# 一.物理内存初始化

## *1.物理内存管理的数据结构：*

## 1.内存节点 pglist_data

mmzone.h

```
typedef struct pglist_data {
    struct zone node_zones[MAX_NR_ZONES];
    struct zonelist node_zonelists[MAX_ZONELISTS];
    int nr_zones;
    #ifdef CONFIG_FLAT_NODE_MEM_MAP /* means !SPARSEMEM */
     struct page *node_mem_map;
          #ifdef CONFIG_CGROUP_MEM_RES_CTLR
               struct page_cgroup *node_page_cgroup;
          #endif
     #endif
     #ifndef CONFIG_NO_BOOTMEM
      struct bootmem_data *bdata;
     #endif
          #ifdef CONFIG_MEMORY_HOTPLUG
               spinlock_t node_size_lock;
          #endif
    unsigned long node_start_pfn;
    unsigned long node_present_pages; /* total number of physical pages */
    unsigned long node_spanned_pages; /* total size of physical page range, including holes */
    int node_id;
    wait_queue_head_t kswapd_wait;
    struct task_struct *kswapd;
    int kswapd_max_order;
} pg_data_t;
```

部分元素解释：

(1) node_zones :节点中包含的内存域(zone)的数据结构。

(2) node_zonelists : 指定备用节点的内存域的的列表。

```
     struct zonelist {
        struct zonelist_cache *zlcache_ptr;          // NULL or &zlcache
        struct zoneref _zonerefs[MAX_ZONES_PER_ZONELIST + 1];
            #ifdef CONFIG_NUMA
                struct zonelist_cache zlcache;               // optional ...
           #endif
};
```

(3) node_mem_map : 在 FLATMEM 和 DISCONTIGMEM（即：在平坦和不连续模型）下，该元素指向保存当前节点中的所有物理页框描述符(page)的内存区域。 在 UMA 结构中系统的唯一内存节点 contig_page_data 的该成员变量指向的地址也保存在全局变量 mem_map 中。
(4) bdata : 指向系统初始化期间使用的内存分配数据结构。
(5) node_start_pfn :当前节点中包含的第一个物理页面号。系统中的所有节点的物理页面号是依次编号的，每个物理页面号是全局唯一的。当在 UMA 系统中，该字段值总是 0.
(6) node_present_pages: 记录当前节点包含物理页面的总数，不包括漏洞。
(7) node_spanned_pages: 记录当前节点的最大物理页号与最小物理页号之差。
(8) kswapd_wait: 是内核线程 kswapd 的等待队列。该内核线程通常睡眠在对应的内存节点的的等待队列上，在分配物理页框的过程中会检查每个 zone 中页面的使用情况，如果某个 zone 的空闲页面小于一个临界值，那么将会唤醒睡眠在等待队列的 kswapd 线程。
(9) kswapd_max_order: 记录了需要从内存节点中交换出多少被占用的页面。

## 2.内存域 zone

mmzone.h

```
struct zone {
    unsigned long watermark[NR_WMARK];
    unsigned long percpu_drift_mark;
    unsigned long       lowmem_reserve[MAX_NR_ZONES];
            #ifdef CONFIG_NUMA
               int node;
               unsigned long       min_unmapped_pages;
               unsigned long       min_slab_pages;
           #endif
    struct per_cpu_pageset __percpu *pageset;
    spinlock_t      lock;
    int                   all_unreclaimable; /* All pages pinned */
           #ifdef CONFIG_MEMORY_HOTPLUG
                seqlock_t       span_seqlock;
           #endif
    struct free_area    free_area[MAX_ORDER];

           #ifndef CONFIG_SPARSEMEM
                unsigned long       *pageblock_flags;
           #endif /* CONFIG_SPARSEMEM */

           #ifdef CONFIG_COMPACTION
                 unsigned int       compact_considered;
                 unsigned int       compact_defer_shift;
           #endif
    ZONE_PADDING(_pad1_)
    spinlock_t      lru_lock;
```

```
    struct zone_lru {
        struct list_head list;
    } lru[NR_LRU_LISTS];
    struct zone_reclaim_stat reclaim_stat;
    unsigned long       pages_scanned;     /* since last reclaim */
    unsigned long       flags;         /* zone flags, see below */
    atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];
    unsigned int inactive_ratio;
    ZONE_PADDING(_pad2_)
    wait_queue_head_t   * wait_table;
    unsigned long       wait_table_hash_nr_entries;
   unsigned long       wait_table_bits;
    struct pglist_data  *zone_pgdat;
    unsigned long       zone_start_pfn;
    unsigned long       spanned_pages;  /* total size, including holes */
    unsigned long       present_pages;  /* amount of memory (excluding holes) */
    const char      *name;
} ____cacheline_internodealigned_in_smp;
```

部分元素解释：

(1)lowmem_reserve: 记录每个内存域中保留的物理页框数。为内核不能睡眠的关键路径分配内存。

(2) pageset: 该字段指向一块内存区域，该区域保存了该内存域中的用于实现每个 cpu 的热，冷页的列表。即为了加快物理页框的分配，为每个 cpu 保存了预先分配的页框。

内核说页是热的，意味着页已经加载到 cpu 高速缓存，冷页则不在高速缓存中。

```
struct per_cpu_pageset {
    struct per_cpu_pages pcp;
          #ifdef CONFIG_NUMA
              s8 expire;
         #endif
         #ifdef CONFIG_SMP
             s8 stat_threshold;
            s8 vm_stat_diff[NR_VM_ZONE_STAT_ITEMS];
         #endif
};

struct per_cpu_pages {
    int count;      /* number of pages in the list */
    int high;       /* high watermark, emptying needed */
    int batch;      /* chunk size for buddy add/remove */
    struct list_head lists[MIGRATE_PCPTYPES];
};
```

(3)free_area: 用于实现伙伴系统，每个数组元素都表示某种固定长度的内存区。

(4)lru[NR_LRU_LISTS]: 用来根据活动情况对内存域中使用的页进行编目。如果内存页访问频繁，则内核认为它是活动的。

(5)reclaim_stat:指定在回收内存时需要扫描 活动和不活动等页的数目。

```
struct zone_reclaim_stat {
    unsigned long       recent_rotated[2];
    unsigned long       recent_scanned[2];
    unsigned long       nr_saved_scan[NR_LRU_LISTS];
};
```

(6) all_unreclaimable:如果所有已分配的页都处于不可回收状态，该字段值为 1.

(7) vm_stat: 保存了关于内存域的统计信息。

(8) wait_table, wait_table_hash_nr_entries, wait_table_bits:

当一个物理页框因与外设交互数据而处于锁定状态(PG_locked)时,一个进程如果访问此类页框需要阻塞，直到页框完成与外设的数据交换，以保持数据的一致性。

内核中为每个内存区设置了一个等待队列表，该表中包含了一定数目的等待队列，物理页通过哈希函数关联到相应的等待队列上，其中 wait_table 保存了 wait_table_hash_nr_entries 个等待队列的队列头。

(9) zone_start_pfn: 是内存域中的第一个物理页号。

(10)ZONE_PADDING:

zone->lock and zone->lru_lock are two of the hottest locks in the kernel. So add a wild amount of padding here to ensure that they fall into separate cachelines. There are very few zone structures in the machine, so space consumption is not a concern here.

```
 struct zone_padding {
    char x[0];
} ____cacheline_internodealigned_in_smp;
```

## 3.物理页框 page

mm_types.h

```
struct page {
    unsigned long flags;        /* Atomic flags, some possibly  updated asynchronously */
    atomic_t _count;        /* Usage count, see below. */
    union {
        atomic_t _mapcount; /* Count of ptes mapped in mms,*/
        struct {        /* SLUB */
            u16 inuse;
            u16 objects;
        };
    };
    union {
        struct {
        unsigned long private;      /* Mapping-private opaque data:
                        * usually used for buffer_heads if PagePrivate set;
                         *used for swp_entry_t if PageSwapCache;
                        * indicates order in the buddy system if PG_buddy is set.
                        */
        struct address_space *mapping;  /* If low bit clear, points to
                        * inode address_space, or NULL.
```

```
                    * If page mapped as anonymous
                    * memory, low bit is set, and
                    * it points to anon_vma object:
                    * see PAGE_MAPPING_ANON below.
                    */
        };
                #if USE_SPLIT_PTLOCKS
                    spinlock_t ptl;
                #endif
        struct kmem_cache *slab;    /* SLUB: Pointer to slab */
        struct page *first_page;    /* Compound tail pages */
    };
    union {
        pgoff_t index;      /* Our offset within mapping. */
        void *freelist;     /* SLUB: freelist req. slab lock */
    };
    struct list_head lru;       /* Pageout list, eg. active_list  protected by zone->lru_lock ! */
            #if defined(WANT_PAGE_VIRTUAL)
                void *virtual;          /* Kernel virtual address (NULL if not kmapped, ie. highmem) */
            #endif /* WANT_PAGE_VIRTUAL */
            #ifdef CONFIG_WANT_PAGE_DEBUG_FLAGS
                unsigned long debug_flags;  /* Use atomic bitops on this */
            #endif
            #ifdef CONFIG_KMEMCHECK
                void *shadow;
            #endif
};
```

部分元素解释：

(1) flags:用于描述页的属性。

```
 * Various page->flags bits:
 *
 * PG_reserved is set for special pages, which can never be swapped out. Some
 * of them might not even exist (eg empty_bad_page)...
 *
 * The PG_private bitflag is set on pagecache pages if they contain filesystem
 * specific data (which is normally at page->private). It can be used by
 * private allocations for its own usage.
 *
 * During initiation of disk I/O, PG_locked is set. This bit is set before I/O
 * and cleared when writeback _starts_ or when read _completes_. PG_writeback
 * is set before writeback starts and cleared when it finishes.
```

*

* PG_locked also pins a page in pagecache, and blocks truncation of the file while it is held.

*

* page_waitqueue(page) is a wait queue of all tasks waiting for the page

* to become unlocked.

*

* PG_uptodate tells whether the page's contents is valid.  When a read

* completes, the page becomes uptodate, unless a disk I/O error happened.

*

* PG_referenced, PG_reclaim are used for page reclaim for anonymous and

* file-backed pagecache (see mm/vmscan.c).

*

* PG_error is set to indicate that an I/O error occurred on this page

* PG_buddy is set to indicate that the page is free and in the buddy system

(2) lru: 用于活动或非活动的链表，或伙伴算法对应的空闲列表。

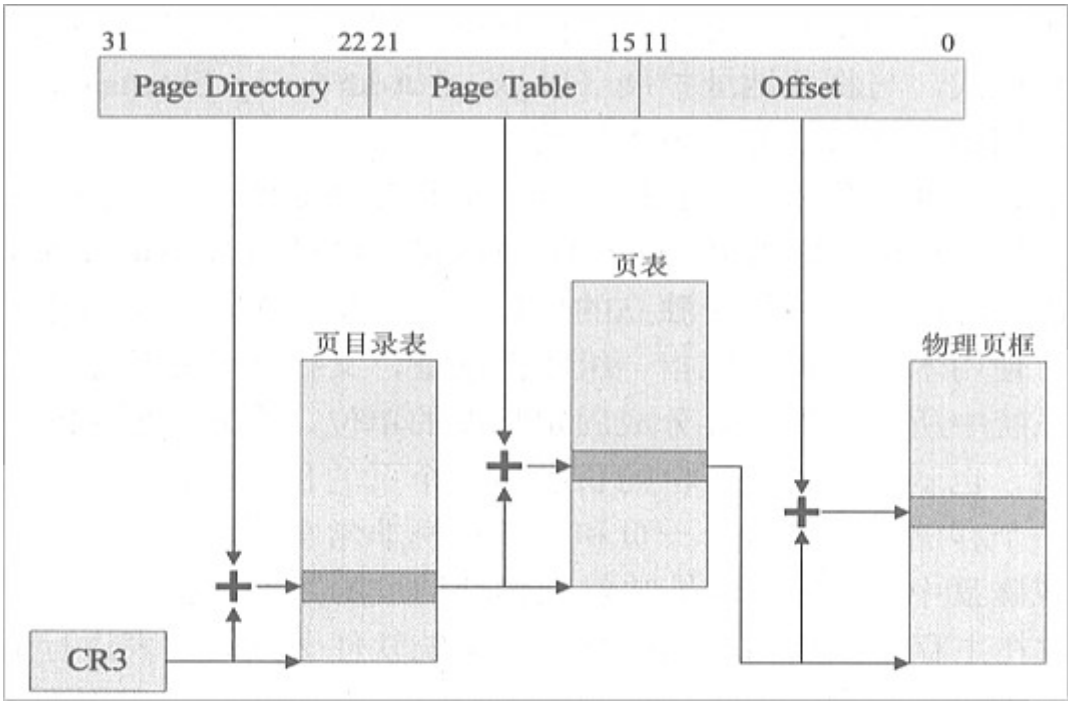
(3) virtual: 在高内存区中(HIGHMEM)物理页框映射到内核地址空间时，保存所映射到的内核线性地址。

## *2.分页机制*

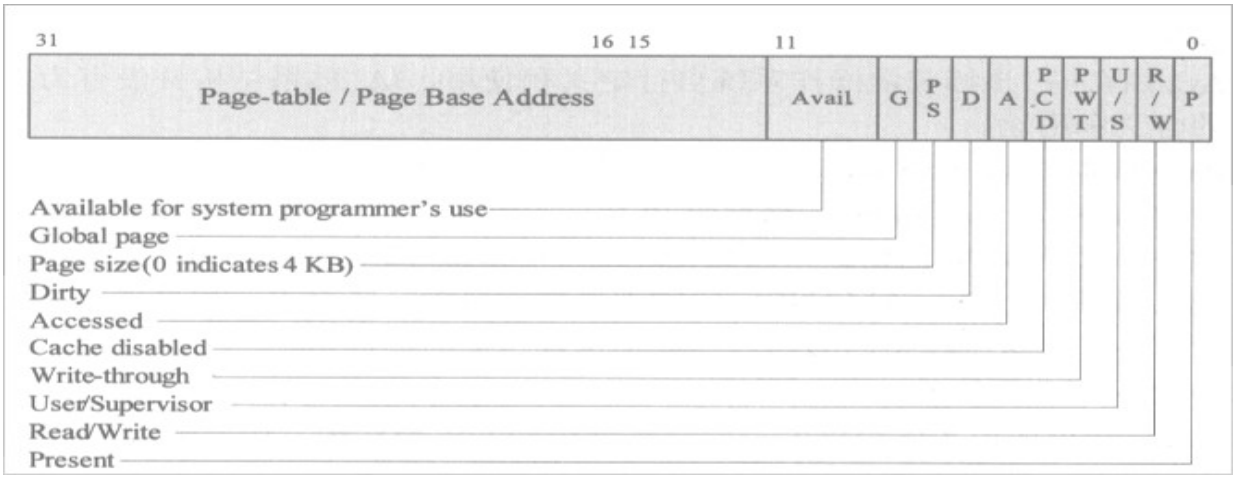
原理：根据程序运行期间的局部行原理，可知程序不必完全装入内存即可有效的运行。基于此原理出现了请求调页技术，它的原理是将开始运行所需的部分代码、数据装入内存，即开始运行。在运行过程中，如果需要访问当前不在主存中代码、数据，则将其调入，继续运行。

采用分页机制，主要是为了避免为页目录项分配比较大的连续内存空间。

## 1.IA32 体系结构下的分页机制：



页表项结构：




```
#define _PAGE_PRESENT     /* is present */
#define _PAGE_RW          /* writeable */
#define _PAGE_USER        /* userspace addressable */
#define _PAGE_PWT         /* page write through */
#define _PAGE_PCD         /* page cache disabled */
#define _PAGE_ACCESSED    /* was accessed (raised by CPU) */
#define _PAGE_DIRTY       /* was written to (raised by CPU) */
#define _PAGE_PSE         /* 4 MB (or 2MB) page */
#define _PAGE_GLOBAL      /* Global TLB entry PPro+ */
```

