16 inuse;
45              u16 objects;
46          };
47      };
48      union {
49          struct {
50          unsigned long private;          /* Mapping-private opaque data:
51                                          * usually used for buffer_heads
52                                          * if PagePrivate set; used for
53                                          * swp_entry_t if PageSwapCache;
54                                          * indicates order in the buddy
55                                          * system if PG_buddy is set.
56                                          */
57          struct address_space *mapping;    /* If low bit clear, points to
58                                          * inode address_space, or NULL.
59                                          * If page mapped as anonymous
60                                          * memory, low bit is set, and
61                                          * it points to anon_vma object:
62                                          * see PAGE_MAPPING_ANON below.
63                                          */
64          };
65 #if USE_SPLIT_PTLOCKS
66          spinlock_t ptl;
67 #endif
```

```
 68                struct kmem_cache *slab;      /* SLUB: Pointer to slab */
 69                struct page *first_page;      /* Compound tail pages */
 70        };
 71        union {
 72                pgoff_t index;          /* Our offset within mapping. */
 73                void *freelist;         /* SLUB: freelist req. slab lock */
 74        };
 75        struct list_head lru;           /* Pageout list, eg. active_list
 76                                         * protected by zone->lru_lock !
 77                                         */
 78        /*
 79         * On machines where all RAM is mapped into kernel address space,
 80         * we can simply calculate the virtual address. On machines with
 81         * highmem some memory is mapped into kernel virtual memory
 82         * dynamically, so we need a place to store that address.
 83         * Note that this field could be 16 bits on x86 ... ;)
 84         *
 85         * Architectures with slow multiplication can define
 86         * WANT_PAGE_VIRTUAL in asm/page.h
 87         */
 88 #if defined(WANT_PAGE_VIRTUAL)
 89        void *virtual;                  /* Kernel virtual address (NULL if
 90                                           not kmapped, ie. highmem) */
 91 #endif /* WANT_PAGE_VIRTUAL */
 92 #ifdef CONFIG_WANT_PAGE_DEBUG_FLAGS
 93        unsigned long debug_flags;    /* Use atomic bitops on this */
 94 #endif
 95
 96 #ifdef CONFIG_KMEMCHECK
 97        /*
 98         * kmemcheck wants to track the status of each byte in a page; this
```

```
 99          * is a pointer to such a status block. NULL if not tracked.
100          */
101         void *shadow;
102 #endif
103 };
```

- Slab，freelist，inuse 用于 slab 分配器。
- flags 存储了体系结构无关的标志，用于描述页的属性。
- _count 使用计数，表示内核中引用该页的次数。
- _mapcount 表示在页表中有多少项指向该页。
- lru 是一个表头，用于在各种链表上维护该页，以便将该页按不同的类别分组。最重要的是活动页和不活动页。
- first_page 内核可以将相邻的页合并为较大的复合页。分组中的第一个页称为首页，而所有其余的页叫做尾页。first_page 指向首页。
- mapping 指定了页帧所在的地址空间。Index 是页帧在映射内部的偏移量。地址空间是一个非常一般的概念。地址空间用于将文件的内容（数据）与装载的内存区关联起来。通过一个技巧，mapping 不仅能够保存一个指针，而且还可以包含一些额外的信息，用于判断页是否关联到地址空间的某个匿名内存区。如果将 mapping 的最低位设为 1，则该指针并不指向 address_space 的实例，而是指向另一个数据结构（anon_vma），该结构对实现匿名页的逆向映射很重要。可以双重使用该指针是因为 address_space 实例总是对齐到 sizeof(long)。
- Private 指向“私有”数据的指针，虚拟内存管理会忽略该数据。根据页的用途，可以用不同的方式使用该指针。大多数情况下它将页与数据缓冲区关联起来。
- virtual 用于高端内存区域中的页，换言之，即无法直接映射到内存中的页。virtual 用于存储该页的虚拟地址。

2.2.1.4 体系结构无关页标志

页的不同属性通过一系列标志描述，存储为 struct page 的 flags 成员中的各个比特位。这样的标志独立于是用的体系结构，因而无法提供特定于 CPU 或计算机的信息（该信息保存在页表中）。

下面列出一些重要的标志：

- PG_locked 指定了页是否锁定。如果该比特置位，内核的其它部分不允许访问该页。这防止了内存管理出现竞态条件，例如，在从硬盘读取数据到页帧时。
- PG_error 在涉及 I/O 操作期间发生错误，则 PG_error 置位。
- PG_referenced 和 PG_active 控制了系统使用该页的活跃程度。在页交互子系统选择换出页时，该信息很重要。
- PG_uptodate 表示页的数据已经从块设备读取，期间没有出错。
- PG_dirty 如果与硬盘上的数据相比，页的内容已经改变，则置位该标志。出于性能的考虑，页并不在每次改变后立即回写。因此内核使用该标志注明页已经改变，可以稍后刷出。
- PG_lru 有助于实现页面回收和切换。内核使用两个最近最少使用(least recently used,lru)链表来区别活动和不活动页。如果页在其中一个链表中，则设置该比特位。还有一个 PG_active 标志，如果页在活动页链表中，则设置该标志。
- PG_highmem 表示页在高端内存中，无法持久映射到内核内存中。
- PG_private 位，如果 page 结构的 private 成员非空，则必须设置 PG_private 位，用于 I/O 的页，可使用该字段将页细分为多个缓冲区，但内核的其他部分也有各种不同的方法，将私有数据附加到页上。
- PG_wrateback，如果页的内容处于向块设备回写的过程中，则需要设置 PG_writeback 位。
- PG_slab 位，如果页是 slab 分配器的一部分，则设置该位。
- PG_swapcache 位，如果页处于交换缓存，则设置改位。在这种情况下，private 包含一个类型为 swap_entry_t 的项。
- PG_reclaim 在可用内存的数量变少时，内核试图周期性地回收页，即剔除不活动、未用的页。在内核决定回收某个特定的页之后，需要设置 PG_reclaim 标志通知。
- PG_buddy 如果页空闲且包含在伙伴系统的列表中，则设置该位。
- PG_compound 表示该页属于一个更大的复合页，复合页由多个毗连的普通页组成。

2.2.1.5 页表

层次化的页表用于支持对大地址空间的快速、高效的管理。

页表用于建立用户进程的虚拟地址空间和系统物理内存（内存、页帧）之间的关联。到目前为止讨论的数据结构主要用于描述内存的机构（划分为节点和内存域），

同时指定了其中包含的页帧的数量和状态（使用中或者空闲）。页表用于向每个进程提供一致的虚拟地址空间。应用程序看到的地址空间是一个连续的内存区。该表也将虚拟内存页映射到物理内存，因而支持共享内存的实现。还可以在不额外增加物理内存的情况下，将页换出到块设备来增加有效的可用内存空间。

内核内存管理总是假定使用四级页表，而不管底层处理器是否如此。默认情况下，该体系结构只使用两级分页系统（在不适用 PAE 扩展的情况下）。因此，第三级和第四级页表由特定于体系的结构代码模拟。

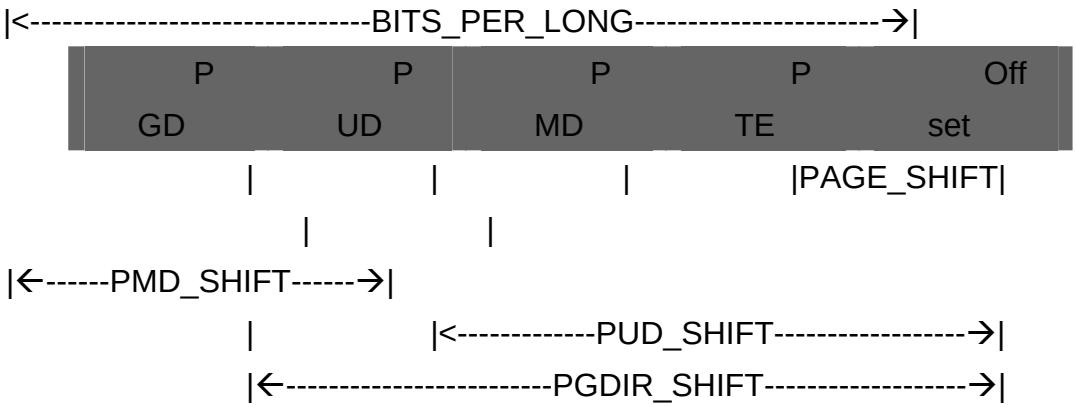
页表管理分为两部分，第一部分依赖于体系结构，而第二部分是体系结构无关的。

内存管理使用 unsigned long 类型的变量来表示页面项的类型。

1、内存地址的分解

根据四级页表结构的需要，虚拟内存地址分为 5 部分（4 个表项用于选择页，1 个索引表示页帧内部的偏移）。

各个体系结构的不仅地址长度不同，而且地址拆分的方式也不同。因此内核定义了宏，用于将地址分解为各个分量。



OFFSET：指定页帧内部的位置。

PMD_SHIFT：页表偏移量和最后一集页表项所需的比特位的总数。该值减去 PAGE_SHIFT 表示索引最后一级页表项所需的比特位数。更重要的一个事实是：该值表明了一个中间层页表项管理的地址空间的大小，即 2^{PMD_SHIFT} 字节。

PUD_SHIFT，PGDIR_SHIFT 含义和上相同。

在各级页目录/页表中所能存储的指针数目，也可以通过宏定义确定。

PTRS_PER_PGD 指定了全局页目录项的数目。

PTRS_PER_PUD 对应于页上层页目录中项的数目。

PTRS_PER_PMD 对应于页中间目录中项的数目。

PTRS_PER_PTE 则是页表中项的数目。

两级页表的体系结构会将 PTRS_PER_PMD 和 PTRS_PER_PUD 定义为 1。这使得内核的剩余部分感觉该体系结构也提供了四级页转换结构。

2、页表的格式

上述定义已经确立了页表项的数目，但没定义器结构。内核提供了 4 个数据结构。

- pgd_t 用于页全局目录
- pud_t 用于上层页目录
- pmd_t 用于中间页目录
- pte_t 用于直接页表项

用于分析页表项的标准函数

pgd_val 将 pte_t 等类型转换转换为变量为 unsigned long 整数。

pud_val

pmd_val

pte_val

pgprot_val

__pgd 将 unsigned long 整数转换为 pgd_t 等类型的变量

__pud

__pmd

__pte

__pgprot

pgd_index 从内存指针和页表项获得下一级页表的地址。

pud_index

pmd_index

pte_index

pgd_present 检测对应的_PRESENT 位是否设置。

pud_present

pmd_present

pte_present

pgd_none　　对 xxx_present 函数的值逻辑取反。如果返回 true，则检测的页不在内存中。

pud_none

pmd_none

pte_none

pgd_clear 删去传递的页表项，通常是将其设置为零

pud_clear

pmd_clear

pte_clear

pgd_bad 检测中间页表项、上次页表、全局页表项是否无效

pud_bad

pmd_bad

pmd_page　　返回保存页数据的 page 结构或者中间页目录的项。

pud_page

pte_page

3、特定与 PTE 的信息

最后一级页表中的项不仅包含了指向页的内存位置的指针，还在上述的多余比特位包含了与页有关的附加信息，尽管这些数据时特定于 CPU 的，它们至少提供了有关页访问控制的一些信息。

_PAGE_PRESENT 指定了虚拟内存页是否存在于内存中。

_PAGE_ACCESSED CPU 每次访问页时，会自动设置该位。内核会定期检查该比特位，以确认页使用的活跃程度。

_PAGE_DIRTY 表示该页是否是“脏的”，即页的内容是否已经修改。

_PAGE_FILE 的数值与_PAGE_DIRTY 相同，当用于不同的上下文。

_PAGE_USER 如果设置了改为，则允许用户空间的代码访问该页，否则只有内核才能访问。

_PAGE_READ、_PAGE_WRITE、_PAGE_EXECUTE 指定了普通的用户进程是否允许读取、写入、执行该页中的机器代码。

_PAGE_BIT_NX，用于将页标记为不可执行。在 IA-32（在 IA-32 系统上，只有启用了可寻址 64Gib 内存的页面地址扩展 PAE 功能时）和 AMD64 可以使用。

2.3 初始化内存管理

2.3.1 启动期间内存管理

linux 内核在 2.6.34 中基本上弃用了 bootmem 分配器，在内核的默认配置中，是不使用 bootmem 分配器的，而是使用了一种分配效率更高的方式，early_res 分配方式。

2.3.1.1 相关数据结构

E820 数据结构：

该数据结构保存内核启动期间通过 BIOS 15 号中断获得内存布局信息。

arch/x86/include/asm/e820.h

```
52 #include <linux/types.h>
53 struct e820entry {
54        __u64 addr; /* start of memory segment */
55        __u64 size; /* size of memory segment */
56        __u32 type; /* type of memory segment */
57 } __attribute__((packed));
58
59 struct e820map {
60        __u32 nr_map;
61        struct e820entry map[E820_X_MAX];
62 };
```

- nr_map:内存区域数目
- addr：起始内存地址
- size：内存块大小

- type：内存类型

arch/x86/kernel/e820.c 中定义全局变量

```
struct e820map e820;
```

全局变量 e820 保存内核启动参数中通过 bios int15 调用传过来的内存布局。

E820 中不仅仅包含 RAM，还有可能包含 ROM，保留内存区等。

```
struct node_active_region {
    unsigned long start_pfn;
    unsigned long end_pfn;
    int nid;
};
```

- start_pfn：起始物理页面的编号
- end_pdf：终止物理页面的编号
- nid：节点编号

mm/page_alloc.c

```
178    static struct node_active_region __meminitdata
early_node_map[MAX_ACTIVE_REGIONS];
```

early_node_map 中保存所有节点的内存（RAM）区域范围。

```
#define MAX_EARLY_RES_X 32

struct early_res {
    u64 start, end;
    char name[15];
    char overlap_ok;
};
static struct early_res early_res_x[MAX_EARLY_RES_X] __initdata;

static int max_early_res __initdata = MAX_EARLY_RES_X;
static struct early_res *early_res __initdata = &early_res_x[0];
```

```
static int early_res_count __initdata;
```

early_res 记录内核中保留内存（不被伙伴系统支配，无法回收释放）的区域。

上述数据结构的关系是：

全局变量 e820 保存了内核启动过程中内存的布局信息。在内核初始化过程中，会根据 e820 初始化全局变量 early_node_map。

而 early_res 在启用伙伴系统之前，记录内核的保留页面，这些页面包括系统启动时内核所占用的页面，从 early_node_map 中分配的页面。

在上述数据结构初始化好后，内核通过从 early_node_map 中去掉在 early_res 保留的页面，建立伙伴系统。伙伴系统建立后，early_node_map 的内存分配方式被启用，从此内核可以享受伙伴系统提供的高效便捷的内存管理功能。

2.3.1.2 reserve_early机制

```
175 static void __init __reserve_early(u64 start, u64 end, char
*name,
176                                     int overlap_ok)
177 {
178         int i;
179         struct early_res *r;
180
181         i = find_overlapped_early(start, end);
182         if (i >= max_early_res)
183                 panic("Too many early reservations");
184         r = &early_res[i];
185         if (r->end)
186                 panic("Overlapping early reservations "
187                         "%llx-%llx %s to %llx-%llx %s\n",
188                         start, end - 1, name ? name : "", r->start,
```

```
189                     r->end - 1, r->name);
190         r->start = start;
191         r->end = end;
192         r->overlap_ok = overlap_ok;
193         if (name)
194             strncpy(r->name, name, sizeof(r->name) - 1);
195         early_res_count++;
196 }
```

在全局变量 early_res 中添加一个内存保留区域。

2.3.1.3 early_node_map内存管理

内核在伙伴系统建立起来之前，分配内存时最终的调用接口会落在 __alloc_memory_core_early 函数上，该函数的内容如下：

该函数的作用是从 early_node_map 中分配一块内存，然后将分配的内存块添加到保留数组 reserve_early 中。

```
3411 #ifdef CONFIG_NO_BOOTMEM
3412 void * __init __alloc_memory_core_early(int nid, u64 size, u64 align,
3413                                 u64 goal, u64 limit)
3414 {
3415         int i;
3416         void *ptr;
3417
3418         /* need to go over early_node_map to find out good range for node
*/
3419         for_each_active_range_index_in_nid(i, nid) {
3420             u64 addr;
3421             u64 ei_start, ei_last;
3422
3423             ei_last = early_node_map[i].end_pfn;
3424             ei_last <<= PAGE_SHIFT;
3425             ei_start = early_node_map[i].start_pfn;
```

```
3426            ei_start <<= PAGE_SHIFT;
3427            addr = find_early_area(ei_start, ei_last,
3428                                   goal, limit, size, align);
3429
3430            if (addr == -1ULL)
3431                    continue;
3432
3433 #if 0
3434            printk(KERN_DEBUG "alloc (nid=%d %llx - %llx) (%llx - %llx)
%llx %llx => %llx\n",
3435                                   nid,
3436                                   ei_start, ei_last, goal, limit, size,
3437                                   align, addr);
3438 #endif
3439
3440            ptr = phys_to_virt(addr);
3441            memset(ptr, 0, size);
3442            reserve_early_without_check(addr, addr + size, "BOOTMEM");
3443            return ptr;
3444        }
3445
3446        return NULL;
3447 }
3448 #endif
```

该函数遍历 early_node_map 的每一个区域，从中找到一个符合分配的内存区。是否符合内核是通过调用 find_early_area 来确定的。

该函数的六个参数如下：

ei_start:e820 内存区起始区域地址

ei_last:e820 内存区域终止地址

start:可供分配内存的起始地址

end:可供分配内存的终止地址

size:期望分配内存的大小

align:对齐方式

```
534 /*
535  * Find a free area with specified alignment in a specific range.
536  * only with the area.between start to end is active range from early_node_map
537  * so they are good as RAM
538  */
539 u64 __init find_early_area(u64 ei_start, u64 ei_last, u64 start, u64 end,
540                         u64 size, u64 align)
541 {
542         u64 addr, last;
543
544         addr = round_up(ei_start, align);
545         if (addr < start)
546                 addr = round_up(start, align);
```

addr 必须属于[start,end]

```
547         if (addr >= ei_last)
548                 goto out;
```

addr 必须不超过 ei_last

```
549         while (bad_addr(&addr, size, align) && addr+size <= ei_last)
550                 ;
551         last = addr + size;
552         if (last > ei_last)
553                 goto out;
```

last 必须属于[ei_start,ei_last]

```
554         if (last > end)
555                 goto out;
556
557         return addr;
```

```
558
559 out:
560         return -1ULL;
561 }
```

上述函数的基本功能是：

在[ei_start,ei_last]区域分配一块大小为 size 的内存。如果 ei_last-ei_start<size，分配失败。在大于的情况下，还要保证分配内存的区间在[start,end]之间。

在上述函数的 549 行 bad_addr(&addr, size, align)函数我们还没有介绍，该函数的作用是检测[addr,addr+size]是否落与某个已经分配出去的内存区域存在交集。如果是则继续查找合适的区域。具体细节看如下函数：

```
482 /* Check for already reserved areas */
483 static inline int __init bad_addr(u64 *addrp, u64 size, u64 align)
484 {
485         int i;
486         u64 addr = *addrp;
487         int changed = 0;
488         struct early_res *r;
489 again:
490         i = find_overlapped_early(addr, addr + size);
491         r = &early_res[i];
492         if (i < max_early_res && r->end) {
493                 *addrp = addr = round_up(r->end, align);
494                 changed = 1;
495                 goto again;
496         }
497         return changed;
498 }
```

490 行调用 find_overlapped_early 检测区间[addr，addr+size]是否包含已经保留的地址空间。

```
 31 static int __init find_overlapped_early(u64 start, u64 end)
 32 {
```

```
33      int i;
34      struct early_res *r;
35
36      for (i = 0; i < max_early_res && early_res[i].end; i++) {
37              r = &early_res[i];
38              if (end > r->start && start < r->end)
39                      break;
40      }
```

遍历每一个保留的区间，查看器是否与[start，end]有交集，如果有，则说明[start,end]中包含保留的内存地址空间。

```
41
42      return i;
43 }
```

如果通过 find_early_area 找到了合适的内存区间，则调用 reserve_early_without_check 函数将该区间加入到全局遍历 early_res 中。

```
298 void __init reserve_early_without_check(u64 start, u64 end,
char *name)
299 {
300     struct early_res *r;
301
302     if (start >= end)
303             return;
304
305     __check_and_double_early_res(start, end);
```

如果 early_res 内存空间不足，则扩充内存。

```
306
307     r = &early_res[early_res_count];
308
309     r->start = start;
310     r->end = end;
311     r->overlap_ok = 0;
```

309-311 设置已分配内存区。

```
312         if (name)
313                 strncpy(r->name, name, sizeof(r->name) - 1);
314         early_res_count++;
315 }
```

2.3.1.4 reserve_early数组的初始化

```
31 void __init i386_start_kernel(void)
32 {
33 #ifdef CONFIG_X86_TRAMPOLINE
34         /*
35          * But first pinch a few for the stack/trampoline stuff
36          * FIXME: Don't need the extra page at 4K, but need to fix
37          * trampoline before removing it. (see the GDT stuff)
38          */
39         reserve_early_overlap_ok(PAGE_SIZE, PAGE_SIZE + PAGE_SIZE,
40                                  "EX TRAMPOLINE");
41 #endif
42
43         reserve_early(__pa_symbol(&_text), __pa_symbol(&__bss_stop),
"TEXT DATA BSS");
44
45 #ifdef CONFIG_BLK_DEV_INITRD
46         /* Reserve INITRD */
47         if (boot_params.hdr.type_of_loader &&
boot_params.hdr.ramdisk_image) {
```

```
48              /* Assume only end is not page aligned */
49              u64 ramdisk_image = boot_params.hdr.ramdisk_image;
50              u64 ramdisk_size    = boot_params.hdr.ramdisk_size;
51              u64 ramdisk_end     = PAGE_ALIGN(ramdisk_image +
ramdisk_size);
52              reserve_early(ramdisk_image, ramdisk_end, "RAMDISK");
53      }
54 #endif
55
56      /* Call the subarch specific early setup function */
57      switch (boot_params.hdr.hardware_subarch) {
58      case X86_SUBARCH_MRST:
59              x86_mrst_early_setup();
60              break;
61      default:
62              i386_default_early_setup();
63              break;
64      }
65
66      /*
67       * At this point everything still needed from the boot loader
68       * or BIOS or kernel text should be early reserved or marked not
69       * RAM in e820. All other memory is free game.
70       */
71
72      start_kernel();
73 }
```

- 43 行：将内核所占页面加入到包 early_res 数组中
- 52 行：将 ramdisk 所占的内存页面加入到 early_res 数组中。
- 62 行：设置 bios 的保留页面。

在伙伴系统建立起来之前，内核通过__alloc_memory_core_early 分配内存，分配的内存会添加到保留数组 early_res 中。

2.3.1.5 early_node_map初始化

在内核初始化期间，调用 initmem_init 函数初始化 early_node_map 函数。

该版本的函数在 arch/x86/mm/init_32.c 中：

```
707 #ifndef CONFIG_NEED_MULTIPLE_NODES
708 void __init initmem_init(unsigned long start_pfn, unsigned long end_pfn,
709                           int acpi, int k8)
710 {
711 #ifdef CONFIG_HIGHMEM
712         highstart_pfn = highend_pfn = max_pfn;
713         if (max_pfn > max_low_pfn)
714                 highstart_pfn = max_low_pfn;
715         e820_register_active_regions(0, 0, highend_pfn);
716         sparse_memory_present_with_active_regions(0);
717         printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
718                 pages_to_mb(highend_pfn - highstart_pfn));
719         num_physpages = highend_pfn;
720         high_memory = (void *) __va(highstart_pfn * PAGE_SIZE - 1) + 1;
721 #else
722         e820_register_active_regions(0, 0, max_low_pfn);
723         sparse_memory_present_with_active_regions(0);
724         num_physpages = max_low_pfn;
725         high_memory = (void *) __va(max_low_pfn * PAGE_SIZE - 1) + 1;
726 #endif
727 #ifdef CONFIG_FLATMEM
```

```
728         max_mapnr = num_physpages;
729 #endif
730         __vmalloc_start_set = true;
731
732         printk(KERN_NOTICE "%ldMB LOWMEM available.\n",
733                         pages_to_mb(max_low_pfn));
734
735         setup_bootmem_allocator();
736 }
737 #endif /* !CONFIG_NEED_MULTIPLE_NODES */
```

该函数调用 e820_register_active_regions

函数参数：

nid: cpu 节点号

start_pfn:起始页面号

last_pnf:终止页面号

```
931 /* Walk the e820 map and register active regions within a node */
932 void __init e820_register_active_regions(int nid, unsigned long start_pfn,
933                                         unsigned long last_pfn)
934 {
935         unsigned long ei_startpfn;
936         unsigned long ei_endpfn;
937         int i;
938
939         for (i = 0; i < e820.nr_map; i++)
940                 if (e820_find_active_region(&e820.map[i],
941                                             start_pfn, last_pfn,
942                                             &ei_startpfn, &ei_endpfn))
943                         add_active_range(nid, ei_startpfn, ei_endpfn);
944 }
```

940 行：调用 e820_find_active_region 检测 e820.map[i]是否为合法的 active region，如果是调用 add_active_range 将其加入到 early_node_map 中。

该函数所在文件为 mm/page_alloc.c 中：
从注释中可以看出：
nid:CPU 节点号
start_pfn：可用物理内存的起始页面号
end_pfn：终止节点的起始页面号

该函数的功能：将[start_pfn,end_pfn]对应的范围保存到 early_node_map 数组中，在后面的过程中该数组将会被 free_area_init_nodes 调用，用于计算各个区域的大小以及是否存在空洞。

```
3972 /**
3973  * add_active_range - Register a range of PFNs backed by physical memory
3974  * @nid: The node ID the range resides on
3975  * @start_pfn: The start PFN of the available physical memory
3976  * @end_pfn: The end PFN of the available physical memory
3977  *
3978  * These ranges are stored in an early_node_map[] and later used by
3979  * free_area_init_nodes() to calculate zone sizes and holes. If the
3980  * range spans a memory hole, it is up to the architecture to ensure
3981  * the memory is not freed by the bootmem allocator. If possible
3982  * the range being registered will be merged with existing ranges.
3983  */
3984 void __init add_active_range(unsigned int nid, unsigned long start_pfn,
```

```
3985                                    unsigned long end_pfn)
3986 {
3987        int i;
3988
3989        mminit_dprintk(MMINIT_TRACE, "memory_register",
3990                "Entering add_active_range(%d, %#lx, %#lx) "
3991                "%d entries of %d used\n",
3992                nid, start_pfn, end_pfn,
3993                nr_nodemap_entries,
MAX_ACTIVE_REGIONS);
3994
3995        mminit_validate_memmodel_limits(&start_pfn,
&end_pfn);
3996
3997        /* Merge with existing active regions if possible */
3998        for (i = 0; i < nr_nodemap_entries; i++) {
3999                if (early_node_map[i].nid != nid)
4000                        continue;
```

3999-4000：节点号不匹配，跳入下一次循环

```
4001
4002                /* Skip if an existing region covers this new one */
4003                if (start_pfn >= early_node_map[i].start_pfn &&
4004                                end_pfn <= early_node_map[i].end_pfn)
4005                        return;
```

4003-4005：已经存在的区域，直接返回

```
4006
4007                /* Merge forward if suitable */
4008                if (start_pfn <= early_node_map[i].end_pfn &&
4009                                end_pfn > early_node_map[i].end_pfn) {
4010                        early_node_map[i].end_pfn = end_pfn;
4011                        return;
```

```
4012            }
```

4008-4012：前向合并

```
4013
4014            /* Merge backward if suitable */
4015            if (start_pfn < early_node_map[i].start_pfn &&
4016                    end_pfn >= early_node_map[i].start_pfn) {
4017                early_node_map[i].start_pfn = start_pfn;
4018                return;
4019            }
```

4014-4019：后向合并

```
4020        }
4021
4022        /* Check that early_node_map is large enough */
4023        if (i >= MAX_ACTIVE_REGIONS) {
4024            printk(KERN_CRIT "More than %d memory regions,
truncating\n",
4025                                    MAX_ACTIVE_REGIONS);
4026            return;
4027        }
4028
4029        early_node_map[i].nid = nid;
4030        early_node_map[i].start_pfn = start_pfn;
4031        early_node_map[i].end_pfn = end_pfn;
4032        nr_nodemap_entries = i + 1;
```

4029-4032：如果无法合并，这建立一个新的项，同时将 nr_nodemap_entries 加一

```
4033 }
```

有一个小小的疑问：为啥不考虑

start_pfn <= early_node_map[i].start_pfn &&

end_pfn > early_node_map[i].end_pfn 的情况？

在看了初始化期间的内存管理各种数据结构后，下面来分析特定于体系结构的内存管理。

2.3.2 物理内存初始化流程

在[start_kernel 之物理内存管理]中已经详细的分析了 特定于体系结构的内存管理 的代码，在本小结中主要讲解内存管理的流程，而不会去详细的分析代码了。

2.3.2.1 内核在内存中的布局

默认情况下，内核被装载到物理内存中的一个固定位置，该位置在编译时确定。

如果启用了故障转储机制，那么也可以配置内核二进制代码在物理内存中的初始位置。配置选项 PHYSICAL_START 用于确定内核在内存中的位置，会受到配置选项 PHYSICAL_ALGN 设置的物理对齐方式的影响。

此外，内核可以编译为可重定位二进制程序，在这种情况下完全忽略编译时给定的物理起始地址。启动装载程序可以判断将内核放置到何处。
物理内存最低几兆的布局一般如下：
0-4K：前 4K 是第一个页帧，一般会忽略，通常保留给 BI0S 使用。
4K-640K：这段空间一般可用，但也不用于内核加载。
640K-1M：内核保留区域，一般用于映射各种 ROM。
1M-_end：加载内核区域的空间。

内核占用的内存分为几个段，其边界保存在变量中。
_text 和_etext 是代码段的起始地址和结束地址，包含了编译后的内核代码。
_etext 和_edata 之间，保存了大部分的内核变量。
_edata_end 初始化数据在内核启动过程结束后不再需要（例如，包含初始化为 0 的所有静态全集变量的 BSS 段）保存在最后一段，从_edata 到_end。

2.3.2.2 初始化步骤

在内核已经载入内存、而初始化的汇编代码程序部分已经执行完毕后，内核调用 setup_arch 函数执行特定于系统的初始化。

特定于体系结构的初始化

setup_arch

default_machine_specific_memory_setup

init_memory_mapping

initmem_init

paging_init

pagetable_init

kmap_init

zone_size_init

free_area_init_ndoes

default_machine_specific_memory_setup 该函数将内核启动参数中有关内存布局的信息保存到全局变量 e820 中。

init_memory_mapping 建立永久内核页表。

initmem_init 将 e820 中是 RAM 的内存区域保存到变量 early_node_map 全局变量中。

paging_init：高端内存初始化（初始化永久内核、临时内核、内存节点、区域变量初始化）。

节点备用内存区域初始化

build_all_zonelists

伙伴系统初始化

mem_init

slab 分配器初始化

kmem_cache_init

2.4 高端内存管理

高端内存是指内核不能通过线性关系映射的内核虚拟地址空间。

CPU 理论上支持的物理地址空间为 0-4G，而内核的虚拟地址空间仅为 1G，如果内核仅仅使用线性地址的关系来映射物理内存的话，则内核无法使用物理内存 1G 以上的空间。内核借助于高端内存管理的方法来是内核可以访问所以的物理地址空间。

在 IA-32 体系中，内核可以直接使用的物理地址空间为 0-896M，也就是说，内核虚拟地址空间[3G，3G+896M]的部分使用线性地址映射，而[3G+896M，4G]的 128M 地址空间则使用高端内存管理的方式。

高端内存的管理分为三种方式：

- 非连续页的分配
- 永久内核映射
- 临时内核映射

2.4.1 非连续页的分配

物理上连续的内存页的分配对内核是最好的，但在物理内存紧缺的情况下，连续页面的分配并不总能成功。用户空间中这不是问题，因为用户空间使用的是逻辑映射。

在内核空间中也使用了同样的技术，内核分配了其高端虚拟内存地址空间的一部分，用于建立虚拟地址空间连续但物理地址空间可能不连续的映射。

在 IA-32 系统中，紧随直接映射（线性映射）的前 892M 物理内存，在插入 8M 的安全空隙之后，是用于管理不连续内存的区域。这一段具有用户空间线性地址空间的所有性质。分配到其中的页可能位于物理地址空间中的任何地方。

2.4.1.1 数据结构

include/linux/vmalloc.h

vm_struct 该结构用于管理一个内存区域