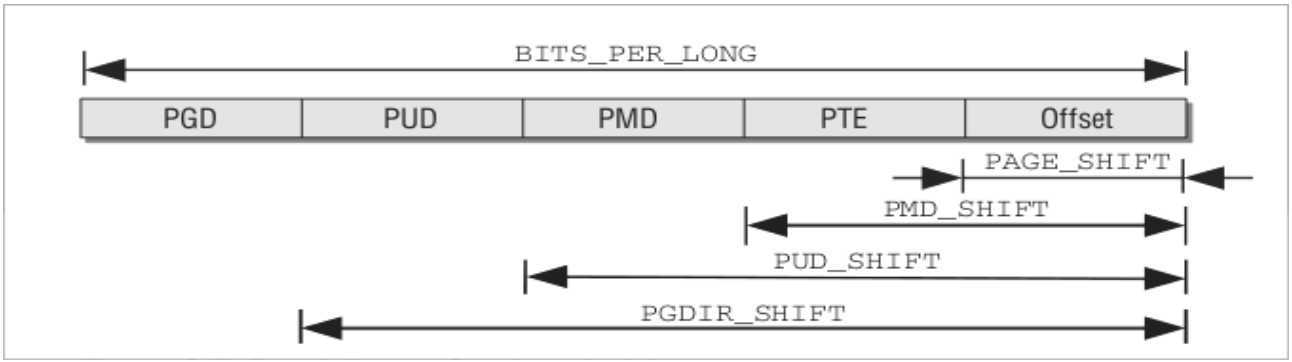
_PAGE_PRESENT:
specifies whether the virtual page is present in RAM memory. This need not necessarily be the case because pages may be swapped out into a swap area
_PAGE_ACCESSED :
is set automatically by the CPU each time the page is accessed. The kernel reg ularly checks the field to establish how actively the page is used (infrequently used pages are good swapping candidates). The bit is set after either read or write access.
_PAGE_DIRTY :

indicates whether the page is ‘‘dirty,’’ that is, whether the page contents have been modified. 每当对一个页框进行写操作时，CPU 就设置这个标志。
_PAGE_USER :
if is set, userspace code is allowed to access the page. Otherwise, only the kernel is allowed to do this (or when the CPU is in system mode).
_PAGE_PWT :
用于设置在写高速缓存(cache)时，采用的是写穿透策略，还是回写法
_PAGE_PCD :
设置为 1 时，高速缓存不对该页进行缓存，否则可以进行缓存。

## 2.linux 采用的分页机制：

linux 内核设计了一种通用的内存映射模型，在不同的体系结构上具体实现时，在落实到该体系结构提供的具体内存映射机制。

为了更好的支持 X86_64 体系结构中的采用的 48 位线性地址，内核采用了更加通用的 4 级分页机制，将线性地址划分为 5 个段。




## *3.页表初始化*

## 1.创建内核页表

head_32.S

```
page_pde_offset = (__PAGE_OFFSET >> 20);
     movl $pa(__brk_base), %edi
    movl $pa(swapper_pg_dir), %edx
    movl $PTE_IDENT_ATTR, %eax
10:
    leal PDE_IDENT_ATTR(%edi),%ecx      /* Create PDE entry */
    movl %ecx,(%edx)           /* Store identity PDE entry */
    movl %ecx,page_pde_offset(%edx)     /* Store kernel PDE entry */
    addl $4,%edx
    movl $1024, %ecx
11:
    stosl  /*将%eax中的内容放到%edi,%edi自动加4字节，%ecx值会自动减一*/
    addl $0x1000,%eax
    loop 11b
    /*
     * End condition: we must map up to the end + MAPPING_BEYOND_END.
     */
    movl $pa(_end) + MAPPING_BEYOND_END + PTE_IDENT_ATTR, %ebp
    cmpl %ebp,%eax
    jb 10b
    addl $__PAGE_OFFSET, %edi
```

```
    movl %edi, pa(_brk_end)
    shrl $12, %eax
    movl %eax, pa(max_pfn_mapped)

    /* Do early initialization of the fixmap area */
    movl $pa(swapper_pg_fixmap)+PDE_IDENT_ATTR,%eax
    movl %eax,pa(swapper_pg_dir+0xffc)
#endif
    jmp 3f
```

部分解释：

1.__brk_base： 是 linux 内核镜像中地址最靠后的一个符号，该符号随后的地址空间用来存放生成的页表信息。

2.swapper_pg_dir 的定义： ENTRY(swapper_pg_dir) .fill 1024,4,0 占用了 1024*4 字节的空间。用于存放系统的全局页目录表。

3._brk_end ：保存了这一阶段创建的最后一个页表项。该变量用作内存初始化的第二个阶段使用。

## 2.激活分页机制

head_32.S

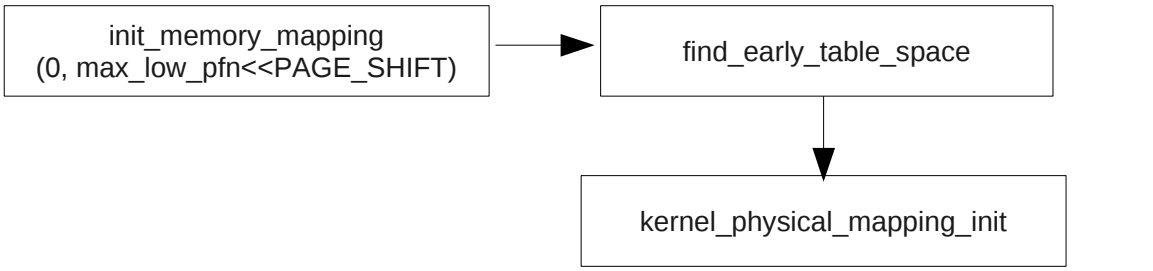
```
ENTRY(initial_page_table)
    .long pa(swapper_pg_dir)

/*
 * Enable paging
 */
    movl pa(initial_page_table), %eax
    movl %eax,%cr3      /* set the page table pointer.. */
    movl %cr0,%eax
    orl  $X86_CR0_PG,%eax
    movl %eax,%cr0      /* ..and set paging (PG) bit */
    ljmp $__BOOT_CS,$1f /* Clear prefetch and normalize %eip */
```

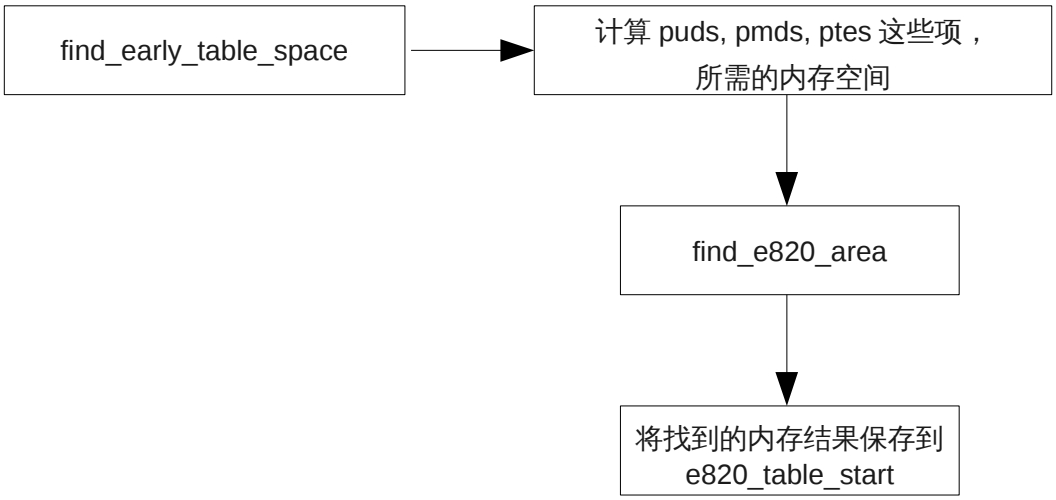
解释：

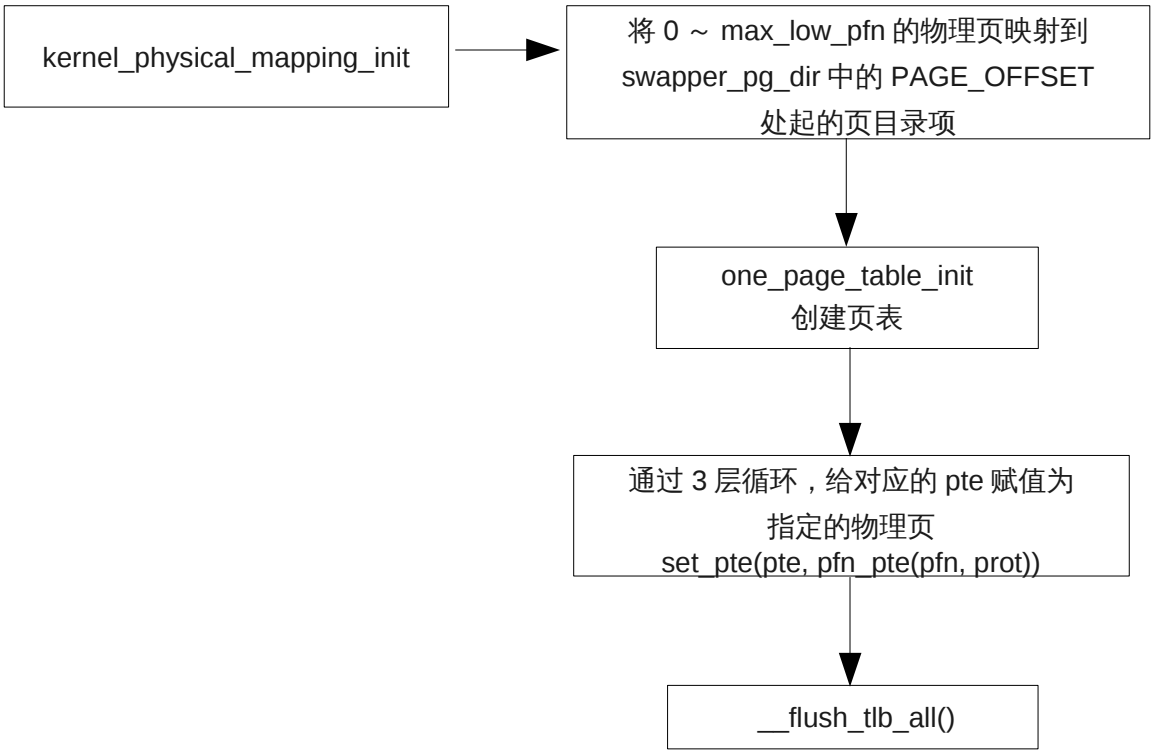
在启用分页机制之前，线性地址即为物理地址。指令寄存器 EIP 中必须是实际的物理地址才能有效访问。在即活分页机制之后，需要修改 EIP 为线性地址(此时线性地址与物理地址不再重合，线性地址大于 0XC0000000)才能有效访问。

## 3.完全初始化内存页表

arch/x86/kernel/setup.c



pte 的组成：

pte = one_page_table_init(pmd);

pte_ofs = pte_index((pfn<<PAGE_SHIFT) + PAGE_OFFSET);

pte += pte_ofs;

set_pte(pte, pfn_pte(pfn, init_prot));

通过上面代码，建立了内核的常规映射地址空间中的虚拟页框和系统低内存物理页框的映射关系，即：虚拟页框的虚拟地址和物理页框的物理地址之间恒定有个偏移量 PAGE_OFFSET。

## *4.物理内存的初始化*

## 1.注册内存相关信息

探测系统可用的物理内存数量，起始地址等相关信息，并将探测的可用物理内存信息保存到预留的内存位置中。该阶段的探测任务是由 BIOS 例程(E820 内存探测)完成

**1.查找 max_pfn， max_low_pfn，highmem_pages 的值：**

(1)e820_end_of_ram_pfn 函数从 e820map 中查找框架内可用的最大物理内存数。将最大物理内存数，赋给 max_pfn 和 max_low_pfn 变量。

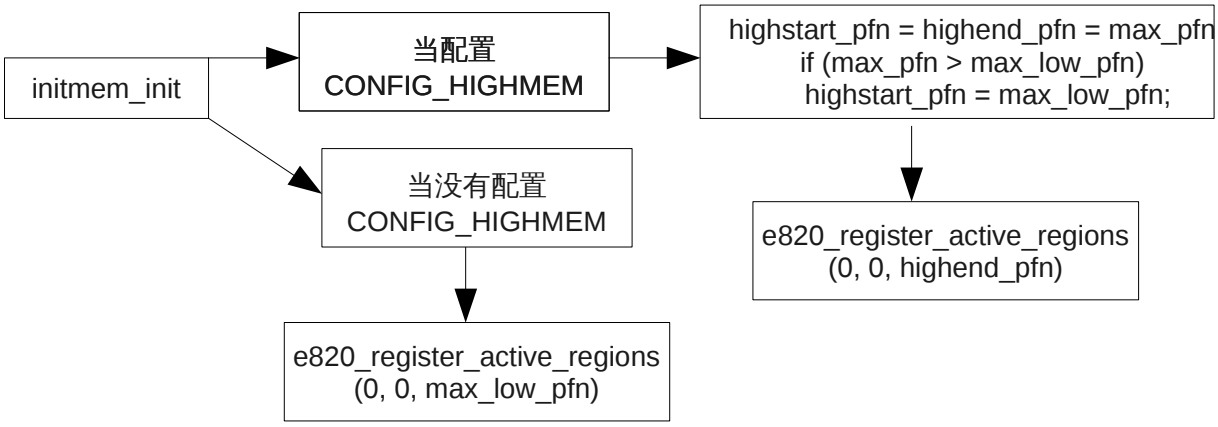
(2)find_low_pfn_rang 更新 max_low_pfn 变量的值，如果 max_pfn 的值小于 MAXMEM_PFN，则

max_low_pfn 的值为 max_pfn。如果 max_pfn 的值大于 MAXMEM_PFN，则 max_low_pfn 的值为 MAXMEM_PFN。然后设置 highmem_pages 的值。

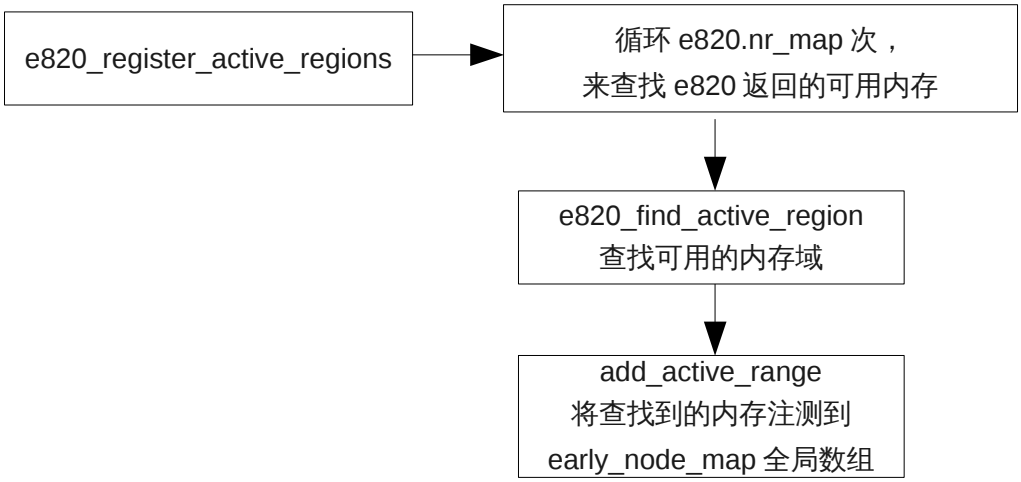
MAXMEM_PFN 为 (VMALLOC_END - PAGE_OFFSET - __VMALLOC_RESERVE)，大约是 896Mib

**2.注册活动内存区**

arch/x86/mm/init_32.c




arch/x86/kernel/e820.c

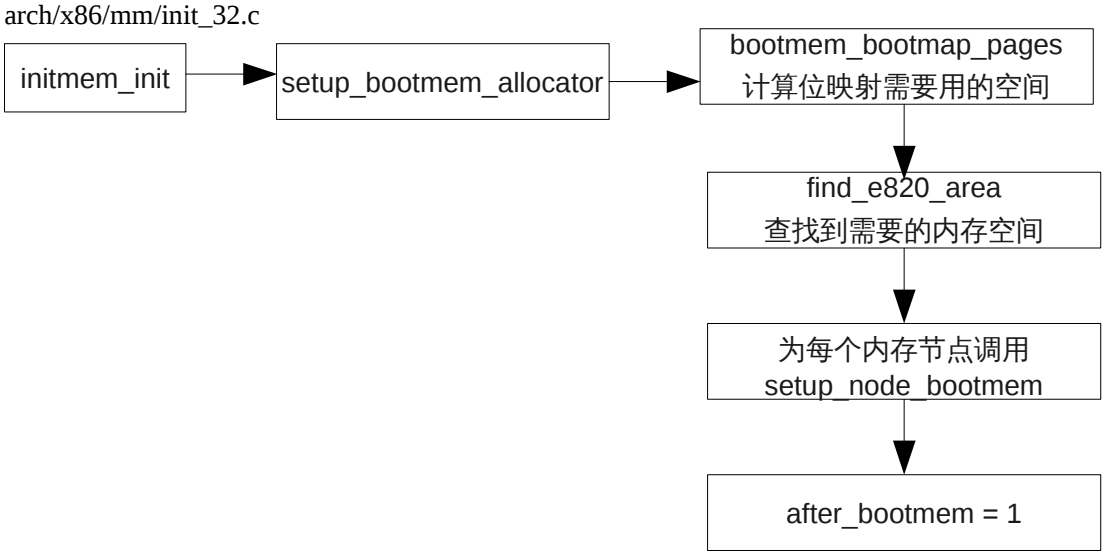


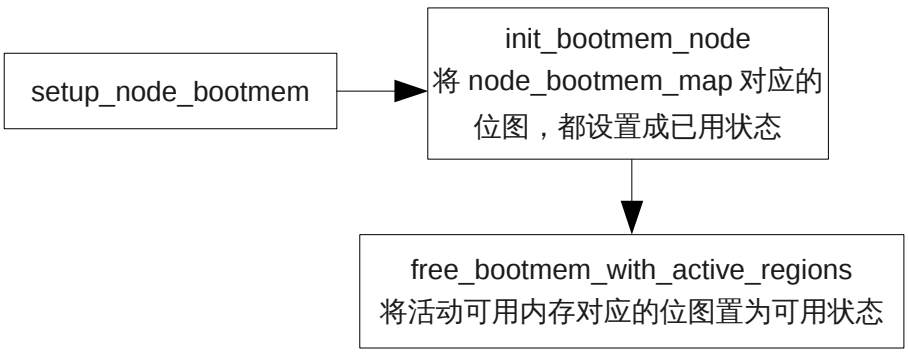

early_node_map 变量的结构体：

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

## 2.创建引导内存分配器

arch/x86/mm/init_32.c

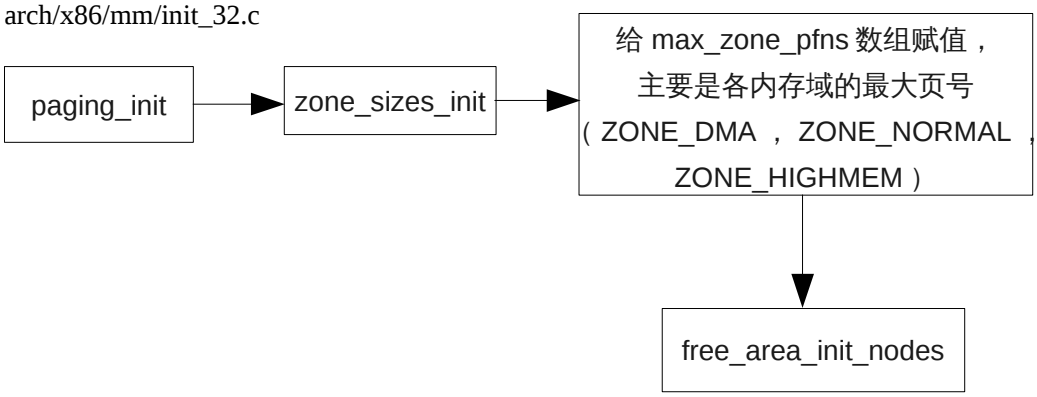


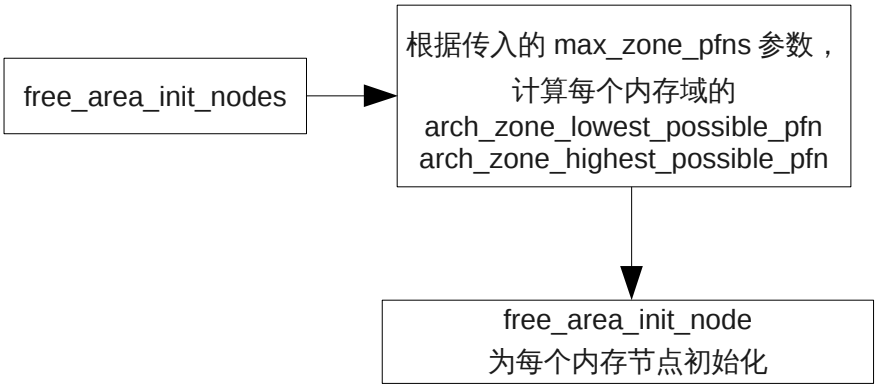



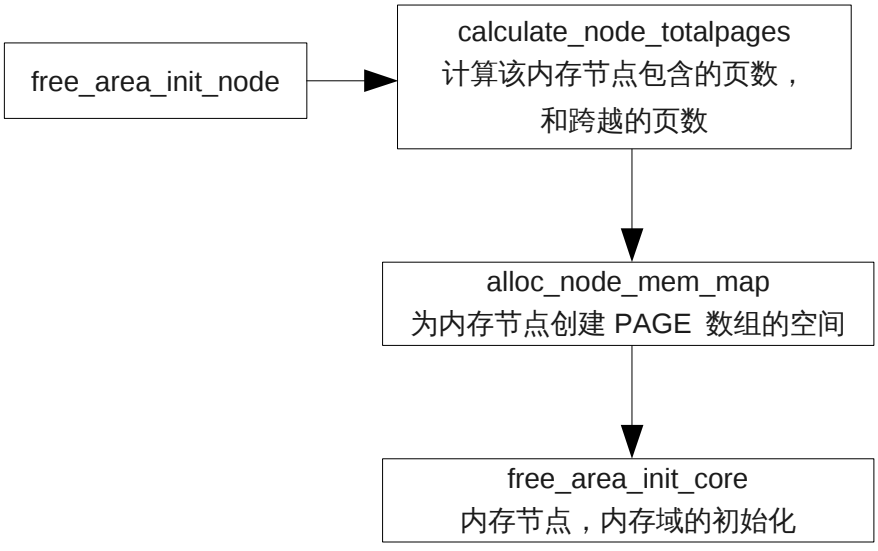
## 3.创建系统内存拓扑结构

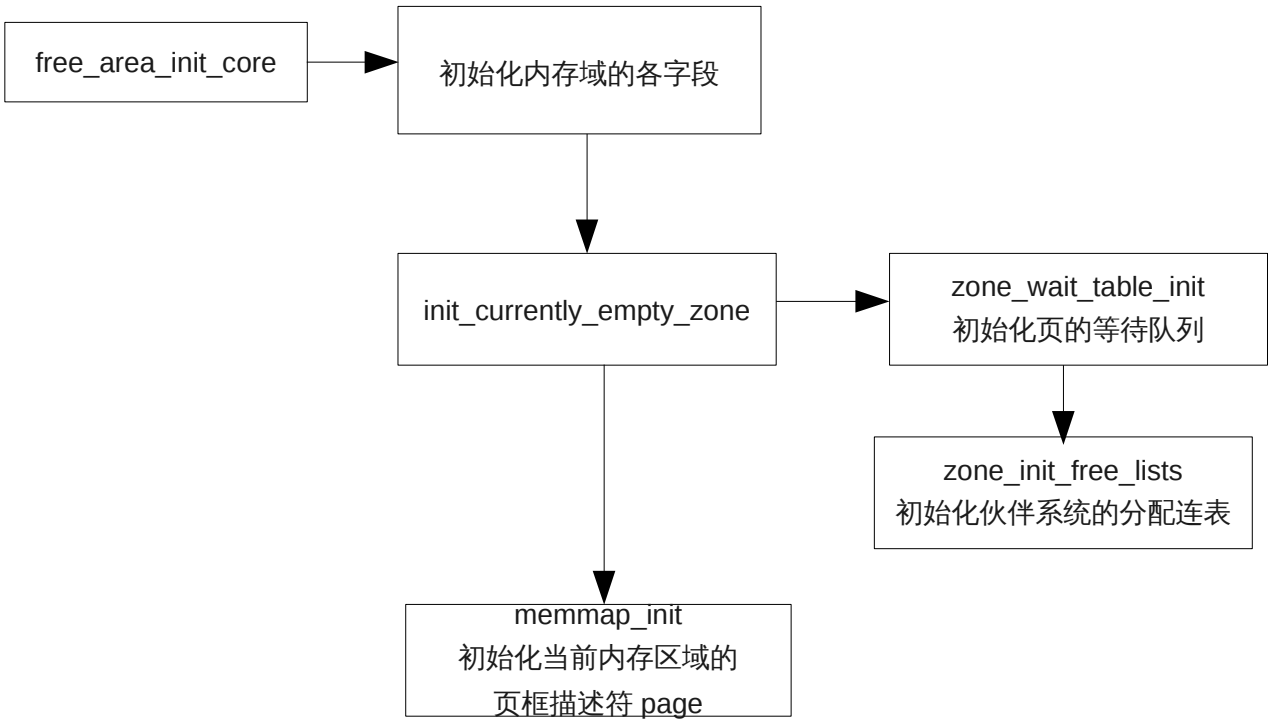
包括 pg_data 内存节点，zone 内存域，页框描述符 page.

arch/x86/mm/init_32.c



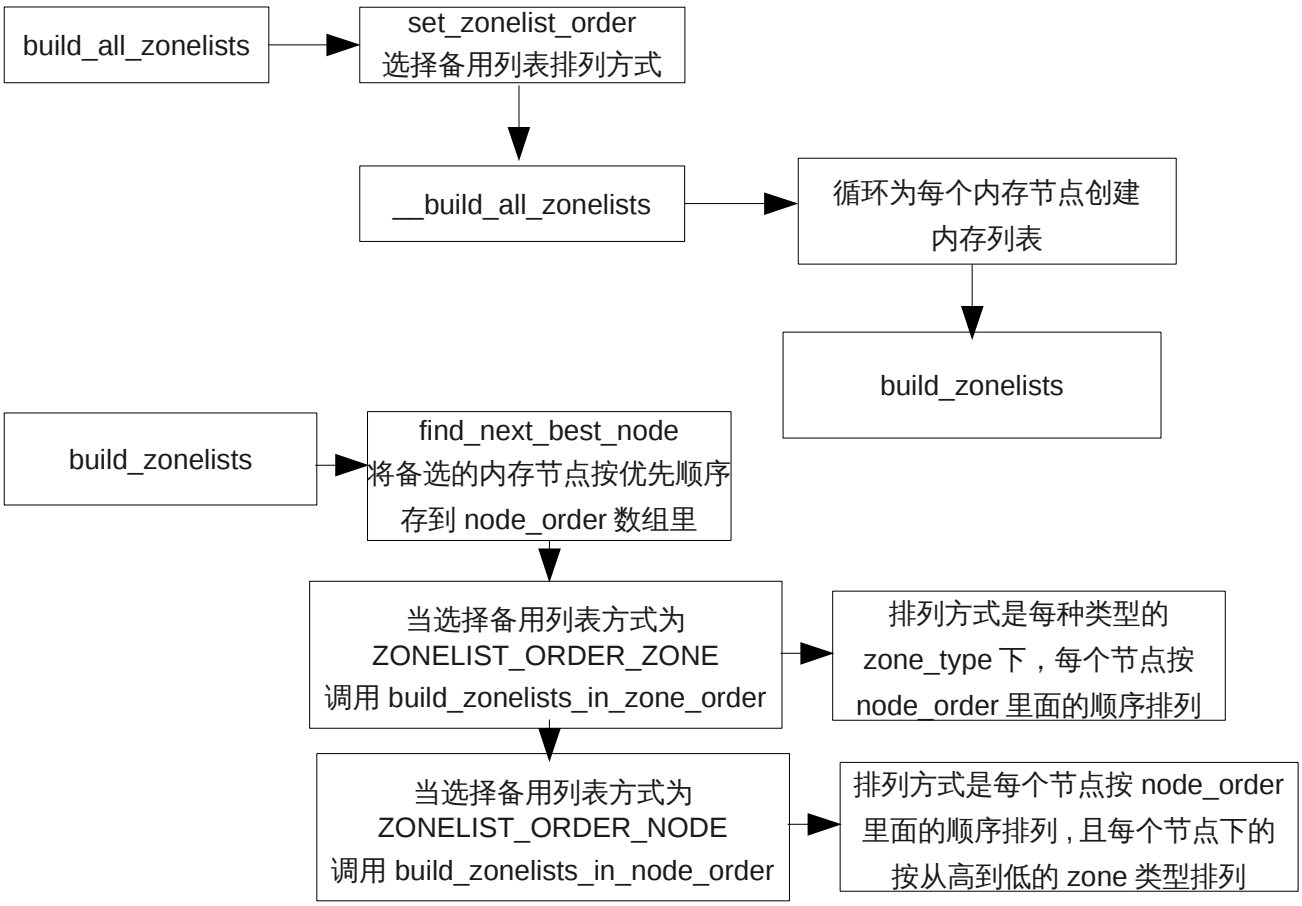






## 4.创建内存域列表

mm/page_alloc.c



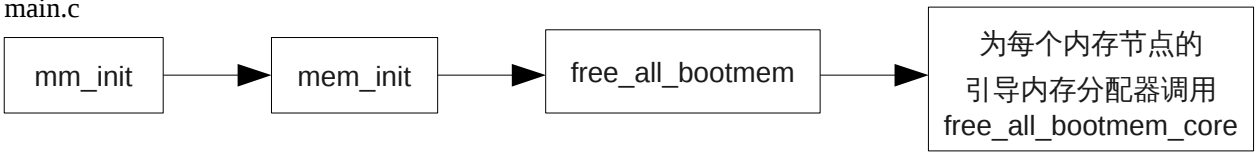

build_zonelists_in_zone_order 代码如下：

```
zonelist = &pgdat->node_zonelists[0];
```

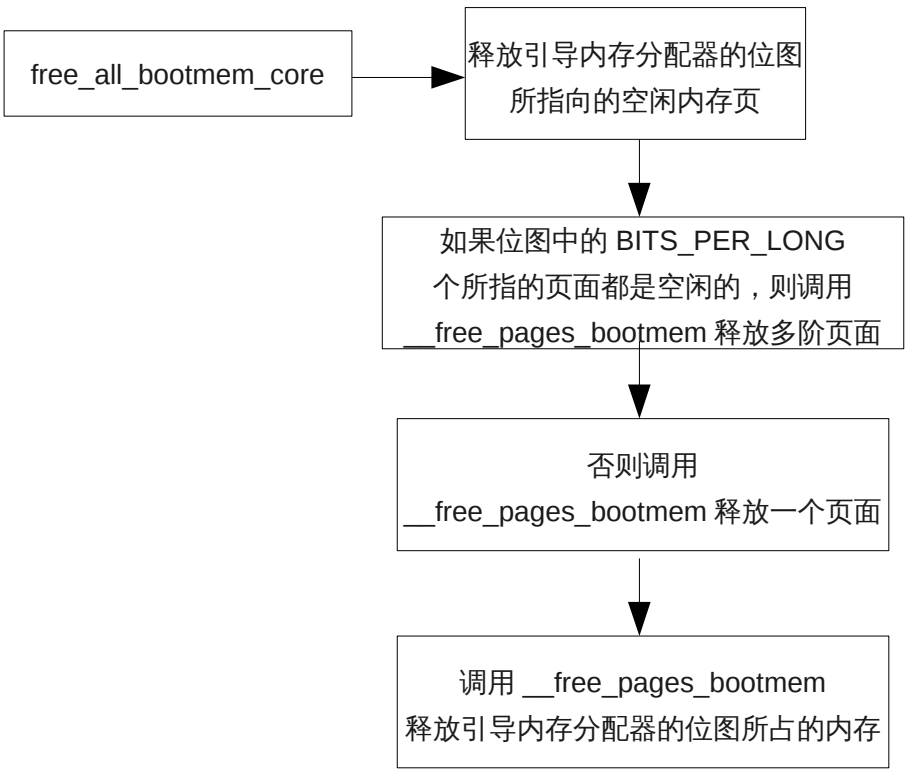
```
pos = 0;
for (zone_type = MAX_NR_ZONES - 1; zone_type >= 0; zone_type--) {
    for (j = 0; j < nr_nodes; j++) {
        node = node_order[j];
        z = &NODE_DATA(node)->node_zones[zone_type];
        if (populated_zone(z)) {
            zoneref_set_zone(z, &zonelist->_zonerefs[pos++]);
            check_highest_zone(zone_type);
        }
    }
}
zonelist->_zonerefs[pos].zone = NULL;
zonelist->_zonerefs[pos].zone_idx = 0;
```