```
26 struct vm_struct {
27         struct vm_struct       *next;
28         void                   *addr;
29         unsigned long           size;
30         unsigned long           flags;
31         struct page           **pages;
32         unsigned int            nr_pages;
33         unsigned long           phys_addr;
34         void                   *caller;
```

```
35 };
```

- next:指向下一个 vm_struct 结构。
- addr:该虚拟内存区域的起始地址。
- size：内存区域的大小加 4K。
- flags：非连续内存区映射的内存类型。
  - VM_ALLOC 表示使用 vmalloc()得到的页；
  - VM_MAP 表示使用 vmap()映射的已经被分配的页；
  - VM_IOREMAP 表示使用 ioremap()映射的硬件设备的板上内存。
- pages：指向 nr_pages 个元素的内存指针，每一个元素为一个 struct page *的指针。
- nr_pages：该内存区域的页面个数。
- pyhs_addr：该字段为 0，除非内存已被创建来映射一个硬件设备的 I/O 共享内存。
- caller：todo

mm/vmalloc.c

vmap_area:红黑树管理内存区间。

```
253 struct vmap_area {
254         unsigned long va_start;
255         unsigned long va_end;
256         unsigned long flags;
257         struct rb_node rb_node;         /* address sorted rbtree */
258         struct list_head list;          /* address sorted list */
259         struct list_head purge_list;    /* "lazy purge" list */
260         void *private;
261         struct rcu_head rcu_head;
262 };
```

va_start：虚拟空间其实地址

va_end：虚拟空间终止地址

flags:

rb_node:红黑树节点

list：

purge_list

private:

rcu_head:

2.4.1.2 非连续内存区域初始化

在内核初始化过程中，会初始化和非连续内存区域相关的数据结构。其调用过程为

start_kernel->mm_init->vmalloc_init

mm/vmalloc.c

```
1088 void __init vmalloc_init(void)
1089 {
1090        struct vmap_area *va;
1091        struct vm_struct *tmp;
1092        int i;
1093
1094        for_each_possible_cpu(i) {
1095                struct vmap_block_queue *vbq;
1096
1097                vbq = &per_cpu(vmap_block_queue, i);
1098                spin_lock_init(&vbq->lock);
1099                INIT_LIST_HEAD(&vbq->free);
1100        }
1101
1102        /* Import existing vmlist entries. */
1103        for (tmp = vmlist; tmp; tmp = tmp->next) {
1104                va = kzalloc(sizeof(struct vmap_area), GFP_NOWAIT);
1105                va->flags = tmp->flags | VM_VM_AREA;
1106                va->va_start = (unsigned long)tmp->addr;
1107                va->va_end = va->va_start + tmp->size;
1108                __insert_vmap_area(va);
1109        }
1110
1111        vmap_area_pcpu_hole = VMALLOC_END;
```

```
1112
1113        vmap_initialized = true;
1114 }
```

- 1103-1109：将每一个 vm_struct 区域添加到红黑树中。
- 1113：标记 vmalloc 已经初始化。

vmlist 是一个全局变量，在 vmalloc 中定义如下。

struct vm_struct *vmlist;

vmlist 作为非连续内存分配区域的表头，记录了所有的非连续内存分配区域。

```
288 static void __insert_vmap_area(struct vmap_area *va)
 289 {
 290        struct rb_node **p = &vmap_area_root.rb_node;
 291        struct rb_node *parent = NULL;
 292        struct rb_node *tmp;
 293
 294        while (*p) {
 295                struct vmap_area *tmp;
 296
 297                parent = *p;
 298                tmp = rb_entry(parent, struct vmap_area, rb_node);
 299                if (va->va_start < tmp->va_end)
 300                        p = &(*p)->rb_left;
 301                else if (va->va_end > tmp->va_start)
 302                        p = &(*p)->rb_right;
 303                else
 304                        BUG();
 305        }
 306
 307        rb_link_node(&va->rb_node, parent, p);
```

```
308         rb_insert_color(&va->rb_node, &vmap_area_root);
309
310         /* address-sort this list so it is usable like the vmlist */
311         tmp = rb_prev(&va->rb_node);
312         if (tmp) {
313                 struct vmap_area *prev;
314                 prev = rb_entry(tmp, struct vmap_area, rb_node);
315                 list_add_rcu(&va->list, &prev->list);
316         } else
317                 list_add_rcu(&va->list, &vmap_area_list);
318 }
```

vmap_area_root 是一个红黑树的根节点，用于加速 vmlist 的查找、分配等。其定义如下：

static struct rb_root vmap_area_root = RB_ROOT;

2.4.1.3 分配与释放

2.4.1.3.1 vmalloc

vmalloc 函数给内核分配一个非连续内存区域。参数 size 表示所请求内存区的大小。

```
1586 /**
1587  *  vmalloc  -  allocate virtually contiguous memory
1588  *  @size:        allocation size
1589  *  Allocate enough pages to cover @size from the page level
1590  *  allocator and map them into contiguous kernel virtual space.
1591  *
1592  *  For tight control over page level allocator and protection flags
1593  *  use __vmalloc() instead.
1594  */
1595 void *vmalloc(unsigned long size)
1596 {
```

```
1597        return __vmalloc_node(size, 1, GFP_KERNEL | __GFP_HIGHMEM,
PAGE_KERNEL,
1598                              -1, __builtin_return_address(0));
1599 }
1600 EXPORT_SYMBOL(vmalloc);
```

vmalloc 调用__vmalloc_node 进行实际的操作。

```
1536 /**
1537  *  __vmalloc_node  -  allocate virtually contiguous memory
1538  *  @size:          allocation size
1539  *  @align:         desired alignment
1540  *  @gfp_mask:      flags for the page level allocator
1541  *  @prot:          protection mask for the allocated pages
1542  *  @node:          node to use for allocation or -1
1543  *  @caller:        caller's return address
1544  *
1545  *  Allocate enough pages to cover @size from the page level
1546  *  allocator with @gfp_mask flags.  Map them into contiguous
1547  *  kernel virtual space, using a pagetable protection of @prot.
1548  */
1549 static void *__vmalloc_node(unsigned long size, unsigned long align,
1550                             gfp_t gfp_mask, pgprot_t prot,
1551                             int node, void *caller)
1552 {
1553         struct vm_struct *area;
1554         void *addr;
1555         unsigned long real_size = size;
1556
1557         size = PAGE_ALIGN(size);
1558         if (!size || (size >> PAGE_SHIFT) > totalram_pages)
1559                 return NULL;
1560
```

```
1561        area = __get_vm_area_node(size, align, VM_ALLOC,
VMALLOC_START,
1562                                  VMALLOC_END, node, gfp_mask, caller);
1563
1564        if (!area)
1565                return NULL;
1566
1567        addr = __vmalloc_area_node(area, gfp_mask, prot, node, caller);
1568
1569        /*
1570         * A ref_count = 3 is needed because the vm_struct and vmap_area
1571         * structures allocated in the __get_vm_area_node() function
contain
1572         * references to the virtual address of the vmalloc'ed block.
1573         */
1574        kmemleak_alloc(addr, real_size, 3, gfp_mask);
1575
1576        return addr;
1577 }
```

- 1557：size 页面对齐。
- 1561：分配虚拟地址空间
- 1567：为对应的虚拟地址空间分配物理页面，并且建立映射关系。

```
1219 static struct vm_struct *__get_vm_area_node(unsigned long size,
1220                unsigned long align, unsigned long flags, unsigned long start,
1221                unsigned long end, int node, gfp_t gfp_mask, void *caller)
1222 {
1223        static struct vmap_area *va;
1224        struct vm_struct *area;
1225
1226        BUG_ON(in_interrupt());
1227        if (flags & VM_IOREMAP) {
```

```
1228            int bit = fls(size);
1229
1230            if (bit > IOREMAP_MAX_ORDER)
1231                    bit = IOREMAP_MAX_ORDER;
1232            else if (bit < PAGE_SHIFT)
1233                    bit = PAGE_SHIFT;
1234
1235            align = 1ul << bit;
1236    }
1237
1238    size = PAGE_ALIGN(size);
1239    if (unlikely(!size))
1240            return NULL;
1241
1242    area = kzalloc_node(sizeof(*area), gfp_mask &
GFP_RECLAIM_MASK, node);
1243    if (unlikely(!area))
1244            return NULL;
1245
1246    /*
1247     * We always allocate a guard page.
1248     */
1249    size += PAGE_SIZE;
1250
1251    va = alloc_vmap_area(size, align, start, end, node, gfp_mask);
1252    if (IS_ERR(va)) {
1253            kfree(area);
1254            return NULL;
1255    }
1256
1257    insert_vmalloc_vm(area, va, flags, caller);
1258    return area;
```

```
1259 }
```

- 1249：size 加上 4k，用于表示两个区域之间的间隙。
- 1251：使用红黑树分配一个 vmap_area 区域。
- 1257：将 vmap_area 转换为 vm_stract 结构，插入到链表中。

```
1469 static void *__vmalloc_area_node(struct vm_struct *area, gfp_t
gfp_mask,
1470                                   pgprot_t prot, int node, void *caller)
1471 {
1472         struct page **pages;
1473         unsigned int nr_pages, array_size, i;
1474         gfp_t nested_gfp = (gfp_mask & GFP_RECLAIM_MASK) |
__GFP_ZERO;
1475
1476         nr_pages = (area->size - PAGE_SIZE) >> PAGE_SHIFT;
1477         array_size = (nr_pages * sizeof(struct page *));
1478
1479         area->nr_pages = nr_pages;
1480         /* Please note that the recursion is strictly bounded. */
1481         if (array_size > PAGE_SIZE) {
1482                 pages = __vmalloc_node(array_size, 1,
nested_gfp|__GFP_HIGHMEM,
1483                                 PAGE_KERNEL, node, caller);
1484                 area->flags |= VM_VPAGES;
1485         } else {
1486                 pages = kmalloc_node(array_size, nested_gfp, node);
1487         }
1488         area->pages = pages;
1489         area->caller = caller;
1490         if (!area->pages) {
1491                 remove_vm_area(area->addr);
1492                 kfree(area);
```

```
1493            return NULL;
1494        }
1495
1496        for (i = 0; i < area->nr_pages; i++) {
1497            struct page *page;
1498
1499            if (node < 0)
1500                page = alloc_page(gfp_mask);
1501            else
1502                page = alloc_pages_node(node, gfp_mask, 0);
1503
1504            if (unlikely(!page)) {
1505                /* Successfully allocated i pages, free them in __vunmap()
*/
1506                area->nr_pages = i;
1507                goto fail;
1508            }
1509            area->pages[i] = page;
1510        }
1511
1512        if (map_vm_area(area, prot, &pages))
1513            goto fail;
1514        return area->addr;
1515
1516 fail:
1517        vfree(area->addr);
1518        return NULL;
1519 }
```

- 1476-1479：计算 pages 数组的大小。
- 1482：如果 pages 数组大小大于一个页面，则通过 __vmalloc_node 分配内存空间，

- 1486：如果 pages 数组小于一个页面，则通过 kmalloc_node 分配空间。
- 1496-1510：循环为每一个页面的虚拟地址空间分配物理页面。并将对应的物理页面 page 结构保存在 pages 数组中。
- 1512：建立虚拟地址和物理地址之间的映射关系。

2.4.1.3.2 vfree

释放通过 vmalloc，vmalloc_32 分配的内存。

```
1395 /**
1396  *  vfree  -  release memory allocated by vmalloc()
1397  *  @addr:        memory base address
1398  *
1399  *  Free the virtually continuous memory area starting at @addr, as
1400  *  obtained from vmalloc(), vmalloc_32() or __vmalloc(). If @addr is
1401  *  NULL, no operation is performed.
1402  *
1403  *  Must not be called in interrupt context.
1404  */
1405 void vfree(const void *addr)
1406 {
1407       BUG_ON(in_interrupt());
1408
1409       kmemleak_free(addr);
1410
1411       __vunmap(addr, 1);
1412 }
1413 EXPORT_SYMBOL(vfree);
```

__vunmap 完成实质性的工作。

```
1353 static void __vunmap(const void *addr, int deallocate_pages)
1354 {
```

```
1355        struct vm_struct *area;
1356
1357        if (!addr)
1358                return;
1359
1360        if ((PAGE_SIZE-1) & (unsigned long)addr) {
1361                WARN(1, KERN_ERR "Trying to vfree() bad address (%p)\n",
addr);
1362                return;
1363        }
1364
1365        area = remove_vm_area(addr);
1366        if (unlikely(!area)) {
1367                WARN(1, KERN_ERR "Trying to vfree() nonexistent vm area
(%p)\n",
1368                                addr);
1369                return;
1370        }
1371
1372        debug_check_no_locks_freed(addr, area->size);
1373        debug_check_no_obj_freed(addr, area->size);
1374
1375        if (deallocate_pages) {
1376                int i;
1377
1378                for (i = 0; i < area->nr_pages; i++) {
1379                        struct page *page = area->pages[i];
1380
1381                        BUG_ON(!page);
1382                        __free_page(page);
1383                }
1384
```

```
1385            if (area->flags & VM_VPAGES)
1386                    vfree(area->pages);
1387            else
1388                    kfree(area->pages);
1389    }
1390
1391    kfree(area);
1392    return;
1393 }
```

- 1365：从 vmlist 和红黑树中去掉内存区域。
- 1378-1383：释放该虚拟内存区域对应的物理内存到伙伴系统中。
- 1385-1387：根据 pages 是通过 vmalloc 分配方式还是通过 kmalloc 分配方式分配的，选择不同的释放函数来释放 pages。
- 1391：释放 area 数据结构占用的内存空间。

2.4.2 永久内核映射

如果需要将高端页帧长期映射到内核地址空间中，需要使用 kmap 函数。需要映射的页用 page 指定，作为该函数的参数。该函数在必要时创建一个映射，并返回一个虚拟地址。

如果没有启用高端支持，该函数比较简单，使用内核线性映射即可。

如果确实存在高端页，则必须使用永久内核映射的方式进行处理。

永久内核映射使用主内核页表中的一个专门的页表，该页表的地址存放在全局变量 pkmap_page_table 中，在内核初始化中已经讲过。

2.4.2.1 数据结构

页表项使用记录数据结构 pkmap_count

```
61 static int pkmap_count[LAST_PKMAP];
```

```
62 static unsigned int last_pkmap_nr;
63 static    __cacheline_aligned_in_smp DEFINE_SPINLOCK(kmap_lock);
64
65 pte_t * pkmap_page_table;
```

pkmap_count 数组是一个容量为 LAST_PKMAP 的数组，每个元素对应于一个持久映射页。该元素值的含义如下：

0：对应的页表项没有映射任何高端内存页框，可以使用。

1：对应的页表项没有映射任何高端内存页面，但是不能使用。因为自从它最后一次使用以来，其相应的 TLB 表项还没有被刷新。

计数器 n（n>1）:相应的页表项映射有一个高端内存页面，并且内核中有 n-1 处使用该页。

为记录高端内存页面与永久内核映射包含的线性地址之间的联系，内核使用了 page_address_htable 散列表。该表包含了一个 page_address_map 数据结构。

```
300 struct page_address_map {
301         struct page *page;
302         void *virtual;
303         struct list_head list;
304 };
```

page：对应页框的 page 结构

virtual：虚拟地址空间

list：指向以一个结构实例

```
312 /*
313  * Hash table bucket
314  */
315 static struct page_address_slot {
316         struct list_head lh;                    /* List of page_address_maps */
317         spinlock_t lock;                  /* Protect this bucket's list */
318 } ____cacheline_aligned_in_smp
page_address_htable[1<<PA_HASH_ORDER];
```

lh：指向 page_address_maps。

2.4.2.2 通过page查找永久内核虚拟地址

Page_address 为 page 页面分配在永久内核的虚拟地址空间中查找一个空闲的表项目。并区分如下情况：

如果 page 不在高端内存中，则通过线性映射返回 page 对应的虚拟地址。

如果 page 在高端内存中，函数就到 page_address_htable 散列表中查找。如果在散列表中找到页面，page_address()就返回它的线性地址，否则返回 NULL。当然，得到了这个线性地址，我们就同意可以通过__pa 宏得到其对应的物理地址。

```
325 /**
326  * page_address - get the mapped virtual address of a page
327  * @page: &struct page to get the virtual address of
328  *
329  * Returns the page's virtual address.
330  */
331 void *page_address(struct page *page)
332 {
333         unsigned long flags;
334         void *ret;
335         struct page_address_slot *pas;
336
337         if (!PageHighMem(page))
338                 return lowmem_page_address(page);
339
340         pas = page_slot(page);
341         ret = NULL;
342         spin_lock_irqsave(&pas->lock, flags);
343         if (!list_empty(&pas->lh)) {
344                 struct page_address_map *pam;
345
346                 list_for_each_entry(pam, &pas->lh, list) {
347                         if (pam->page == page) {
```

```
348                         ret = pam->virtual;
349                         goto done;
350                 }
351             }
352         }
353 done:
354         spin_unlock_irqrestore(&pas->lock, flags);
355         return ret;
356 }
```

2.4.2.3 分配和释放

2.4.2.3.1 kmap

arch/x86/mm/highmem_32.c

```
 5 void *kmap(struct page *page)
 6 {
 7         might_sleep();
 8         if (!PageHighMem(page))
 9                 return page_address(page);
10         return kmap_high(page);
11 }
```

8-9：如果是非高端页面，直接通过线性地址返回。

10：如果是高端内存，调用 kmap_high 进行实际的操作。

```
189 /**
190  * kmap_high - map a highmem page into memory
191  * @page: &struct page to map
192  *
193  * Returns the page's virtual memory address.
194  *
195  * We cannot call this from interrupts, as it may block.
```

```
196   */
197 void *kmap_high(struct page *page)
198 {
199         unsigned long vaddr;
200
201         /*
202          * For highmem pages, we can't trust "virtual" until
203          * after we have the lock.
204          */
205         lock_kmap();
206         vaddr = (unsigned long)page_address(page);
207         if (!vaddr)
208                 vaddr = map_new_virtual(page);
209         pkmap_count[PKMAP_NR(vaddr)]++;
210         BUG_ON(pkmap_count[PKMAP_NR(vaddr)] < 2);
211         unlock_kmap();
212         return (void*) vaddr;
213 }
```

206：先从散列表中查找是否已经映射 page。

207-208：如果没有映射，调用 map_new_virtual 建立映射。

209：将虚拟地址对应的页表项引用计数加。

```
139 static inline unsigned long map_new_virtual(struct page *page)
140 {
141         unsigned long vaddr;
142         int count;
143
144 start:
145         count = LAST_PKMAP;
146         /* Find an empty entry */
147         for (;;) {
```

```
148             last_pkmap_nr = (last_pkmap_nr + 1) & LAST_PKMAP_MASK;
149             if (!last_pkmap_nr) {
150                     flush_all_zero_pkmaps();
151                     count = LAST_PKMAP;
152             }
153             if (!pkmap_count[last_pkmap_nr])
154                     break;    /* Found a usable entry */
155             if (--count)
156                     continue;
157
158             /*
159              * Sleep for somebody else to unmap their entries
160              */
161             {
162                     DECLARE_WAITQUEUE(wait, current);
163
164                     __set_current_state(TASK_UNINTERRUPTIBLE);
165                     add_wait_queue(&pkmap_map_wait, &wait);
166                     unlock_kmap();
167                     schedule();
168                     remove_wait_queue(&pkmap_map_wait, &wait);
169                     lock_kmap();
170
171                     /* Somebody else might have mapped it while we slept */
172                     if (page_address(page))
173                             return (unsigned long)page_address(page);
174
175                     /* Re-start */
176                     goto start;
177             }
178     }
```

```
179         vaddr = PKMAP_ADDR(last_pkmap_nr);
180         set_pte_at(&init_mm, vaddr,
181                 &(pkmap_page_table[last_pkmap_nr]), mk_pte(page,
kmap_prot));
182
183         pkmap_count[last_pkmap_nr] = 1;
184         set_page_address(page, (void *)vaddr);
185
186         return vaddr;
187 }
```

- 1513-1514：找到一个空闲的表项，跳出。
- 179：空闲表项对应的虚拟地址。
- 180：设置对应的页表项
- 183：引用计数加 1。
- 184：将页框加入到散列表中。

需要注意的是，上述函数是一个阻塞函数，当所有的永久内核虚拟地址空间都被映射了时，会进入 161-177 行进入睡眠状态，等待空闲的项出现。

2.4.2.3.2 kunmap

```
13 void kunmap(struct page *page)
14 {
15         if (in_interrupt())
16                 BUG();
17         if (!PageHighMem(page))
18                 return;
19         kunmap_high(page);
20 }
```

调用 kunmap_high 进行具体的工作。

```
243 /**
```

```
244   * kunmap_high - map a highmem page into memory
245   * @page: &struct page to unmap
246   *
247   * If ARCH_NEEDS_KMAP_HIGH_GET is not defined then this may be
called
248   * only from user context.
249   */
250 void kunmap_high(struct page *page)
251 {
252         unsigned long vaddr;
253         unsigned long nr;
254         unsigned long flags;
255         int need_wakeup;
256
257         lock_kmap_any(flags);
258         vaddr = (unsigned long)page_address(page);
259         BUG_ON(!vaddr);
260         nr = PKMAP_NR(vaddr);
261
262         /*
263          * A count must never go down to zero
264          * without a TLB flush!
265          */
266         need_wakeup = 0;
267         switch (--pkmap_count[nr]) {
268         case 0:
269                 BUG();
270         case 1:
271                 /*
272                  * Avoid an unnecessary wake_up() function call.
273                  * The common case is pkmap_count[] == 1, but
274                  * no waiters.
```

```
275                 * The tasks queued in the wait-queue are guarded
276                 * by both the lock in the wait-queue-head and by
277                 * the kmap_lock.    As the kmap_lock is held here,
278                 * no need for the wait-queue-head's lock.    Simply
279                 * test if the queue is empty.
280                 */
281                need_wakeup = waitqueue_active(&pkmap_map_wait);
282        }
283        unlock_kmap_any(flags);
284
285        /* do wake-up, if needed, race-free outside of the spin lock */
286        if (need_wakeup)
287                wake_up(&pkmap_map_wait);
288 }
```

- 258：有 page 找到虚拟地址
- 260：由虚拟地址找到索引号
- 267 行：将 pkmap_count 对应的项目减一。

2.4.3 临时内核映射

刚才可以看到，kmap 函数不能用于中断处理程序，因为它可能进入睡眠状态。内核提供了一个备选映射函数，其执行是原子，成为 kmap_atomic。

2.4.3.1 分配与释放

2.4.3.1.1 kmap_atomic

```
51 void *kmap_atomic(struct page *page, enum km_type type)
52 {
53        return kmap_atomic_prot(page, type, kmap_prot);
54 }
```

```
22 /*
```

```
23   * kmap_atomic/kunmap_atomic is significantly faster than kmap/kunmap
because
24   * no global lock is needed and because the kmap code must perform a
global TLB
25   * invalidation when the kmap pool wraps.
26   *
27   * However when holding an atomic kmap it is not legal to sleep, so
atomic
28   * kmaps are appropriate for short, tight code paths only.
29   */
30 void *kmap_atomic_prot(struct page *page, enum km_type type, pgprot_t
prot)
31 {
32       enum fixed_addresses idx;
33       unsigned long vaddr;
34
35       /* even !CONFIG_PREEMPT needs this, for in_atomic in
do_page_fault */
36       pagefault_disable();
37
38       if (!PageHighMem(page))
39             return page_address(page);
40
41       debug_kmap_atomic(type);
42
43       idx = type + KM_TYPE_NR*smp_processor_id();
44       vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);
45       BUG_ON(!pte_none(*(kmap_pte-idx)));
46       set_pte(kmap_pte-idx, mk_pte(page, prot));
47
48       return (void *)vaddr;
49 }
```

38-39 行：非高端内存，直接通过线性地址映射。

43-44 行：通过 km_type 得到虚拟地址。

46 行：设置对应的页表项。

2.4.3.1.2 kunmap_atomic

```
56 void kunmap_atomic(void *kvaddr, enum km_type type)
57 {
58        unsigned long vaddr = (unsigned long) kvaddr & PAGE_MASK;
59        enum fixed_addresses idx = type +
KM_TYPE_NR*smp_processor_id();
60
61        /*
62         * Force other mappings to Oops if they'll try to access this pte
63         * without first remap it.    Keeping stale mappings around is a bad
idea
64         * also, in case the page changes cacheability attributes or becomes
65         * a protected page in a hypervisor.
66         */
67        if (vaddr == __fix_to_virt(FIX_KMAP_BEGIN+idx))
68              kpte_clear_flush(kmap_pte-idx, vaddr);
69        else {
70 #ifdef CONFIG_DEBUG_HIGHMEM
71              BUG_ON(vaddr < PAGE_OFFSET);
72              BUG_ON(vaddr >= (unsigned long)high_memory);
73 #endif
74        }
75
76        pagefault_enable();
77 }
```

2.5 伙伴系统

2.5.1 伙伴系统产生的原因

内核应该为分配一组连续的页框建立一种健壮、高效的分配策略。为此，需要解决著名的内存管理问题，也就是所谓的外部碎片。频繁地请求和释放不同大小的一组连续页框，必然导致在已经分配的块内分散了许多小块的空闲页表。由此带来的问题是，即使有足够的空闲页框可以满足请求，但要分配大块的连续页框可能无法满足。

本质上讲，避免外部碎片的方法可以有两种：

- 利用分页单元把一组非连续的空闲页框映射到连续的线性地址区间。（用户空间的地址分配正是如此）。
- 开发一种适当的技术用来记录现存的空闲连续页框的情况，以尽量避免为满足对小块的请求而分割大的空闲块。

基于以下三种原因，内核首选第二种方法：

- 在某些情况下，连续的页框确实是必须的，因为连续的线性地址不足以满足要求。典型例子是给 DMA 处理器分配缓冲区的内存请求时，因为当在一次单独的 I/O 操作中传送几个磁盘扇区的数据时，DMA 忽略分页单元而直接访问地址总线，因此，所请求的缓冲区必须位于连续的页框中。
- 即使连续页框的分配不是很重要，但它在保持内核页表不变方面所起的作用也是不容忽视的。频繁的修改页表势必导致平均访问内存次数增加，因为这会使得 CPU 频繁地刷新转换后援缓冲区（TLB）的内容。
- 内核通过 4M 的页可以加快访问大块连续的物理内存。这样减少了转换后援缓冲区的失效率，因此提高了访问内存的平均速度。

Linux 采用著名的伙伴系统（buddy system）算法来解决该外部碎片问题。伙伴系统基于一种相对简单但却效率惊人的算法，伴随 linux 内核几乎 40 年。它结合了优秀内存分配器的两个关键特征：速度和效率。

2.5.2 伙伴系统的算法原理

伙伴系统作用于内存区上。它将内存区域中所有的空闲页框分为 11 个块链表，每个块链表分别包含大小为 1，2，4，8，16，32，64，128，256，512，1024 个连续的页框。每个块的第一个页框的物理地址是该块大小的整数倍。

分配页框算法很简单：

假设请求 256 个页框的块。先在 256 个页框的链表中检测是否有空闲块。如果没有，则会在下一个更大的页块中查找。也就是，在 512 个页框的链表中查找。如果存在这样的空闲块，则内核将 512 的页框分为两等份，一个用作满足请求，一个插入到 256 个页框的链表中。如果 512 中没找到，就在 1024 中找。找到后，拿出 256 个满足需求，剩余的 768 分为两部分，512 个插入到 512 的链表中，256 的插入到 256 的链表中。如果 1024 链表中没有空闲的页框，算法放弃并发出出错信号。

页框释放算法是上述过程的逆过程：

内核将大小为 b 的一对空闲伙伴合并为大小为 2b 的单独块。满足以下条件的两个块称为伙伴：

- 两个块具有相同的大小
- 它们的物理地址连续
- 第一个块的第一个页框的物理地址是 2b 的倍数。

该算法是迭代的，如果它成功合并所释放的块，它会试图合并 2b 的块，以再次试图形成更大的块。

2.5.3 数据结构

伙伴系统是基于内存区域的。

```
280 struct zone {
……
314     struct free_area     free_area[MAX_ORDER];
……
418 } ____cacheline_internodealigned_in_smp;
```

```
57 struct free_area {
58     struct list_head     free_list[MIGRATE_TYPES];
```

```
59        unsigned long          nr_free;
60 };
```

nr_free 指定了当前内存区域中空闲页块的数目。

free_list 是用于链接空闲页的链表。页链表中包含了大小相同的连续内存区域。该变量是一个数组，为何是数组呢，下面仔细讲讲。

在内核版本 2.6.23 之前，free_list 就是一个双链表。但为了满足内核苛刻的条件，在后来 free_list 变成了数组，目的仍然是避免碎片。

2.5.4 防碎片数据结构

伙伴系统在 free_list 变成数组之前，工作的非常好。但在 linux 内存管理方面，有一个长期存在的问题：在系统启动并长期运行后，物理内存会产生很多碎片。

很长时间来，物理内存碎片的确是 linux 内核的弱点之一。在 linux 2.6.24 内核开发期间，防止碎片的方法最终加入内核。有一点需要明确，文件系统也有碎片，该领域的碎片问题主要是通过碎片合并工具解决，它们分析文件系统，重新排序已分配的存储块，从而建立较大的连续存储区。理论上，该方法对物理内存也是可能的，但由于许多物理内存页不能移动到任意位置，阻碍了该方法的实施。因此，内核反碎片的方法是从最初开始尽可能防止碎片。

内核将已分配页划分为如下三种类型：

- 不可移动页：在内存中固定的位置，不能移动到其他地方。核心内核分配的大多数内存属于该类别。
- 可回收页：不能直接移动，但可以删去，其内容可以从某些源重新生成。例如，映射自文件的数据属于该类别。kswapd 守护进程会根据页的访问频繁程度，周期性的释放此类内存。
- 可移动页：属于用户空间应用程序的页属于该类别。

注意，从最初开始，内存并未划分为可移动性不同的区，这些是在运行时动态生成的。

```
38 #define MIGRATE_UNMOVABLE         0
39 #define MIGRATE_RECLAIMABLE       1
```

```
40 #define MIGRATE_MOVABLE     2
41 #define MIGRATE_PCPTYPES    3 /* the number of types on the pcp lists */
42 #define MIGRATE_RESERVE     3
43 #define MIGRATE_ISOLATE     4 /* can't allocate from here */
44 #define MIGRATE_TYPES       5
```

MIGRATE_UNMOVABLE、MIGRATE_RECLAIMABLE、MIGRATE_MOVABLE 已经介绍。

MIGRATE_RESERVE 是保留的区域，在前面三种列表分配都失败后，可以从该列表分配。该列表的内容在内核初始化函数 setup_zone_migrate_reserve 中进行设置。

如果内核无法满足针对某一给定迁移类型的分配请求，将会如何。

内核提供了一个备用列表，规定了在无法满足请求的情况下，接下来该使用那种迁移类型。