## 5.废除引导内存分配器

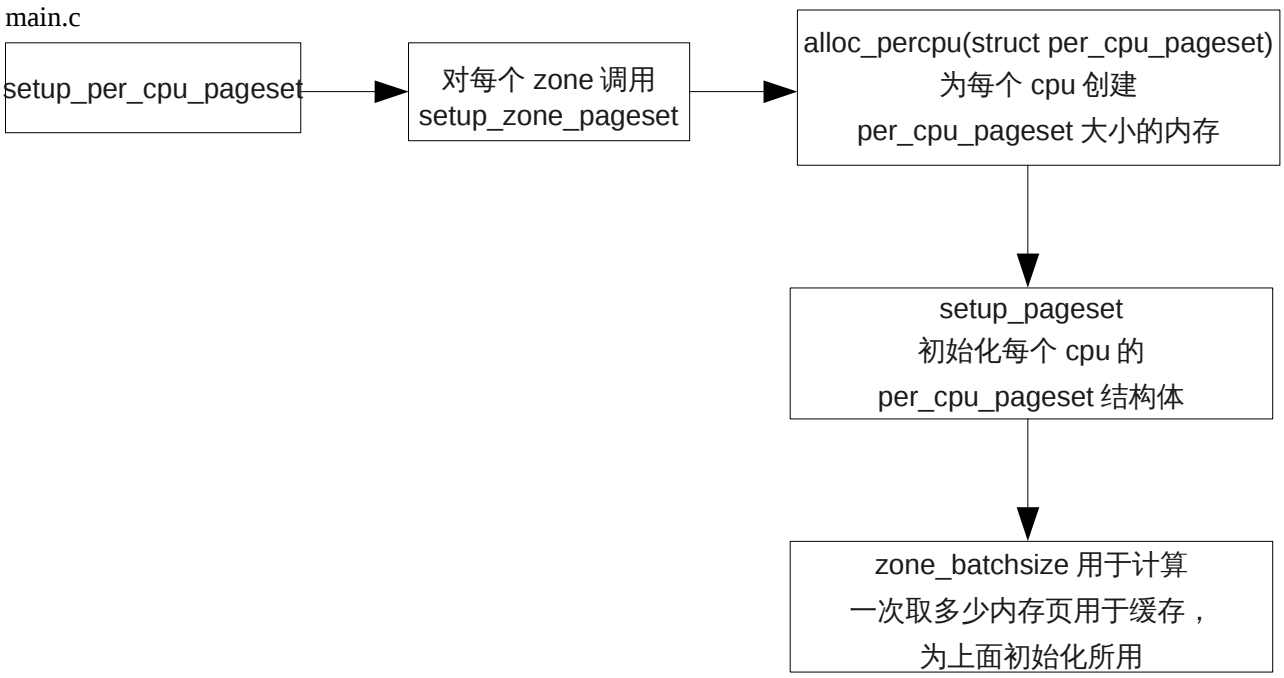
main.c




mm/bootmem.c



## 6.冷热缓存的初始化

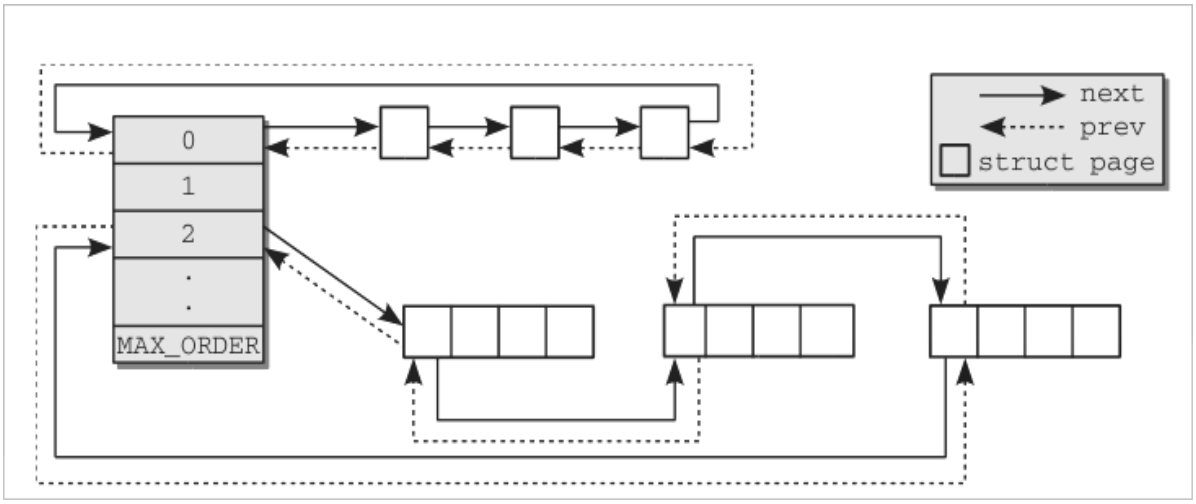



# 二.物理内存的分配

## *1.伙伴系统内存分配*

## 1.伙伴系统概述

### *1.伙伴系统的总体图：*



***2.内核将分配页分为3类：***

不可移动页：Non-movable pages have a fixed position in memory and cannot be moved anywhere else. Most allocations of the core kernel fall into this category.

可回收页 ：Reclaimable pages cannot be moved directly, but they can be deleted and their contents regenerated from some source. Data mapped from files fall into this category,。
for instance. Reclaimable pages are periodically freed by the kswapd daemon depending on how often they are accessed.

可移动页：Movable pages can be moved around as desired. Pages that belong to userspace applications fall into this category. They are mapped via page tables. If they are copied into a new location, the page table entries can be updated accordingly, and the application won't notice anything.

***3.内存域zone中包含的伙伴系统的分配结构：***

mmzone.h

```
#define MIGRATE_UNMOVABLE     0
#define MIGRATE_RECLAIMABLE   1
#define MIGRATE_MOVABLE       2
#define MIGRATE_PCPTYPES      3 /* the number of types on the pcp lists */
#define MIGRATE_RESERVE       3
#define MIGRATE_ISOLATE       4 /* can't allocate from here */
#define MIGRATE_TYPES         5

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

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

部分元素解释：

nr_free：统计了所有列表上空闲页的数目。

free_list[MIGRATE_TYPES]：每种迁移类型都对应一个空闲列表。

### *4.伙伴系统自身的备用列表*

在指定的列表中无法满足分配请求时，按如下顺序查找，别的迁移列表

```
static int fallbacks[MIGRATE_TYPES][MIGRATE_TYPES-1] = {
    [MIGRATE_UNMOVABLE]   = { MIGRATE_RECLAIMABLE, MIGRATE_MOVABLE,
MIGRATE_RESERVE },
    [MIGRATE_RECLAIMABLE] = { MIGRATE_UNMOVABLE,   MIGRATE_MOVABLE,
MIGRATE_RESERVE },
    [MIGRATE_MOVABLE]     = { MIGRATE_RECLAIMABLE, MIGRATE_UNMOVABLE,
MIGRATE_RESERVE },
    [MIGRATE_RESERVE]     = { MIGRATE_RESERVE,     MIGRATE_RESERVE,
MIGRATE_RESERVE }, /* Never used */
};
```

### *5.迁移类型列表的启用*

pageblock_order，pageblock_nr_pages 这两个全局变量，pageblock_order 表示内核认为是“大”的一个分配阶，

pageblock_nr_pages 是该阶所对应的页数。

如果各迁移类型的链表中没有一块较大的连续内存，那么页面迁移不会有任何好处，因此在可用内存少时，内核会关闭该特性。

在 pageblock_nr_pages 函数中：

```
    if (vm_total_pages < (pageblock_nr_pages * MIGRATE_TYPES))
        page_group_by_mobility_disabled = 1; //表示不起用页面迁移
    else
        page_group_by_mobility_disabled = 0;
```

### *6.分配标志转换成对应的迁移类型*

allocflags_to_migratetype(gfp_t gfp_flags) 函数的主要实现：

```
 if (unlikely(page_group_by_mobility_disabled))
          return MIGRATE_UNMOVABLE;
    /* Group based on mobility */
    return (((gfp_flags & __GFP_MOVABLE) != 0) << 1) |
        ((gfp_flags & __GFP_RECLAIMABLE) != 0);
```

如果停用了页面迁移特性，则所有的页都是不可移动的页。否则，该函数返回值可以直接用作

free_area. free_list 的数组索引。

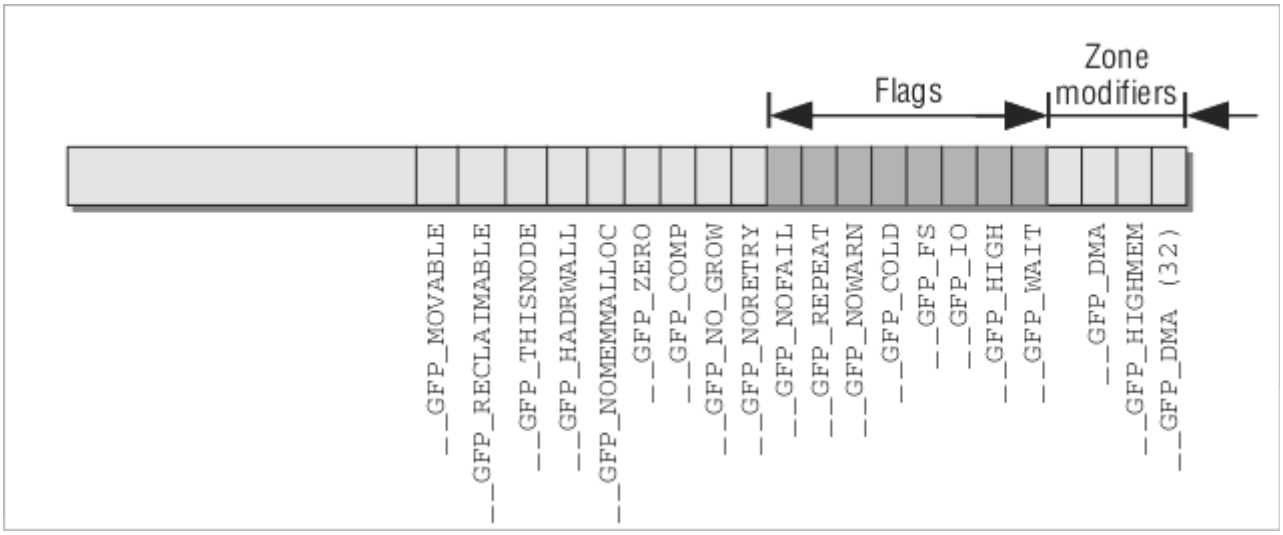
### *7.虚拟内存域 ZONE_MOVABLE*

```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
ZONE_DMA
#endif
#ifdef CONFIG_ZONE_DMA32
    ZONE_DMA32,
#endif
    ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
    ZONE_HIGHMEM,
#endif
    ZONE_MOVABLE,
    __MAX_NR_ZONES
};
```

ZONE_MOVABLE 内存域可能位于高端或普通内存域。与系统中所有其他的内存域相反，
ZONE_MOVABLE 并不关联到任何硬件上有意义的内存范围。实际上该内存域中的内存取自高端内存域或普通内存域。所以 ZONE_MOVABLE 是一个虚拟内存域。

## 2.伙伴系统内存分配器

### *1.分配掩码*



伙伴系统的分配函数，都需要指定对应的分配掩码，来决定分配页的属性。(图有点旧)

*1.*内存域修饰符

分配掩码的低 4 位，表示内存域修饰符。定义从哪个内存域分配所需的页。

```
#define __GFP_DMA   ((__force gfp_t)0x01u)
#define __GFP_HIGHMEM   ((__force gfp_t)0x02u)
#define __GFP_DMA32 ((__force gfp_t)0x04u)
#define __GFP_MOVABLE   ((__force gfp_t)0x08u)  /* Page is movable */
#define GFP_ZONEMASK    (__GFP_DMA|__GFP_HIGHMEM|__GFP_DMA32|__GFP_MOVABLE)
```

通过分配掩码，决定使用的内存域函数：

```
enum zone_type gfp_zone(gfp_t flags){
    enum zone_type z;
    int bit = flags & GFP_ZONEMASK;
    z = (GFP_ZONE_TABLE >> (bit * ZONES_SHIFT)) &
                    ((1 << ZONES_SHIFT) – 1);
.....
}

#define GFP_ZONE_TABLE ( \
    (ZONE_NORMAL << 0 * ZONES_SHIFT)              \
    | (OPT_ZONE_DMA << __GFP_DMA * ZONES_SHIFT)       \
    | (OPT_ZONE_HIGHMEM << __GFP_HIGHMEM * ZONES_SHIFT)    \
    | (OPT_ZONE_DMA32 << __GFP_DMA32 * ZONES_SHIFT)       \
    | (ZONE_NORMAL << __GFP_MOVABLE * ZONES_SHIFT)        \
    | (OPT_ZONE_DMA << (__GFP_MOVABLE | __GFP_DMA) * ZONES_SHIFT)   \
    | (ZONE_MOVABLE << (__GFP_MOVABLE | __GFP_HIGHMEM) * ZONES_SHIFT)\
    | (OPT_ZONE_DMA32 << (__GFP_MOVABLE | __GFP_DMA32) * ZONES_SHIFT)\
)
```

通过将 GFP_ZONE_TABLE 宏所包含的各情况的内存域的位列表，用指定的内存域描述符的值进行移位，然后通过((1 << ZONES_SHIFT) – 1)，求得需要分配的内存域。

*2.*分配掩码的其他标志位

```
#define __GFP_WAIT  ((__force gfp_t)0x10u)  /* Can wait and reschedule? */
#define __GFP_HIGH  ((__force gfp_t)0x20u)  /* Should access emergency pools? */
#define __GFP_IO    ((__force gfp_t)0x40u)  /* Can start physical IO? */
```

#define __GFP_FS    ((__force gfp_t)0x80u)  /* Can call down to low-level FS? */

#define __GFP_COLD  ((__force gfp_t)0x100u) /* Cache-cold page required */

#define __GFP_NOWARN    ((__force gfp_t)0x200u) /* Suppress page allocation failure warning */

#define __GFP_REPEAT    ((__force gfp_t)0x400u) /* Try hard to allocate the memory, but the allocation attempt might  fail*/

#define __GFP_NOFAIL    ((__force gfp_t)0x800u) /* the caller cannot handle allocation failures */

#define __GFP_NORETRY   ((__force gfp_t)0x1000u)

#define __GFP_COMP  ((__force gfp_t)0x4000u)/* Add compound page metadata */

#define __GFP_ZERO  ((__force gfp_t)0x8000u)/* Return zeroed page on success */

#define __GFP_NOMEMALLOC ((__force gfp_t)0x10000u) /* Don't use emergency reserves */

#define __GFP_HARDWALL   ((__force gfp_t)0x20000u) /* Enforce hardwall cpuset memory allocs */

#define __GFP_THISNODE  ((__force gfp_t)0x40000u)/* No fallback, no policies */

#define __GFP_RECLAIMABLE ((__force gfp_t)0x80000u) /* Page is reclaimable */

标志解释：

**__GFP_WAIT:** indicates that the memory request may be interrupted; that is, the scheduler is free to select another process during the request, or the request can be interrupted by a more important event. The allocator is also permitted to wait for an event on a queue (and to put the process to sleep) before memory is returned.

**__GFP_HIGH**: is set if the request is very important, that is, when the kernel urgently needs memory. This flag is always used when failure to allocate memory would have massive consequences for the kernel resulting in a threat to system stability or even a system crash.

**__GFP_IO** : specifies that the kernel can perform I/O operations during an attempt to find fresh memory. In real terms, this means that if the kernel begins to swap out pages during memory allocation, the selected pages may be written to hard disk only if this flag is set.

**__GFP_FS**: allows the kernel to perform VFS operations. This must be prevented in kernel layers linked with the VFS layer because interactions of this kind could cause endless recursive calls.

**__GFP_COLD**: is set if allocation of a ‘‘cold‘‘ page that is not resident in the CPU cache is required.

**__GFP_RECLAIMABLE and __GFP_MOVABLE**: are required by the page mobility mechanism. As their names indicate, they mark that the allocated memory will be reclaimable or movable, respectively. This influences from which sublist of the freelist the page or pages will be taken.

**__GFP_HARDWALL**: is meaningful on NUMA systems only. It limits memory allocation to the nodes associated with the CPUs assigned to a process. The flag is meaningless if a process is allowed to run on all CPUs (this is the default). It only has an explicit effect if the CPUs on which a process may run are limited.

**__GFP_THISNODE**: also only makes sense on NUMA systems. If the bit is set, then fallback to other nodes is not permitted, and mem

*3.*分配掩码的标志组合

#define GFP_ATOMIC  (__GFP_HIGH)

#define GFP_NOIO    (__GFP_WAIT)

#define GFP_NOFS    (__GFP_WAIT | __GFP_IO)

#define GFP_KERNEL  (__GFP_WAIT | __GFP_IO | __GFP_FS)

#define GFP_TEMPORARY   (__GFP_WAIT | __GFP_IO | __GFP_FS |  __GFP_RECLAIMABLE)

#define GFP_USER    (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL)

#define GFP_HIGHUSER   (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL | \
__GFP_HIGHMEM)

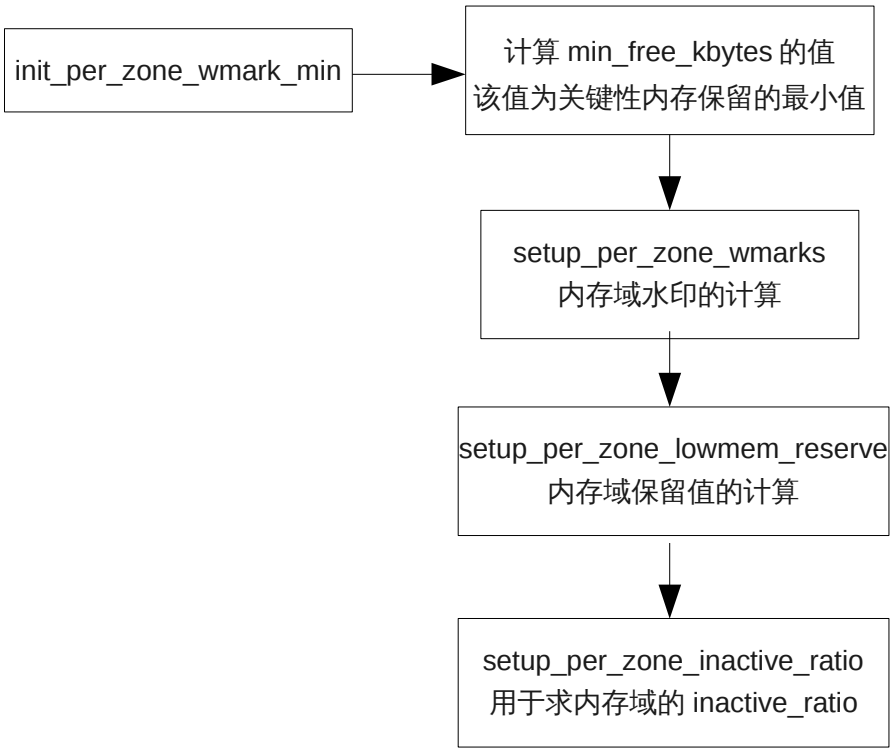
#define GFP_HIGHUSER_MOVABLE (__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HARDWALL | __GFP_HIGHMEM | __GFP_MOVABLE)

#define GFP_IOFS (__GFP_IO | __GFP_FS)

### *2.内存域水印(watermark)*

*1.*内存域水印的初始化

init_per_zone_wmark_min 函数由内核在启动期间调用，无需显示调用。




*2.*检查空闲内存是否满足内存水印

控制水印标记

#define ALLOC_WMARK_MIN WMARK_MIN

#define ALLOC_WMARK_LOW WMARK_LOW

#define ALLOC_WMARK_HIGH WMARK_HIGH

#define ALLOC_NO_WATERMARKS 0x04 /* don't check watermarks at all */

#define ALLOC_HARDER 0x10 /* try to alloc harder */

#define ALLOC_HIGH 0x20 /* __GFP_HIGH set */

#define ALLOC_CPUSET 0x40 /* check for correct cpuset */

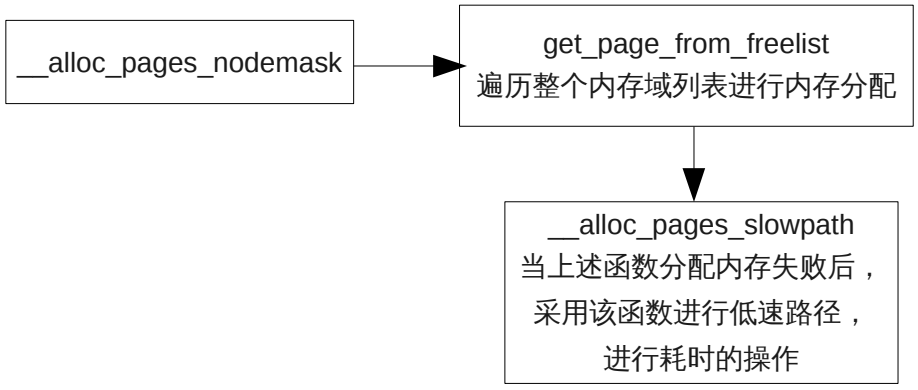
```
int zone_watermark_ok(struct zone *z, int order, unsigned long mark,
        int classzone_idx, int alloc_flags)
{
```

```
    long min = mark;
    long free_pages = zone_nr_free_pages(z) - (1 << order) + 1;
    int o;
    if (alloc_flags & ALLOC_HIGH)
        min -= min / 2;
    if (alloc_flags & ALLOC_HARDER)
        min -= min / 4;
      if (free_pages <= min + z->lowmem_reserve[classzone_idx])
        return 0;
    for (o = 0; o < order; o++) {
        free_pages -= z->free_area[o].nr_free << o;
          min >>= 1;  /* Require fewer higher order pages to be free */
     if (free_pages <= min)
            return 0;
 }
    return 1;
}
```

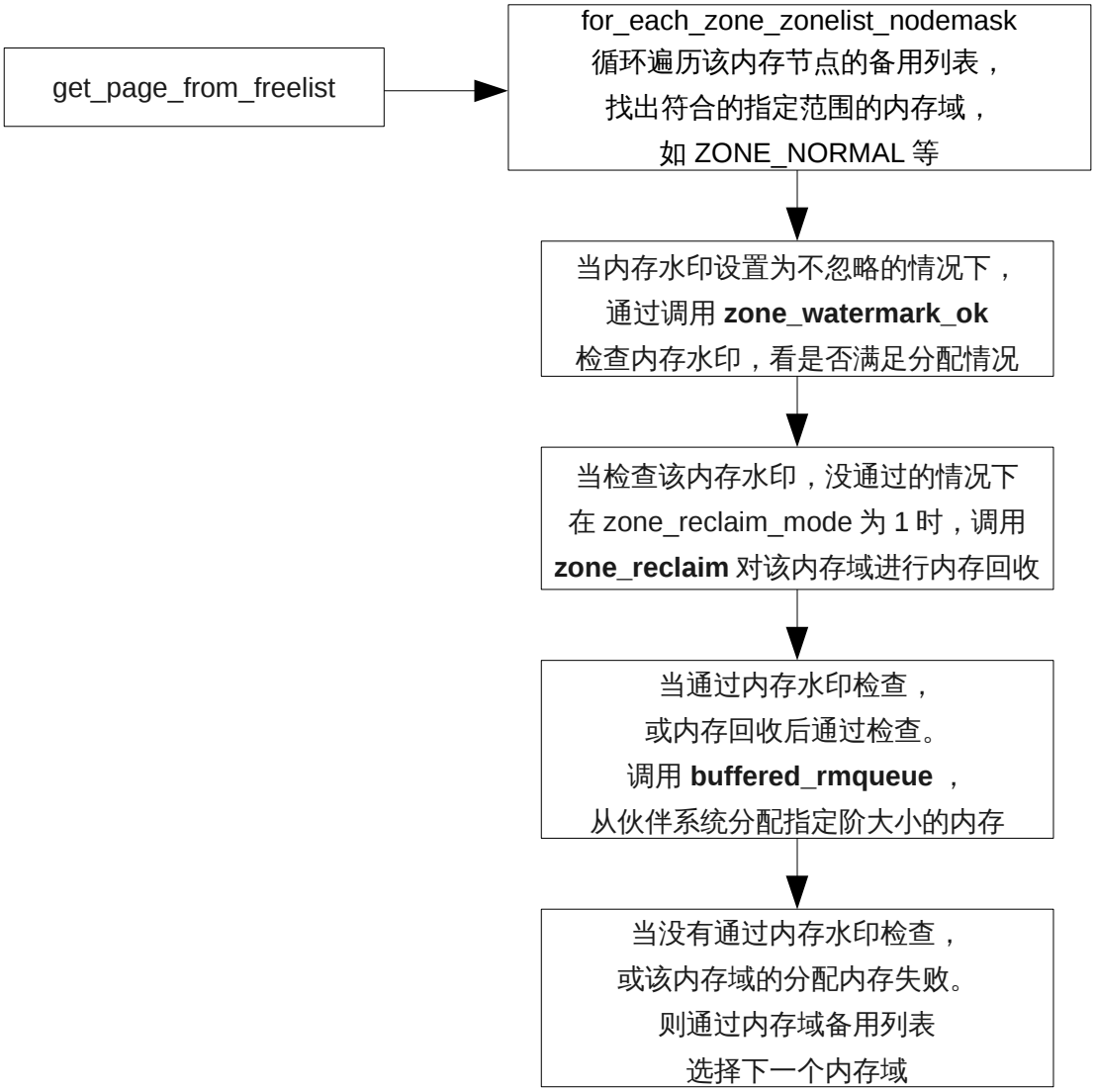
### *3.伙伴系统分配内存*

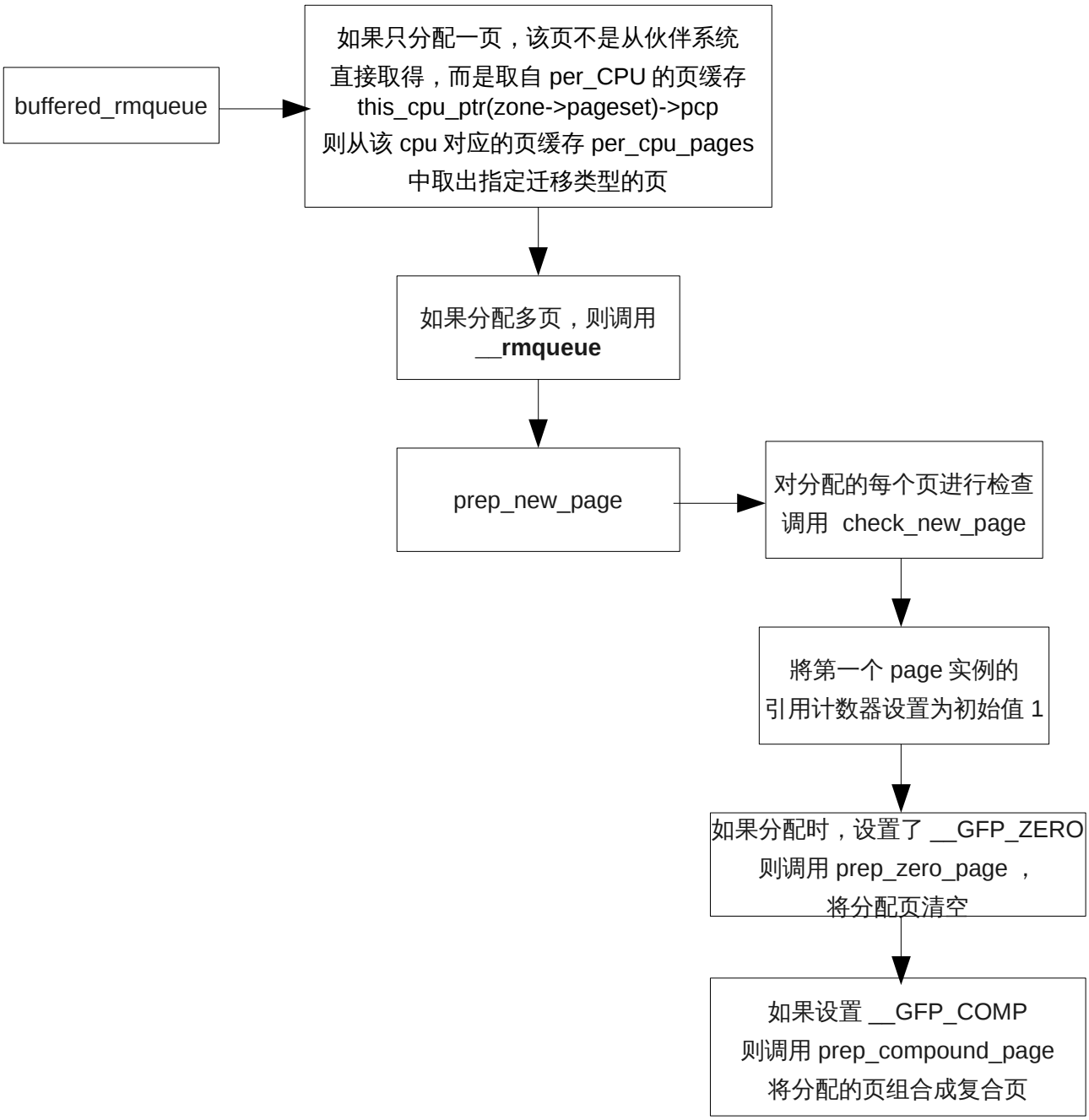
__alloc_pages_nodemask 函数是伙伴系统分配器的“心脏”。不同的伙伴系统的分配函数接口，最终都要追溯该函数。

mm/page_alloc.c

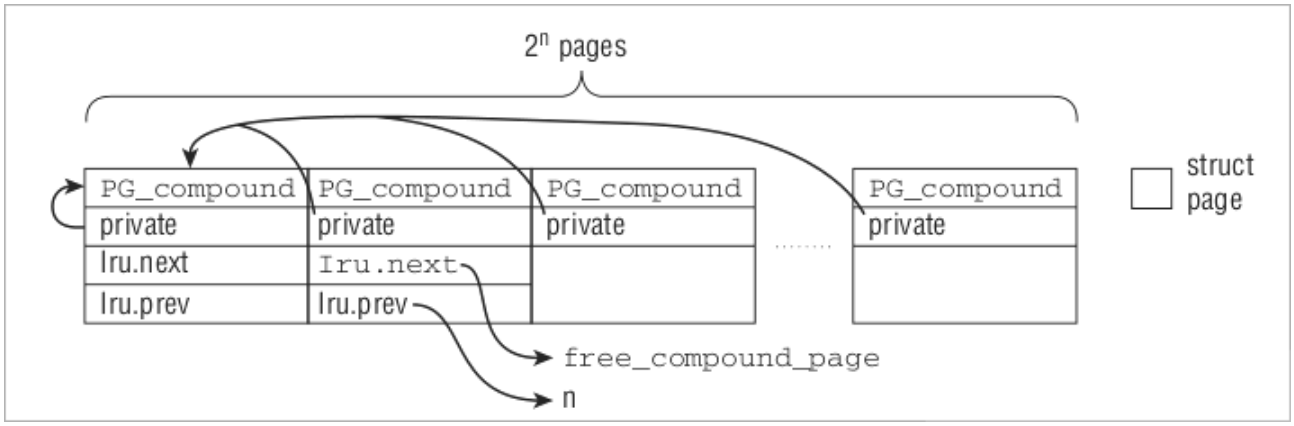


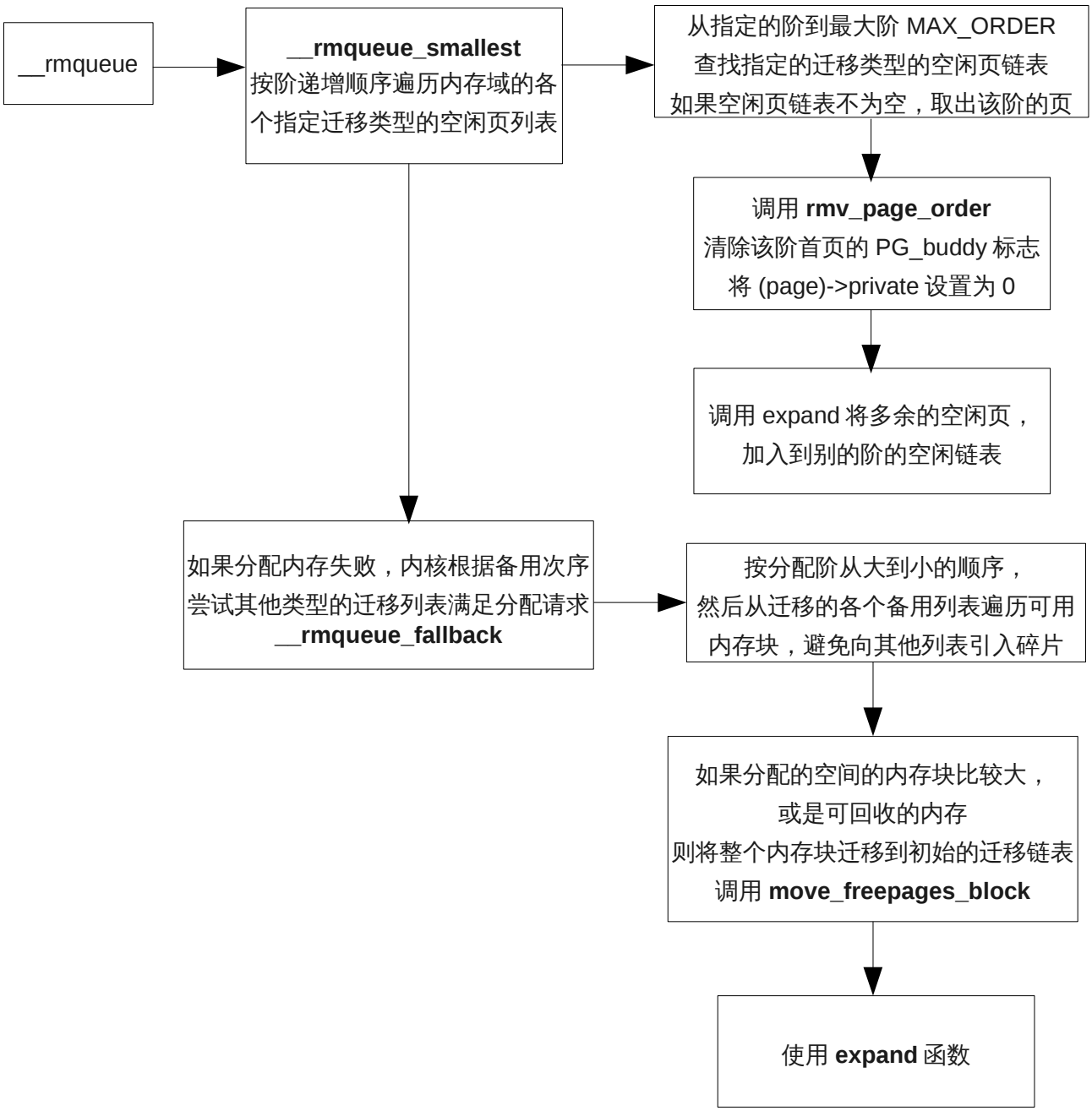

__alloc_pages_slowpath 将会在后面讲解。

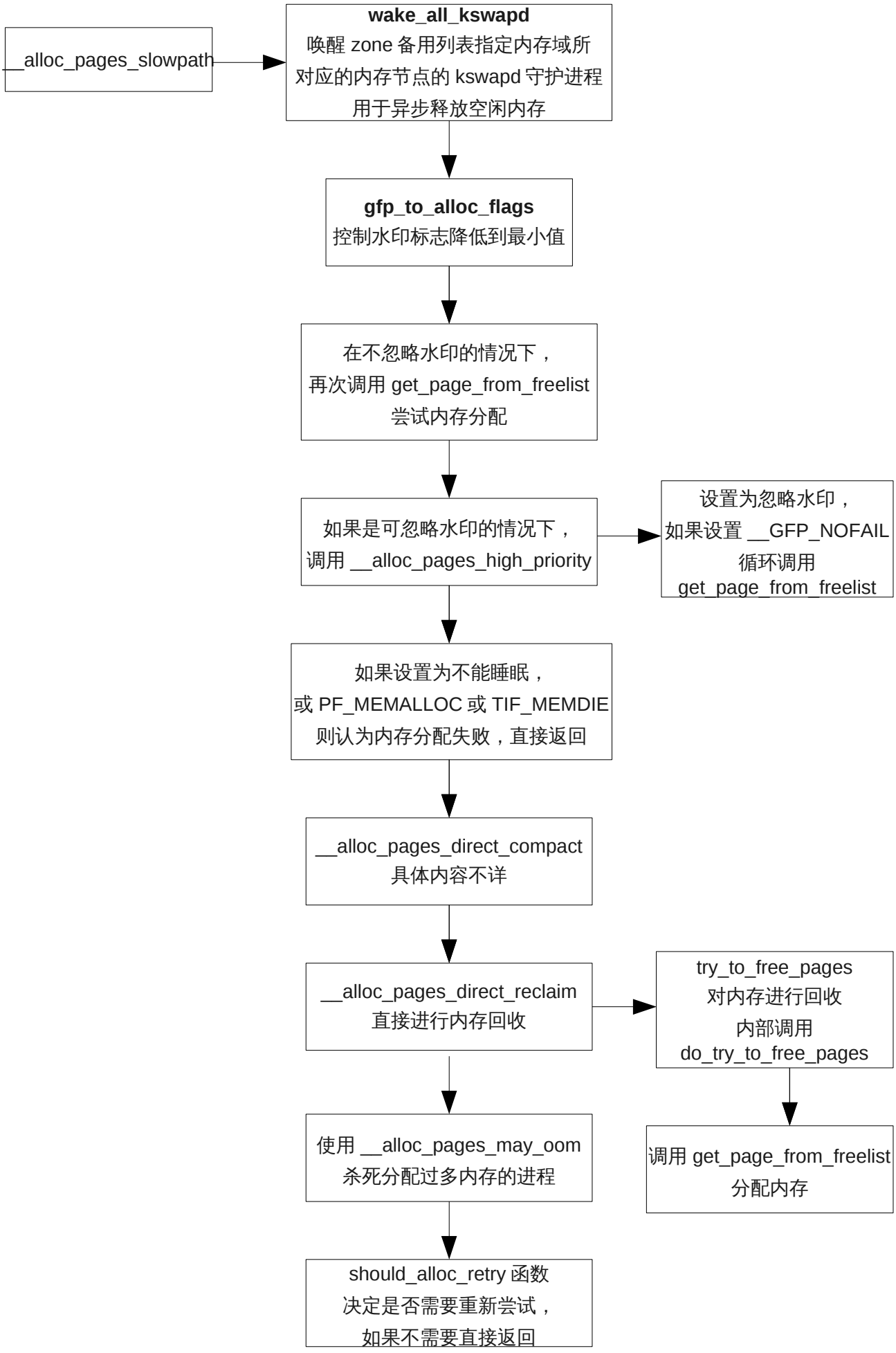

get_page_from_freelist
for_each_zone_zonelist_nodemask
循环遍历该内存节点的备用列表，
找出符合的指定范围的内存域，
如 ZONE_NORMAL 等
当内存水印设置为不忽略的情况下，
通过调用 zone_watermark_ok
检查内存水印，看是否满足分配情况
当检查该内存水印，没通过的情况下
在 zone_reclaim_mode 为 1 时，调用
zone_reclaim 对该内存域进行内存回收
当通过内存水印检查，
或内存回收后通过检查。
调用 buffered_rmqueue ，
从伙伴系统分配指定阶大小的内存
当没有通过内存水印检查，
或该内存域的分配内存失败。
则通过内存域备用列表
选择下一个内存域


buffered_rmqueue
如果只分配一页，该页不是从伙伴系统直接取得，而是取自 per_CPU 的页缓存
this_cpu_ptr(zone->pageset)->pcp
则从该 cpu 对应的页缓存 per_cpu_pages 中取出指定迁移类型的页
如果分配多页，则调用 __rmqueue
prep_new_page
对分配的每个页进行检查
调用 check_new_page
将第一个 page 实例的引用计数器设置为初始值 1
如果分配时，设置了 __GFP_ZERO
则调用 prep_zero_page ，将分配页清空
如果设置 __GFP_COMP
则调用 prep_compound_page
将分配的页组合成复合页

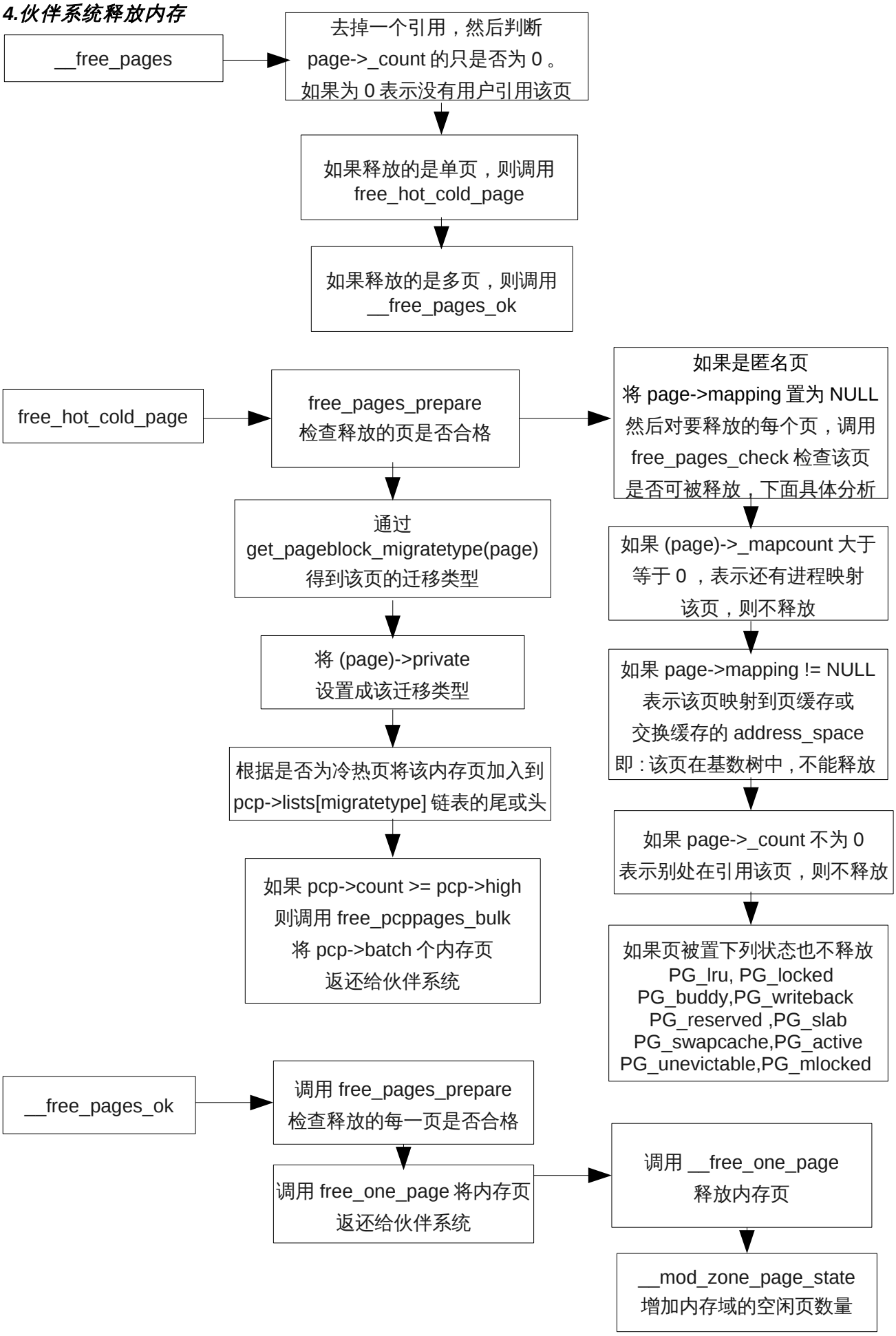
复合页的结构：

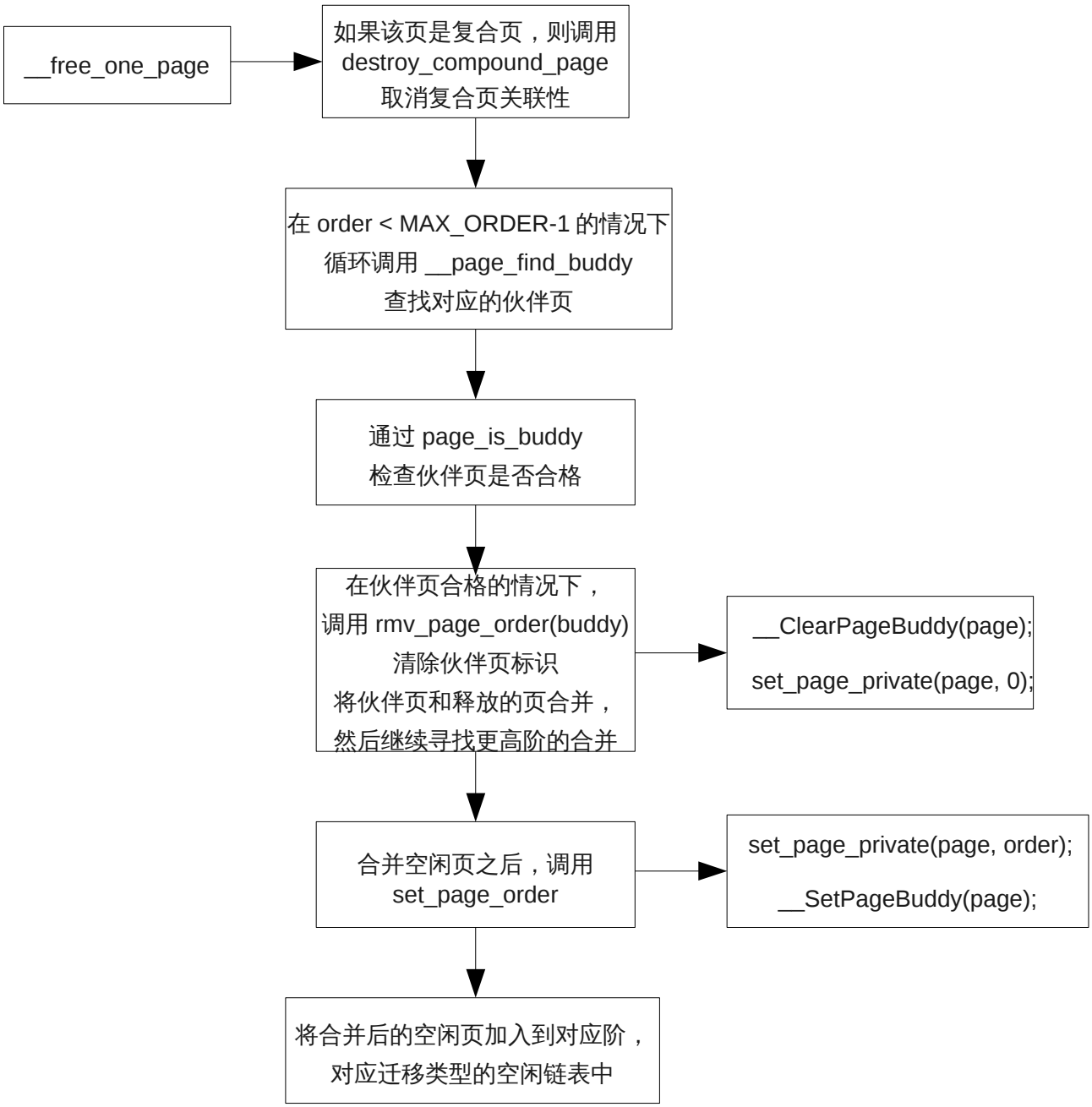

2^n pages
PG_compound
private
lru.next
lru.prev
PG_compound
private
lru.next
lru.prev
PG_compound
private
PG_compound
private
struct page
free_compound_page
n


__rmqueue
__rmqueue_smallest
按阶递增顺序遍历内存域的各个指定迁移类型的空闲页列表
从指定的阶到最大阶 MAX_ORDER
查找指定的迁移类型的空闲页链表
如果空闲页链表不为空，取出该阶的页
调用 rmv_page_order
清除该阶首页的 PG_buddy 标志
将 (page)->private 设置为 0
调用 expand 将多余的空闲页，
加入到别的阶的空闲链表
如果分配内存失败，内核根据备用次序
尝试其他类型的迁移列表满足分配请求
__rmqueue_fallback
按分配阶从大到小的顺序，
然后从迁移的各个备用列表遍历可用
内存块，避免向其他列表引入碎片
如果分配的空间的内存块比较大，
或是可回收的内存
则将整个内存块迁移到初始的迁移链表
调用 move_freepages_block
使用 expand 函数


__alloc_pages_slowpath
wake_all_kswapd
唤醒 zone 备用列表指定内存域所
对应的内存节点的 kswapd 守护进程
用于异步释放空闲内存
gfp_to_alloc_flags
控制水印标志降低到最小值
在不忽略水印的情况下，
再次调用 get_page_from_freelist
尝试内存分配
如果是可忽略水印的情况下，
调用 __alloc_pages_high_priority
设置为忽略水印，
如果设置 __GFP_NOFAIL
循环调用
get_page_from_freelist
如果设置为不能睡眠，
或 PF_MEMALLOC 或 TIF_MEMDIE
则认为内存分配失败，直接返回
__alloc_pages_direct_compact
具体内容不详
__alloc_pages_direct_reclaim
直接进行内存回收
try_to_free_pages
对内存进行回收
内部调用
do_try_to_free_pages
调用 get_page_from_freelist
分配内存
使用 __alloc_pages_may_oom
杀死分配过多内存的进程
should_alloc_retry 函数
决定是否需要重新尝试，
如果不需要直接返回

### *4.伙伴系统释放内存*


__free_pages
去掉一个引用，然后判断
page->_count 的只是否为 0 。
如果为 0 表示没有用户引用该页
如果释放的是单页，则调用
free_hot_cold_page
如果释放的是多页，则调用
__free_pages_ok
free_hot_cold_page
free_pages_prepare
检查释放的页是否合格
如果是匿名页
将 page->mapping 置为 NULL
然后对要释放的每个页，调用
free_pages_check 检查该页
是否可被释放，下面具体分析
通过
get_pageblock_migratetype(page)
得到该页的迁移类型
如果 (page)->_mapcount 大于
等于 0 ，表示还有进程映射
该页，则不释放
将 (page)->private
设置成该迁移类型
如果 page->mapping != NULL
表示该页映射到页缓存或
交换缓存的 address_space
即 : 该页在基数树中，不能释放
根据是否为冷热页将该内存页加入到
pcp->lists[migratetype] 链表的尾或头
如果 page->_count 不为 0
表示别处在引用该页，则不释放
如果 pcp->count >= pcp->high
则调用 free_pcppages_bulk
将 pcp->batch 个内存页
返还给伙伴系统
如果页被置下列状态也不释放
PG_lru, PG_locked
PG_buddy,PG_writeback
PG_reserved ,PG_slab
PG_swapcache,PG_active
PG_unevictable,PG_mlocked
__free_pages_ok
调用 free_pages_prepare
检查释放的每一页是否合格
调用 free_one_page 将内存页
返还给伙伴系统
调用 __free_one_page
释放内存页
__mod_zone_page_state
增加内存域的空闲页数量




查找对应伙伴的示意图：

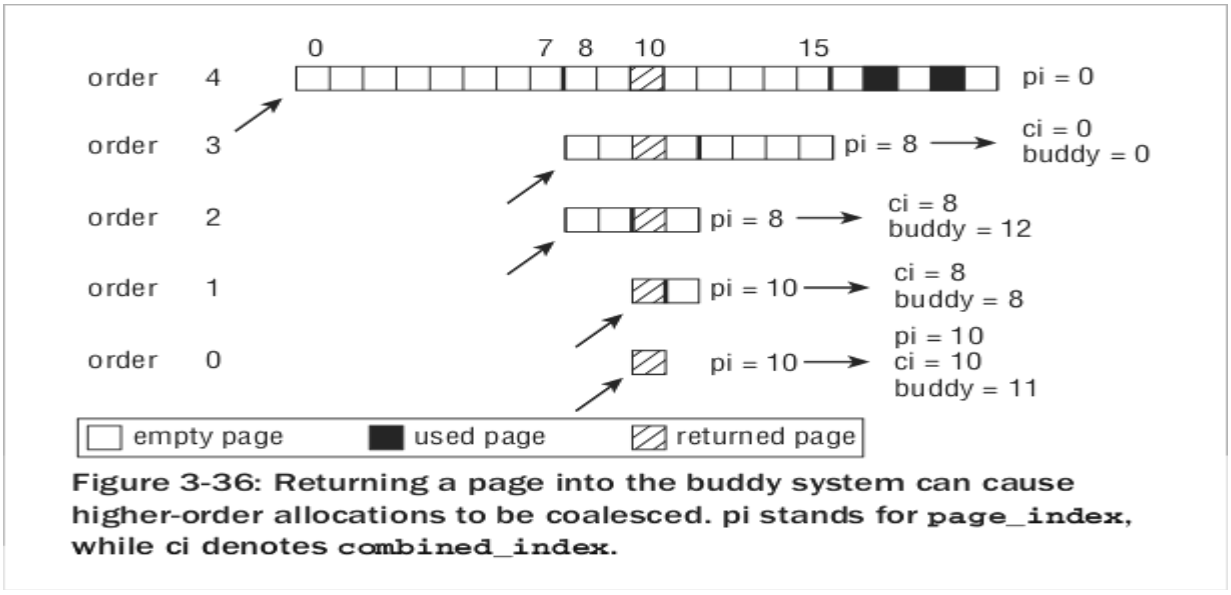
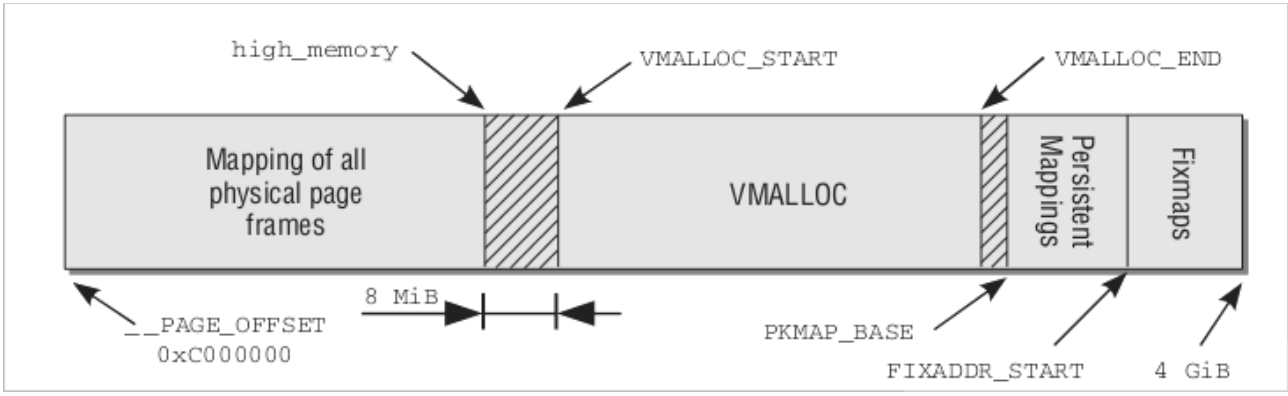



**Figure 3-36: Returning a page into the buddy system can cause higher-order allocations to be coalesced. pi stands for `page_index`, while ci denotes `combined_index`.**

## *2.内核地址空间*

IA-32：




## 1.常规映射地址空间

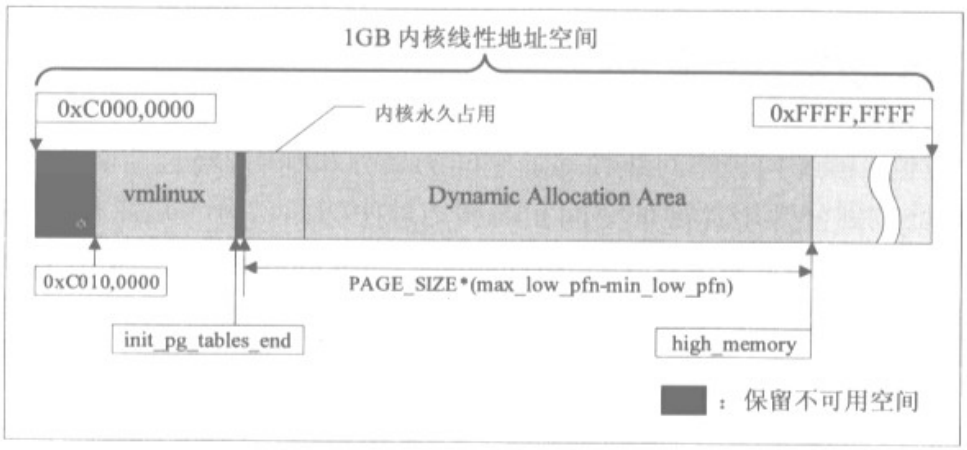


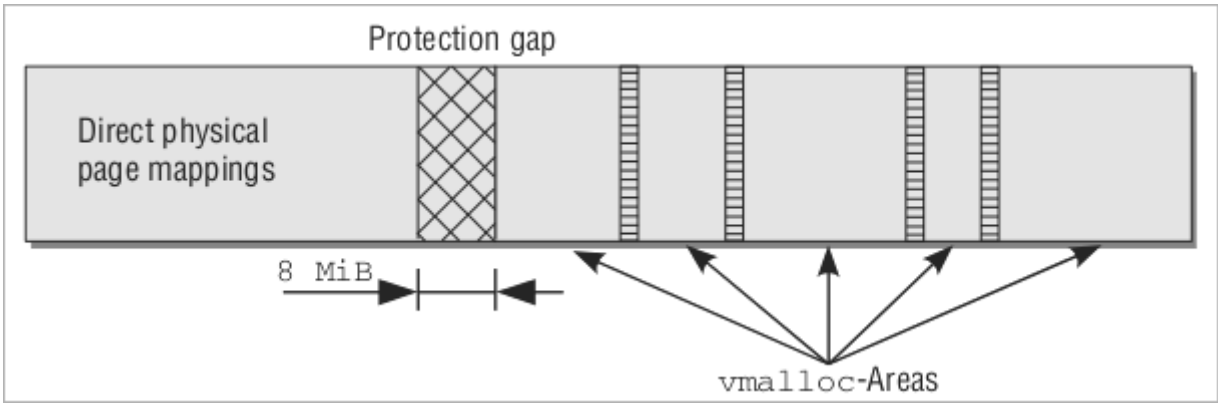

图 3.10　常规映射地址空间

- 内核可执行镜像部分，保存了内核镜像（vmLinux）指令段和数据段，不可回收和换出。
- 内核永久占用区部分，主要在系统初始化过程中由 Boot Memory Allocator 分配给内核，它存放了系统关键的核心数据结构，不可回收和换出。
- 系统动态分配低内存，由伙伴算法（Buddy Algorithm）进行管理，可以动态地分配给内核、用户进程使用，可以回收和换出。

虽然系统低物理内存页框通过内核页表被映射到了内核线性地址空间中，但是这并不意味着内核获得了这些物理页框的使用权；对比用户进程，只要有页表项建立了用户地址空间虚拟页面到物理页框的映射，用户进程就拥有对相应物理页框的使用权。一个物理页框到底是否空闲是由伙伴算法维护的数据结构决定的，在内核执行路访问和使用一个物理页框时，需要首先调用伙伴算法的页框分配接口函数、获取物理页框，然后才能使用。同样，用户进程也要通过该过程，才能访问和使用一个物理页框。

在系统初始化过程中，完全可以不构建多余的内核页表项，而是在运行过程中，按需申请、释放物理页框和创建、销毁对应的页表项。但是这一过程会修改内核页表，为了保证页表的一致性，需要刷新系统 TLB，这对系统性能会有严重危害。在 linux 内核的实现中，采用预先创建内核页表的方法避免 TLB 刷新带来的危害。

## 2.VMALLOC 分配内存




***1.数据结构***

描述 vmalloc 分配的子区域