```
765 /*
766  * This array describes the order lists are fallen back to when
767  * the free lists for the desirable migrate type are depleted
768  */
769 static int fallbacks[MIGRATE_TYPES][MIGRATE_TYPES-1] = {
770     [MIGRATE_UNMOVABLE]   = { MIGRATE_RECLAIMABLE, MIGRATE_MOVABLE,   MIGRATE_RESERVE },
771     [MIGRATE_RECLAIMABLE] = { MIGRATE_UNMOVABLE, MIGRATE_MOVABLE,   MIGRATE_RESERVE },
772     [MIGRATE_MOVABLE]     = { MIGRATE_RECLAIMABLE, MIGRATE_UNMOVABLE, MIGRATE_RESERVE },
773     [MIGRATE_RESERVE]     = { MIGRATE_RESERVE, MIGRATE_RESERVE,   MIGRATE_RESERVE }, /* Never used */
774 };
```

2.5.5 伙伴系统内存分配

2.5.5.1 分配器API

伙伴系统分配器有如下是个 API：

- alloc_pages(gfp_t gfp_mask, unsigned int order)分配 2^{order} 页并返回一个 struct page 的实例，表示分配内存的起始页。alloc_pages(gfp_t gfp_mask)是前者在 order=0 下的简化。
- get_zeroed_page(gfp_t gfp_mask)分配一页并返回 page 实例，页对应的内存填充 0.
- __get_free_pages(gfp_t gfp_mask, unsigned int order)和 __get_free_pages(gfp_t gfp_mask)工作方式与 alloc_pages 相同，但返回分配内存块的虚拟地址，而不是 page 实例

上述所有的函数最终都会调用 alloc_pages_node 函数，该函数是伙伴系统的发射台。

2.5.5.2 辅助函数

首先定义一些函数的使用标志，用以控制到达各个水印时指定的临界状态时的行为。

```
1263 /* The ALLOC_WMARK bits are used as an index to zone->watermark */
1264 #define ALLOC_WMARK_MIN        WMARK_MIN
1265 #define ALLOC_WMARK_LOW        WMARK_LOW
1266 #define ALLOC_WMARK_HIGH       WMARK_HIGH
1267 #define ALLOC_NO_WATERMARKS 0x04 /* don't check watermarks at all */
1268
1269 /* Mask to get the watermark bits */
1270 #define ALLOC_WMARK_MASK       (ALLOC_NO_WATERMARKS-1)
1271
1272 #define ALLOC_HARDER            0x10 /* try to alloc harder */
1273 #define ALLOC_HIGH          0x20 /* __GFP_HIGH set */
1274 #define ALLOC_CPUSET            0x40 /* check for correct cpuset */
```

默认情况下，只有内存区域中包含的数目至少为 zone->page_high 时，才能分配页，这对应于 ALLOC_WMARK_HIGH。如果要用较低 zone->page_low 或者最低 zone->page_min 设置，则必须设置 ALLOC_WMARK_LOW，ALLOC_WMARK_MIN。ALLOC_HARDER 通知伙伴系统在急需内存时放宽条件。在分配高端内存域时，ALLOC_HIGH 进一步放宽限制。最后，ALLOC_CPUSET 告诉内核，内存只能从当前 CPU 允许运行的 CPU 相关联的内存节点分配，该选项只对 NUMA 有意义。

2.5.5.2.1 zone_watermark_ok函数

该函数根据设置的标志判断能否从给定的内存域分配内存。

```
1371 /*
1372  * Return 1 if free pages are above 'mark'. This takes into account the order
1373  * of the allocation.
1374  */
1375 int zone_watermark_ok(struct zone *z, int order, unsigned long mark,
1376                       int classzone_idx, int alloc_flags)
1377 {
1378         /* free_pages my go negative - that's OK */
1379         long min = mark;
1380         long free_pages = zone_page_state(z, NR_FREE_PAGES) - (1 << order) + 1;
1381         int o;
1382
1383         if (alloc_flags & ALLOC_HIGH)
1384                 min -= min / 2;
1385         if (alloc_flags & ALLOC_HARDER)
1386                 min -= min / 4;
1387
1388         if (free_pages <= min + z->lowmem_reserve[classzone_idx])
1389                 return 0;
1390         for (o = 0; o < order; o++) {
```

```
1391            /* At the next order, this order's pages become unavailable */
1392            free_pages -= z->free_area[o].nr_free << o;
1393
1394            /* Require fewer higher order pages to be free */
1395            min >>= 1;
1396
1397            if (free_pages <= min)
1398                    return 0;
1399    }
1400    return 1;
1401 }
```

1380：获得指定内存区域的空闲页面数。

1383-1386：在解释了 ALLOC_HIGH，ALLOC_HARDER 标志之后（将最小标记降低到当前值的一半或者四分之三）。

1388-1389 行：如果空闲的页面数比最小值和 lowmem_reserve 指定的紧急分配值之后小，则不允许分配。

1390-1399 行：否则，代码遍历所有小于当前阶的分配阶，从 free_pages 中减去当前分配阶的所有空闲页，同时，所需空闲页的最小值减半，如果 free_pages 小于 min，则不允许分配。

2.5.5.2.2 get_page_from_freelist

该函数是伙伴系统使用的一个重要的辅助函数。它通过标志集和分配阶来判断可否进行分配，如果可以，则发起实际的操作。

```
1523 /*
1524  * get_page_from_freelist goes through the zonelist trying to allocate
1525  * a page.
1526  */
1527 static struct page *
1528 get_page_from_freelist(gfp_t gfp_mask, nodemask_t *nodemask, unsigned int order,
1529            struct zonelist *zonelist, int high_zoneidx, int alloc_flags,
1530            struct zone *preferred_zone, int migratetype)
1531 {
```

```
1532        struct zoneref *z;
1533        struct page *page = NULL;
1534        int classzone_idx;
1535        struct zone *zone;
1536        nodemask_t *allowednodes = NULL;/* zonelist_cache approximation */
1537        int zlc_active = 0;         /* set if using zonelist_cache */
1538        int did_zlc_setup = 0;          /* just call zlc_setup() one time */
1539
1540        classzone_idx = zone_idx(preferred_zone);
1541 zonelist_scan:
1542        /*
1543         * Scan zonelist, looking for a zone with enough free.
1544         * See also cpuset_zone_allowed() comment in kernel/cpuset.c.
1545         */
1546        for_each_zone_zonelist_nodemask(zone, z, zonelist,
1547                                        high_zoneidx, nodemask) {
1548                if (NUMA_BUILD && zlc_active &&
1549                        !zlc_zone_worth_trying(zonelist, z, allowednodes))
1550                                continue;
1551                if ((alloc_flags & ALLOC_CPUSET) &&
1552                        !cpuset_zone_allowed_softwall(zone, gfp_mask))
1553                                goto try_next_zone;
1554
1555                BUILD_BUG_ON(ALLOC_NO_WATERMARKS < NR_WMARK);
1556                if (!(alloc_flags & ALLOC_NO_WATERMARKS)) {
1557                        unsigned long mark;
1558                        int ret;
1559
1560                        mark = zone->watermark[alloc_flags & ALLOC_WMARK_MASK];
1561                        if (zone_watermark_ok(zone, order, mark,
1562                                    classzone_idx, alloc_flags))
```

```
1563                    goto try_this_zone;
1564
1565            if (zone_reclaim_mode == 0)
1566                    goto this_zone_full;
1567
1568            ret = zone_reclaim(zone, gfp_mask, order);
1569            switch (ret) {
1570            case ZONE_RECLAIM_NOSCAN:
1571                    /* did not scan */
1572                    goto try_next_zone;
1573            case ZONE_RECLAIM_FULL:
1574                    /* scanned but unreclaimable */
1575                    goto this_zone_full;
1576            default:
1577                    /* did we reclaim enough */
1578                    if (!zone_watermark_ok(zone, order, mark,
1579                                    classzone_idx, alloc_flags))
1580                            goto this_zone_full;
1581            }
1582    }
1583
1584 try_this_zone:
1585    page = buffered_rmqueue(preferred_zone, zone, order,
1586                            gfp_mask, migratetype);
1587    if (page)
1588            break;
1589 this_zone_full:
1590    if (NUMA_BUILD)
1591            zlc_mark_zone_full(zonelist, z);
1592 try_next_zone:
1593    if (NUMA_BUILD && !did_zlc_setup && nr_online_nodes > 1) {
```

```
1594                 /*
1595                  * we do zlc_setup after the first zone is tried but only
1596                  * if there are multiple nodes make it worthwhile
1597                  */
1598                 allowednodes = zlc_setup(zonelist, alloc_flags);
1599                 zlc_active = 1;
1600                 did_zlc_setup = 1;
1601             }
1602         }
1603
1604     if (unlikely(NUMA_BUILD && page == NULL && zlc_active)) {
1605             /* Disable zlc cache for second zonelist scan */
1606             zlc_active = 0;
1607             goto zonelist_scan;
1608         }
1609         return page;
1610 }
```

1546-1547 行：循环遍历该内存区域备用分配列表中的每一个内存区域。

1561-1563：检测该区域是否有足够的空闲页面，如果有则跳转到 1585

1568：如果区域没有足够的空闲页面，检测 zone_reclaim_mode 是否为 1，如果是，则调用 zone_reclaim 来回收内存。

1570-1581：根据 zone_reclaim 返回值，确定跳转的目标。

1585-1586：调用 buffered_rmqueue 进行实质性的内存分配。

1587-1588：如果分配成功，则退出循环。

1590-1601：根据前面的运行结果，设置相应的标志位。

1604-1608：确定是否进行第二次扫描。

2.5.5.3 伙伴系统启动函数

alloc_pages_node

```
281 static inline struct page *alloc_pages_node(int nid, gfp_t gfp_mask,
```

```
282                                     unsigned int order)
283 {
284         /* Unknown node is current node */
285         if (nid < 0)
286                 nid = numa_node_id();
287
288         return __alloc_pages(gfp_mask, order, node_zonelist(nid,
gfp_mask));
289 }
```

```
274 static inline struct page *
275 __alloc_pages(gfp_t gfp_mask, unsigned int order,
276                 struct zonelist *zonelist)
277 {
278         return __alloc_pages_nodemask(gfp_mask, order, zonelist, NULL);
279 }
```

下面的函数是伙伴系统的心脏。

```
1944 /*
1945  * This is the 'heart' of the zoned buddy allocator.
1946  */
1947 struct page *
1948 __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
1949                         struct zonelist *zonelist, nodemask_t *nodemask)
1950 {
1951         enum zone_type high_zoneidx = gfp_zone(gfp_mask);
1952         struct zone *preferred_zone;
1953         struct page *page;
1954         int migratetype = allocflags_to_migratetype(gfp_mask);
1955
1956         gfp_mask &= gfp_allowed_mask;
```

```
1957
1958        lockdep_trace_alloc(gfp_mask);
1959
1960        might_sleep_if(gfp_mask & __GFP_WAIT);
1961
1962        if (should_fail_alloc_page(gfp_mask, order))
1963                return NULL;
1964
1965        /*
1966         * Check the zones suitable for the gfp_mask contain at least one
1967         * valid zone. It's possible to have an empty zonelist as a result
1968         * of GFP_THISNODE and a memoryless node
1969         */
1970        if (unlikely(!zonelist->_zonerefs->zone))
1971                return NULL;
1972
1973        /* The preferred zone is used for statistics later */
1974        first_zones_zonelist(zonelist, high_zoneidx, nodemask,
&preferred_zone);
1975        if (!preferred_zone)
1976                return NULL;
1977
1978        /* First allocation attempt */
1979        page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL,
nodemask, order,
1980                        zonelist, high_zoneidx,
ALLOC_WMARK_LOW|ALLOC_CPUSET,
1981                        preferred_zone, migratetype);
1982        if (unlikely(!page))
1983                page = __alloc_pages_slowpath(gfp_mask, order,
1984                                zonelist, high_zoneidx, nodemask,
1985                                preferred_zone, migratetype);
```

```
1986
1987        trace_mm_page_alloc(page, order, gfp_mask, migratetype);
1988        return page;
1989 }
1990 EXPORT_SYMBOL(__alloc_pages_nodemask);
```

该函数的逻辑很清楚，先使用 get_page_from_freelist 进入快速分配内存路径，如果失败，则进入慢速分配路径。

1979-1981：进入快速分配路径

1983-1985：进入慢速分配路径。

在快速分配失败的情况下，进入慢速分配路径。

进入慢速分配路径，意味着内存剩余空间不是很多，内核需要加大分配力度。

内核遍历备用列表中的每一个内存区域，调用 wakeup_kswapd。该函数会唤醒负责换出页的 kswapd 守护进程。（1843-1844）

在交互守护进程唤醒后，内核进行更积极的尝试，修改一些当前特定情况下更有可能分配成功的标志，设置分配标志将水印降低到最小值进行分配。（1854-1856）

在上面的努力失败后，内核检测当前的分配是否允许设置 ALLOC_NO_WATERMARKS，如果允许，则调用 __alloc_pages_high_priority 进行无水印标志分配。（1862-1865）。如果忽略水印也分配失败，但设置了__GFP_NOFAIL 标志，内核会进入无线循环，首先等待（通过 congestion_wait）块设备层结束占线。接下来再次尝试，直至成功。（1752-1753）

如果上面过程失败，内核是原子分配，或者设置了 PF_MEMALLOC，则分配内存失败。（1871-1876）。否则进入下面的过程。

调用__alloc_pages_direct_reclaim 函数，进入真的慢分配路径。

内核通过 cond_resched 函数重调度的时机，防止花费过多的时间分配内存，导致其他进程处于饥渴状态。（1707）。分页机制提供了目前很少使用的一个选项，将很少使用的页换出到块介质，以便有更多的物理内存空间。但改选项非常耗时，可能导致进程进入睡眠状态。try_to_free_pages 是该过程的辅助函数，用于查找当前不急需的页，以便换出。在该分配设置了 PF_MEMALLOC 标志后，会掉调用该函数，用于向其它内核代码表明后续的内存分配都这样搜索（1711）。

try_to_free_pages（1716）被设置（1711）/清除（1720）标志的代码间隔起来，try_to_free_pages 自身也需要获得内存，由于要获得新内存还需要额外分配一点内存（相当矛盾的情形），该函数在内存管理方面享有最高优先级。上述标志设置达到了这一目的，设置 PF_MEMALLOC 分配标志的内存分配非常积极。该标志的设置，也保证了 try_to_free_pages 不会递归调用。如果此前设置了 PF_MEMALLOC，那么__alloc_pages 已经返回。

如果需要多页，per-cpu 变量也会拿回到伙伴系统。（1725）

通过上面的努力后，在试图分配内存。如果这次努力还是失败，并且内核可能执行影响 VFS 层的调用而又没有设置 GFP_NORETRY 标志，同时 oom_killer_disabled 为假，则进行 OOM Killer。（1895-1902）。这里不讨论 oom 的实现细节。请注意，该机制选择一个内核认为分配了过多内存的进程，并杀死该进程。这有很大的几率获得较多的空闲页。但杀死一个进程未必立即出现多余 $2^{PAGE\_ALLOC\_COSTLY\_ORDER}$ 页的连续内存，其中 PAGE_ALLOC_COSTLY_ORDER 为 3，如果要分配如此大的内存区（1876-1877），那么内核不会杀死该进程，而是承认分配内存失败。否则的话，跳转到 restart（1916），重新扫描分配内存。

如果内核没有设置__GFP_NORETRY（1617），并且

- 分配内存小于 8 个页面（1625）或者
- 分配标志设置了__GFP_REPEAT 且 pages_reclaimed < (1 << order))（1635）或者
- 分配标志设置了__GFP_NOFAIL（1642）

则内核会跳转到 rebalance（1922-1925）重新扫描分配内存。

```
1808 static inline struct page *
1809 __alloc_pages_slowpath(gfp_t gfp_mask, unsigned int order,
1810         struct zonelist *zonelist, enum zone_type high_zoneidx,
1811         nodemask_t *nodemask, struct zone *preferred_zone,
1812         int migratetype)
1813 {
1814         const gfp_t wait = gfp_mask & __GFP_WAIT;
1815         struct page *page = NULL;
1816         int alloc_flags;
```

```
1817        unsigned long pages_reclaimed = 0;
1818        unsigned long did_some_progress;
1819        struct task_struct *p = current;
1820
1821        /*
1822         * In the slowpath, we sanity check order to avoid ever trying to
1823         * reclaim >= MAX_ORDER areas which will never succeed. Callers
may
1824         * be using allocators in order of preference for an area that is
1825         * too large.
1826         */
1827        if (order >= MAX_ORDER) {
1828                WARN_ON_ONCE(!(gfp_mask & __GFP_NOWARN));
1829                return NULL;
1830        }
1831
1832        /*
1833         * GFP_THISNODE (meaning __GFP_THISNODE,
__GFP_NORETRY and
1834         * __GFP_NOWARN set) should not cause reclaim since the
subsystem
1835         * (f.e. slab) using GFP_THISNODE may choose to trigger reclaim
1836         * using a larger set of nodes after it has established that the
1837         * allowed per node queues are empty and that nodes are
1838         * over allocated.
1839         */
1840        if (NUMA_BUILD && (gfp_mask & GFP_THISNODE) ==
GFP_THISNODE)
1841                goto nopage;
1842
1843 restart:
1844        wake_all_kswapd(order, zonelist, high_zoneidx);
```

1843-1844 行：遍历所有的内存区域，对每一个区域调用 wakeup_kswapd。

```
1845
1846        /*
1847         * OK, we're below the kswapd watermark and have kicked
background
1848         * reclaim. Now things get more complex, so set up alloc_flags
according
1849         * to how we want to proceed.
1850         */
1851        alloc_flags = gfp_to_alloc_flags(gfp_mask);
1852
1853        /* This is the last chance, in general, before the goto nopage. */
1854        page = get_page_from_freelist(gfp_mask, nodemask, order,
zonelist,
1855                high_zoneidx, alloc_flags &
~ALLOC_NO_WATERMARKS,
1856                preferred_zone, migratetype);
```

1851-1856：设置一些在该特定环境下更容易分配成功的标志，调用 get_page_from_freelist 尝试内存的分配。

```
1857        if (page)
1858            goto got_pg;
1859
1860 rebalance:
1861        /* Allocate without watermarks if the context allows */
1862        if (alloc_flags & ALLOC_NO_WATERMARKS) {
1863            page = __alloc_pages_high_priority(gfp_mask, order,
1864                    zonelist, high_zoneidx, nodemask,
1865                    preferred_zone, migratetype);
```

1862-1865：忽略水位标志，加强分配力度进行内存分配。

```
1866            if (page)
1867                goto got_pg;
```

```
1868        }
1869
1870        /* Atomic allocations - we can't balance anything */
1871        if (!wait)
1872                goto nopage;
1873
1874        /* Avoid recursion of direct reclaim */
1875        if (p->flags & PF_MEMALLOC)
1876                goto nopage;
```

1871-1876 行：如果是原子分配，且设置了 PF_MEMALLOC 分配标志，到现在还没分配成功的话，则放弃进一步的尝试，分配内存失败。

```
1877
1878        /* Avoid allocations with no watermarks from looping endlessly */
1879        if (test_thread_flag(TIF_MEMDIE) && !(gfp_mask &
__GFP_NOFAIL))
1880                goto nopage;
```

1879-1880 行：如果分配内存时间超市，且没有设置__GFP_NOFAIL 标志为，则则放弃进一步的尝试，分配内存失败。

```
1881
1882        /* Try direct reclaim and then allocating */
1883        page = __alloc_pages_direct_reclaim(gfp_mask, order,
1884                                        zonelist, high_zoneidx,
1885                                        nodemask,
1886                                        alloc_flags, preferred_zone,
1887                                        migratetype, &did_some_progress);
```

1883-1887 行：进入慢搜索路径，调用 try_to_free_pages 换出最近最少使用的页面，然后尝试分配内存。

```
1888        if (page)
1889                goto got_pg;
1890
1891        /*
```

```
1892         * If we failed to make any progress reclaiming, then we are
1893         * running out of options and have to consider going OOM
1894         */
1895        if (!did_some_progress) {
1896                if ((gfp_mask & __GFP_FS) && !(gfp_mask &
__GFP_NORETRY)) {
1897                        if (oom_killer_disabled)
1898                                goto nopage;
1899                        page = __alloc_pages_may_oom(gfp_mask, order,
1900                                        zonelist, high_zoneidx,
1901                                        nodemask, preferred_zone,
1902                                        migratetype);
```

1895-1902 行：如果条件满足，尝试杀掉一个进程，然后在进行内存分配。

```
1903                        if (page)
1904                                goto got_pg;
1905
1906                        /*
1907                         * The OOM killer does not trigger for high-order
1908                         * ~__GFP_NOFAIL allocations so if no progress is being
1909                         * made, there are no other options and retrying is
1910                         * unlikely to help.
1911                         */
1912                        if (order > PAGE_ALLOC_COSTLY_ORDER &&
1913                                        !(gfp_mask & __GFP_NOFAIL))
1914                                goto nopage;
1915
1916                        goto restart;
1917                }
1918        }
1919
1920        /* Check if we should retry the allocation */
```

```
1921        pages_reclaimed += did_some_progress;
1922        if (should_alloc_retry(gfp_mask, order, pages_reclaimed)) {
1923                /* Wait for some write requests to complete then retry */
1924                congestion_wait(BLK_RW_ASYNC, HZ/50);
1925                goto rebalance;
1926        }
```

1922-1926：检查是否需要重新进行内存分配扫描。

```
1927
1928 nopage:
1929        if (!(gfp_mask & __GFP_NOWARN) && printk_ratelimit()) {
1930                printk(KERN_WARNING "%s: page allocation failure."
1931                        " order:%d, mode:0x%x\n",
1932                        p->comm, order, gfp_mask);
1933                dump_stack();
1934                show_mem();
1935        }
1936        return page;
1937 got_pg:
1938        if (kmemcheck_enabled)
1939                kmemcheck_pagealloc_alloc(page, order, gfp_mask);
1940        return page;
1941
1942 }
```

2.5.5.3.1 __alloc_pages_high_priority

该函数忽略水印标记分配内存，如果分配标记设置了__GFP_NOFAIL，则会循环分配，直到成功为止。

```
1735 /*
1736  * This is called in the allocator slow-path if the allocation request is of
```

```
1737    * sufficient urgency to ignore watermarks and take other desperate
measures
1738    */
1739 static inline struct page *
1740 __alloc_pages_high_priority(gfp_t gfp_mask, unsigned int order,
1741        struct zonelist *zonelist, enum zone_type high_zoneidx,
1742        nodemask_t *nodemask, struct zone *preferred_zone,
1743        int migratetype)
1744 {
1745        struct page *page;
1746
1747        do {
1748              page = get_page_from_freelist(gfp_mask, nodemask, order,
1749                    zonelist, high_zoneidx, ALLOC_NO_WATERMARKS,
1750                    preferred_zone, migratetype);
1751
1752              if (!page && gfp_mask & __GFP_NOFAIL)
1753                    congestion_wait(BLK_RW_ASYNC, HZ/50);
1754        } while (!page && (gfp_mask & __GFP_NOFAIL));
1755
1756        return page;
1757 }
```

2.5.5.3.2 __alloc_pages_direct_reclaim

该函数真正的进入慢分配路径，内核通过 cond_resched 函数重调度的时机，防止花费过多的时间分配内存，导致其他进程处于饥渴状态。(1707)。分页机制提供了目前很少使用的一个选项，将很少使用的页换出到块介质，以便有更多的物理内存空间。但改选项非常耗时，可能导致进程进入睡眠状态。try_to_free_pages 是该过程的辅助函数，用于查找当前不急需的页，以便换出。在该分配设置了 PF_MEMALLOC 标志后，会掉调用该函数，用于向其它内核代码表明后续的内存分配都这样搜索（1711）。

try_to_free_pages（1716）被设置（1711）/清除（1720）标志的代码间隔起来，try_to_free_pages 自身也需要获得内存，由于要获得新内存还需要额外分配一点内存（相当矛盾的情形），该函数在内存管理方面享有最高优先级。上述标志设置达到了这一目的，设置 PF_MEMALLOC 分配标志的内存分配非常积极。该标志的设置，也保证了 try_to_free_pages 不会递归调用。如果此前设置了 PF_MEMALLOC，那么__alloc_pages 已经返回。

如果需要多页，per-cpu 变量也会拿回到伙伴系统。（1725）

```
1696 /* The really slow allocator path where we enter direct reclaim */
1697 static inline struct page *
1698 __alloc_pages_direct_reclaim(gfp_t gfp_mask, unsigned int order,
1699         struct zonelist *zonelist, enum zone_type high_zoneidx,
1700         nodemask_t *nodemask, int alloc_flags, struct zone
*preferred_zone,
1701         int migratetype, unsigned long *did_some_progress)
1702 {
1703         struct page *page = NULL;
1704         struct reclaim_state reclaim_state;
1705         struct task_struct *p = current;
1706
1707         cond_resched();
1708
1709         /* We now go into synchronous reclaim */
1710         cpuset_memory_pressure_bump();
1711         p->flags |= PF_MEMALLOC;
1712         lockdep_set_current_reclaim_state(gfp_mask);
1713         reclaim_state.reclaimed_slab = 0;
1714         p->reclaim_state = &reclaim_state;
1715
1716         *did_some_progress = try_to_free_pages(zonelist, order, gfp_mask,
nodemask);
1717
1718         p->reclaim_state = NULL;
```

```
1719        lockdep_clear_current_reclaim_state();
1720        p->flags &= ~PF_MEMALLOC;
1721
1722        cond_resched();
1723
1724        if (order != 0)
1725                drain_all_pages();
1726
1727        if (likely(*did_some_progress))
1728                page = get_page_from_freelist(gfp_mask, nodemask, order,
1729                                        zonelist, high_zoneidx,
1730                                        alloc_flags, preferred_zone,
1731                                        migratetype);
1732        return page;
1733 }
```

2.5.5.3.3 __alloc_pages_may_oom

通过上面的努力后，在试图分配内存。如果这次努力还是失败，并且内核可能执行影响 VFS 层的调用而又没有设置 GFP_NORETRY 标志，同时 oom_killer_disabled 为假，则进行 OOM Killer。（1895-1902）。这里不讨论 oom 的实现细节。请注意，该机制选择一个内核认为分配了过多内存的进程，并杀死该进程。这有很大的几率获得较多的空闲页。但杀死一个进程未必立即出现多余 $2^{PAGE\_ALLOC\_COSTLY\_ORDER}$ 页的连续内存，其中 PAGE_ALLOC_COSTLY_ORDER 为 3，如果要分配如此大的内存区（1876-1877），那么内核不会杀死该进程，而是承认分配内存失败。否则的话，跳转到 restart（1916），重新扫描分配内存。

```
1648 static inline struct page *
1649 __alloc_pages_may_oom(gfp_t gfp_mask, unsigned int order,
1650        struct zonelist *zonelist, enum zone_type high_zoneidx,
1651        nodemask_t *nodemask, struct zone *preferred_zone,
1652        int migratetype)
1653 {
```

```
1654        struct page *page;
1655
1656        /* Acquire the OOM killer lock for the zones in zonelist */
1657        if (!try_set_zone_oom(zonelist, gfp_mask)) {
1658                schedule_timeout_uninterruptible(1);
1659                return NULL;
1660        }
1661
1662        /*
1663         * Go through the zonelist yet one more time, keep very high
watermark
1664         * here, this is only to catch a parallel oom killing, we must fail if
1665         * we're still under heavy pressure.
1666         */
1667        page = get_page_from_freelist(gfp_mask|__GFP_HARDWALL,
nodemask,
1668                order, zonelist, high_zoneidx,
1669                ALLOC_WMARK_HIGH|ALLOC_CPUSET,
1670                preferred_zone, migratetype);
1671        if (page)
1672                goto out;
1673
1674        if (!(gfp_mask & __GFP_NOFAIL)) {
1675                /* The OOM killer will not help higher order allocs */
1676                if (order > PAGE_ALLOC_COSTLY_ORDER)
1677                        goto out;
1678                /*
1679                 * GFP_THISNODE contains __GFP_NORETRY and we
never hit this.
1680                 * Sanity check for bare calls of __GFP_THISNODE, not real
OOM.
1681                 * The caller should handle page allocation failure by itself if
```

```
1682                 * it specifies __GFP_THISNODE.
1683                 * Note: Hugepage uses it but will hit
PAGE_ALLOC_COSTLY_ORDER.
1684                 */
1685                if (gfp_mask & __GFP_THISNODE)
1686                        goto out;
1687        }
1688        /* Exhausted what can be done so it's blamo time */
1689        out_of_memory(zonelist, gfp_mask, order, nodemask);
1690
1691 out:
1692        clear_zonelist_oom(zonelist, gfp_mask);
1693        return page;
1694 }
```

2.5.5.3.4 should_alloc_retry

如果内核没有设置__GFP_NORETRY（1617），并且

- 分配内存小于 8 个页面（1625）或者
- 分配标志设置了__GFP_REPEAT 且 pages_reclaimed < (1 << order))（1635）或者
- 分配标志设置了__GFP_NOFAIL（1642）

则内核会跳转到 rebalance（1922-1925）重新扫描分配内存。

```
1612 static inline int
1613 should_alloc_retry(gfp_t gfp_mask, unsigned int order,
1614                                unsigned long pages_reclaimed)
1615 {
1616        /* Do not loop if specifically requested */
1617        if (gfp_mask & __GFP_NORETRY)
1618                return 0;
1619
1620        /*
```

```
1621         * In this implementation, order <=
PAGE_ALLOC_COSTLY_ORDER
1622         * means __GFP_NOFAIL, but that may not be true in other
1623         * implementations.
1624         */
1625        if (order <= PAGE_ALLOC_COSTLY_ORDER)
1626                return 1;
1627
1628        /*
1629         * For order > PAGE_ALLOC_COSTLY_ORDER, if
__GFP_REPEAT is
1630         * specified, then we retry until we no longer reclaim any pages
1631         * (above), or we've reclaimed an order of pages at least as
1632         * large as the allocation's order. In both cases, if the
1633         * allocation still fails, we stop retrying.
1634         */
1635        if (gfp_mask & __GFP_REPEAT && pages_reclaimed < (1 <<
order))
1636                return 1;
1637
1638        /*
1639         * Don't let big-order allocations loop unless the caller
1640         * explicitly requests that.
1641         */
1642        if (gfp_mask & __GFP_NOFAIL)
1643                return 1;
1644
1645        return 0;
1646 }
```

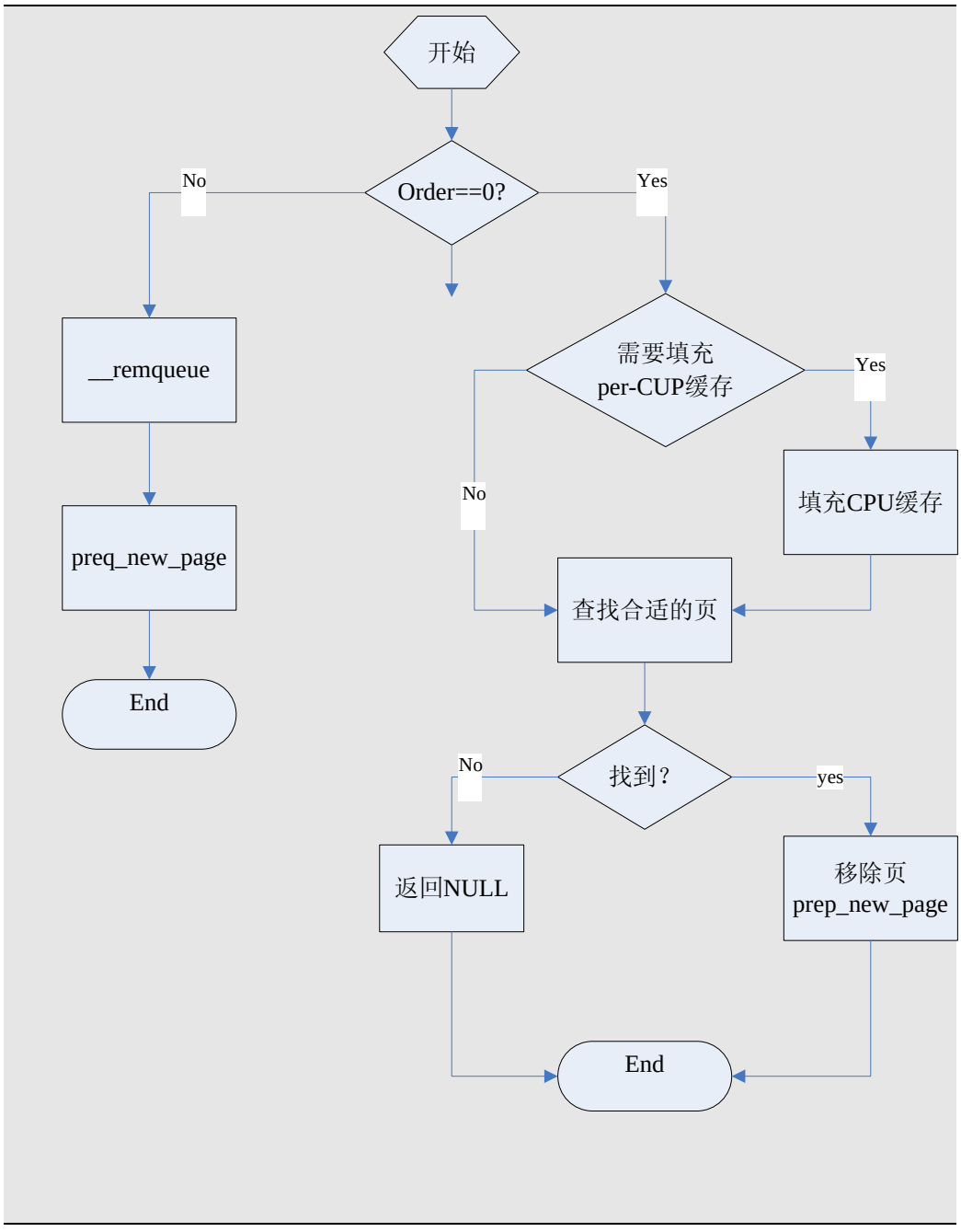
2.5.5.4 从伙伴系统移除选择的页buffered_rmqueue

如果内核找到适当的内存区域，具有足够的空闲页可分配，那么还需要做两件事情来完成任务。

- 检查这些页是否是连续的（到目前为止，只知道有许多空闲页）。
- 按伙伴系统的方式从 free_lists 中移除这些页，该过程有可能分解并重排内存区。

内核将工作交给由 get_page_from_freelist 调用的函数 buffered_rmqueue。

该函数的执行流程如下图所示：