```
struct vm_struct {
    struct vm_struct    *next;
    void            *addr;
    unsigned long       size;
    unsigned long       flags;
    struct page     **pages;
    unsigned int        nr_pages;
    phys_addr_t     phys_addr;
    void            *caller;
};
```

部分元素解释：

(1)addr:定义了分配的子区域在虚拟地址空间的起始地址。

(2)phys_addr:仅当 ioremap 映射了由物理地址描述的物理内存区域时才需要。

(3)pages:指向 page 指针的数组。

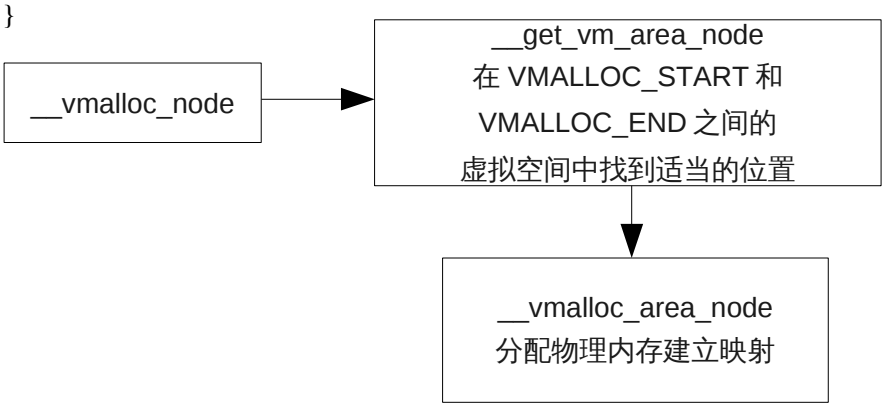
(4)flags:用于指定内存区类型：

```
#define VM_IOREMAP  0x00000001  /* ioremap() and friends */
#define VM_ALLOC    0x00000002  /* vmalloc() */
#define VM_MAP      0x00000004  /* vmap()ed pages */
#define VM_USERMAP  0x00000008  /* suitable for remap_vmalloc_range */
#define VM_VPAGES   0x00000010  /* buffer for pages was vmalloc'ed */
```

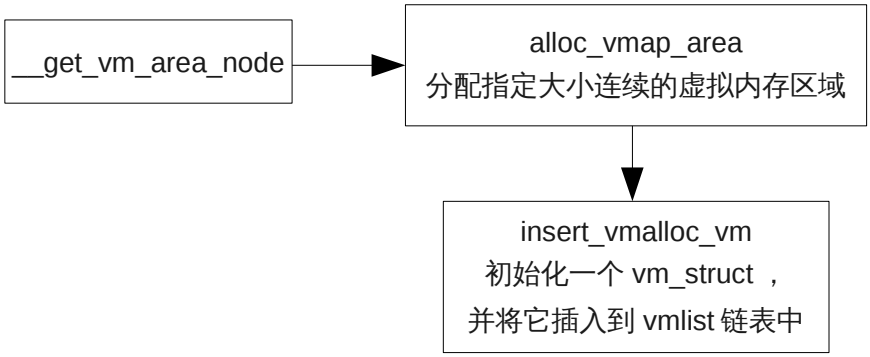
### *2.分配内存函数*

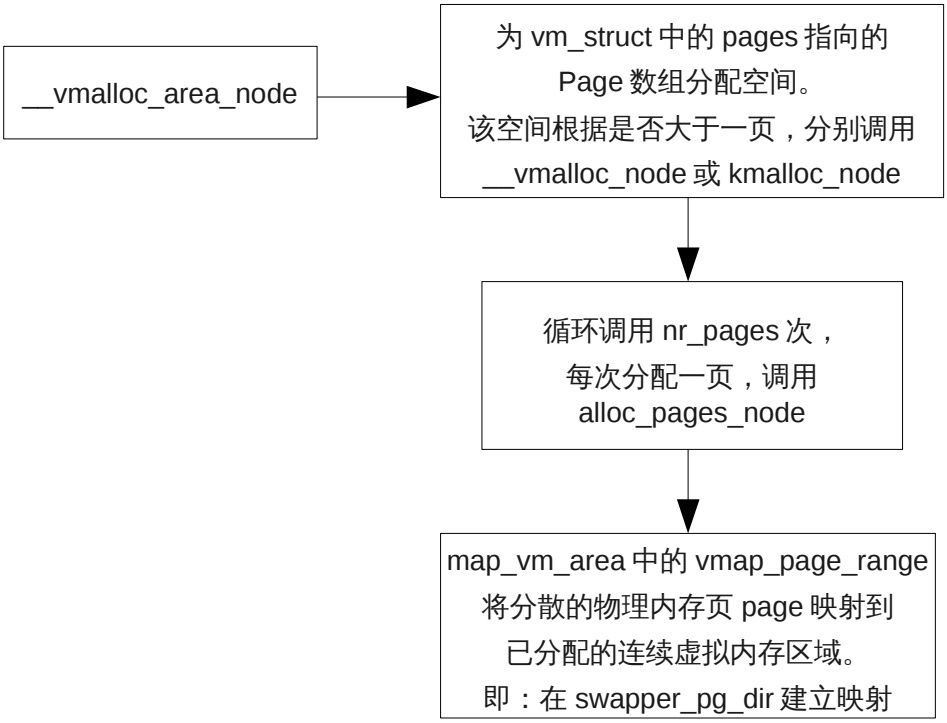
vmalloc.c