```
1190 /*
1191  * Really, prep_compound_page() should be called from __rmqueue_bulk().  But
1192  * we cheat by calling it from here, in the order > 0 path.  Saves a branch
1193  * or two.
1194  */
1195 static inline
1196 struct page *buffered_rmqueue(struct zone *preferred_zone,
1197                         struct zone *zone, int order, gfp_t gfp_flags,
1198                         int migratetype)
1199 {
1200         unsigned long flags;
1201         struct page *page;
1202         int cold = !!(gfp_flags & __GFP_COLD);
1203
1204 again:
1205         if (likely(order == 0)) {
1206                 struct per_cpu_pages *pcp;
1207                 struct list_head *list;
1208
1209                 local_irq_save(flags);
1210                 pcp = &this_cpu_ptr(zone->pageset)->pcp;
1211                 list = &pcp->lists[migratetype];
1212                 if (list_empty(list)) {
1213                         pcp->count += rmqueue_bulk(zone, 0,
1214                                         pcp->batch, list,
1215                                         migratetype, cold);
1216                         if (unlikely(list_empty(list)))
```

```
1217                    goto failed;
1218            }
1219
1220            if (cold)
1221                    page = list_entry(list->prev, struct page, lru);
1222            else
1223                    page = list_entry(list->next, struct page, lru);
1224
1225            list_del(&page->lru);
1226            pcp->count--;
1227    } else {
1228            if (unlikely(gfp_flags & __GFP_NOFAIL)) {
1229                    /*
1230                     * __GFP_NOFAIL is not to be used in new code.
1231                     *
1232                     * All __GFP_NOFAIL callers should be fixed so that they
1233                     * properly detect and handle allocation failures.
1234                     *
1235                     * We most definitely don't want callers attempting to
1236                     * allocate greater than order-1 page units with
1237                     * __GFP_NOFAIL.
1238                     */
1239                    WARN_ON_ONCE(order > 1);
1240            }
1241            spin_lock_irqsave(&zone->lock, flags);
1242            page = __rmqueue(zone, order, migratetype);
1243            spin_unlock(&zone->lock);
1244            if (!page)
1245                    goto failed;
1246            __mod_zone_page_state(zone, NR_FREE_PAGES, -(1 <<
order));
```

```
1247        }
1248
1249        __count_zone_vm_events(PGALLOC, zone, 1 << order);
1250        zone_statistics(preferred_zone, zone);
1251        local_irq_restore(flags);
1252
1253        VM_BUG_ON(bad_range(zone, page));
1254        if (prep_new_page(page, order, gfp_flags))
1255                goto again;
1256        return page;
1257
1258 failed:
1259        local_irq_restore(flags);
1260        return NULL;
1261 }
```

该函数先检查要分配的页面是否为1个页面，如果是则直接从per-CPU缓存中分配（1205）。否则的话调用__rmqueue 从伙伴系统中分配（1242）。

当从 per-CPU 缓存中分配时，检查是否有指定迁移类型的页可用，如果有，则直接分配。如果没有，调用 rmqueue_bulk（1213）从伙伴系统中分配页作为缓存，然后在分配。

当分配完成返回页之前，会调用 prep_new_page（1254）做一些准备工作，以便内核可以处理这些页。

```
709 static int prep_new_page(struct page *page, int order, gfp_t gfp_flags)
 710 {
 711        int i;
 712
 713        for (i = 0; i < (1 << order); i++) {
 714                struct page *p = page + i;
 715                if (unlikely(check_new_page(p)))
 716                        return 1;
```

```
717     }
```

713-717：检查所分配的内存块中的每一页是否都符合要求。

```
718
719     set_page_private(page, 0);
720     set_page_refcounted(page);
```

719-720：将第一个页的引用计数设为 1。

```
721
722     arch_alloc_page(page, order);
723     kernel_map_pages(page, 1 << order, 1);
724
725     if (gfp_flags & __GFP_ZERO)
726         prep_zero_page(page, order, gfp_flags);
```

725-726：如果设置聊__GFP_ZERO 标志，将分配的页内容清零。

```
727
728     if (order && (gfp_flags & __GFP_COMP))
729         prep_compound_page(page, order);
```

728-729 行：如果设置了复合页标志__GFP_COMP，则将分配的块中的页进行相应的设置。

```
730
731     return 0;
732 }
```

该函数进行复合页初始化。

```
321 void prep_compound_page(struct page *page, unsigned long order)
322 {
323     int i;
324     int nr_pages = 1 << order;
325
326     set_compound_page_dtor(page, free_compound_page);
327     set_compound_order(page, order);
328     __SetPageHead(page);
```

```
329         for (i = 1; i < nr_pages; i++) {
330                 struct page *p = page + i;
331
332                 __SetPageTail(p);
333                 p->first_page = page;
334         }
335 }
```

2.5.5.4.1 rmqueue_bulk

为 per-CPU 缓存重伙伴系统中申请页面。

```
957 /*
958  * Obtain a specified number of elements from the buddy allocator, all
under
959  * a single hold of the lock, for efficiency.  Add them to the supplied list.
960  * Returns the number of new pages which were placed at *list.
961  */
962 static int rmqueue_bulk(struct zone *zone, unsigned int order,
963                         unsigned long count, struct list_head *list,
964                         int migratetype, int cold)
965 {
966         int i;
967
968         spin_lock(&zone->lock);
969         for (i = 0; i < count; ++i) {
970                 struct page *page = __rmqueue(zone, order, migratetype);
971                 if (unlikely(page == NULL))
972                         break;
973
974                 /*
975                  * Split buddy pages returned by expand() are received here
976                  * in physical page order. The page is added to the callers and
977                  * list and the list head then moves forward. From the callers
```

```
978                 * perspective, the linked list is ordered by page number in
979                 * some conditions. This is useful for IO devices that can
980                 * merge IO requests if the physical pages are ordered
981                 * properly.
982                 */
983                if (likely(cold == 0))
984                        list_add(&page->lru, list);
985                else
986                        list_add_tail(&page->lru, list);
987                set_page_private(page, migratetype);
988                list = &page->lru;
989        }
990        __mod_zone_page_state(zone, NR_FREE_PAGES, -(i << order));
991        spin_unlock(&zone->lock);
992        return i;
993 }
```

该函数通过 for 循环（969-989）行，每次通过__rmqueue（970）函数从伙伴系统中分配一个页面，进行相应的设置，链入 per-CPU 缓存链表。

2.5.5.4.2 __rmqueue

```
927 /*
928  * Do the hard work of removing an element from the buddy allocator.
929  * Call me with the zone->lock already held.
930  */
931 static struct page *__rmqueue(struct zone *zone, unsigned int order,
932                                        int migratetype)
933 {
934        struct page *page;
935
936 retry_reserve:
937        page = __rmqueue_smallest(zone, order, migratetype);
```

```
938
939         if (unlikely(!page) && migratetype != MIGRATE_RESERVE) {
940                 page = __rmqueue_fallback(zone, order, migratetype);
941
942                 /*
943                  * Use MIGRATE_RESERVE rather than fail an allocation. goto
944                  * is used because __rmqueue_smallest is an inline function
945                  * and we want just one call site
946                  */
947                 if (!page) {
948                         migratetype = MIGRATE_RESERVE;
949                         goto retry_reserve;
950                 }
951         }
952
953         trace_mm_page_alloc_zone_locked(page, order, migratetype);
954         return page;
955 }
```

该函数的路径很简单，

从指定的迁移类型中分配__rmqueue_smallest（937），如果分配失败，则

从备用迁移列表中分配（940）。

2.5.5.4.2.1 __rmqueue_smallest

该函数由一个循环组成，按递增顺序遍历同一迁移类型的各个阶，知道找到一个空闲的块为止。

```
734 /*
735  * Go through the free lists for the given migratetype and remove
736  * the smallest available page from the freelists
737  */
738 static inline
739 struct page *__rmqueue_smallest(struct zone *zone, unsigned int order,
```

```
740                                     int migratetype)
741 {
742         unsigned int current_order;
743         struct free_area * area;
744         struct page *page;
745
746         /* Find a page of the appropriate size in the preferred list */
747         for (current_order = order; current_order < MAX_ORDER;
++current_order) {
748                 area = &(zone->free_area[current_order]);
749                 if (list_empty(&area->free_list[migratetype]))
750                         continue;
751
752                 page = list_entry(area->free_list[migratetype].next,
753                                                 struct page, lru);
754                 list_del(&page->lru);
755                 rmv_page_order(page);
756                 area->nr_free--;
```

752 行：从链表中分配一个空闲块

754 行：将空闲块从链表中删除

755 行：清除 page 页面的伙伴标志

766 行：内存区域块数目减 1

```
757                 expand(zone, page, order, current_order, area, migratetype);
```

757 行：如果需要分配的内存块小于所选择的内存块，那么需要按伙伴系统的原理分裂成小的块，则是通过 expand 函数实现的。

```
758                 return page;
759         }
760
761         return NULL;
762 }
```

2.5.5.4.2.2 __rmqueue_fallback

该函数通过两次循环来遍历备用迁移列表中的空闲块。

外层循环控制分配的阶数，此处阶是按从大到小的递减的（867-868）。这与通常的策略相反，内核的依据是：如果无法避免分配迁移类型的不同的内存块，那么就分配一个尽可能大的内存块。如果优先选择小的内存块，则会向其它列表引入碎片。

特别列表 MIGRATE_RESERVE 需要特殊处理。

内层循环（867-868）控制不同的迁移类型。

```
857 /* Remove an element from the buddy allocator from the fallback list */
858 static inline struct page *
859 __rmqueue_fallback(struct zone *zone, int order, int start_migratetype)
860 {
861         struct free_area * area;
862         int current_order;
863         struct page *page;
864         int migratetype, i;
865
866         /* Find the largest possible block of pages in the other list */
867         for (current_order = MAX_ORDER-1; current_order >= order;
868                                         --current_order) {
869                 for (i = 0; i < MIGRATE_TYPES - 1; i++) {
870                         migratetype = fallbacks[start_migratetype][i];
871
872                         /* MIGRATE_RESERVE handled later if necessary */
873                         if (migratetype == MIGRATE_RESERVE)
874                                 continue;
875
876                         area = &(zone->free_area[current_order]);
877                         if (list_empty(&area->free_list[migratetype]))
878                                 continue;
879
```

```
880                 page = list_entry(area->free_list[migratetype].next,
881                         struct page, lru);
882                 area->nr_free--;
883
884                 /*
885                  * If breaking a large block of pages, move all free
886                  * pages to the preferred allocation list. If falling
887                 * back for a reclaimable kernel allocation, be more
888                  * agressive about taking ownership of free pages
889                  */
890                 if (unlikely(current_order >= (pageblock_order >> 1)) ||
891                         start_migratetype == MIGRATE_RECLAIMABLE
||
892                         page_group_by_mobility_disabled) {
893                     unsigned long pages;
894                     pages = move_freepages_block(zone, page,
895                                     start_migratetype);
896
897                     /* Claim the whole block if over half of it is free */
898                     if (pages >= (1 << (pageblock_order-1)) ||
899                                 page_group_by_mobility_disabled)
900                         set_pageblock_migratetype(page,
901                                     start_migratetype);
902
903                     migratetype = start_migratetype;
904                 }
905
906                 /* Remove the page from the freelists */
907                 list_del(&page->lru);
908                 rmv_page_order(page);
909
910                 /* Take ownership for orders >= pageblock_order */
```

```
911                     if (current_order >= pageblock_order)
912                             change_pageblock_range(page, current_order,
913                                                     start_migratetype);
914
915                     expand(zone, page, order, current_order, area,
migratetype);
916
917                     trace_mm_page_alloc_extfrag(page, order, current_order,
918                             start_migratetype, migratetype);
919
920                     return page;
921             }
922     }
923
924     return NULL;
925 }
```

如果需要分解来自其他迁移列表的空闲内存块，内核的处理办法是：

如果剩余部分是一个较大的内存块（890），则将整个内存块都转到当前分配类型对应的迁移列表，这样可以减少碎片（894-895）。

如果是在分配可回收内存，那么内核从一个迁移列表移动到另一个时，会更加积极。此类分配经常突发涌现，导致许多小的可回收内存块散步到所有的迁移列表。为避免此类情况，分配 MIGRATE_RECLAIMABLE 内存块时，剩余的总是迁移到可回收列表（890-904）。

2.5.5.4.2.3 expand

该函数短小精悍，用于处理获得的内存块的阶数大于需要分配的阶的情况。

该函数通过循环将当前的块分为前半部分和后半部分（684-685），后半部分插入到对应阶的 free_list 列表中（688），前半部分继续下一轮的循环，直到将所有空闲的块都分配到对应的阶的 free_list 中，循环结束，返回需要分配的块。

```
663 /*
664  * The order of subdivision here is critical for the IO subsystem.
```

```
665   * Please do not alter this order without good reasons and regression
666   * testing. Specifically, as large blocks of memory are subdivided,
667   * the order in which smaller blocks are delivered depends on the order
668   * they're subdivided in this function. This is the primary factor
669   * influencing the order in which pages are delivered to the IO
670   * subsystem according to empirical testing, and this is also justified
671   * by considering the behavior of a buddy system containing a single
672   * large block of memory acted on by a series of small allocations.
673   * This behavior is a critical factor in sglist merging's success.
674   *
675   * -- wli
676   */
677 static inline void expand(struct zone *zone, struct page *page,
678         int low, int high, struct free_area *area,
679         int migratetype)
680 {
681         unsigned long size = 1 << high;
682
683         while (high > low) {
684                 area--;
685                 high--;
686                 size >>= 1;
687                 VM_BUG_ON(bad_range(zone, &page[size]));
688                 list_add(&page[size].lru, &area->free_list[migratetype]);
689                 area->nr_free++;
690                 set_page_order(&page[size], high);
691         }
692 }
```

2.5.6 伙伴系统内存释放

__free_pages 是一个基础函数，用于作为伙伴系统释放页面的总入口。

该函数的基本流程是：

先判断是否为单页：

是：则调用 free_hot_cold_page(page, 0)释放页面。（2032）

否：调用__free_pages_ok(page, order)释放页面。（2034）

```
2028 void __free_pages(struct page *page, unsigned int order)
2029 {
2030        if (put_page_testzero(page)) {
2031                if (order == 0)
2032                        free_hot_cold_page(page, 0);
2033                else
2034                        __free_pages_ok(page, order);
2035        }
2036 }
```

不论是 free_hot_cold_page 还是__free_pages_ok 函数，最终都会调用 free_one_page 函数。该函数与其名称不符合，其不仅处理单页情况，也处理复合页的释放。

2.5.6.1 free_one_page

```
590 static void free_one_page(struct zone *zone, struct page *page,
int order,
591                                int migratetype)
592 {
593        spin_lock(&zone->lock);
594        zone->all_unreclaimable = 0;
595        zone->pages_scanned = 0;
596
597        __mod_zone_page_state(zone, NR_FREE_PAGES, 1
<< order);
```

```
598         __free_one_page(page, zone, order, migratetype);
599         spin_unlock(&zone->lock);
600 }
```

__free_one_page 函数

```
449 /*
450  * Freeing function for a buddy system allocator.
451  *
452  * The concept of a buddy system is to maintain direct-mapped table
453  * (containing bit values) for memory blocks of various "orders".
454  * The bottom level table contains the map for the smallest allocatable
455  * units of memory (here, pages), and each level above it describes
456  * pairs of units from the levels below, hence, "buddies".
457  * At a high level, all that happens here is marking the table entry
458  * at the bottom level available, and propagating the changes upward
459  * as necessary, plus some accounting needed to play nicely with other
460  * parts of the VM system.
461  * At each level, we keep a list of pages, which are heads of continuous
462  * free pages of length of (1 << order) and marked with PG_buddy.
Page's
463  * order is recorded in page_private(page) field.
464  * So when we are allocating or freeing one, we can derive the state of
the
465  * other.  That is, if we allocate a small block, and both were
466  * free, the remainder of the region must be split into blocks.
467  * If a block is freed, and its buddy is also free, then this
468  * triggers coalescing into a block of larger size.
469  *
470  * -- wli
471  */
472
473 static inline void __free_one_page(struct page *page,
```

```
474             struct zone *zone, unsigned int order,
475             int migratetype)
476 {
477         unsigned long page_idx;
478
478
479         if (unlikely(PageCompound(page)))
480             if (unlikely(destroy_compound_page(page, order)))
481                 return;
482
483         VM_BUG_ON(migratetype == -1);
484
485         page_idx = page_to_pfn(page) & ((1 << MAX_ORDER) - 1);
```

485 行：page_idx 表示 page 在 mem_map 数组中的下标模 1023 的值。

```
486
487         VM_BUG_ON(page_idx & ((1 << order) - 1));
488         VM_BUG_ON(bad_range(zone, page));
489
490         while (order < MAX_ORDER-1) {
491             unsigned long combined_idx;
492             struct page *buddy;
493
494             buddy = __page_find_buddy(page, page_idx, order);
```

494 行：寻找 page 块的伙伴。

```
406 static inline struct page *
407 __page_find_buddy(struct page *page, unsigned long page_idx, unsigned int order)
408 {
409         unsigned long buddy_idx = page_idx ^ (1 << order);
410
```

```
411         return page + (buddy_idx - page_idx);
412 }
```

要看懂上述函数，必须清楚如下几点：

1、伙伴算法的块的分配的方法，大小为 2^x 个页框的块，他的起始地址一定是 2^x 的倍数

2、在把两个大小为 2^(x-1)的块合并为大小为 2^x 的块的时候，需要考虑目前回收的块是 buddy 里面的第一个还是第二个，判断的依据就是 page_idx 的 2^order 这一位，如果为 1，则必须向前寻找 buddy，如果为 0，则必须向后寻找 buddy，这也是异或的精妙之处，如果写成 if else 可能更好理解，但是有了奇技淫巧才显得 linux 源代码的高深。

3、当 order 固定时，一个 page_idx 在指数为 order 的 free_list 中只可能有一个伙伴。

```
495             if (!page_is_buddy(page, buddy, order))
496                     break;
```

495-496 行：判断 buddy 是否为 page 的伙伴。

420-427 行的注释很清楚的说明了伙伴的条件。

1、buddy 不能在空洞中。

2、伙伴在伙伴系统中。

3、伙伴和 page 需要有相同的阶数

4、page 和 buddy 在相同的内存区域中。

```
420 /*
421  * This function checks whether a page is free && is the buddy
422  * we can do coalesce a page and its buddy if
423  * (a) the buddy is not in a hole &&
424  * (b) the buddy is in the buddy system &&
425  * (c) a page and its buddy have the same order &&
426  * (d) a page and its buddy are in the same zone.
427  *
428  * For recording whether a page is in the buddy system, we use PG_buddy.
429  * Setting, clearing, and testing PG_buddy is serialized by zone->lock.
```

```
 430  *
 431  * For recording page's order, we use page_private(page).
 432  */
 433 static inline int page_is_buddy(struct page *page, struct page
*buddy,
 434                                                 int order)
 435 {
 436         if (!pfn_valid_within(page_to_pfn(buddy)))
 437                 return 0;
 438
 439         if (page_zone_id(page) != page_zone_id(buddy))
 440                 return 0;
 441
 442         if (PageBuddy(buddy) && page_order(buddy) == order) {
 443                 VM_BUG_ON(page_count(buddy) != 0);
 444                 return 1;
 445         }
 446         return 0;
 447 }
```

```
 497
 498             /* Our buddy is free, merge with it and move up one order. */
 499             list_del(&buddy->lru);
 500             zone->free_area[order].nr_free--;
```

499-500 行：将伙伴冲阶 order 中移除，同时减少阶 order 中 nr_free。

```
 501             rmv_page_order(buddy);
 502             combined_idx = __find_combined_index(page_idx, order);
```

502 行：计算伙伴合并后新的 idx

```
 503             page = page + (combined_idx - page_idx);
 504             page_idx = combined_idx;
 505             order++;
```

503-505 行：计算伙伴的头 page，得到伙伴的 idx，order++，进行下一轮循环。

```
506         }
507         set_page_order(page, order);
```

507 行：设置 page 的阶数

```
508         list_add(&page->lru,
509                 &zone->free_area[order].free_list[migratetype]);
510         zone->free_area[order].nr_free++;
```

508-510 行：将伙伴添加到对应的阶中，且该阶的 nr_free++。

```
511 }
```

2.6 slab分配器

2.6.1 slab分配器产生的原因

C 程序中 malloc 函数，在需要分配若干字节内存时使用。该函数可以控制分配单位为字节。

内核需要经常分配内存，虽然伙伴系统可以分配内存，但其分配单位为页面，如果为一个需要 10 字节的申请分配一个页面，显示是一种浪费。内核的解决方案是将页拆分为更小的单位，可以容纳更多的小对象。

内核使用 slab 分配器解决上述问题。

2.6.2 数据结构

slab 分配器中的一个重要概念是缓存。一个缓存包含一些列的 slab，每一个 slab 由几个连续的物理页面组成，这些页面包含相同类型的对象。

在内核中，缓存的结构如下：

Include/linux/slab_def.h

2.6.2.1 kmem_cache数据结构

```
19 /*
20  * struct kmem_cache
21  *
```

```
22   * manages a cache.
23   */
24
25 struct kmem_cache {
26 /* 1) per-cpu data, touched during every alloc/free */
27       struct array_cache *array[NR_CPUS];
```

27:per-CPU 数据，每次分配/释放时访问，加快分配速度。

array 是一个指针数组，数组的长度是系统中 CPU 数目的个数。每个数组项指向一个 array_cache 示例，表示系统中的一个 CPU。

```
28 /* 2) Cache tunables. Protected by cache_chain_mutex */
29       unsigned int batchcount;
30       unsigned int limit;
31       unsigned int shared;
32
33       unsigned int buffer_size;
34       u32 reciprocal_buffer_size;
```

28-34 行：可调整的缓存参数

batchcount：在 per-CPU 列表为空时，从缓存的 slab 中获取对象的数目。它还表示在缓存增长时分配的对象数目。

limit：指定了 per-CPU 列表中保存的对象的最大数目。

```
35 /* 3) touched by every alloc & free from the backend */
36
37       unsigned int flags;       /* constant flags */
38       unsigned int num;          /* # of objs per slab */
39
```

37-38 行：常数标志。

```
40 /* 4) cache_grow/shrink */
41       /* order of pgs per slab (2^n) */
42       unsigned int gfporder;
43
44       /* force GFP flags, e.g. GFP_DMA */
```

```
45      gfp_t gfpflags;
46
47      size_t colour;              /* cache colouring range */
48      unsigned int colour_off;    /* colour offset */
49      struct kmem_cache *slabp_cache;
50      unsigned int slab_size;
51      unsigned int dflags;        /* dynamic flags */
52                                          ¢] 223L, 5300C
53      /* constructor func */
54      void (*ctor)(void *obj);
55
```

41-55 行：缓存的增长/缩减变量

```
56 /* 5) cache creation/removal */
57      const char *name;
58      struct list_head next;
59
```

57-58 行：缓存创建/删去相关变量

```
60 /* 6) statistics */

61 #ifdef CONFIG_DEBUG_SLAB
62      unsigned long num_active;
63      unsigned long num_allocations;
64      unsigned long high_mark;
65      unsigned long grown;
66      unsigned long reaped;
67      unsigned long errors;
68      unsigned long max_freeable;
69      unsigned long node_allocs;
70      unsigned long node_frees;
71      unsigned long node_overflow;    ¢] 223L, 5300C
72      atomic_t allochit;
```

```
73          atomic_t allocmiss;
74          atomic_t freehit;
75          atomic_t freemiss;
76
77          /*
78           * If debugging is enabled, then the allocator can add additional
79           * fields and/or padding to every object. buffer_size contains the total
80           * object size including these internal fields, the following two
81           * variables contain the offset to the user object and its size.
82           */
83          int obj_offset;
84          int obj_size;
85 #endif /* CONFIG_DEBUG_SLAB */
```

61-85 行：统计量

```
86
87          /*
88           * We put nodelists[] at the end of kmem_cache, because we want to size
89           * this array to nr_node_ids slots instead of MAX_NUMNODES
90           * (see kmem_cache_init())
91           * We still use [MAX_NUMNODES] and not [1] or [0] because cache_cache
92           * is statically defined, so we reserve the max number of nodes.
93           */
94          struct kmem_list3 *nodelists[MAX_NUMNODES];
```

94 行：slab 链表表头数据结构。Nodelists 是一个数组，每个数组项对应系统中一个可能的节点。

```
95          /*
96           * Do not add fields after nodelists[]
97           */
98 };
```

2.6.2.2 数据结构array_cache

```
255 /*
256  * struct array_cache
257  *
258  * Purpose:
259  * - LIFO ordering, to hand out cache-warm objects from _alloc
260  * - reduce the number of linked list operations
261  * - reduce spinlock operations
262  *
263  * The limit is stored in the per-cpu structure to reduce the data cache
264  * footprint.
265  *
266  */
267 struct array_cache {
268         unsigned int avail;
269         unsigned int limit;
270         unsigned int batchcount;
271         unsigned int touched;
272         spinlock_t lock;
273         void *entry[];    /*
274                           * Must have this definition in here for the proper
275                           * alignment of array_cache. Also simplifies accessing
276                           * the entries.
277                           */
278 };
```

avail:保存当前可用对象的数目。

entry：一个伪数组，方便访问 array_cache 实例之后缓存中的各个对象。

2.6.2.3 kmem_list3 数据结构

```
290 /*
291  * The slab lists for all objects.
292  */
293 struct kmem_list3 {
294       struct list_head slabs_partial; /* partial list first, better asm
code       */
295       struct list_head slabs_full;
296       struct list_head slabs_free;
```

部分、满、空闲 slab 的链表头

```
297       unsigned long free_objects;
298       unsigned int free_limit;
299       unsigned int colour_next;     /* Per-node cache coloring */
300       spinlock_t list_lock;
301       struct array_cache *shared; /* shared per node */
302       struct array_cache **alien; /* on other nodes */
303       unsigned long next_reap;      /* updated without locking */
304       int free_touched;             /* updated without locking */
305 };
```

297-305 行：和页面回收的相关数据结构

2.6.3 slab分配器的初始化

为使用 slab 机制，内核在初始化中需要为 slab 建立初始环境。

start_kernel/mm_init /kmen_cache_init

为初始化 slab 数据结构，内核若干小于 1 页的内存，这些内存最适合使用 kmalloc 调用，但在 slab 分配器建立之前，是不能使用 kmalloc 分配内存的。

内核使用编译时创建的静态数据，为 slab 的初始化提供内存。

kmen_cache_init 的主要过程是：

1、创建系统中的第一个 slab 缓存，为 kmem_cache 实例提供内存。第一个 slab 缓存的实例使用编译时的静态数据结构 initarray_cache 用作 per-CPU 数组，缓存名称为 cache_cache。

2、初始化一般性的的缓存，为使用用 kmalloc 的做好准备。

```
1372  * Initialisation.  Called after the page allocator have been initialised and
1373  * before smp_init().
1374  */
1375 void __init kmem_cache_init(void)
1376 {
1377         size_t left_over;
1378         struct cache_sizes *sizes;
1379         struct cache_names *names;
1380         int i;
1381         int order;
1382         int node;
1383
1384         if (num_possible_nodes() == 1)
1385                 use_alien_caches = 0;
1386
```

```
1387         for (i = 0; i < NUM_INIT_LISTS; i++) {
1388                 kmem_list3_init(&initkmem_list3[i]);
1389                 if (i < MAX_NUMNODES)
1390                         cache_cache.nodelists[i] = NULL;
1391         }
```

初始化全局变量 initkmem_list3

```
307 /*
308  * Need this for bootstrapping a per node allocator.
309  */
310 #define NUM_INIT_LISTS (3 * MAX_NUMNODES)
```

```
311 struct kmem_list3 __initdata
initkmem_list3[NUM_INIT_LISTS];
312 #define CACHE_CACHE 0
313 #define SIZE_AC MAX_NUMNODES
314 #define SIZE_L3 (2 * MAX_NUMNODES)
```

initkmem_list3 是一个数组，数组的每个元素为 kmem_list3,数组的个数为最大节点数的 3 倍。

```
1392        set_up_list3s(&cache_cache, CACHE_CACHE);
```

初始化全局变量 cache_cache.

```
1393
1394        /*
1395         * Fragmentation resistance on low memory - only use bigger
1396         * page orders on machines with more than 32MB of memory.
1397         */
1398        if (totalram_pages > (32 << 20) >> PAGE_SHIFT)
1399                slab_break_gfp_order = BREAK_GFP_ORDER_HI;
1400
1401        /* Bootstrap is tricky, because several objects are allocated
1402         * from caches that do not exist yet:
1403         * 1) initialize the cache_cache cache: it contains the struct
1404         *      kmem_cache structures of all caches, except cache_cache
itself:
1405         *      cache_cache is statically allocated.
1406         *      Initially an __init data area is used for the head array and the
1407         *      kmem_list3 structures, it's replaced with a kmalloc allocated
1408         *      array at the end of the bootstrap.
1409         * 2) Create the first kmalloc cache.
1410         *      The struct kmem_cache for the new cache is allocated
normally.
1411         *      An __init data area is used for the head array.
1412         * 3) Create the remaining kmalloc caches, with minimally sized
1413         *      head arrays.
```

```
1414        * 4) Replace the __init data head arrays for cache_cache and the
first
1415        *       kmalloc cache with kmalloc allocated arrays.
1416        * 5) Replace the __init data for kmem_list3 for cache_cache and
1417        *       the other cache's with kmalloc allocated memory.
1418        * 6) Resize the head arrays of the kmalloc caches to their final
sizes.
1419        */
1420
1421       node = numa_node_id();
1422
1423       /* 1) create the cache_cache */
1424       INIT_LIST_HEAD(&cache_chain);
1425       list_add(&cache_cache.next, &cache_chain);
```

将 cache_cache 连接到链表头 cache_chain 中。

```
1426       cache_cache.colour_off = cache_line_size();
1427       cache_cache.array[smp_processor_id()] = &initarray_cache.cache;
```

使用静态全局变量作为 cache_cache 缓存的 per-CPU 实例。

```
591 static struct arraycache_init initarray_cache __initdata =
592       { {0, BOOT_CPUCACHE_ENTRIES, 1, 0} };
593 static struct arraycache_init initarray_generic =
594       { {0, BOOT_CPUCACHE_ENTRIES, 1, 0} };
595
596 /* internal cache of cache description objs */
597 static struct kmem_cache cache_cache = {
598       .batchcount = 1,
599       .limit = BOOT_CPUCACHE_ENTRIES,
600       .shared = 1,
601       .buffer_size = sizeof(struct kmem_cache),
602       .name = "kmem_cache",
603 };
```

initarray_cache 是内型为 arraycache_init 结构体，它的元素包含一个 array_cache 的示例，以及一个指针数组。

```
284 #define BOOT_CPUCACHE_ENTRIES    1
285 struct arraycache_init {
286         struct array_cache cache;
287         void *entries[BOOT_CPUCACHE_ENTRIES];
288 };
```

```
1428        cache_cache.nodelists[node] = &initkmem_list3[CACHE_CACHE +
node];
1429
1430        /*
1431         * struct kmem_cache size depends on nr_node_ids, which
1432         * can be less than MAX_NUMNODES.
1433         */
1434        cache_cache.buffer_size = offsetof(struct kmem_cache, nodelists)
+
1435                                nr_node_ids * sizeof(struct kmem_list3 *);
1436 #if DEBUG
1437        cache_cache.obj_size = cache_cache.buffer_size;
1438 #endif
1439        cache_cache.buffer_size = ALIGN(cache_cache.buffer_size,
1440                                        cache_line_size());
1441        cache_cache.reciprocal_buffer_size =
1442                reciprocal_value(cache_cache.buffer_size);
1443
1444        for (order = 0; order < MAX_ORDER; order++) {
1445                cache_estimate(order, cache_cache.buffer_size,
1446                        cache_line_size(), 0, &left_over, &cache_cache.num);
1447                if (cache_cache.num)
1448                        break;
1449        }
```

```
1450        BUG_ON(!cache_cache.num);
1451        cache_cache.gfporder = order;
1452        cache_cache.colour = left_over / cache_cache.colour_off;
1453        cache_cache.slab_size = ALIGN(cache_cache.num *
sizeof(kmem_bufctl_t) +
1454                                      sizeof(struct slab), cache_line_size());
1455
```

1426-1455 行：初始化 cache_cache 字段的相关变量

```
1456        /* 2+3) create the kmalloc caches */
1457        sizes = malloc_sizes;
1458        names = cache_names;
```

Malloc_sizes，cache_names 是两个指针数组，其中分别存放一般性缓存的大小和名称。

```
1459
1460        /*
1461         * Initialize the caches that provide memory for the array cache and
the
1462         * kmem_list3 structures first.    Without this, further allocations will
1463         * bug.
1464         */
1465
1466        sizes[INDEX_AC].cs_cachep =
kmem_cache_create(names[INDEX_AC].name,
1467                                      sizes[INDEX_AC].cs_size,
1468                                      ARCH_KMALLOC_MINALIGN,
1469                                      ARCH_KMALLOC_FLAGS|SLAB_PANIC,
1470                                      NULL);
```

创建大小为(sizeof(struct arraycache_init)的普通缓存。

```
349 #define INDEX_AC index_of(sizeof(struct arraycache_init))
350 #define INDEX_L3 index_of(sizeof(struct kmem_list3))
```

```
1471
1472        if (INDEX_AC != INDEX_L3) {
1473            sizes[INDEX_L3].cs_cachep =
1474                kmem_cache_create(names[INDEX_L3].name,
1475                    sizes[INDEX_L3].cs_size,
1476                    ARCH_KMALLOC_MINALIGN,
1477                    ARCH_KMALLOC_FLAGS|SLAB_PANIC,
1478                    NULL);
1479        }
```

如果 arraycache_init 和 kmem_list3 大小相等，则共用同一普通缓存，否则创建大小为 sizeof(struct kmem_list3)的缓存。

这两个普通缓存非常重要，因为它是下面其它缓存的基础。因为每个 kmem_cache 缓存都需要分配 arraycache_init 和 kmem_list3 大小的对象，而这些对象需要通过 kmalloc 来分配，而 kmalloc 的分配有是建立的上述两个普通缓存的基础上的。

这里有一个问题大家必须明白，即为分配 arraycache_init 和 kmem_list3 建立的缓存也是一个 kmem_cache 实例，它们也需要 arraycache_init 和 kmem_list3，所以也需要 kmalloc 的分配，但是 kmalloc 的分配在 arraycache_init 和 kmem_list3 的缓存还没建立起来是不可使用的，这里就存在一个鸡与蛋的问题，起始从上面的代码可以看出，内核的解决办法是：为分配 arraycache_init 和 kmem_list3 的而建立的缓存的 arraycache_init 和 kmem_list3 使用静态的全局变量。具体的实现就不深入进去了，等到讲解虚拟内存系统是在详细讨论。

```
1480
1481        slab_early_init = 0;
1482
1483        while (sizes->cs_size != ULONG_MAX) {
1484            /*
1485             * For performance, all the general caches are L1 aligned.
1486             * This should be particularly beneficial on SMP boxes, as it
1487             * eliminates "false sharing".
1488             * Note for systems short on memory removing the alignment
will
```

```
1489             * allow tighter packing of the smaller caches.
1490             */
1491            if (!sizes->cs_cachep) {
1492                sizes->cs_cachep = kmem_cache_create(names->name,
1493                        sizes->cs_size,
1494                        ARCH_KMALLOC_MINALIGN,
1495                        ARCH_KMALLOC_FLAGS|SLAB_PANIC,
1496                        NULL);
1497            }
1498 #ifdef CONFIG_ZONE_DMA
1499            sizes->cs_dmacachep = kmem_cache_create(
1500                        names->name_dma,
1501                        sizes->cs_size,
1502                        ARCH_KMALLOC_MINALIGN,
1503                        ARCH_KMALLOC_FLAGS|SLAB_CACHE_DMA|
1504                            SLAB_PANIC,
1505                        NULL);
1506 #endif
1507            sizes++;
1508            names++;
1509        }
```

1456-1509 行：建立剩下的一般性缓存。

```
1510        /* 4) Replace the bootstrap head arrays */
1511        {
1512            struct array_cache *ptr;
1513
1514            ptr = kmalloc(sizeof(struct arraycache_init), GFP_NOWAIT);
```

前面的一般性缓存已经建立，现在可以使用 kmalloc 分配内存了。

```
1515
1516            BUG_ON(cpu_cache_get(&cache_cache) !=
&initarray_cache.cache);
```

1516 行：首先判断 cache_cache 的 per-CPU 变量是否是从静态全局变量 initarray_cache 中来的，如果不是这内核初始化肯定是出问题了。

```
1517            memcpy(ptr, cpu_cache_get(&cache_cache),
1518                   sizeof(struct arraycache_init));
1519            /*
1520             * Do not assume that spinlocks can be initialized via memcpy:
1521             */
1522            spin_lock_init(&ptr->lock);
1523
1524            cache_cache.array[smp_processor_id()] = ptr;
```

替换掉之前的静态 per-CPU 变量。

```
1525
1526            ptr = kmalloc(sizeof(struct arraycache_init), GFP_NOWAIT);
1527
1528
BUG_ON(cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep)
1529                   != &initarray_generic.cache);
1530            memcpy(ptr,
cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep),
1531                   sizeof(struct arraycache_init));
1532            /*
1533             * Do not assume that spinlocks can be initialized via memcpy:
1534             */
1535            spin_lock_init(&ptr->lock);
1536
1537
malloc_sizes[INDEX_AC].cs_cachep->array[smp_processor_id()] =
1538                   ptr;
1539    }
```

1510-1539 行：替换 bootstrap

```
1540    /* 5) Replace the bootstrap kmem_list3's */
1541    {
```

```
1542            int nid;
1543
1544            for_each_online_node(nid) {
1545                    init_list(&cache_cache, &initkmem_list3[CACHE_CACHE
+ nid], nid);
1546
1547                    init_list(malloc_sizes[INDEX_AC].cs_cachep,
1548                            &initkmem_list3[SIZE_AC + nid], nid);
1549
1550                    if (INDEX_AC != INDEX_L3) {
1551                            init_list(malloc_sizes[INDEX_L3].cs_cachep,
1552                                    &initkmem_list3[SIZE_L3 + nid], nid);
1553                    }
```

1545-1553：替换掉缓存 cache_cache，arraycache_init，kmem_list3 的 kmem_list3 变量。

```
1554            }
1555    }
1556
1557    g_cpucache_up = EARLY;
```

设置 g_cpucache_up 为 EARLY。g_cpucache_up 的状态控制着建立缓存时 per-cache 的分配机制。

```
1558 }
```

2.6.4 缓存的创建与销毁

2.6.4.1 创建缓存kmem_cache_create

创建新缓存是一个冗长的过程，kmem_cache_create 代码流程如下：

2.6.4.1.1 kmem_cache_create

参数有效性检查

计算对齐值
分配缓存结构
确定 slab 头的存储位置
calculate_slab_order
    cache_estimate 迭代计算缓存长度
计算颜色
将缓存插入到 cache_chain 中。

```
2068 /**
2069  * kmem_cache_create - Create a cache.
2070  * @name: A string which is used in /proc/slabinfo to identify this cache.
2071  * @size: The size of objects to be created in this cache.
2072  * @align: The required alignment for the objects.
2073  * @flags: SLAB flags
2074  * @ctor: A constructor for the objects.
2075  *
2076  * Returns a ptr to the cache on success, NULL on failure.
2077  * Cannot be called within a int, but can be interrupted.
2078  * The @ctor is run when new pages are allocated by the cache.
2079  *
2080  * @name must be valid until the cache is destroyed. This implies that
2081  * the module calling this has to destroy the cache before getting unloaded.
2082  * Note that kmem_cache_name() is not guaranteed to return the same pointer,
2083  * therefore applications must manage it themselves.
2084  *
2085  * The flags are
2086  *
2087  * %SLAB_POISON - Poison the slab with a known test pattern (a5a5a5a5)
2088  * to catch references to uninitialised memory.
2089  *
2090  * %SLAB_RED_ZONE - Insert `Red' zones around the allocated memory to check
2091  * for buffer overruns.
2092  *
```

```
2093   * %SLAB_HWCACHE_ALIGN - Align the objects in this cache to a hardware
2094   * cacheline.  This can be beneficial if you're counting cycles as closely
2095   * as davem.
2096   */
2097 struct kmem_cache *
2098 kmem_cache_create (const char *name, size_t size, size_t align,
2099         unsigned long flags, void (*ctor)(void *))
2100 {
2101         size_t left_over, slab_size, ralign;
2102         struct kmem_cache *cachep = NULL, *pc;
2103         gfp_t gfp;
2104
2105         /*
2106          * Sanity checks... these are all serious usage bugs.
2107          */
2108         if (!name || in_interrupt() || (size < BYTES_PER_WORD) ||
2109                 size > KMALLOC_MAX_SIZE) {
2110                 printk(KERN_ERR "%s: Early error in slab %s\n", __func__,
2111                                 name);
2112                 BUG();
2113         }
2114
2115         /*
2116          * We use cache_chain_mutex to ensure a consistent view of
2117          * cpu_online_mask as well.  Please see cpuup_callback
2118          */
2119         if (slab_is_available()) {
2120                 get_online_cpus();
2121                 mutex_lock(&cache_chain_mutex);
2122         }
2123
```

```
2124        list_for_each_entry(pc, &cache_chain, next) {
2125                char tmp;
2126                int res;
2127
2128                /*
2129                 * This happens when the module gets unloaded and doesn't
2130                 * destroy its slab cache and no-one else reuses the vmalloc
2131                 * area of the module.    Print a warning.
2132                 */
2133                res = probe_kernel_address(pc->name, tmp);
2134                if (res) {
2135                        printk(KERN_ERR
2136                                "SLAB: cache with size %d has lost its name\n",
2137                                pc->buffer_size);
2138                        continue;
2139                }
2140
2141                if (!strcmp(pc->name, name)) {
2142                        printk(KERN_ERR
2143                                "kmem_cache_create: duplicate cache %s\n", name);
2144                        dump_stack();
2145                        goto oops;
2146                }
2147        }
2148
2149 #if DEBUG
2150        WARN_ON(strchr(name, ' ')); /* It confuses parsers */
2151 #if FORCED_DEBUG
2152        /*
2153         * Enable redzoning and last user accounting, except for caches with
2154         * large objects, if the increased size would increase the object size
```

```
2155         * above the next power of two: caches with object sizes just above a
2156         * power of two have a significant amount of internal fragmentation.
2157         */
2158        if (size < 4096 || fls(size - 1) == fls(size-1 + REDZONE_ALIGN +
2159                                        2 * sizeof(unsigned long long)))
2160                flags |= SLAB_RED_ZONE | SLAB_STORE_USER;
2161        if (!(flags & SLAB_DESTROY_BY_RCU))
2162                flags |= SLAB_POISON;
2163 #endif
2164        if (flags & SLAB_DESTROY_BY_RCU)
2165                BUG_ON(flags & SLAB_POISON);
2166 #endif
2167        /*
2168         * Always checks flags, a caller might be expecting debug support which
2169         * isn't available.
2170         */
2171        BUG_ON(flags & ~CREATE_MASK);
2172
2173        /*
2174         * Check that size is in terms of words.  This is needed to avoid
2175         * unaligned accesses for some archs when redzoning is used, and makes
2176         * sure any on-slab bufctl's are also correctly aligned.
2177         */
2178        if (size & (BYTES_PER_WORD - 1)) {
2179                size += (BYTES_PER_WORD - 1);
2180                size &= ~(BYTES_PER_WORD - 1);
2181        }
```

2178-2181 行：按计算机字处理对齐对象。

```
2182
2183        /* calculate the final buffer alignment: */
2184
```

```
/* 1) arch recommendation: can be overridden for debug */
2186        if (flags & SLAB_HWCACHE_ALIGN) {
2187                /*
2188                 * Default alignment: as specified by the arch code.   Except if
2189                 * an object is really small, then squeeze multiple objects into
2190                 * one cacheline.
2191                 */
2192                ralign = cache_line_size();
2193                while (size <= ralign / 2)
2194                        ralign /= 2;
2195        } else {
2196                ralign = BYTES_PER_WORD;
2197        }
```

2186-2197 行：如果设置了 SLAB_HWCACHE_ALIGN 标志，则内核按照特定于体系结构函数 cache_line_size 给出的值来对齐数据。内核会保证将尽可能多的对象填充到一个缓存中。如果没有设置 SLAB_HWCACHE_ALIGN 标志，则对齐值为机器字。

```
2198
2199        /*
2200         * Redzoning and user store require word alignment or possibly larger.
2201         * Note this will be overridden by architecture or caller mandated
2202         * alignment if either is greater than BYTES_PER_WORD.
2203         */
2204        if (flags & SLAB_STORE_USER)
2205                ralign = BYTES_PER_WORD;
2206
2207        if (flags & SLAB_RED_ZONE) {
2208                ralign = REDZONE_ALIGN;
2209                /* If redzoning, ensure that the second redzone is suitably
2210                 * aligned, by adjusting the object size accordingly. */
2211                size += REDZONE_ALIGN - 1;
2212                size &= ~(REDZONE_ALIGN - 1);
```

```
2213        }
2214
2215        /* 2) arch mandated alignment */
2216        if (ralign < ARCH_SLAB_MINALIGN) {
2217                ralign = ARCH_SLAB_MINALIGN;
2218        }
```

2215-2218：处理体系结构强制的最小对齐值。

```
2219        /* 3) caller mandated alignment */
2220        if (ralign < align) {
2221                ralign = align;
2222        }
```

2219-2222：调用者强制的对齐值。

```
2223        /* disable debug if necessary */
2224        if (ralign > __alignof__(unsigned long long))
2225                flags &= ~(SLAB_RED_ZONE | SLAB_STORE_USER);
2226        /*
2227         * 4) Store it.
2228         */
2229        align = ralign;
```

2229 行：保存对齐值。

```
2230
2231        if (slab_is_available())
2232                gfp = GFP_KERNEL;
2233        else
2234                gfp = GFP_NOWAIT;
2235
2236        /* Get cache's description obj. */
2237        cachep = kmem_cache_zalloc(&cache_cache, gfp);
```

2237：分配缓存对象。

```
2238        if (!cachep)
2239                goto oops;
```

```
2240
2241 #if DEBUG
2242        cachep->obj_size = size;
```

2242：设置缓存中对象的大小。

```
2243
/*
2245         * Both debugging options require word-alignment which is calculated
2246         * into align above.
2247         */
2248        if (flags & SLAB_RED_ZONE) {
2249                /* add space for red zone words */
2250                cachep->obj_offset += sizeof(unsigned long long);
2251                size += 2 * sizeof(unsigned long long);
2252        }
2253        if (flags & SLAB_STORE_USER) {
2254                /* user store requires one word storage behind the end of
2255                 * the real object. But if the second red zone needs to be
2256                 * aligned to 64 bits, we must allow that much space.
2257                 */
2258                if (flags & SLAB_RED_ZONE)
2259                        size += REDZONE_ALIGN;
2260                else
2261                        size += BYTES_PER_WORD;
2262        }
2263 #if FORCED_DEBUG && defined(CONFIG_DEBUG_PAGEALLOC)
2264        if (size >= malloc_sizes[INDEX_L3 + 1].cs_size
2265                && cachep->obj_size > cache_line_size() && size < PAGE_SIZE) {
2266                cachep->obj_offset += PAGE_SIZE - size;
2267                size = PAGE_SIZE;
2268        }
2269 #endif
```

```
2270 #endif
2271
2272        /*
2273         * Determine if the slab management is 'on' or 'off' slab.
2274         * (bootstrapping cannot cope with offslab caches so don't do
2275         * it too early on. Always use on-slab management when
2276         * SLAB_NOLEAKTRACE to avoid recursive calls into kmemleak)
2277         */
2278        if ((size >= (PAGE_SIZE >> 3)) && !slab_early_init &&
2279                !(flags & SLAB_NOLEAKTRACE))
2280                /*
2281                 * Size is large, assume best to place the slab management obj
2282                 * off-slab (should allow better packing of objs).
2283                 */
2284                flags |= CFLGS_OFF_SLAB;
```

2278-2284 行：如果对象长度大于页面的 1/8，则 slab 头存储在 slab 之外。

```
2285
2286        size = ALIGN(size, align);
```

2286 行：将 size 设置到对齐值。

```
2287
2288        left_over = calculate_slab_order(cachep, size, align, flags);
```

2288：通过 calculate_slab_order 函数，设置 slab 长度。

```
2289
2290        if (!cachep->num) {
2291                printk(KERN_ERR
2292                        "kmem_cache_create: couldn't create cache %s.\n", name);
2293                kmem_cache_free(&cache_cache, cachep);
2294                cachep = NULL;
2295                goto oops;
2296        }
```

```
2297        slab_size = ALIGN(cachep->num * sizeof(kmem_bufctl_t)
2298                    + sizeof(struct slab), align);
```

2297-2298：计算 slab 头部长度，并对齐到 align。

```
2299
2300        /*
2301         * If the slab has been placed off-slab, and we have enough space then
2302         * move it on-slab. This is at the expense of any extra colouring.
2303         */
2304        if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
2305                flags &= ~CFLGS_OFF_SLAB;
2306                left_over -= slab_size;
2307        }
2308
2309        if (flags & CFLGS_OFF_SLAB) {
2310                /* really off slab. No need for manual alignment */
2311                slab_size =
2312                        cachep->num * sizeof(kmem_bufctl_t) + sizeof(struct slab);
2313
2314 #ifdef CONFIG_PAGE_POISONING
2315                /* If we're going to use the generic kernel_map_pages()
2316                 * poisoning, then it's going to smash the contents of
2317                 * the redzone and userword anyhow, so switch them off.
2318                 */
2319                if (size % PAGE_SIZE == 0 && flags & SLAB_POISON)
2320                        flags &= ~(SLAB_RED_ZONE | SLAB_STORE_USER);
2321 #endif
2322        }
2323
2324        cachep->colour_off = cache_line_size();
2325        /* Offset must be a multiple of the alignment. */
2326        if (cachep->colour_off < align)
```

```
2327            cachep->colour_off = align;
2328        cachep->colour = left_over / cachep->colour_off;
2329        cachep->slab_size = slab_size;
2330        cachep->flags = flags;
2331        cachep->gfpflags = 0;
2332        if (CONFIG_ZONE_DMA_FLAG && (flags & SLAB_CACHE_DMA))
2333            cachep->gfpflags |= GFP_DMA;
2334        cachep->buffer_size = size;
2335        cachep->reciprocal_buffer_size = reciprocal_value(size);
```

2324-2335：设置缓存中的一些其它变量。

```
2336
2337        if (flags & CFLGS_OFF_SLAB) {
2338            cachep->slabp_cache = kmem_find_general_cachep(slab_size, 0u);
2339            /*
2340             * This is a possibility for one of the malloc_sizes caches.
2341             * But since we go off slab only for object size greater than
2342             * PAGE_SIZE/8, and malloc_sizes gets created in ascending order,
2343             * this should not happen at all.
2344             * But leave a BUG_ON for some lucky dude.
2345             */
2346            BUG_ON(ZERO_OR_NULL_PTR(cachep->slabp_cache));
2347        }
2348        cachep->ctor = ctor;
2349        cachep->name = name;
2350
2351        if (setup_cpu_cache(cachep, gfp)) {
2352            __kmem_cache_destroy(cachep);
2353            cachep = NULL;
2354            goto oops;
2355        }
```

2351-2354 行：处理了 slab 布局之后，创建缓存还需要设置 per-CPU 缓存，该过程委托给 setup_cpu_cache 函数。

```
2356
2357        /* cache setup completed, link it into the list */
2358        list_add(&cachep->next, &cache_chain);
```

2358 行：将创建的缓存添加到 cache_chain 链表中。

```
2359 oops:
2360        if (!cachep && (flags & SLAB_PANIC))
2361                panic("kmem_cache_create(): failed to create slab `%s'\n",
2362                        name);
2363        if (slab_is_available()) {
2364                mutex_unlock(&cache_chain_mutex);
2365                put_online_cpus();
2366        }
2367        return cachep;
2368 }
```

2.6.4.1.2 setup_cpu_cache

该函数创建缓存的 per-CPU 缓存。

```
2014 static int __init_refok setup_cpu_cache(struct kmem_cache *cachep,
gfp_t gfp)
2015 {
2016        if (g_cpucache_up == FULL)
2017                return enable_cpucache(cachep, gfp);
```

2016-2017 行：如果 g_cpucache_up 为 FULL，也就是说大小为 sizeof(array_cache)和大小为 sizeof(kmem_list3)通用缓存已经建立，也就是可以使用 kmalloc 分配内存，则调用 enable_cpucache 来初始化 per-CPU 缓存。

```
2018
2019        if (g_cpucache_up == NONE) {
2020                /*
```

```
2021             * Note: the first kmem_cache_create must create the cache
2022             * that's used by kmalloc(24), otherwise the creation of
2023             * further caches will BUG().
2024             */
2025            cachep->array[smp_processor_id()] =
&initarray_generic.cache;
```

2025 行：如果不能使用 kmalloc，则暂时使用初始化静态对象来设置 per-CPU。

```
2026
2027            /*
2028             * If the cache that's used by kmalloc(sizeof(kmem_list3)) is
2029             * the first cache, then we need to set up all its list3s,
2030             * otherwise the creation of further caches will BUG().
2031             */
2032            set_up_list3s(cachep, SIZE_AC);
```

2032 行：使用初始化静态对象设置 kmlist3

```
2033            if (INDEX_AC == INDEX_L3)
2034                g_cpucache_up = PARTIAL_L3;
2035            else
2036                g_cpucache_up = PARTIAL_AC;
```

2035-2036 行：改变 g_cpucache_up 状态。

```
2037        } else {
2038            cachep->array[smp_processor_id()] =
2039                kmalloc(sizeof(struct arraycache_init), gfp);
```

2038-2039 行：kmalloc 可用，则通过 kmalloc 分配。

```
2040
2041            if (g_cpucache_up == PARTIAL_AC) {
2042                set_up_list3s(cachep, SIZE_L3);
2043                g_cpucache_up = PARTIAL_L3;
```

2042-2043 行：可以进入这个分支的一定是建立 L3 缓存的时候，这时 AC 已建立，L3 正在建立，2042 行建立好后，L3 就可以使用了。所以设置 g_cupcache_up 标志为 L3。

```
2044            } else {
2045                    int node;
2046                    for_each_online_node(node) {
2047                            cachep->nodelists[node] =
2048                                    kmalloc_node(sizeof(struct kmem_list3),
2049                                            gfp, node);
2050                            BUG_ON(!cachep->nodelists[node]);
2051                            kmem_list3_init(cachep->nodelists[node]);
```

2045-2051 行：L3 标志已建立，则通过 kmalloc 分配。

```
2052                    }
2053            }
2054    }
2055    cachep->nodelists[numa_node_id()]->next_reap =
2056                    jiffies + REAPTIMEOUT_LIST3 +
2057                    ((unsigned long)cachep) % REAPTIMEOUT_LIST3;
2058
2059    cpu_cache_get(cachep)->avail = 0;
2060    cpu_cache_get(cachep)->limit = BOOT_CPUCACHE_ENTRIES;
2061    cpu_cache_get(cachep)->batchcount = 1;
2062    cpu_cache_get(cachep)->touched = 0;
2063    cachep->batchcount = 1;
2064    cachep->limit = BOOT_CPUCACHE_ENTRIES;
2065    return 0;
2066 }
```

2.6.4.1.3 calculate_slab_order 计算一个slab缓存的大小

```
1944 /**
```

```
1945  * calculate_slab_order - calculate size (page order) of slabs
1946  * @cachep: pointer to the cache that is being created
1947  * @size: size of objects to be created in this cache.
1948  * @align: required alignment for the objects.
1949  * @flags: slab allocation flags
1950  *
1951  * Also calculates the number of objects per slab.
1952  *
1953  * This could be made much more intelligent.  For now, try to avoid using
1954  * high order pages for slabs.  When the gfp() functions are more friendly
1955  * towards high-order requests, this should be changed.
1956  */
1957 static size_t calculate_slab_order(struct kmem_cache *cachep,
1958                         size_t size, size_t align, unsigned long flags)
1959 {
1960         unsigned long offslab_limit;
1961         size_t left_over = 0;
1962         int gfporder;
1963
1964         for (gfporder = 0; gfporder <= KMALLOC_MAX_ORDER; gfporder++) {
1965                 unsigned int num;
1966                 size_t remainder;
1967
1968                 cache_estimate(gfporder, size, align, flags, &remainder, &num);
1969                 if (!num)
1970                         continue;
1971
1972                 if (flags & CFLGS_OFF_SLAB) {
```

```
1973                /*
1974                 * Max number of objs-per-slab for caches which
1975                 * use off-slab slabs. Needed to avoid a possible
1976                 * looping condition in cache_grow().
1977                 */
1978                offslab_limit = size - sizeof(struct slab);
1979                offslab_limit /= sizeof(kmem_bufctl_t);
1980
1981                if (num > offslab_limit)
1982                        break;
1983        }
1984
1985        /* Found something acceptable - save it away */
1986        cachep->num = num;
1987        cachep->gfporder = gfporder;
1988        left_over = remainder;
```

1986-1988 行：设置数目，分配阶数。

```
1989
1990        /*
1991         * A VFS-reclaimable slab tends to have most allocations
1992         * as GFP_NOFS and we really don't want to have to be
allocating
1993         * higher-order pages when we are unable to shrink dcache.
1994         */
1995        if (flags & SLAB_RECLAIM_ACCOUNT)
1996                break;
1997
1998        /*
1999         * Large number of objects is good, but very large slabs are
2000         * currently bad for the gfp()s.
2001         */
```

```
2002            if (gfporder >= slab_break_gfp_order)
2003                break;
2004
/*
2006             * Acceptable internal fragmentation?
2007             */
2008            if (left_over * 8 <= (PAGE_SIZE << gfporder))
2009                break;
2010        }
2011        return left_over;
2012 }
```

该函数迭代查找理想的 slab 长度。基于给定对象长度，cache_estimate 函数针对特定的页数，来计算对象的数目、浪费的空间、着色需要的空间。calculate_slab_order 会循环调用该函数，直到内核认为满意为止。

通过系统不断摸索，cache_estimate 找到一个布局，可以有如下要素描述：

- size：对象长度
- gfp_order：分配阶数
- num：slab 上对象的数目
- left_over：浪费的空间

head 指定 slab 头的大小，该布局对应如下公式：

PAGE_SIZE<<gfp_order = head + num*size + left_over(1)

如果 slab 头在 slab 外，则 head=0。否则，

head = sizeof(struct slab) + sizeof(kmem_bufctl_t)(2)

内核给出的迭代结束条件是：

- 8*left_over 小于 slab 长度，及浪费的空间小于 1/8 对象大小。(2008-2009)
- gfp_order 大于 slab_break_gfp_order。(2002-2003)
- slab 头在 slab 之外，num 大于 offslab_limit(1972-1983)

```
747 /*
```

```
748   * Calculate the number of objects and left-over bytes for a given buffer
size.
749   */
750 static void cache_estimate(unsigned long gfporder, size_t buffer_size,
751                            size_t align, int flags, size_t *left_over,
752                            unsigned int *num)
753 {
754         int nr_objs;
755         size_t mgmt_size;
756         size_t slab_size = PAGE_SIZE << gfporder;
757
758         /*
759          * The slab management structure can be either off the slab or
760          * on it. For the latter case, the memory allocated for a
761          * slab is used for:
762          *
763          * - The struct slab
764          * - One kmem_bufctl_t for each object
765          * - Padding to respect alignment of @align
766          * - @buffer_size bytes for each object
767          *
768          * If the slab management structure is off the slab, then the
769          * alignment will already be calculated into the size. Because
770          * the slabs are all pages aligned, the objects will be at the
771          * correct alignment when allocated.
772          */
773         if (flags & CFLGS_OFF_SLAB) {
774                 mgmt_size = 0;
775                 nr_objs = slab_size / buffer_size;
776
    if (nr_objs > SLAB_LIMIT)
778                         nr_objs = SLAB_LIMIT;
```

```
779        } else {
780                /*
781                 * Ignore padding for the initial guess. The padding
782                 * is at most @align-1 bytes, and @buffer_size is at
783                 * least @align. In the worst case, this result will
784                 * be one greater than the number of objects that fit
785                 * into the memory allocation when taking the padding
786                 * into account.
787                 */
788                nr_objs = (slab_size - sizeof(struct slab)) /
789                          (buffer_size + sizeof(kmem_bufctl_t));
790
791                /*
792                 * This calculated number will be either the right
793                 * amount, or one greater than what we want.
794                 */
795                if (slab_mgmt_size(nr_objs, align) + nr_objs*buffer_size
796                                > slab_size)
797                        nr_objs--;
798
799                if (nr_objs > SLAB_LIMIT)
800                        nr_objs = SLAB_LIMIT;
801
802                mgmt_size = slab_mgmt_size(nr_objs, align);
803        }
804        *num = nr_objs;
805        *left_over = slab_size - nr_objs*buffer_size - mgmt_size;
806 }
```

2.6.4.2 缓存销毁kmem_cache_destroy

```
2527 /**
```

```
2528  * kmem_cache_destroy - delete a cache
2529  * @cachep: the cache to destroy
2530  *
2531  * Remove a &struct kmem_cache object from the slab cache.
2532  *
2533  * It is expected this function will be called by a module when it is
2534  * unloaded.  This will remove the cache completely, and avoid a duplicate
2535  * cache being allocated each time a module is loaded and unloaded, if the
2536  * module doesn't have persistent in-kernel storage across loads and unloads.
2537  *
2538  * The cache must be empty before calling this function.
2539  *
2540  * The caller must guarantee that noone will allocate memory from the cache
2541  * during the kmem_cache_destroy().
2542  */
2543 void kmem_cache_destroy(struct kmem_cache *cachep)
2544 {
2545       BUG_ON(!cachep || in_interrupt());
2546
2547       /* Find the cache in the chain of caches. */
2548       get_online_cpus();
2549       mutex_lock(&cache_chain_mutex);
2550       /*
2551        * the chain is never empty, cache_cache is never destroyed
2552        */
2553       list_del(&cachep->next);
2554       if (__cache_shrink(cachep)) {
2555             slab_error(cachep, "Can't free all objects");
```

```
2556                list_add(&cachep->next, &cache_chain);
2557                mutex_unlock(&cache_chain_mutex);
2558                put_online_cpus();
2559                return;
2560        }
2561
2562        if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU))
2563                rcu_barrier();
2564
2565        __kmem_cache_destroy(cachep);
2566        mutex_unlock(&cache_chain_mutex);
2567        put_online_cpus();
2568 }
2569 EXPORT_SYMBOL(kmem_cache_destroy);
```

2.6.5 特定对象的分配与释放

2.6.5.1 对象分配kmem_cache_alloc

```
3563 /**
3564    * kmem_cache_alloc - Allocate an object
3565    * @cachep: The cache to allocate from.
3566    * @flags: See kmalloc().
3567    *
3568    * Allocate an object from this cache.    The flags are only relevant
3569    * if the cache has no available objects.
3570    */
3571 void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
3572 {
3573        void *ret = __cache_alloc(cachep, flags,
__builtin_return_address(0));
```

```
3574
3575        trace_kmem_cache_alloc(_RET_IP_, ret,
3576                               obj_size(cachep), cachep->buffer_size, flags);
3577
3578        return ret;
3579 }
```

调用__cache_alloc 完成实际的工作

```
3399 static __always_inline void *
3400 __cache_alloc(struct kmem_cache *cachep, gfp_t flags, void *caller)
3401 {
3402        unsigned long save_flags;
3403        void *objp;
3404
3405        flags &= gfp_allowed_mask;
3406
3407        lockdep_trace_alloc(flags);
3408
3409        if (slab_should_failslab(cachep, flags))
3410                return NULL;
3411
3412        cache_alloc_debugcheck_before(cachep, flags);
3413        local_irq_save(save_flags);
3414        objp = __do_cache_alloc(cachep, flags);
3415        local_irq_restore(save_flags);
3416        objp = cache_alloc_debugcheck_after(cachep, flags, objp, caller);
3417        kmemleak_alloc_recursive(objp, obj_size(cachep), 1,
cachep->flags,
3418                                 flags);
3419        prefetchw(objp);
3420
3421        if (likely(objp))
```

```
3422            kmemcheck_slab_alloc(cachep, flags, objp, obj_size(cachep));
3423
3424        if (unlikely((flags & __GFP_ZERO) && objp))
3425            memset(objp, 0, obj_size(cachep));
3426
3427        return objp;
3428 }
```

__do_cache_alloc 会调用____cache_alloc 来完成工作。

```
3106 static inline void *____cache_alloc(struct kmem_cache *cachep, gfp_t
flags)
3107 {
3108        void *objp;
3109        struct array_cache *ac;
3110
3111        check_irq_off();
3112
3113        ac = cpu_cache_get(cachep);
3114        if (likely(ac->avail)) {
3115            STATS_INC_ALLOCHIT(cachep);
3116            ac->touched = 1;
3117            objp = ac->entry[--ac->avail];
```

3115-3117 行：如果缓存中还有空间可用，直接从缓存获取。

```
3118        } else {
3119            STATS_INC_ALLOCMISS(cachep);
3120            objp = cache_alloc_refill(cachep, flags);
3121            /*
3122             * the 'ac' may be updated by cache_alloc_refill(),
3123             * and kmemleak_erase() requires its correct value.
3124             */
3125            ac = cpu_cache_get(cachep);
```

3120-3125 行：如果缓存不可用，则调用 cache_alloc_refill 分配。

```
3126        }
3127        /*
3128         * To avoid a false negative, if an object that is in one of the
3129         * per-CPU caches is leaked, we need to make sure kmemleak
doesn't
3130         * treat the array pointers as a reference to the object.
3131         */
3132        if (objp)
3133                kmemleak_erase(&ac->entry[ac->avail]);
3134        return objp;
3135 }
```

2.6.5.1.1 cache_alloc_refill

```
2937 static void *cache_alloc_refill(struct kmem_cache *cachep, gfp_t flags)
2938 {
2939        int batchcount;
2940        struct kmem_list3 *l3;
2941        struct array_cache *ac;
2942        int node;
2943
2944 retry:
2945        check_irq_off();
2946        node = numa_node_id();
2947        ac = cpu_cache_get(cachep);
2948        batchcount = ac->batchcount;
2949        if (!ac->touched && batchcount > BATCHREFILL_LIMIT) {
2950                /*
2951                 * If there was little recent activity on this cache, then
2952                 * perform only a partial refill.    Otherwise we could generate
2953                 * refill bouncing.
2954                 */
```

```
2955            batchcount = BATCHREFILL_LIMIT;
2956        }
2957        l3 = cachep->nodelists[node];
2958
2959        BUG_ON(ac->avail > 0 || !l3);
2960        spin_lock(&l3->list_lock);
2961
2962        /* See if we can refill from the shared array */
2963        if (l3->shared && transfer_objects(ac, l3->shared, batchcount)) {
2964            l3->shared->touched = 1;
2965            goto alloc_done;
2966        }
2967
2968        while (batchcount > 0) {
2969            struct list_head *entry;
2970            struct slab *slabp;
2971            /* Get slab alloc is to come from. */
2972            entry = l3->slabs_partial.next;
2973            if (entry == &l3->slabs_partial) {
2974                l3->free_touched = 1;
2975                entry = l3->slabs_free.next;
2976                if (entry == &l3->slabs_free)
2977                    goto must_grow;
2978            }
```

2972-2978：先遍历部分空闲的 slab，如果没有，在遍历空闲的 slab，如果没有，则必须增加缓存。

```
2979
2980            slabp = list_entry(entry, struct slab, list);
2981            check_slabp(cachep, slabp);
2982            check_spinlock_acquired(cachep);
2983
```

```
2984            /*
2985             * The slab was either on partial or free list so
2986             * there must be at least one object available for
2987             * allocation.
2988             */
2989            BUG_ON(slabp->inuse >= cachep->num);
2990
2991            while (slabp->inuse < cachep->num && batchcount--) {
2992                    STATS_INC_ALLOCED(cachep);
2993                    STATS_INC_ACTIVE(cachep);
2994                    STATS_SET_HIGH(cachep);
2995
2996                    ac->entry[ac->avail++] = slab_get_obj(cachep, slabp,
2997                                                          node);
```

2996-2997：调用 slab_get_obj 分配一个对象。

```
2998            }
2999            check_slabp(cachep, slabp);
3000
3001            /* move slabp to correct slabp list: */
3002            list_del(&slabp->list);
3003            if (slabp->free == BUFCTL_END)
3004                    list_add(&slabp->list, &l3->slabs_full);
3005            else
3006                    list_add(&slabp->list, &l3->slabs_partial);
3007    }
3008
3009 must_grow:
3010    l3->free_objects -= ac->avail;
3011 alloc_done:
3012    spin_unlock(&l3->list_lock);
3013
```

```
3014        if (unlikely(!ac->avail)) {
3015                int x;
3016                x = cache_grow(cachep, flags | GFP_THISNODE, node,
NULL);
```

3016：如果没有空闲的 slab，则必须调用 cache_grow 增加 slab 缓存。

```
3017
3018                /* cache_grow can reenable interrupts, then ac could change. */
3019                ac = cpu_cache_get(cachep);
3020                if (!x && ac->avail == 0)       /* no objects in sight? abort */
3021                        return NULL;
3022
3023                if (!ac->avail)         /* objects refilled by interrupt? */
3024                        goto retry;
3025        }
3026        ac->touched = 1;
3027        return ac->entry[--ac->avail];
3028 }
```

```
2676 static void *slab_get_obj(struct kmem_cache *cachep, struct slab *slabp,
2677                        int nodeid)
2678 {
2679        void *objp = index_to_obj(cachep, slabp, slabp->free);
2680        kmem_bufctl_t next;
2681
2682        slabp->inuse++;
2683        next = slab_bufctl(slabp)[slabp->free];
2684 #if DEBUG
2685        slab_bufctl(slabp)[slabp->free] = BUFCTL_FREE;
2686        WARN_ON(slabp->nodeid != nodeid);
2687 #endif
2688        slabp->free = next;
```

```
2689
2690        return objp;
2691 }
```

2.6.5.1.2 cache_grow

该函数的流程是：

计算偏移量和下一个颜色

kmem_getpages->alloc_pages_node(伙伴系统分配页块)

alloc_slabmgt

设置页指针

cache_init_objs

将 slab 添加到缓存

```
2737 /*
2738  * Grow (by 1) the number of slabs within a cache.  This is called by
2739  * kmem_cache_alloc() when there are no active objs left in a cache.
2740  */
2741 static int cache_grow(struct kmem_cache *cachep,
2742                 gfp_t flags, int nodeid, void *objp)
2743 {
2744         struct slab *slabp;
2745         size_t offset;
2746         gfp_t local_flags;
2747         struct kmem_list3 *l3;
2748
2749         /*
2750          * Be lazy and only check for valid flags here,  keeping it out of the
2751          * critical path in kmem_cache_alloc().
2752          */
2753         BUG_ON(flags & GFP_SLAB_BUG_MASK);
2754         local_flags = flags &
(GFP_CONSTRAINT_MASK|GFP_RECLAIM_MASK);
```

```
2755
2756        /* Take the l3 list lock to change the colour_next on this node */
2757        check_irq_off();
2758        l3 = cachep->nodelists[nodeid];
2759        spin_lock(&l3->list_lock);
2760
2761        /* Get colour for the slab, and cal the next value. */
2762        offset = l3->colour_next;
2763        l3->colour_next++;
2764        if (l3->colour_next >= cachep->colour)
2765                l3->colour_next = 0;
2766        spin_unlock(&l3->list_lock);
2767
2768        offset *= cachep->colour_off;
```

计算偏移和下一个颜色。

```
2769
2770        if (local_flags & __GFP_WAIT)
2771                local_irq_enable();
2772
2773        /*
2774         * The test for missing atomic flag is performed here, rather than
2775         * the more obvious place, simply to reduce the critical path length
2776         * in kmem_cache_alloc(). If a caller is seriously mis-behaving they
2777         * will eventually be caught here (where it matters).
2778         */
2779        kmem_flagcheck(cachep, flags);
2780
2781        /*
2782         * Get mem for the objs.    Attempt to allocate a physical page from
2783         * 'nodeid'.
2784         */
```

```
2785        if (!objp)
2786            objp = kmem_getpages(cachep, local_flags, nodeid);
```

调用伙伴系统分配页块。

```
2787        if (!objp)
2788            goto failed;
2789
2790        /* Get slab management. */
2791        slabp = alloc_slabmgmt(cachep, objp, offset,
2792                local_flags & ~GFP_CONSTRAINT_MASK, nodeid);
```

2791-2792：分配 slab 头部管理信息，如果 slab 头部在外，则使用 slabp_cache 指向的普通缓存分配数据。如果 slab 在内部，则已经分配。两种情况下，都需要初始化 slab 头部一些信息。

```
2793        if (!slabp)
2794            goto opps1;
2795
2796        slab_map_pages(cachep, slabp, objp);
```

设置 slab 各页与缓存和 slab 之间的关系。

```
2797
2798        cache_init_objs(cachep, slabp);
```

初始化 slab 中所有对象。

```
2799
2800        if (local_flags & __GFP_WAIT)
2801            local_irq_disable();
2802        check_irq_off();
2803        spin_lock(&l3->list_lock);
2804
2805        /* Make slab active. */
2806        list_add_tail(&slabp->list, &(l3->slabs_free));
```

slab 添加到 slabs_freel 链表上。

```
2807        STATS_INC_GROWN(cachep);
2808        l3->free_objects += cachep->num;
```

```
2809        spin_unlock(&l3->list_lock);
2810        return 1;
2811 opps1:
2812        kmem_freepages(cachep, objp);
2813 failed:
2814        if (local_flags & __GFP_WAIT)
2815                local_irq_disable();
2816        return 0;
2817 }
```

```
2713 /*
2714    * Map pages beginning at addr to the given cache and slab. This is
required
2715    * for the slab allocator to be able to lookup the cache and slab of a
2716    * virtual address for kfree, ksize, kmem_ptr_validate, and slab
debugging.
2717    */
2718 static void slab_map_pages(struct kmem_cache *cache, struct slab
*slab,
2719                            void *addr)
2720 {
2721        int nr_pages;
2722        struct page *page;
2723
2724        page = virt_to_page(addr);
2725
2726        nr_pages = 1;
2727        if (likely(!PageCompound(page)))
2728                nr_pages <<= cache->gfporder;
2729
2730        do {
2731                page_set_cache(page, cache);
```

```
2732                page_set_slab(page, slab);
2733                page++;
2734        } while (--nr_pages);
2735 }
```

2.6.5.2 对象的释放kmem_cache_free

```
3735 /**
3736  * kmem_cache_free - Deallocate an object
3737  * @cachep: The cache the allocation was from.
3738  * @objp: The previously allocated object.
3739  *
3740  * Free an object which was previously allocated from this
3741  * cache.
3742  */
3743 void kmem_cache_free(struct kmem_cache *cachep, void *objp)
3744 {
3745         unsigned long flags;
3746
3747         local_irq_save(flags);
3748         debug_check_no_locks_freed(objp, obj_size(cachep));
3749         if (!(cachep->flags & SLAB_DEBUG_OBJECTS))
3750                 debug_check_no_obj_freed(objp, obj_size(cachep));
3751         __cache_free(cachep, objp);
3752         local_irq_restore(flags);
3753
3754         trace_kmem_cache_free(_RET_IP_, objp);
3755 }
3756 EXPORT_SYMBOL(kmem_cache_free);
```

调用__cache_free 完成工作

```
3528 /*
3529  * Release an obj back to its cache. If the obj has a constructed state, it must
3530  * be in this state _before_ it is released.  Called with disabled ints.
3531  */
3532 static inline void __cache_free(struct kmem_cache *cachep, void *objp)
3533 {
3534         struct array_cache *ac = cpu_cache_get(cachep);
3535
3536         check_irq_off();
3537         kmemleak_free_recursive(objp, cachep->flags);
3538         objp = cache_free_debugcheck(cachep, objp, __builtin_return_address(0));
3539
3540         kmemcheck_slab_free(cachep, objp, obj_size(cachep));
3541
3542         /*
3543          * Skip calling cache_free_alien() when the platform is not numa.
3544          * This will avoid cache misses that happen while accessing slabp (which
3545          * is per page memory  reference) to get nodeid. Instead use a global
3546          * variable to skip the call, which is mostly likely to be present in
3547          * the cache.
3548          */
3549         if (nr_online_nodes > 1 && cache_free_alien(cachep, objp))
3550                 return;
3551
3552         if (likely(ac->avail < ac->limit)) {
3553                 STATS_INC_FREEHIT(cachep);
3554                 ac->entry[ac->avail++] = objp;
3555                 return;
```

如果缓存未满，直接放入缓存。

```
3556        } else {
3557                STATS_INC_FREEMISS(cachep);
3558                cache_flusharray(cachep, ac);
3559                ac->entry[ac->avail++] = objp;
```

如果缓存已满，则需要使用 cache_flusharray 将一部分对象移回 slab。

```
3560        }
3561 }
```

2.6.5.2.1 cache_flusharray

该函数实现的功能是：将对象从缓存移动到 slab，并将剩余的对象向数组起始处移动。

```
3477 static void cache_flusharray(struct kmem_cache *cachep, struct
array_cache *ac)
3478 {
3479        int batchcount;
3480        struct kmem_list3 *l3;
3481        int node = numa_node_id();
3482
3483        batchcount = ac->batchcount;
3484 #if DEBUG
3485        BUG_ON(!batchcount || batchcount > ac->avail);
3486 #endif
3487        check_irq_off();
3488        l3 = cachep->nodelists[node];
3489        spin_lock(&l3->list_lock);
3490        if (l3->shared) {
3491                struct array_cache *shared_array = l3->shared;
3492                int max = shared_array->limit - shared_array->avail;
3493                if (max) {
3494                        if (batchcount > max)
```

```
3495                    batchcount = max;
3496                memcpy(&(shared_array->entry[shared_array->avail]),
3497                        ac->entry, sizeof(void *) * batchcount);
3498                shared_array->avail += batchcount;
3499                goto free_done;
3500            }
```

如果 shared 不空，则直接移动到 shared，不用移回 slab。

```
3501        }
3502
3503        free_block(cachep, ac->entry, batchcount, node);
```

调用 free_block 将对象移回 slab。

```
3504 free_done:
3505 #if STATS
3506        {
3507            int i = 0;
3508            struct list_head *p;
3509
3510            p = l3->slabs_free.next;
3511            while (p != &(l3->slabs_free)) {
3512                struct slab *slabp;
3513
3514                slabp = list_entry(p, struct slab, list);
3515                BUG_ON(slabp->inuse);
3516
3517                i++;
3518                p = p->next;
3519            }
3520            STATS_SET_FREEABLE(cachep, i);
3521        }
3522 #endif
3523        spin_unlock(&l3->list_lock);
```

```
3524        ac->avail -= batchcount;
3525        memmove(ac->entry, &(ac->entry[batchcount]), sizeof(void
*)*ac->avail);
```

将剩余的对象向数组前端移动。

```
3526 }
```

2.6.5.2.2 free_block

```
3430 /*
3431   * Caller needs to acquire correct kmem_list's list_lock
3432   */
3433 static void free_block(struct kmem_cache *cachep, void **objpp, int
nr_objects,
3434                        int node)
3435 {
3436        int i;
3437        struct kmem_list3 *l3;
3438
3439        for (i = 0; i < nr_objects; i++) {
3440              void *objp = objpp[i];
3441              struct slab *slabp;
3442
3443              slabp = virt_to_slab(objp);
```

根据地址得到 page，根据 page 得到所属 slab。

```
3444              l3 = cachep->nodelists[node];
3445              list_del(&slabp->list);
3446              check_spinlock_acquired_node(cachep, node);
3447              check_slabp(cachep, slabp);
3448              slab_put_obj(cachep, slabp, objp, node);
```

将对象放回 slab。

```
3449              STATS_DEC_ACTIVE(cachep);
```

```
3450            l3->free_objects++;
3451            check_slabp(cachep, slabp);
3452
3453            /* fixup slab chains */
3454            if (slabp->inuse == 0) {
3455                if (l3->free_objects > l3->free_limit) {
3456                    l3->free_objects -= cachep->num;
3457                    /* No need to drop any previously held
3458                     * lock here, even if we have a off-slab slab
3459                     * descriptor it is guaranteed to come from
3460                     * a different cache, refer to comments before
3461                     * alloc_slabmgmt.
3462                     */
3463                    slab_destroy(cachep, slabp);
3464                } else {
3465                    list_add(&slabp->list, &l3->slabs_free);
3466                }
```

3454-3466 行：如果 slab 未使用，则将 slab 返回空链表。如果缓存中的对象数目超过了预定义的数目 l3->free_limit，则调用 slab_destroy 将 slab 返回给伙伴系统。

```
3467            } else {
3468                /* Unconditionally move a slab to the end of the
3469                 * partial list on free - maximum time for the
3470                 * other objects to be freed, too.
3471                 */
3472                list_add_tail(&slabp->list, &l3->slabs_partial);
```

将 slab 挂入到 partial 链表。

```
3473            }
3474        }
3475 }
```

2.6.6 通用对象的创建和释放

2.6.6.1 创建kmalloc

kmalloc 函数根据 size 是否是产量，执行不同的流程。

如果是常量，则先找到对应的缓存，然后在缓存中分配。

否则调用__kmalloc

```
128 static __always_inline void *kmalloc(size_t size, gfp_t flags)
129 {
130         struct kmem_cache *cachep;
131         void *ret;
132
133         if (__builtin_constant_p(size)) {
134                 int i = 0;
135
136                 if (!size)
137                         return ZERO_SIZE_PTR;
138
139 #define CACHE(x) \
140                 if (size <= x) \
141                         goto found; \
142                 else \
143                         i++;
144 #include <linux/kmalloc_sizes.h>
145 #undef CACHE
146                 return NULL;
147 found:
148 #ifdef CONFIG_ZONE_DMA
149                 if (flags & GFP_DMA)
150                         cachep = malloc_sizes[i].cs_dmacachep;
151                 else
152 #endif
```

```
153                     cachep = malloc_sizes[i].cs_cachep;
154
155             ret = kmem_cache_alloc_notrace(cachep, flags);
156
157             trace_kmalloc(_THIS_IP_, ret,
158                           size, slab_buffer_size(cachep), flags);
159
160             return ret;
161         }
162         return __kmalloc(size, flags);
163 }
```

```
3715 void *__kmalloc(size_t size, gfp_t flags)
3716 {
3717         return __do_kmalloc(size, flags, __builtin_return_address(0));
3718 }
```

```
3685 /**
3686  * __do_kmalloc - allocate memory
3687  * @size: how many bytes of memory are required.
3688  * @flags: the type of memory to allocate (see kmalloc).
3689  * @caller: function caller for debug tracking of the caller
3690  */
3691 static __always_inline void *__do_kmalloc(size_t size, gfp_t flags,
3692                                           void *caller)
3693 {
3694         struct kmem_cache *cachep;
3695         void *ret;
3696
3697         /* If you want to save a few bytes .text space: replace
3698          * __ with kmem_.
```

```
3699         * Then kmalloc uses the uninlined functions instead of the inline
3700         * functions.
3701         */
3702        cachep = __find_general_cachep(size, flags);
```

在通用缓存中查找缓存。

```
3703        if (unlikely(ZERO_OR_NULL_PTR(cachep)))
3704                return cachep;
3705        ret = __cache_alloc(cachep, flags, caller);
```

调用__cache_alloc 分配。

```
3706
3707        trace_kmalloc((unsigned long) caller, ret,
3708                      size, cachep->buffer_size, flags);
3709
3710        return ret;
3711 }
```

2.6.6.2 释放kfree

```
3758 /**
3759  * kfree - free previously allocated memory
3760  * @objp: pointer returned by kmalloc.
3761  *
3762  * If @objp is NULL, no operation is performed.
3763  *
3764  * Don't free memory not originally allocated by kmalloc()
3765  * or you will run into trouble.
3766  */
3767 void kfree(const void *objp)
3768 {
3769        struct kmem_cache *c;
3770        unsigned long flags;
```

```
3771
3772        trace_kfree(_RET_IP_, objp);
3773
3774        if (unlikely(ZERO_OR_NULL_PTR(objp)))
3775                return;
3776        local_irq_save(flags);
3777        kfree_debugcheck(objp);
3778        c = virt_to_cache(objp);
```

有地址到缓存。

```
3779        debug_check_no_locks_freed(objp, obj_size(c));
3780        debug_check_no_obj_freed(objp, obj_size(c));
3781        __cache_free(c, (void *)objp);
```

调用__cache_free 释放对象。

```
3782        local_irq_restore(flags);
3783 }
3784 EXPORT_SYMBOL(kfree);
```

3. 虚拟内存（进程地址空间）管理

进程的地址空间按 3:1 被分为用户态和内核态两部分。所有进程共有一份内核态的地址空间，而用户态的地址空间为没个进程态所独有。

用户进程的虚拟地址地址空间是 linux 的一个重要抽象：它向没个运行进程提供了同样的系统视图，使得多个进程可以同时运行，而不会干扰到其它进程的内容。

进程的虚拟地址空间提供了物理内存按需调度的基石，它允许一个进程在比其需要的内存小的物理内存中运行。

在物理内存管理中，可以看到内核中的函数以相当直截了当的方式获得动态内存：

- 从伙伴系统中获取页框:__get_free_pages( ) __alloc_pages()
- slab 分配器:kmem_cache_alloc kmalloc

- 非连续物理内存分配：vmalloc vmalloc_32

这个函数都会尽量获得物理内存，内核使用这些简单的方法基于如下两个原因：

- 内核是操作系统中优先级最高的成分。如果某个内核函数请求动态内存，那么，必定有正当的理由，因此，没有必要延迟这个请求。
- 内核信任自己。所有内核函数都被假定是没有错误的，内核函数不必插入针对错误编程的任何保护措施。

当给用户态进程分配内存时，情况完全不同：

进程对动态内存的请求被认为是不紧迫的。一般来见，内核总是尽量推迟给用户态进程分配动态内存。

用户进程是不可信任的，内核必须能随时准备捕获用户态进程引起的所有寻址错误。

3.1 相关数据结构

进程地址空间由允许进程使用的全部线性地址空间组成。没个进程看到的线性地址空间集合是不同的。各个进程的线性地址空间互不关联。

与进程地址空间有关的信息全都包含在一个叫做内存描述符的结构中。

3.1.1 内存描述符mm_struct

include/linux/mm_types.h

```
222 struct mm_struct {
223         struct vm_area_struct * mmap;           /* list of VMAs */
224         struct rb_root mm_rb;
225         struct vm_area_struct * mmap_cache; /* last find_vma result */
226 #ifdef CONFIG_MMU
227         unsigned long (*get_unmapped_area) (struct file *filp,
228                                 unsigned long addr, unsigned long len,
229                                 unsigned long pgoff, unsigned long flags);
230         void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
231 #endif
232         unsigned long mmap_base;            /* base of mmap area */
```

```
233         unsigned long task_size;            /* size of task vm space */
234         unsigned long cached_hole_size;         /* if non-zero, the largest hole
below free_area_cache */
235         unsigned long free_area_cache;          /* first hole of size
cached_hole_size or larger */
236         pgd_t * pgd;
237         atomic_t mm_users;              /* How many users with user space?
*/
238         atomic_t mm_count;              /* How many references to "struct
mm_struct" (users count as 1) */
239         int map_count;                  /* number of VMAs */
240         struct rw_semaphore mmap_sem;
241         spinlock_t page_table_lock;         /* Protects page tables and some
counters */
242
243         struct list_head mmlist;            /* List of maybe swapped mm's.
These are globally strung
244                                     * together off init_mm.mmlist, and are
protected
245                                     * by mmlist_lock
246                                     */
247
248
249         unsigned long hiwater_rss;    /* High-watermark of RSS usage */
250         unsigned long hiwater_vm;     /* High-water virtual memory usage */
251
252         unsigned long total_vm, locked_vm, shared_vm, exec_vm;
253         unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
254         unsigned long start_code, end_code, start_data, end_data;
255         unsigned long start_brk, brk, start_stack;
256         unsigned long arg_start, arg_end, env_start, env_end;
257
```

```
258         unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for
/proc/PID/auxv */
259
260         /*
261          * Special counters, in some configurations protected by the
262          * page_table_lock, in other configurations by being atomic.
263          */
264         struct mm_rss_stat rss_stat;
265
266         struct linux_binfmt *binfmt;
267
268         cpumask_t cpu_vm_mask;
269
270         /* Architecture-specific MM context */
271         mm_context_t context;
272
273         /* Swap token stuff */
274         /*
275          * Last value of global fault stamp as seen by this process.
276          * In other words, this value gives an indication of how long
277          * it has been since this task got the token.
278          * Look at mm/thrash.c
279          */

280         unsigned int faultstamp;
281         unsigned int token_priority;
282         unsigned int last_interval;
283
284         unsigned long flags; /* Must use atomic bitops to access the bits */
285
286         struct core_state *core_state; /* coredumping support */
287 #ifdef CONFIG_AIO
```

```
289         struct hlist_head     ioctx_list;
290 #endif
291 #ifdef CONFIG_MM_OWNER
292         /*
293          * "owner" points to a task that is regarded as the canonical
294          * user/owner of this mm. All of the following must be true in
295          * order for it to be changed:
296          *
297          * current == mm->owner
298          * current->mm != mm
299          * new_owner->mm == mm
300          * new_owner->alloc_lock is held
301          */
302         struct task_struct *owner;
303 #endif
304
305 #ifdef CONFIG_PROC_FS
306         /* store ref to file /proc/<pid>/exe symlink points to */
307         struct file *exe_file;
308         unsigned long num_exe_file_vmas;
309 #endif
310 #ifdef CONFIG_MMU_NOTIFIER
311         struct mmu_notifier_mm *mmu_notifier_mm;
312 #endif
313 };
```

mmap：指向线性区对象的链表头。

mm_rb：指向线性区对象的红-黑树的根。

get_unmaped_area：在进程空间中搜索有效线性地址区间的方法

unmap_area：释放线性区地址空间时调用的方法。

mmap_base：标志第一个分区的匿名线性区或者文件内存映射的线性地址。（虚拟地址空间中用于内存映射的开始地址）。

pgd：指向页全局目录

mm_user：次使用计数器

mm_count：主使用计数器

map_count：线性区的个数。

start_code：可执行代码的起始地址

end_code：可执行代码的最后地址。

start_data：已初始化数据的开始地址

end_data：已初始化数据的结束地址

start_blk：堆得开始地址

brk：堆得当前最后地址。

start_stack：用户态堆栈的开始地址

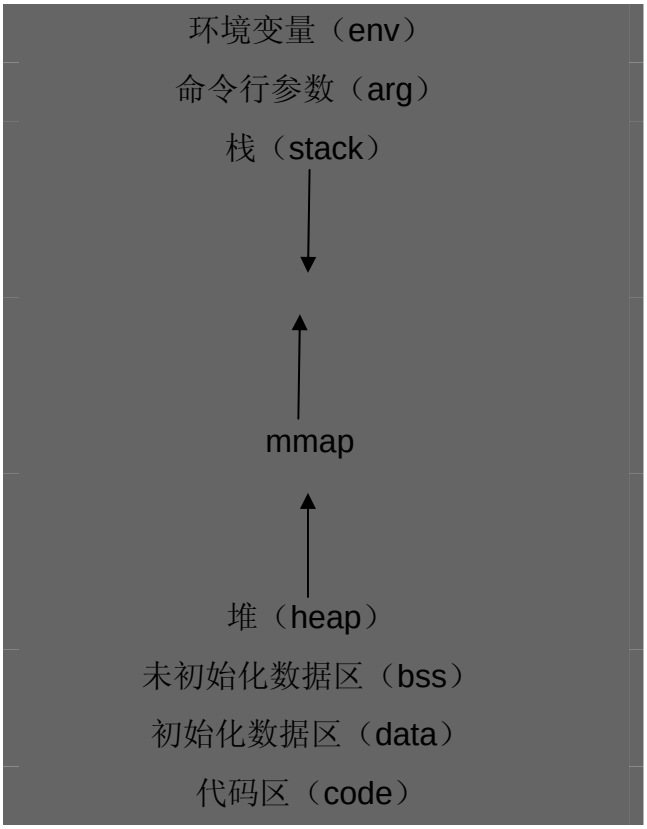
arg_start：命令行参数的开始地址

arg_end：命令行参数的最后地址

env_start：环境变量的开始地址

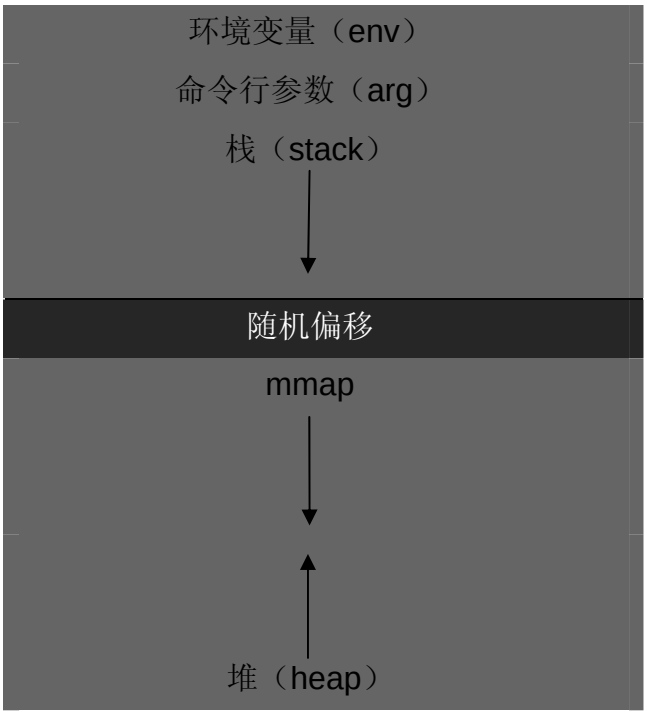
env_end：环境变量的最后地址。

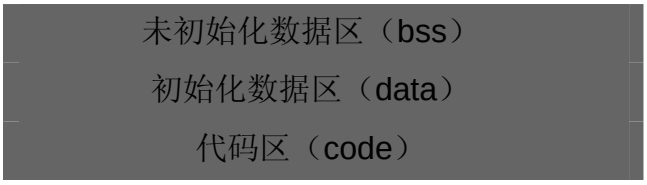
进程用户空间经典地址布局图



在经典的用户空间布局中，mmap_base 开始于 1G 处，这意味着堆空间最多可以分配 1G 的空间。很显然这不是很理想。

上述问题在于，内存映射位于虚拟地址空间的中间，在内核 2.6.7 版本之后，为 IA-32 引入了一个新的虚拟地址空间布局





上述方式是使用固定值限制栈空间的最大长度。由于栈是有界的，因此安置内存映射区域可以再栈末端立即开始。与经典方法相比，该区域是自顶向下扩展。

PF_RANDOMIZE 标志

如果进程设置了该标志，则内核不会为栈和内存映射的起点选择固定位置，而是在每次新进程启动时随机改变这些值的设置。这样做的好处是：给栈溢出带来了难度。

3.1.2 线性区vm_area_struct

```
124 /*
125  * This struct defines a memory VMM memory area. There is one of these
126  * per VM-area/task.    A VM area is any part of the process virtual
memory
127  * space that has a special rule for the page-fault handlers (ie a shared
128  * library, the executable area etc).
129  */
130 struct vm_area_struct {
131         struct mm_struct * vm_mm;     /* The address space we belong to. */
132         unsigned long vm_start;         /* Our start address within vm_mm. */
133         unsigned long vm_end;           /* The first byte after our end address
134                                            within vm_mm. */
135
136         /* linked list of VM areas per task, sorted by address */
137         struct vm_area_struct *vm_next;
138
139         pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
140         unsigned long vm_flags;         /* Flags, see mm.h. */
141
142         struct rb_node vm_rb;
```

```
143
144         /*
145          * For areas with an address space and backing store,
146          * linkage into the address_space->i_mmap prio tree, or
147          * linkage to the list of like vmas hanging off its node, or
148          * linkage of vma in the address_space->i_mmap_nonlinear list.
149          */
150         union {
151                 struct {
152                         struct list_head list;
153                         void *parent;     /* aligns with prio_tree_node parent */
154                         struct vm_area_struct *head;
155                 } vm_set;
156
157                 struct raw_prio_tree_node prio_tree_node;
158         } shared;
159
160         /*
161          * A file's MAP_PRIVATE vma can be in both i_mmap tree and
anon_vma
162          * list, after a COW of one of the file pages.    A MAP_SHARED vma
163          * can only be in the i_mmap tree.    An anonymous MAP_PRIVATE,
stack
164          * or brk vma (with NULL file) can only be in an anon_vma list.
165          */
166         struct list_head anon_vma_chain; /* Serialized by mmap_sem &
167                                           * page_table_lock */
168         struct anon_vma *anon_vma;    /* Serialized by page_table_lock */
169
170         /* Function pointers to deal with this struct. */
171         const struct vm_operations_struct *vm_ops;
172
```

```
173         /* Information about our backing store: */
174         unsigned long vm_pgoff;         /* Offset (within vm_file) in
PAGE_SIZE
175                                         units, *not* PAGE_CACHE_SIZE */
176         struct file * vm_file;          /* File we map to (can be NULL). */
177         void * vm_private_data;         /* was vm_pte (shared mem) */
178         unsigned long vm_truncate_count;/* truncate_count or restart_addr */
179
180 #ifndef CONFIG_MMU
181         struct vm_region *vm_region;       /* NOMMU mapping region */
182 #endif
183 #ifdef CONFIG_NUMA
184         struct mempolicy *vm_policy;       /* NUMA policy for the VMA */
185 #endif
186 };
```

vm_mm：指向线性区所在的内存描述符

vm_start，vm_end：线性区的开始和结束地址

vm_next：指向下一个线性区

vm_page_port：线性区中页框的访问权限

vm_flags：线性区标志

vm_rb：用于红-黑树的结构

vm_file：指向映射文件的文件对象

vm_private_data：指向内存区的私有数据

anon_vma_chain 和 anon_vma 用与管理源自匿名映射的共享页。指向相同页的映射都保存在一个双向链表上，anon_vma_chain 为表头。

shared：从文件到进程的虚拟地址空间的映射。

对于有地址空间和后备存储区域来说，shared 连接到 address_space->i_mmap 优先树或者连接到悬挂在优先树之外、类似的一组虚拟内存区域的链表，或连接到 address_space->i_mmap_nonlinear 链表中的虚拟内存区域。

3.2 基本函数

内核提供了各种函数来操作进程的虚拟内存区域。

3.2.1 find_vma

find_vma：该函数查找用户地址空间中结束地址在给定地址之后的第一个区域。及满足 addr < vm_area_struct->vm_end 条件的第一个区域。

该函数返回 vm_area_struct 指针或者是 NULL。如果返回 NULL，表示 addr 之后的地址还没有虚拟地址区域。

该函数过程很简单，就不分析了。

```
1568 /* Look up the first VMA which satisfies  addr < vm_end,  NULL if none.
*/
1569 struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long
addr)
1570 {
1571        struct vm_area_struct *vma = NULL;
1572
1573        if (mm) {
1574              /* Check the cache first. */
1575              /* (Cache hit rate is typically around 35%.) */
1576              vma = mm->mmap_cache;
1577              if (!(vma && vma->vm_end > addr && vma->vm_start <= addr))
{
1578                    struct rb_node * rb_node;
1579
1580                    rb_node = mm->mm_rb.rb_node;
1581                    vma = NULL;
1582
1583                    while (rb_node) {
1584                          struct vm_area_struct * vma_tmp;
1585
1586                          vma_tmp = rb_entry(rb_node,
```

```
1587                                    struct vm_area_struct, vm_rb);
1588
1589                            if (vma_tmp->vm_end > addr) {
1590                                    vma = vma_tmp;
1591                                    if (vma_tmp->vm_start <= addr)
1592                                            break;
1593                                    rb_node = rb_node->rb_left;
1594                            } else
1595                                    rb_node = rb_node->rb_right;
1596                    }
1597                    if (vma)
1598                            mm->mmap_cache = vma;
1599            }
1600    }
1601    return vma;
1602 }
```

3.2.2 get_unmapped_area：查找空闲的地址空间。

```
1536 unsigned long
1537 get_unmapped_area(struct file *file, unsigned long addr, unsigned long
len,
1538            unsigned long pgoff, unsigned long flags)
1539 {
1540    unsigned long (*get_area)(struct file *, unsigned long,
1541                              unsigned long, unsigned long, unsigned long);
1542
1543    unsigned long error = arch_mmap_check(addr, len, flags);
1544    if (error)
1545            return error;
1546
```

```
1547        /* Careful about overflows.. */
1548        if (len > TASK_SIZE)
1549                return -ENOMEM;
1550
1551        get_area = current->mm->get_unmapped_area;
1552        if (file && file->f_op && file->f_op->get_unmapped_area)
1553                get_area = file->f_op->get_unmapped_area;
1554        addr = get_area(file, addr, len, pgoff, flags);
1555        if (IS_ERR_VALUE(addr))
1556                return addr;
1557
1558        if (addr > TASK_SIZE - len)
1559                return -ENOMEM;
1560        if (addr & ~PAGE_MASK)
1561                return -EINVAL;
1562
1563        return arch_rebalance_pgtables(addr, len);
1564 }
```

该函数会调用 arch_get_unmapped_area 来完成实质性的工作。

```
1363 unsigned long
1364 arch_get_unmapped_area(struct file *filp, unsigned long addr,
1365                unsigned long len, unsigned long pgoff, unsigned long flags)
1366 {
1367        struct mm_struct *mm = current->mm;
1368        struct vm_area_struct *vma;
1369        unsigned long start_addr;
1370
1371        if (len > TASK_SIZE)
1372                return -ENOMEM;
1373
```

```
1374        if (flags & MAP_FIXED)
1375                return addr;
1376
1377        if (addr) {
1378                addr = PAGE_ALIGN(addr);
1379                vma = find_vma(mm，addr);
1380                if (TASK_SIZE - len >= addr &&
1381                        (!vma || addr + len <= vma->vm_start))
1382                        return addr;
1383        }
1384        if (len > mm->cached_hole_size) {
1385                        start_addr = addr = mm->free_area_cache;
1386        } else {
1387                        start_addr = addr = TASK_UNMAPPED_BASE;
1388                        mm->cached_hole_size = 0;
1389        }
1390
1391 full_search:
1392        for (vma = find_vma(mm, addr); ; vma = vma->vm_next) {
1393                /* At this point:    (!vma || addr < vma->vm_end). */
1394                if (TASK_SIZE - len < addr) {
1395                        /*
1396                         * Start a new search - just in case we missed
1397                         * some holes.
1398                         */
1399                        if (start_addr != TASK_UNMAPPED_BASE) {
1400                                addr = TASK_UNMAPPED_BASE;
1401                                        start_addr = addr;
1402                                mm->cached_hole_size = 0;
1403                                goto full_search;
1404                        }
```

```
1405                    return -ENOMEM;
1406            }
1407            if (!vma || addr + len <= vma->vm_start) {
1408                    /*
1409                     * Remember the place where we stopped the search:
1410                     */
1411                    mm->free_area_cache = addr + len;
1412                    return addr;
1413            }
```

如果没找到，或者找到了但是该区域之前的空闲地址足够本次分配，则返回 addr。

```
1414            if (addr + mm->cached_hole_size < vma->vm_start)
1415                    mm->cached_hole_size = vma->vm_start - addr;
1416            addr = vma->vm_end;
1417    }
1418 }
```

3.2.3 vma_merge区域合并

```
746 struct vm_area_struct *vma_merge(struct mm_struct *mm,
747                    struct vm_area_struct *prev, unsigned long addr,
748                    unsigned long end, unsigned long vm_flags,
749                    struct anon_vma *anon_vma, struct file *file,
750                    pgoff_t pgoff, struct mempolicy *policy)
751 {
752     pgoff_t pglen = (end - addr) >> PAGE_SHIFT;
753     struct vm_area_struct *area, *next;
754     int err;
755
756     /*
757      * We later require that vma->vm_flags == vm_flags,
758      * so this tests vma->vm_flags & VM_SPECIAL, too.
```

```
759          */
760         if (vm_flags & VM_SPECIAL)
761                 return NULL;
762
763         if (prev)
764                 next = prev->vm_next;
765         else
766                 next = mm->mmap;
767         area = next;
768         if (next && next->vm_end == end)                /* cases 6, 7, 8 */
769                 next = next->vm_next;
770
771         /*
772          * Can it merge with the predecessor?
773          */
774         if (prev && prev->vm_end == addr &&
775                         mpol_equal(vma_policy(prev), policy) &&
776                         can_vma_merge_after(prev, vm_flags,
777                                         anon_vma, file, pgoff)) {
```

前一个和新的区域可以合并

```
778                 /*
779                  * OK, it can.  Can we now merge in the successor as well?
780                  */
781                 if (next && end == next->vm_start &&
782                                 mpol_equal(policy, vma_policy(next)) &&
783                                 can_vma_merge_before(next, vm_flags,
784                                         anon_vma, file, pgoff+pglen) &&
785                                 is_mergeable_anon_vma(prev->anon_vma,
786                                                 next->anon_vma)) {
```

后一个和新的区域可以合并

```
787                                         /* cases 1, 6 */
```

```
788                     err = vma_adjust(prev, prev->vm_start,
789                         next->vm_end, prev->vm_pgoff, NULL);
790             } else                          /* cases 2, 5, 7 */
791                     err = vma_adjust(prev, prev->vm_start,
792                         end, prev->vm_pgoff, NULL);
793             if (err)
794                     return NULL;
795             return prev;
796     }
797
798     /*
799      * Can this new request be merged in front of next?
800      */
801     if (next && end == next->vm_start &&
802                     mpol_equal(policy, vma_policy(next)) &&
803                     can_vma_merge_before(next, vm_flags,
804                             anon_vma, file, pgoff+pglen)) {
805             if (prev && addr < prev->vm_end)       /* case 4 */
806                 err = vma_adjust(prev, prev->vm_start,
807                         addr, prev->vm_pgoff, NULL);
808             else                            /* cases 3, 8 */
809                     err = vma_adjust(area, addr, next->vm_end,
810                         next->vm_pgoff - pglen, NULL);
811             if (err)
812                     return NULL;
813             return area;
814     }
815
816     return NULL;
817 }
```

3.2.4 find_vm_prepare

```
350 static struct vm_area_struct *
351 find_vma_prepare(struct mm_struct *mm, unsigned long addr,
352             struct vm_area_struct **pprev, struct rb_node ***rb_link,
353             struct rb_node ** rb_parent)
354 {
355         struct vm_area_struct * vma;
356         struct rb_node ** __rb_link, * __rb_parent, * rb_prev;
357
358         __rb_link = &mm->mm_rb.rb_node;
359         rb_prev = __rb_parent = NULL;
360         vma = NULL;
361
362         while (*__rb_link) {
363             struct vm_area_struct *vma_tmp;
364
365             __rb_parent = *__rb_link;
366             vma_tmp = rb_entry(__rb_parent, struct vm_area_struct,
vm_rb);
367
368             if (vma_tmp->vm_end > addr) {
369                 vma = vma_tmp;
370                 if (vma_tmp->vm_start <= addr)
371                     break;
372                 __rb_link = &__rb_parent->rb_left;
373             } else {
374                 rb_prev = __rb_parent;
375                 __rb_link = &__rb_parent->rb_right;
376             }
377         }
378
```

```
    *pprev = NULL;
    380         if (rb_prev)
     381                 *pprev = rb_entry(rb_prev, struct vm_area_struct, vm_rb);
     382         *rb_link = __rb_link;
     383         *rb_parent = __rb_parent;
     384         return vma;
     385 }
```

该函数获取下列信息：

前一个区域的 vm_area_struct 区域。返回 vma 的情况有两种：

- addr 在[vma->start,vma->end]之间
- addr 小于 vma->end，addr 也小于 vma->start。

保存新区域节点中的父节点

包含该区域自身的叶节点。

3.2.5 Insert_vm_struct

```
    473 static void __insert_vm_struct(struct mm_struct *mm, struct
vm_area_struct *vma)
     474 {
     475         struct vm_area_struct *__vma, *prev;
     476         struct rb_node **rb_link, *rb_parent;
     477
     478         __vma = find_vma_prepare(mm, vma->vm_start,&prev, &rb_link,
&rb_parent);
     479         BUG_ON(__vma && __vma->vm_start < vma->vm_end);
     480         __vma_link(mm, vma, prev, rb_link, rb_parent);
     481         mm->map_count++;
     482 }
```

将新节点插入到链表和红黑树中。

3.3 内存映射mmap

一个进程可以通过系统调用 mmap，将一个已打开文件的内容映射到它的用户空间。

在内核中，提供了两个系统调用 mmap，mmap2。有些体系实现了两个版本，有些则只实现了一个。

函数的界面如下：

```
64 asmlinkage long sys_mmap(unsigned long addr, unsigned long len,
unsigned long port, unsigned long flags , unsigned long fd, unsigned long off);
```

addr：期望的虚拟地址空间开始地址

len：映射区间的长度

port：访问权限

flags：标志集

fd：打开文件的句柄

off：映射在文件中开始的位置。

较重要的可设置的标志集：

MAP_FIXED：指定除了给定地址之外，不能将其他地址用于映射。

MAP_SHARED：如果一个对象（通常是文件）在几个进程间共享时，必须使用该标志。

MAP_PRIVATE：创建一个与数据源分离的私有映射，对映射区域的写入操作不影响文件中的数据。

MAP_ANONYMOUS：创建于任何数据都不相关的匿名映射，fd 和 off 可以忽略。

该函数最终会调用 do_mmap_off 完成实际的操作。

```
943 unsigned long do_mmap_pgoff(struct file *file, unsigned long addr,
944                 unsigned long len, unsigned long prot,
945                 unsigned long flags, unsigned long pgoff)
946 {
947     struct mm_struct * mm = current->mm;
948     struct inode *inode;
949     unsigned int vm_flags;
```

```
950         int error;
951         unsigned long reqprot = prot;
952
953         /*
954          * Does the application expect PROT_READ to imply
PROT_EXEC?
955          *
956          * (the exception is when the underlying filesystem is noexec
957          *    mounted, in which case we dont add PROT_EXEC.)
958          */
959         if ((prot & PROT_READ) && (current->personality &
READ_IMPLIES_EXEC))
960                 if (!(file && (file->f_path.mnt->mnt_flags & MNT_NOEXEC)))
961                         prot |= PROT_EXEC;
962
963         if (!len)
964                 return -EINVAL;
965
966         if (!(flags & MAP_FIXED))
967                 addr = round_hint_to_min(addr);
968
      /* Careful about overflows.. */
970         len = PAGE_ALIGN(len);
971         if (!len)
972                 return -ENOMEM;
973
974         /* offset overflow? */
975         if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
976                         return -EOVERFLOW;
977
978         /* Too many mappings? */
979         if (mm->map_count > sysctl_max_map_count)
```

```
980             return -ENOMEM;
981
982     /* Obtain the address to map to. we verify (or select) it and ensure
983      * that it represents a valid section of the address space.
984      */
985     addr = get_unmapped_area(file, addr, len, pgoff, flags);
```

获得虚拟内存区的开始地址。

```
986     if (addr & ~PAGE_MASK)
987             return addr;
988
989     /* Do simple checking here so the lower-level routines won't have
990      * to. we assume access permissions have been handled by the
open
991      * of the memory object, so we don't do any here.
992      */
993     vm_flags = calc_vm_prot_bits(prot) | calc_vm_flag_bits(flags) |
994             mm->def_flags | VM_MAYREAD | VM_MAYWRITE |
VM_MAYEXEC;
```

标志集设置

```
995
996     if (flags & MAP_LOCKED)
997             if (!can_do_mlock())
998                     return -EPERM;
999
    /* mlock MCL_FUTURE? */
1001    if (vm_flags & VM_LOCKED) {
1002            unsigned long locked, lock_limit;
1003            locked = len >> PAGE_SHIFT;
1004            locked += mm->locked_vm;
1005            lock_limit = rlimit(RLIMIT_MEMLOCK);
1006            lock_limit >>= PAGE_SHIFT;
```

```
1007            if (locked > lock_limit && !capable(CAP_IPC_LOCK))
1008                    return -EAGAIN;
1009    }
1010
1011    inode = file ? file->f_path.dentry->d_inode : NULL;
1012
1013    if (file) {
1014            switch (flags & MAP_TYPE) {
1015            case MAP_SHARED:
1016                    if ((prot&PROT_WRITE)
&& !(file->f_mode&FMODE_WRITE))
1017                            return -EACCES;
1018
1019                    /*
1020                     * Make sure we don't allow writing to an append-only
1021                     * file..
1022                     */
1023                    if (IS_APPEND(inode) && (file->f_mode &
FMODE_WRITE))
1024                            return -EACCES;
1025
1026                    /*
1027                     * Make sure there are no mandatory locks on the file.
1028                     */
1029                    if (locks_verify_locked(inode))
1030                            return -EAGAIN;
1031
1032                    vm_flags |= VM_SHARED | VM_MAYSHARE;
1033                    if (!(file->f_mode & FMODE_WRITE))
1034                            vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
1035
1036                    /* fall through */
```

```
1037            case MAP_PRIVATE:
1038                    if (!(file->f_mode & FMODE_READ))
1039                            return -EACCES;
1040                    if (file->f_path.mnt->mnt_flags & MNT_NOEXEC) {
1041                            if (vm_flags & VM_EXEC)
1042                                    return -EPERM;
1043                            vm_flags &= ~VM_MAYEXEC;
1044                    }
1045
1046                    if (!file->f_op || !file->f_op->mmap)
1047                            return -ENODEV;
1048                    break;
1049
1050            default:
1051                    return -EINVAL;
1052            }
1053    } else {
1054            switch (flags & MAP_TYPE) {
1055            case MAP_SHARED:
1056                    /*
1057                     * Ignore pgoff.
1058                     */
1059                    pgoff = 0;
1060                    vm_flags |= VM_SHARED | VM_MAYSHARE;
1061                    break;
1062            case MAP_PRIVATE:
1063                    /*
1064                     * Set pgoff according to addr for anon_vma.
1065                     */
1066                    pgoff = addr >> PAGE_SHIFT;
1067                    break;
```

```
1068            default:
1069                    return -EINVAL;
1070            }
1071    }
1072
1073    error = security_file_mmap(file, reqprot, prot, flags, addr, 0);
```

安全检查

```
1074    if (error)
1075            return error;
1076
1077    return mmap_region(file, addr, len, flags, vm_flags, pgoff);
```

调用 mmap_region 完成实质工作。

```
1078 }
1079 EXPORT_SYMBOL(do_mmap_pgoff);
```

```
1193 unsigned long mmap_region(struct file *file, unsigned long addr,
1194                         unsigned long len, unsigned long flags,
1195                         unsigned int vm_flags, unsigned long pgoff)
1196 {
1197     struct mm_struct *mm = current->mm;
1198     struct vm_area_struct *vma, *prev;
1199     int correct_wcount = 0;
1200     int error;
1201     struct rb_node **rb_link, *rb_parent;
1202     unsigned long charged = 0;
1203     struct inode *inode =   file ? file->f_path.dentry->d_inode : NULL;
1204
1205     /* Clear old maps */
1206     error = -ENOMEM;
1207 munmap_back:
1208     vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
```

```
1209        if (vma && vma->vm_start < addr + len) {
1210                if (do_munmap(mm, addr, len))
1211                        return -ENOMEM;
1212                goto munmap_back;
1213        }
```

在之前已经检测 addr 符合各项标准了，这里为何还要调用 find_vma_prepare 进行检查。如果 find_vma_prepare 返回 vma 非空，且 1209 行条件成立，则

[vma->start,vma->end]与[addr,addr+len]有交集，所以需要调用 do_munmap 函数。该函数很少被调用，但是在某些情况也也会被调用：

在内核中分配虚拟空间时执行流程可能被暂停，而该进程 clone 出来的现场可能被调用，同时可能也映射了同一区间的虚拟内存，这将导致上述情况的发生。

```
1214
1215        /* Check against address space limit. */
1216        if (!may_expand_vm(mm, len >> PAGE_SHIFT))
1217                return -ENOMEM;
```

检查是否操作了内存空间限制

```
1218
1219        /*
1220         * Set 'VM_NORESERVE' if we should not account for the
1221         * memory use of this mapping.
1222         */
1223        if ((flags & MAP_NORESERVE)) {
1224                /* We honor MAP_NORESERVE if allowed to overcommit */
1225                if (sysctl_overcommit_memory != OVERCOMMIT_NEVER)
1226                        vm_flags |= VM_NORESERVE;
1227
1228                /* hugetlb applies strict overcommit unless MAP_NORESERVE
*/
1229                if (file && is_file_hugepages(file))
1230                        vm_flags |= VM_NORESERVE;
1231        }
```

```
1232
1233        /*
1234         * Private writable mapping: check memory availability
1235         */
1236        if (accountable_mapping(file, vm_flags)) {
1237                charged = len >> PAGE_SHIFT;
1238                if (security_vm_enough_memory(charged))
1239                        return -ENOMEM;
1240                vm_flags |= VM_ACCOUNT;
1241        }
1242
1243        /*
1244         * Can we just expand an old mapping?
1245         */
1246        vma = vma_merge(mm, prev, addr, addr + len, vm_flags, NULL, file,
pgoff, NULL);
```

将新分配的 vma 和它前面的 vma、后面的 vma 合并（有可能时）。

```
1247        if (vma)
1248                goto out;
```

合并成功，调整到 out。

```
1249
1250        /*
1251         * Determine the object being mapped and call the appropriate
1252         * specific mapper. the address has already been validated, but
1253         * not unmapped, but the maps are removed from the list.
1254         */
1255        vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
```

分配 vma 实例。

```
1256        if (!vma) {
1257                error = -ENOMEM;
1258                goto unacct_error;
```

```
1259        }
1260
1261        vma->vm_mm = mm;
1262        vma->vm_start = addr;
1263        vma->vm_end = addr + len;
1264        vma->vm_flags = vm_flags;
1265        vma->vm_page_prot = vm_get_page_prot(vm_flags);
1266        vma->vm_pgoff = pgoff;
1267        INIT_LIST_HEAD(&vma->anon_vma_chain);
```

设置 vma 的相关变量。

```
1268
1269        if (file) {
1270                error = -EINVAL;
1271                if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
1272                        goto free_vma;
1273                if (vm_flags & VM_DENYWRITE) {
1274                        error = deny_write_access(file);
1275                        if (error)
1276                                goto free_vma;
1277                        correct_wcount = 1;
1278                }
1279                vma->vm_file = file;
1280                get_file(file);
1281                error = file->f_op->mmap(file, vma);
```

设定 vma 和文件有关的变量。

```
1282                if (error)
1283                        goto unmap_and_free_vma;
1284                if (vm_flags & VM_EXECUTABLE)
1285                        added_exe_file_vma(mm);
1286
1287                /* Can addr have changed??
```

```
1288                 *
1289                 * Answer: Yes, several device drivers can do it in their
1290                 *         f_op->mmap method. -DaveM
1291                 */
1292                addr = vma->vm_start;
1293                pgoff = vma->vm_pgoff;
1294                vm_flags = vma->vm_flags;
1295        } else if (vm_flags & VM_SHARED) {
1296                error = shmem_zero_setup(vma);
1297                if (error)
1298                        goto free_vma;
1299        }
1300
1301        if (vma_wants_writenotify(vma)) {
1302                pgprot_t pprot = vma->vm_page_prot;
1303
1304                /* Can vma->vm_page_prot have changed??
1305                 *
1306                 * Answer: Yes, drivers may have changed it in their
1307                 *         f_op->mmap method.
1308                 *
1309                 * Ensures that vmas marked as uncached stay that way.
1310                 */
1311                vma->vm_page_prot = vm_get_page_prot(vm_flags &
~VM_SHARED);
1312                if (pgprot_val(pprot) == pgprot_val(pgprot_noncached(pprot)))
1313                        vma->vm_page_prot =
pgprot_noncached(vma->vm_page_prot);
1314        }
1315
1316        vma_link(mm, vma, prev, rb_link, rb_parent);
1317        file = vma->vm_file;
```

```
1318
1319        /* Once vma denies write, undo our temporary denial count */
1320        if (correct_wcount)
1321                atomic_inc(&inode->i_writecount);
1322 out:
1323        perf_event_mmap(vma);
1324
1325        mm->total_vm += len >> PAGE_SHIFT;
1326        vm_stat_account(mm, vm_flags, file, len >> PAGE_SHIFT);
1327        if (vm_flags & VM_LOCKED) {
1328                if (!mlock_vma_pages_range(vma, addr, addr + len))
1329                        mm->locked_vm += (len >> PAGE_SHIFT);
1330        } else if ((flags & MAP_POPULATE) && !(flags &
MAP_NONBLOCK))
1331                make_pages_present(addr, addr + len);
1332        return addr;
```

检查成功返回 addr。

```
1333
1334 unmap_and_free_vma:
1335        if (correct_wcount)
1336                atomic_inc(&inode->i_writecount);
1337        vma->vm_file = NULL;
1338        fput(file);
1339
1340        /* Undo any partial mapping done by a device driver. */
1341        unmap_region(mm, vma, prev, vma->vm_start, vma->vm_end);
1342        charged = 0;
1343 free_vma:
1344        kmem_cache_free(vm_area_cachep, vma);
1345 unacct_error:
1346        if (charged)
```

```
1347                vm_unacct_memory(charged);
1348        return error;
1349 }
```

3.4 堆的管理brk

堆是进程中用于动态分配变量和数据的内存区域。堆的管理对应用程序员不直接可见的。因为它依赖于标准库提供的各个辅助函数（其中最重要的是 malloc）来分配任意长度的内存区。malloc 和内核之间的经典接口是 brk 系统调用，负责扩展/收缩堆。新近的 malloc 实现使用了一种组合方法，使用 brk 和匿名映射。该方法提供了更好的性能，而且在分配较大的内存区时具有某些优点。

堆是一个连续的内存区域，在扩展时从底地址向高地址扩充。前文提到的 mm_struct 结构，包含了堆在虚拟地址空间中的开始和当前结束地址（start_brk 和 brk）。

该函数最终会调用 do_brk 来实现实质性的工作。

该函数和上节讨论的 mmap 函数基本一致，就不具体查看了。

```
2119 /*
2120  *  this is really a simplified "do_mmap".  it only handles
2121  *  anonymous maps.  eventually we may be able to do some
2122  *  brk-specific accounting here.
2123  */
2124 unsigned long do_brk(unsigned long addr, unsigned long len)
2125 {
2126         struct mm_struct * mm = current->mm;
2127         struct vm_area_struct * vma, * prev;
2128         unsigned long flags;
2129         struct rb_node ** rb_link, * rb_parent;
2130         pgoff_t pgoff = addr >> PAGE_SHIFT;
2131         int error;
2132
2133         len = PAGE_ALIGN(len);
2134         if (!len)
2135                 return addr;
2136
```

```
2137        error = security_file_mmap(NULL, 0, 0, 0, addr, 1);
2138        if (error)
2139                return error;
2140
2141        flags = VM_DATA_DEFAULT_FLAGS | VM_ACCOUNT |
mm->def_flags;
2142
2143        error = get_unmapped_area(NULL, addr, len, 0, MAP_FIXED);
2144        if (error & ~PAGE_MASK)
2145                return error;
2146
2147        /*
2148         * mlock MCL_FUTURE?
2149         */
2150        if (mm->def_flags & VM_LOCKED) {
2151                unsigned long locked, lock_limit;
2152                locked = len >> PAGE_SHIFT;
2153                locked += mm->locked_vm;
2154                lock_limit = rlimit(RLIMIT_MEMLOCK);
2155                lock_limit >>= PAGE_SHIFT;
2156                if (locked > lock_limit && !capable(CAP_IPC_LOCK))
2157                        return -EAGAIN;
2158        }
2159
2160        /*
2161         * mm->mmap_sem is required to protect against another thread
2162         * changing the mappings in case we sleep.
2163         */
2164        verify_mm_writelocked(mm);
2165
2166        /*
2167         * Clear old maps.  this also does some error checking for us
```

```
2168         */
2169    munmap_back:
2170         vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
2171         if (vma && vma->vm_start < addr + len) {
2172              if (do_munmap(mm, addr, len))
2173                   return -ENOMEM;
2174              goto munmap_back;
2175         }
2176
2177         /* Check against address space limits *after* clearing old maps... */
2178         if (!may_expand_vm(mm, len >> PAGE_SHIFT))
2179              return -ENOMEM;
2180
2181         if (mm->map_count > sysctl_max_map_count)
2182              return -ENOMEM;
2183
2184         if (security_vm_enough_memory(len >> PAGE_SHIFT))
2185              return -ENOMEM;
2186
2187         /* Can we just expand an old private anonymous mapping? */
2188         vma = vma_merge(mm, prev, addr, addr + len, flags,
2189                                   NULL, NULL, pgoff, NULL);
2190         if (vma)
2191              goto out;
2192
2193         /*
2194          * create a vma struct for an anonymous mapping
2195          */
2196         vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
2197         if (!vma) {
2198              vm_unacct_memory(len >> PAGE_SHIFT);
```

```
2199                return -ENOMEM;
2200        }
2201
2202        INIT_LIST_HEAD(&vma->anon_vma_chain);
2203        vma->vm_mm = mm;
2204        vma->vm_start = addr;
2205        vma->vm_end = addr + len;
2206        vma->vm_pgoff = pgoff;
2207        vma->vm_flags = flags;
2208        vma->vm_page_prot = vm_get_page_prot(flags);
2209        vma_link(mm, vma, prev, rb_link, rb_parent);
2210 out:
2211        mm->total_vm += len >> PAGE_SHIFT;
2212        if (flags & VM_LOCKED) {
2213                if (!mlock_vma_pages_range(vma, addr, addr + len))
2214                        mm->locked_vm += (len >> PAGE_SHIFT);
2215        }
2216        return addr;
2217 }
2218
2219 EXPORT_SYMBOL(do_brk);
```

3.5 删除映射munmap

```
2008 /* Munmap is split into 2 main parts -- this part which finds
2009    * what needs doing, and the areas themselves, which do the
2010    * work.    This now handles partial unmappings.
2011    * Jeremy Fitzhardinge <jeremy@goop.org>
2012    */
2013 int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
2014 {
2015        unsigned long end;
```

```
2016        struct vm_area_struct *vma, *prev, *last;
2017
2018        if ((start & ~PAGE_MASK) || start > TASK_SIZE || len >
TASK_SIZE-start)
2019                return -EINVAL;
2020
2021        if ((len = PAGE_ALIGN(len)) == 0)
2022                return -EINVAL;
2023
2024        /* Find the first overlapping VMA */
2025        vma = find_vma_prev(mm, start, &prev);
2026        if (!vma)
2027                return 0;
2028        /* we have    start < vma->vm_end    */
2029
2030        /* if it doesn't overlap, we have nothing.. */
2031        end = start + len;
2032        if (vma->vm_start >= end)
2033                return 0;
2034
2035        /*
2036         * If we need to split any vma, do it now to save pain later.
2037         *
2038         * Note: mremap's move_vma VM_ACCOUNT handling assumes a
partially
2039         * unmapped vm_area_struct will remain in use: so lower split_vma
2040         * places tmp vma above, and higher split_vma places tmp vma
below.
2041         */
2042        if (start > vma->vm_start) {
```

start 在[vma->vm_start,vma->vm_end]之间。使用 start 将 vma 划分为[vma->vm_start,start],[start,vma->vm_end]两部分。

```
2043            int error;
2044
2045            /*
2046             * Make sure that map_count on return from munmap() will
2047             * not exceed its limit; but let map_count go just above
2048             * its limit temporarily, to help free resources as expected.
2049             */
2050            if (end < vma->vm_end && mm->map_count >=
sysctl_max_map_count)
2051                    return -ENOMEM;
2052
2053            error = __split_vma(mm, vma, start, 0);
2054            if (error)
2055                    return error;
2056            prev = vma;
2057    }
2058
2059    /* Does it split the last one? */
2060    last = find_vma(mm, end);
2061    if (last && end > last->vm_start) {
2062            int error = __split_vma(mm, last, end, 1);
```

如果 end 在[last->vm_start,last->vm_end]之间，将 last 分为[last->vm_start,end]与[end,last->vm_end]两部分。

```
2063            if (error)
2064                    return error;
2065    }
2066    vma = prev? prev->vm_next: mm->mmap;
2067
2068    /*
2069     * unlock any mlock()ed ranges before detaching vmas
2070     */
2071    if (mm->locked_vm) {
```

```
2072            struct vm_area_struct *tmp = vma;
2073            while (tmp && tmp->vm_start < end) {
2074                    if (tmp->vm_flags & VM_LOCKED) {
2075                            mm->locked_vm -= vma_pages(tmp);
2076                            munlock_vma_pages_all(tmp);
2077                    }
2078                    tmp = tmp->vm_next;
2079            }
2080    }
2081
2082    /*
2083     * Remove the vma's, and unmap the actual pages
2084     */
```

上述代码将包含在进程地址空间的线性地址区间中的所有线性区从链表中解除链接。该过程就是链表和红黑树的操作，具体过程就不详细查看了。

```
2085    detach_vmas_to_be_unmapped(mm, vma, prev, end);
```

将要删除的线性区的描述符放在一个排好序的链表中。

```
2086    unmap_region(mm, vma, prev, start, end);
2087
2088    /* Fix up all other VM information */
2089    remove_vma_list(mm, vma);
2090
2091    return 0;
2092 }
2093
2094 EXPORT_SYMBOL(do_munmap);
```

```
1867 /*
1868  * Get rid of page table information in the indicated region.
1869  *
```

```
1870   * Called with the mm semaphore held.
1871   */
1872 static void unmap_region(struct mm_struct *mm,
1873         struct vm_area_struct *vma, struct vm_area_struct *prev,
1874         unsigned long start, unsigned long end)
1875 {
1876     struct vm_area_struct *next = prev? prev->vm_next: mm->mmap;
1877     struct mmu_gather *tlb;
1878     unsigned long nr_accounted = 0;
1879
1880     lru_add_drain();
1881     tlb = tlb_gather_mmu(mm, 0);
1882     update_hiwater_rss(mm);
1883     unmap_vmas(&tlb, vma, start, end, &nr_accounted, NULL);
```

unmap_vmas 扫描线性地址空间中的所有页表，调用相关函数反复释放相应的页。

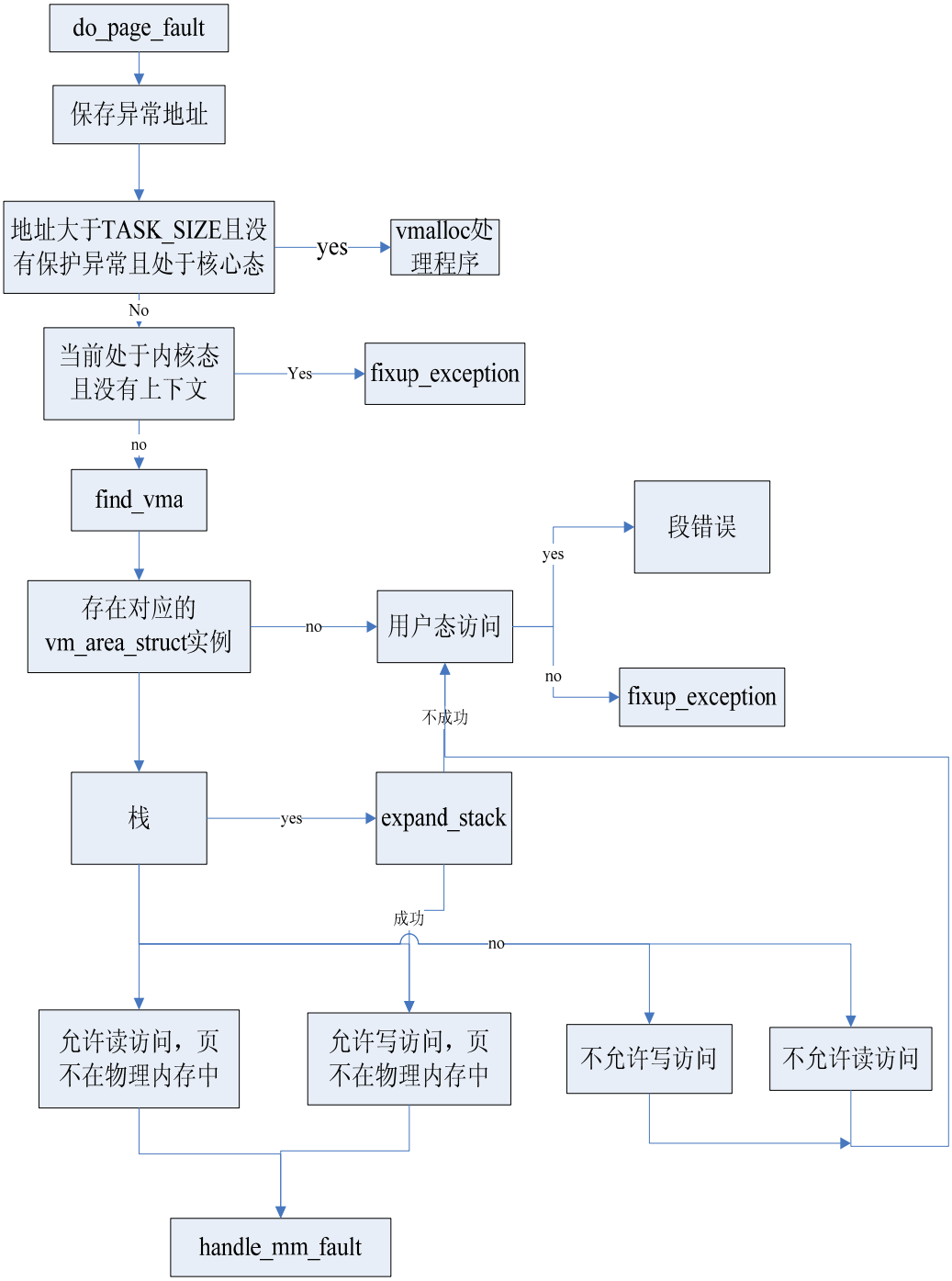
```
1884     vm_unacct_memory(nr_accounted);
1885     free_pgtables(tlb, vma, prev? prev->vm_end:
FIRST_USER_ADDRESS,
1886                   next? next->vm_start: 0);
```

回收上一步已经清空的进程页表。

```
1887     tlb_finish_mmu(tlb, start, end);
1888 }
```

3.6 缺页异常处理

在实际需要某个虚拟内存区域的数据之前，虚拟和物理内存之间的关系并不会建立。如果进程访问的虚拟地址部分尚未分配与之关联的页框，则处理器会引发一个缺页异常。




```
941 /*
942  * This routine handles page faults.  It determines the address,
943  * and the problem, and then passes it off to one of the appropriate
944  * routines.
945  */
946 dotraplinkage void __kprobes
```

```
947 do_page_fault(struct pt_regs *regs, unsigned long error_code)
948 {
949         struct vm_area_struct *vma;
950         struct task_struct *tsk;
951         unsigned long address;
952         struct mm_struct *mm;
953         int write;
954         int fault;
955
956         tsk = current;
957         mm = tsk->mm;
958
959         /* Get the faulting address: */
960         address = read_cr2();
```

当异常发生时，CPU 控制单元将引起缺页的地址保存到 CR2 寄存器中。read_cr2 将读取该地址，并保存引起缺页异常的地址到 address 中。

```
961
962         /*
963          * Detect and handle instructions that would cause a page fault for
964          * both a tracked kernel page and a userspace page.
965          */
966         if (kmemcheck_active(regs))
967                 kmemcheck_hide(regs);
968         prefetchw(&mm->mmap_sem);
969
970         if (unlikely(kmmio_fault(regs, address)))
971                 return;
972
973         /*
974          * We fault-in kernel-space virtual memory on-demand. The
975          * 'reference' page table is init_mm.pgd.
```

```
976         *
977         * NOTE! We MUST NOT take any locks for this case. We may
978         * be in an interrupt or a critical region, and should
979         * only copy the information from the master page table,
980         * nothing more.
981         *
982         * This verifies that the fault happens in kernel space
983         * (error_code & 4) == 0, and that the fault was not a
984         * protection error (error_code & 9) == 0.
985         */
986        if (unlikely(fault_in_kernel_space(address))) {
```

引起异常的地址在内核空间

```
987             if (!(error_code & (PF_RSVD | PF_USER | PF_PROT))) {
988                     if (vmalloc_fault(address) >= 0)
```

处理由 vmalloc 区域引起的异常。

```
989                             return;
990
991                     if (kmemcheck_fault(regs, address, error_code))
992                             return;
993             }
994
995             /* Can handle a stale RO->RW TLB: */
996             if (spurious_fault(error_code, address))
997                     return;
998
/* kprobes don't want to hook the spurious faults: */
1000            if (notify_page_fault(regs))
1001                    return;
1002            /*
1003             * Don't take the mm semaphore here. If we fixup a prefetch
1004             * fault we could otherwise deadlock:
```

```
1005                 */
1006                bad_area_nosemaphore(regs, error_code, address);
1007
1008                return;
1009        }
1010
1011        /* kprobes don't want to hook the spurious faults: */
1012        if (unlikely(notify_page_fault(regs)))
1013                return;
1014        /*
1015         * It's safe to allow irq's after cr2 has been saved and the
1016         * vmalloc fault has been handled.
1017         *
1018         * User-mode registers count as a user access even for any
1019         * potential system fault or CPU buglet:
1020         */
1021        if (user_mode_vm(regs)) {
1022                local_irq_enable();
1023                error_code |= PF_USER;
1024        } else {
1025                if (regs->flags & X86_EFLAGS_IF)
1026                        local_irq_enable();
1027        }
1028
1029        if (unlikely(error_code & PF_RSVD))
1030                pgtable_bad(regs, error_code, address);
1031
1032        perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS, 1, 0, regs,
address);
1033
1034        /*
1035         * If we're in an interrupt, have no user context or are running
```

```
1036         * in an atomic region then we must not take the fault:
1037         */
1038        if (unlikely(in_atomic() || !mm)) {
1039                bad_area_nosemaphore(regs, error_code, address);
1040                return;
1041        }
```

如果是在中断期间，没有用户上下文或者处于原子码操作范围内，则无法处理该异常。

```
1042
1043        /*
1044         * When running in the kernel we expect faults to occur only to
1045         * addresses in user space.    All other faults represent errors in
1046         * the kernel and should generate an OOPS.    Unfortunately, in the
1047         * case of an erroneous fault occurring in a code path which already
1048         * holds mmap_sem we will deadlock attempting to validate the fault
1049         * against the address space.    Luckily the kernel only validly
1050         * references user space from well defined areas of code, which are
1051         * listed in the exceptions table.
1052         *
1053         * As the vast majority of faults will be valid we will only perform
1054         * the source reference check when there is a possibility of a
1055         * deadlock. Attempt to lock the address space, if we cannot we
then
1056         * validate the source. If this is invalid we can skip the address
1057         * space check, thus avoiding the deadlock:
1058         */
1059        if (unlikely(!down_read_trylock(&mm->mmap_sem))) {
1060                if ((error_code & PF_USER) == 0 &&
1061                    !search_exception_tables(regs->ip)) {
1062                        bad_area_nosemaphore(regs, error_code, address);
1063                        return;
```

```
1064            }
```

处理由内核态访问用户空间引起的异常。

```
1065            down_read(&mm->mmap_sem);
1066        } else {
1067            /*
1068             * The above down_read_trylock() might have succeeded in
1069             * which case we'll have missed the might_sleep() from
1070             * down_read():
1071             */
1072            might_sleep();
1073        }
1074
1075        vma = find_vma(mm, address);
```

查找引起异常的 vma 区域。

```
1076        if (unlikely(!vma)) {
1077            bad_area(regs, error_code, address);
1078            return;
1079        }
```

如果 vma 不存在，则说明访问越界，调整到 bad_area 处。

```
1080        if (likely(vma->vm_start <= address))
1081            goto good_area;
```

如果 address 在 vma 中间，则跳转到 good_area。

```
1082        if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {
1083            bad_area(regs, error_code, address);
1084            return;
1085        }
```

如果 vma 不空，且 address 不在 vma 中间，并且 vma 不是堆栈去，则跳转到 bad_area。

```
1086        if (error_code & PF_USER) {
1087            /*
1088             * Accessing the stack below %sp is always a bug.
```

```
1089             * The large cushion allows instructions like enter
1090             * and pusha to work. ("enter $65535, $31" pushes
1091             * 32 pointers and then decrements %sp by 65535.)
1092             */
1093            if (unlikely(address + 65536 + 32 * sizeof(unsigned long) <
regs->sp)) {
1094                bad_area(regs, error_code, address);
1095                return;
1096            }
```

如果 vma 是堆栈区，但是 address+ 65536 + 32 * sizeof(unsigned long)<sp, 则跳转到 bad_area。

```
1097        }
1098        if (unlikely(expand_stack(vma, address))) {
1099            bad_area(regs, error_code, address);
1100            return;
1101        }
```

如果上面的检查通过，说明是由于栈扩展引起的异常，调用 expand_stack 来处理。

```
1102
1103        /*
1104         * Ok, we have a good vm_area for this memory access, so
1105         * we can handle it..
1106         */
1107 good_area:
1108        write = error_code & PF_WRITE;
1109
1110        if (unlikely(access_error(error_code, write, vma))) {
1111            bad_area_access_error(regs, error_code, address);
1112            return;
1113        }
```

如果

```
1114
1115        /*
1116         * If for any reason at all we couldn't handle the fault,
1117         * make sure we exit gracefully rather than endlessly redo
1118         * the fault:
1119         */
1120        fault = handle_mm_fault(mm, vma, address, write ?
FAULT_FLAG_WRITE : 0);
```

调用 handle_mm_fault 处理正常的异常。

```
1121
1122        if (unlikely(fault & VM_FAULT_ERROR)) {
1123                mm_fault_error(regs, error_code, address, fault);
1124                return;
1125        }
1126
1127        if (fault & VM_FAULT_MAJOR) {
1128                tsk->maj_flt++;
1129                perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS_MAJ, 1,
0,
1130                              regs, address);
1131        } else {
1132                tsk->min_flt++;
1133                perf_sw_event(PERF_COUNT_SW_PAGE_FAULTS_MIN, 1,
0,
1134                              regs, address);
1135        }
1136
1137        check_v8086_mode(regs, address, tsk);
1138
1139        up_read(&mm->mmap_sem);
1140 }
```

3.6.1 内核态下缺页异常的处理

在访问内核地址空间时，缺页异常可能被各种条件触发：

- 内核中程序设计的错误导致访问地址不正确。这是真正的错误，在稳定版内核中不应该发生。
- 内核通过用户空间传递的参数，访问了无效的地址。
- 访问 vmalloc 分配的区域，触发缺页异常。

前两种情况是真正的错误，内核必须进行额外的检查。vmalloc 的情况是导致缺页异常的合理原因。直至对应的缺页异常发生，vmalloc 区域中的修改都不会传输到进程的页表。

在处理不是由于 vmalloc 引起的缺页异常时，异常修正(exception fixup)机制是一个最后的手段。内核编译了一个列表，列出了所有可能执行未授权内存访问访问操作的危险代码。这个“异常表”在链接内核映像时创建，在二进制文件中位于__start_exception_table 和__end_exception_table 之间。

3.6.1.1 vmalloc异常处理

```
239 /*
240  * 32-bit:
241  *
242  *     Handle a fault on the vmalloc or module mapping area
243  */
244 static noinline __kprobes int vmalloc_fault(unsigned long address)
245 {
246         unsigned long pgd_paddr;
247         pmd_t *pmd_k;
248         pte_t *pte_k;
249
250         /* Make sure we are in vmalloc area: */
251         if (!(address >= VMALLOC_START && address <
VMALLOC_END))
252                 return -1;
253
```

```
254         /*
255          * Synchronize this task's top level page-table
256          * with the 'reference' page table.
257          *
258          * Do _not_ use "current" here. We might be inside
259          * an interrupt in the middle of a task switch..
260          */
261         pgd_paddr = read_cr3();
262         pmd_k = vmalloc_sync_one(__va(pgd_paddr), address);
263         if (!pmd_k)
264                 return -1;
265
266         pte_k = pte_offset_kernel(pmd_k, address);
267         if (!pte_present(*pte_k))
268                 return -1;
269
270         return 0;
271 }
```

251：检查 address 地址是否在 vmalloc 范围内，不在返回-1。

261：读取当前进程页全局目录表指针。

262：调用 vmalloc_sync_one 同步主内核页表到当前进程页表。

266：同步后取 address 对应的页表项。

267：如果页表项对应的页框不在内存中，则表示访问出错，返回-1。vmalloc 异常处理失败。

```
180 static inline pmd_t *vmalloc_sync_one(pgd_t *pgd, unsigned long address)
181 {
182         unsigned index = pgd_index(address);
183         pgd_t *pgd_k;
```

```
184         pud_t *pud, *pud_k;
185         pmd_t *pmd, *pmd_k;
186
187         pgd += index;
188         pgd_k = init_mm.pgd + index;
189
190         if (!pgd_present(*pgd_k))
191                 return NULL;
```

主内核页目录该项不存在，则返回 NULL。

```
192
193         /*
194          * set_pgd(pgd, *pgd_k); here would be useless on PAE
195          * and redundant with the set_pmd() on non-PAE. As would
196          * set_pud.
197          */
198         pud = pud_offset(pgd, address);
199         pud_k = pud_offset(pgd_k, address);
200         if (!pud_present(*pud_k))
201                 return NULL;
```

主内核中页上级目录项不存在，返回 NULL

```
202
203         pmd = pmd_offset(pud, address);
204         pmd_k = pmd_offset(pud_k, address);
205         if (!pmd_present(*pmd_k))
206                 return NULL;
```

主内核中页中级目录项不存在，返回 NULL

```
207
208         if (!pmd_present(*pmd))
209                 set_pmd(pmd, *pmd_k);
```

当前进程的该项为空，则用主内核的填充。

```
210         else
```

```
211             BUG_ON(pmd_page(*pmd) != pmd_page(*pmd_k));
```

如果非空但两者不一致，说明内核出现了错误。

```
212
213         return pmd_k;
214 }
```

3.6.1.2 exception fixup

当在内核态访问用户空间的地址异常时，内核会通过 exception fixup 进行最后的努力。

内核编译了一个列表，列出了所有可能执行未授权内存访问访问操作的危险代码。这个“异常表”在链接内核映像时创建，在二进制文件中位于 __start_exception_table 和__end_exception_table 之间。每一项都是一个 exception_table_entry 结构体，定义如下：

```
struct exception_table_entry {
        unsigned long insn, fixup;
};
```

insn：内核认为可能发生异常的虚拟地址。

fixup：异常发生时执行恢复到那个代码的地址。

```
44 /* Given an address, look for it in the exception tables. */
45 const struct exception_table_entry *search_exception_tables(unsigned long addr)
46 {
47      const struct exception_table_entry *e;
48
49      e = search_extable(__start___ex_table, __stop___ex_table-1, addr);
50      if (!e)
51          e = search_module_extables(addr);
52      return e;
53 }
```

```
 6 int fixup_exception(struct pt_regs *regs)
 7 {
 8         const struct exception_table_entry *fixup;
 9
10 #ifdef CONFIG_PNPBIOS
11         if (unlikely(SEGMENT_IS_PNP_CODE(regs->cs))) {
12                 extern u32 pnp_bios_fault_eip, pnp_bios_fault_esp;
13                 extern u32 pnp_bios_is_utter_crap;
14                 pnp_bios_is_utter_crap = 1;
15                 printk(KERN_CRIT "PNPBIOS fault.. attempting recovery.\n");
16                 __asm__ volatile(
17                         "movl %0, %%esp\n\t"
18                         "jmp *%1\n\t"
19                         : : "g" (pnp_bios_fault_esp), "g" (pnp_bios_fault_eip));
20                 panic("do_trap: can't hit this");
21         }
22 #endif
23
24         fixup = search_exception_tables(regs->ip);
```

查找 fixup

```
25         if (fixup) {
26                 /* If fixup is less than 16, it means uaccess error */
27                 if (fixup->fixup < 16) {
28                         current_thread_info()->uaccess_err = -EFAULT;
29                         regs->ip += fixup->fixup;
30                         return 1;
31                 }
32                 regs->ip = fixup->fixup;
```

使用 fixup 地址替换 regs 中的 ip。

```
33                 return 1;
```

```
34      }
35
36      return 0;
37 }
```

3.6.2 用户态缺页异常

3.6.2.1 expand stack

在 do_page_fault 中，看到 find_vma 查找到 vma，但 address 不在 vma 范围内的唯一种合法的情况就是当

address+ 65536 + 32 * sizeof(unsigned long)>=sp 时，

异常的引起可能是由于 push 导致的堆栈扩充。

最终会调用 expand_downwards 函数来处理这种情况。

```
1748 static int expand_downwards(struct vm_area_struct *vma,
1749                               unsigned long address)
1750 {
1751     int error;
1752
1753     /*
1754      * We must make sure the anon_vma is allocated
1755      * so that the anon_vma locking is not a noop.
1756      */
1757     if (unlikely(anon_vma_prepare(vma)))
1758         return -ENOMEM;
1759
1760     address &= PAGE_MASK;
1761     error = security_file_mmap(NULL, 0, 0, 0, address, 1);
1762     if (error)
1763         return error;
1764
1765     anon_vma_lock(vma);
1766
```

```
1767        /*
1768         * vma->vm_start/vm_end cannot change under us because the
caller
1769         * is required to hold the mmap_sem in read mode.    We need the
1770         * anon_vma lock to serialize against concurrent expand_stacks.
1771         */
1772
1773        /* Somebody else might have raced and expanded it already */
1774        if (address < vma->vm_start) {
1775                unsigned long size, grow;
1776
size = vma->vm_end - address;
1778                grow = (vma->vm_start - address) >> PAGE_SHIFT;
1779
1780                error = acct_stack_growth(vma, size, grow);
```

扩充堆栈

```
1781                if (!error) {
1782                        vma->vm_start = address;
```

扩充堆栈

```
1783                        vma->vm_pgoff -= grow;
1784                }
1785        }
1786        anon_vma_unlock(vma);
1787        return error;
1788 }
```

3.6.2.2 其它合法访问的处理

在用户空间中，查找到了 vma，同时 address 也在 vma 中间，这所名异常是合法情况，这种情况交由 handle_mm_fault 统一处理。

```
3107 /*
```

```
3108  * By the time we get here, we already hold the mm semaphore
3109  */
3110 int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma,
3111             unsigned long address, unsigned int flags)
3112 {
3113     pgd_t *pgd;
3114     pud_t *pud;
3115     pmd_t *pmd;
3116     pte_t *pte;
3117
3118     __set_current_state(TASK_RUNNING);
3119
3120     count_vm_event(PGFAULT);
3121
3122     /* do counter updates before entering really critical section. */
3123     check_sync_rss_stat(current);
3124
3125     if (unlikely(is_vm_hugetlb_page(vma)))
3126         return hugetlb_fault(mm, vma, address, flags);
3127
3128     pgd = pgd_offset(mm, address);
3129     pud = pud_alloc(mm, pgd, address);
3130     if (!pud)
3131         return VM_FAULT_OOM;
3132     pmd = pmd_alloc(mm, pud, address);
3133     if (!pmd)
3134         return VM_FAULT_OOM;
3135     pte = pte_alloc_map(mm, pmd, address);
3136     if (!pte)
3137         return VM_FAULT_OOM;
3138
```

分配 pud，pmd，pte，分配好后，调用 3139 行处理。

```
3139        return handle_pte_fault(mm, vma, address, pte, pmd, flags);
```

处理各种情况的缺页异常。

```
3140 }
```

```
3054 static inline int handle_pte_fault(struct mm_struct *mm,
3055                 struct vm_area_struct *vma, unsigned long address,
3056                 pte_t *pte, pmd_t *pmd, unsigned int flags)
3057 {
3058         pte_t entry;
3059         spinlock_t *ptl;
3060
3061         entry = *pte;
3062         if (!pte_present(entry)) {
```

页不在物理内存中

```
3063                 if (pte_none(entry)) {
3064                         if (vma->vm_ops) {
3065                                 if (likely(vma->vm_ops->fault))
3066                                         return do_linear_fault(mm, vma, address,
3067                                                 pte, pmd, flags, entry);
```

处理线性地址异常

```
3068                         }
3069                         return do_anonymous_page(mm, vma, address,
3070                                                 pte, pmd, flags);
```

处理匿名页异常

```
3071                 }
3072                 if (pte_file(entry))
3073                         return do_nonlinear_fault(mm, vma, address,
3074                                         pte, pmd, flags, entry);
```

处理非线性地址

```
3075            return do_swap_page(mm, vma, address,
3076                    pte, pmd, flags, entry);
```

处理交换页异常

```
3077    }
3078
3079    ptl = pte_lockptr(mm, pmd);
3080    spin_lock(ptl);
3081    if (unlikely(!pte_same(*pte, entry)))
3082            goto unlock;
3083    if (flags & FAULT_FLAG_WRITE) {
3084            if (!pte_write(entry))
3085                    return do_wp_page(mm, vma, address,
3086                            pte, pmd, ptl, entry);
```

页在物理内存中，写时复制

```
3087            entry = pte_mkdirty(entry);
3088    }
3089    entry = pte_mkyoung(entry);
3090    if (ptep_set_access_flags(vma, address, pte, entry, flags &
FAULT_FLAG_WRITE)) {
3091            update_mmu_cache(vma, address, pte);
3092    } else {
3093            /*
3094             * This is needed only for protection faults but the arch code
3095             * is not yet telling us if this is a protection fault or not.
3096             * This still avoids useless tlb flushes for .text page faults
3097             * with threads.
3098             */
3099            if (flags & FAULT_FLAG_WRITE)
3100                    flush_tlb_page(vma, address);
3101    }
3102 unlock:
```

```
3103        pte_unmap_unlock(pte, ptl);
3104        return 0;
3105 }
```

如果页不在物理内存中，即(!pte_present(entry)为真，则分三种情况

- 如果没有对应的页表项（3063），则内核必须从头开始加载该页，对匿名映射调用 do_anonymous_page 按需分配（3069）。对 vm_ops 非空的情况下，调用 do_linear_fault 函数按需调页（3073）。
- 如果该页标记为不存在，而也表项中保存了相关信息，则意味着页已经换出，因为必须从系统的某个交换区换入（3075）。
- 非线性映射已经换出的部分不能像普通页那样存取，必须先回复非线性关联。调用 do_nonliner_fault 处理。

如果页在物理内存中，该区域对页赋予了写权限，而硬件的存取机制没有赋予。则必须调用 do_wp_page 处理写时复制（3085）。

3.6.3 bad_area处理

对于非常页面异常范围，最终会调用__bad_area_nosemaphore 进行善后处理。

```
703 static void
 704 __bad_area_nosemaphore(struct pt_regs *regs, unsigned long
error_code,
 705                        unsigned long address, int si_code)
 706 {
 707        struct task_struct *tsk = current;
 708
 709        /* User mode accesses just cause a SIGSEGV */
 710        if (error_code & PF_USER) {
```

如果是用户态异常

```
 711                /*
 712                 * It's possible to have interrupts off here:
 713                 */
```

```
714             local_irq_enable();
715
716             /*
717              * Valid to do another page fault here because this one came
718              * from user space:
719              */
720             if (is_prefetch(regs, error_code, address))
721                     return;
722
723             if (is_errata100(regs, address))
724                     return;
725
726             if (unlikely(show_unhandled_signals))
727                     show_signal_msg(regs, error_code, address, tsk);
728
729             /* Kernel addresses are always protection faults: */
730             tsk->thread.cr2         = address;
731             tsk->thread.error_code    = error_code | (address >=
TASK_SIZE);
732             tsk->thread.trap_no = 14;
733
734             force_sig_info_fault(SIGSEGV, si_code, address, tsk);
```

发送 SIGSEGV 信号给引起异常的进程。

```
735
736             return;
737     }
738
739     if (is_f00f_bug(regs, address))
740             return;
741
742     no_context(regs, error_code, address);
```

对于内核态异常，调用 no_context 进行最后一次努力。

```
743 }
```

```
623 static noinline void
624 no_context(struct pt_regs *regs, unsigned long error_code,
625            unsigned long address)
626 {
627     struct task_struct *tsk = current;
628     unsigned long *stackend;
629     unsigned long flags;
630     int sig;
631
632     /* Are we prepared to handle this kernel fault? */
633     if (fixup_exception(regs))
634         return;
```

调用 fixup_exception 进行异常修补。

```
635
636     /*
637      * 32-bit:
638      *
639      *   Valid to do another page fault here, because if this fault
640      *   had been triggered by is_prefetch fixup_exception would have
641      *   handled it.
642      *
643      * 64-bit:
644      *
645      *   Hall of shame of CPU/BIOS bugs.
646      */
647     if (is_prefetch(regs, error_code, address))
648         return;
649
```

```
650         if (is_errata93(regs, address))
651                 return;
652
653         /*
654          * Oops. The kernel tried to access some bad page. We'll have to
655          * terminate things with extreme prejudice:
656          */
657         flags = oops_begin();
658
659         show_fault_oops(regs, error_code, address);
660
661         stackend = end_of_stack(tsk);
662         if (tsk != &init_task && *stackend != STACK_END_MAGIC)
663                 printk(KERN_ALERT "Thread overran stack, or stack
corrupted\n");
664
665         tsk->thread.cr2         = address;
666         tsk->thread.trap_no = 14;
667         tsk->thread.error_code    = error_code;
668
669         sig = SIGKILL;
670         if (__die("Oops", regs, error_code))
671                 sig = 0;
672
673         /* Executive summary in case the body of the oops scrolled away */
674         printk(KERN_EMERG "CR2: %016lx\n", address);
675
676         oops_end(flags, regs, sig);
```

oops 处理。

```
677 }
```

4. 参考书籍

[1] linux 内核 2.6.34 源码
[2] 深入了解 linux 内核 （第三版）
[3] linxu 内核源代码情景分析 毛德超
[4] 深入 linxu 内核架构
[5] http://blog.csdn.net/yunsongice(网络资源)