```
void *vmalloc(unsigned long size)
{
    return __vmalloc_node(size, 1, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL,
                                                        -1, __builtin_return_address(0));
}
```



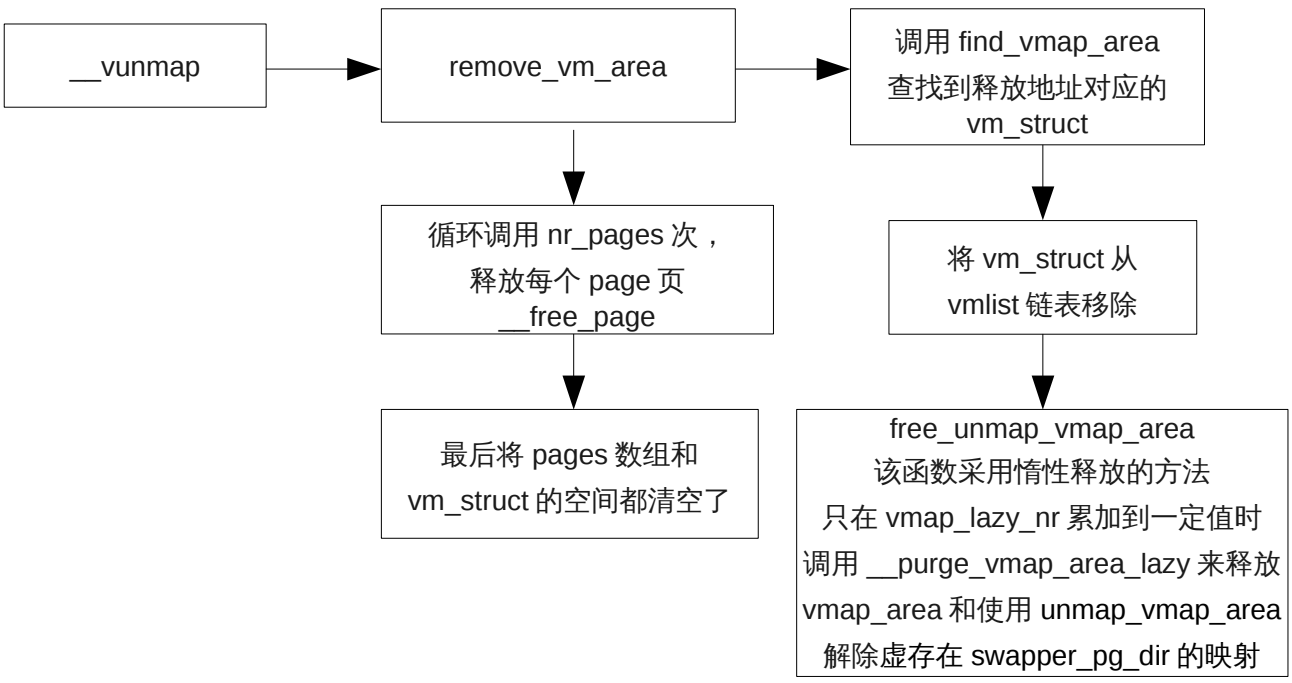

```
#define VMALLOC_START   ((unsigned long)high_memory + VMALLOC_OFFSET)
#ifdef CONFIG_HIGHMEM
# define VMALLOC_END    (PKMAP_BASE - 2 * PAGE_SIZE)
#else
# define VMALLOC_END    (FIXADDR_START - 2 * PAGE_SIZE)
#endif
```






vmalloc 将 gfp_mask 设置为 GFP_KERNEL | __GFP_HIGHMEM，通过该参数指示系统尽可能从 ZONE_HIGHMEM 内存域分配页帧。低端内存域的页帧比较宝贵。

***3.释放内存函数***



# 3.固定映射地址空间

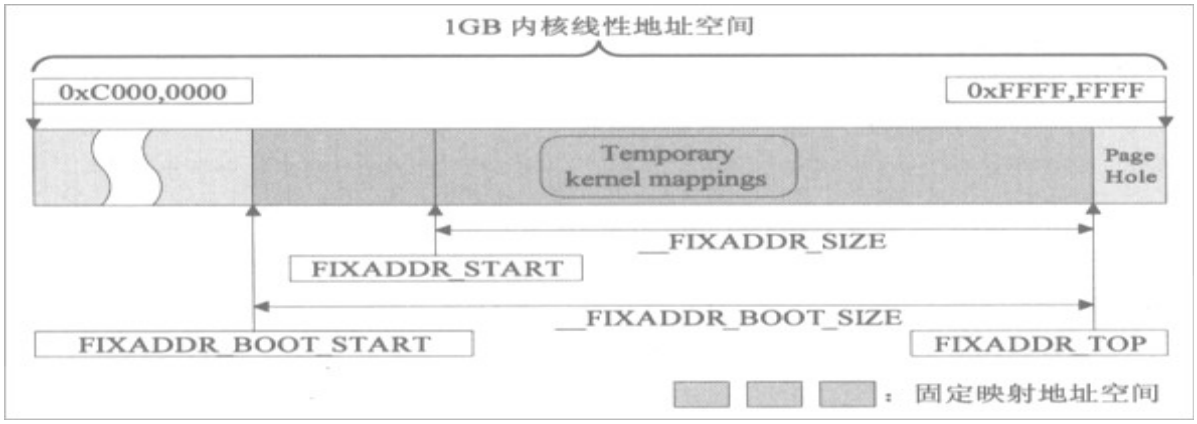
### *1.数据结构：*

```
enum fixed_addresses {
.....
#ifdef CONFIG_X86_32
    FIX_KMAP_BEGIN, /* reserved pte's for temporary kernel mappings */
    FIX_KMAP_END = FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1,
.....
__end_of_permanent_fixed_addresses,
    FIX_BTMAP_BEGIN = FIX_BTMAP_END + TOTAL_FIX_BTMAPS - 1,
.....
 __end_of_fixed_addresses
}
```

上面的每一个枚举元素对应于一个预留的虚拟页面，一对位置连续但取值不连续的枚举元素对应于一组预留的虚拟页面。这些预留的虚拟页面有特定的用途。比如： FIX_KMAP_BEGIN 和 FIX_KMAP_END 这对枚举元素对应的一组虚拟页面空间用于实现临时内核映射。

### *2.枚举值和虚拟地址的转换*

```
unsigned long __FIXADDR_TOP = 0xfffff000;
#define FIXADDR_TOP ((unsigned long)__FIXADDR_TOP)

#define __fix_to_virt(x)    (FIXADDR_TOP - ((x) << PAGE_SHIFT))
#define __virt_to_fix(x)    ((FIXADDR_TOP - ((x)&PAGE_MASK)) >> PAGE_SHIFT)

#define FIXADDR_SIZE    (__end_of_permanent_fixed_addresses << PAGE_SHIFT)
#define FIXADDR_BOOT_SIZE   (__end_of_fixed_addresses << PAGE_SHIFT)
#define FIXADDR_START       (FIXADDR_TOP - FIXADDR_SIZE)
#define FIXADDR_BOOT_START  (FIXADDR_TOP - FIXADDR_BOOT_SIZE)
```



### *3.设置、解除固定映射*

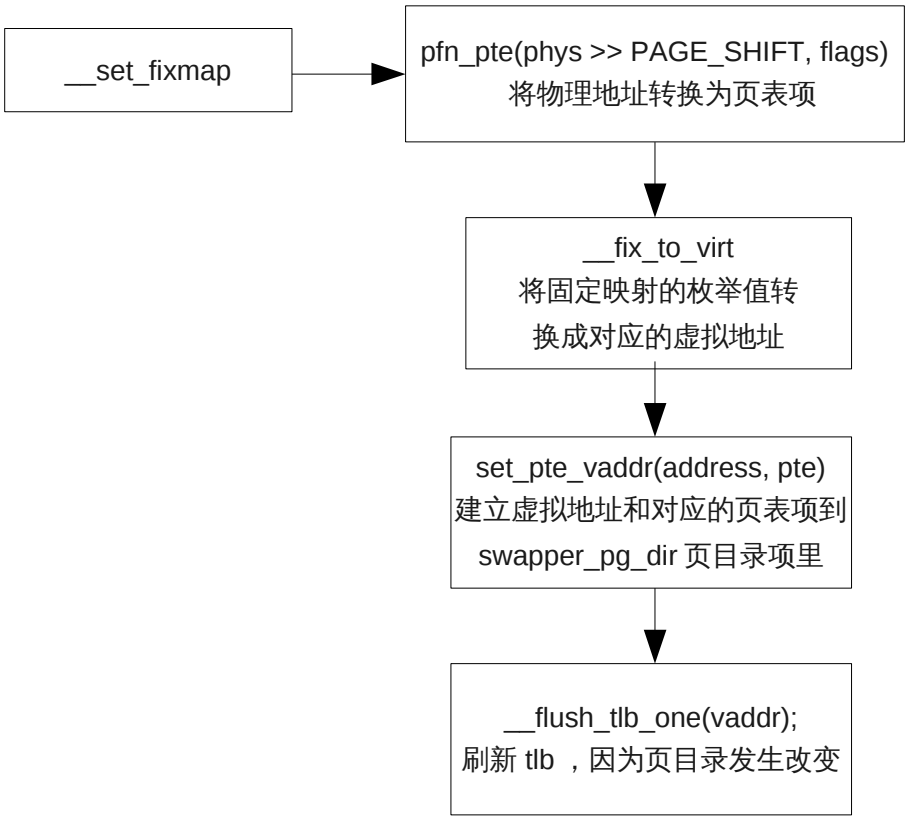
#define set_fixmap(idx, phys)  \

  __set_fixmap(idx, phys, PAGE_KERNEL)

#define clear_fixmap(idx)  \

  __set_fixmap(idx, 0, __pgprot(0))

arch/x86/include/asm/fixmap.h




## 4.长久内核映射空间

### *1.地址空间范围*

该地址空间是否存在依赖于内核选项 CONFIG_HIGHMEM 通过该保留空间，可以在系统高物理内存页框和内核线性地址之间建立比较比较长久的映射。

长久映射地址空间的起始地址为 PKMAP_BASE，大小为 LAST_PKMAP 个虚拟页面。

#define PKMAP_BASE ((FIXADDR_BOOT_START - PAGE_SIZE * (LAST_PKMAP + 1)) & PMD_MASK)

#ifdef CONFIG_X86_PAE

#define LAST_PKMAP 512

#else

#define LAST_PKMAP 1024

#endif

通过长久内核映射空间，内核最多可以同时使用 LAST_PKMAP*PAGE_SIZE（4MB）的高端物理内存。

### *2.虚拟页面的引用计数*

对于长久映射空间的每个虚拟页面，内核为之设置一个整数变量来记录该虚拟页面的状态。该变量为长久映射虚拟页面的引用计数。所有引用计数依次保存在 static int pkmap_count[LAST_PKMAP]中。

根据引用计数的取值的不同，对应的虚拟页面分为 3 种状态：

- 取值为 0：表明该元素对应的虚拟页面没有映射到任何高物理内存页框，处于可用状态，可以将该虚拟页面通过对应的内核页表项映射到任意高物理内存页框。
- 取值为 1：表明该元素对应的虚拟页面映射到了某一高物理内存页框，但是没有内核组件依赖于该映射关系、访问和使用对应的物理页框。这种类型的虚拟页面只有在解除当前的映射关系后，才可以建立新的映射、被再次使用。

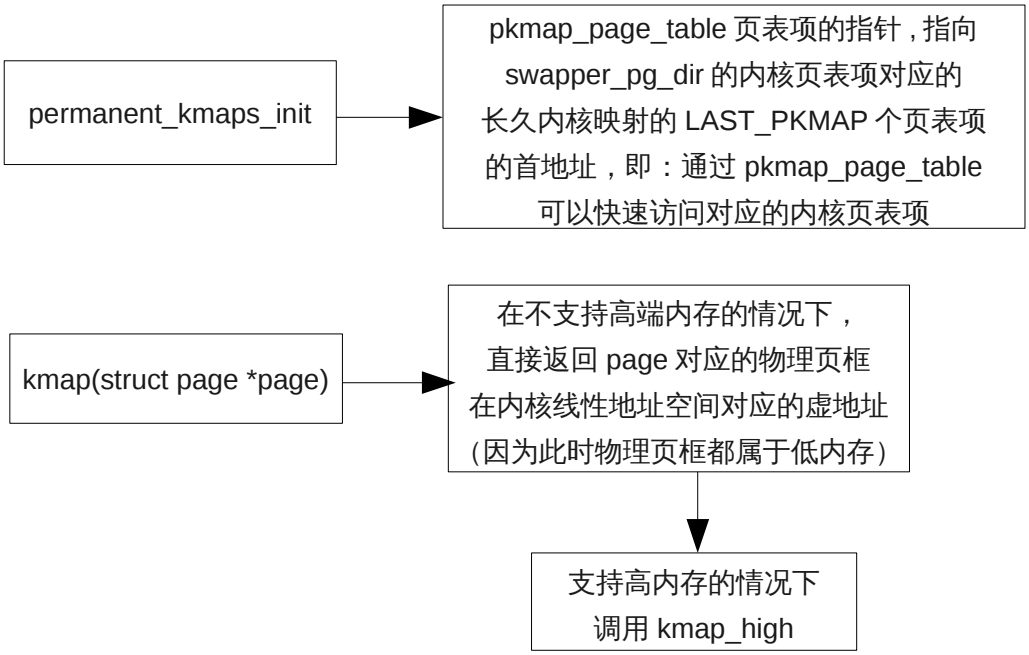
解除映射关系，需要刷新对应页表项的 TLB。刷新 TLB 对系统性能有很大的危害，为了避免对 TLB 的频繁刷新，在长久内核映射的设计中，只有当长久内核映射空间中所有的虚拟页面都被使用了一遍、没有可用虚拟页面时，才调用函数 flush_all_zero_pkmaps()将虚拟页面的引用计数设置为 0，然后重置对应页表项，解除映射关系，最后刷新系统 TLB。这样就最大限度地保证了系统性能不受危害。

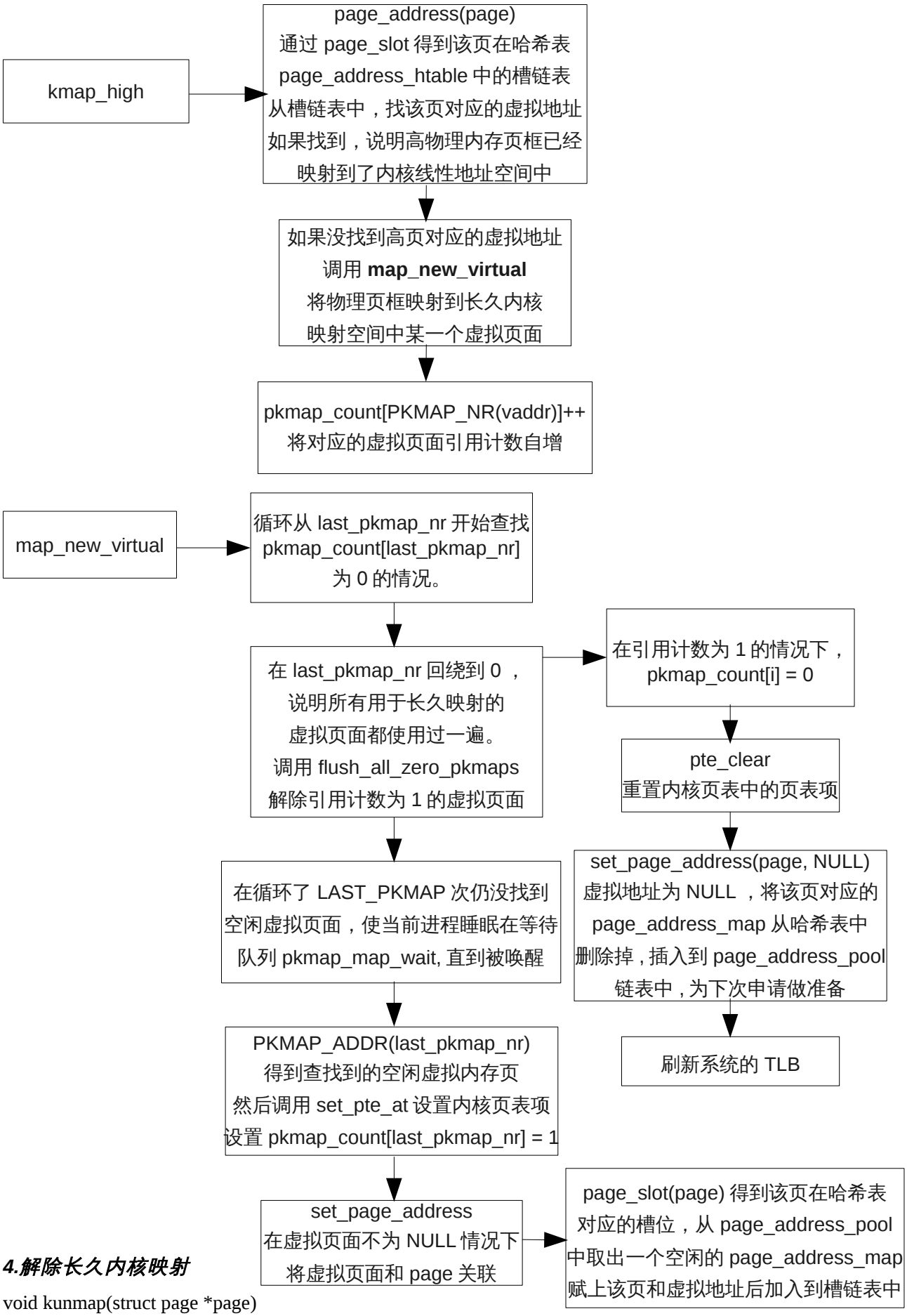
- 取值为 N（N 大于 1）：表明该元素对应的虚拟页面映射到了一个高物理内存页框，且目前有 N−1 个内核组件依赖于该映射关系，通过该映射关系访问和使用对应的物理页框。

### *3.设置长久内核映射*

arch/x86/mm/highmem_32.c

kmap 函数用于将参数 page 对应高物理内存页框映射到长久内核映射空间中的某一虚拟页面中，并返回该虚拟页面在内核线性地址空间中的地址。



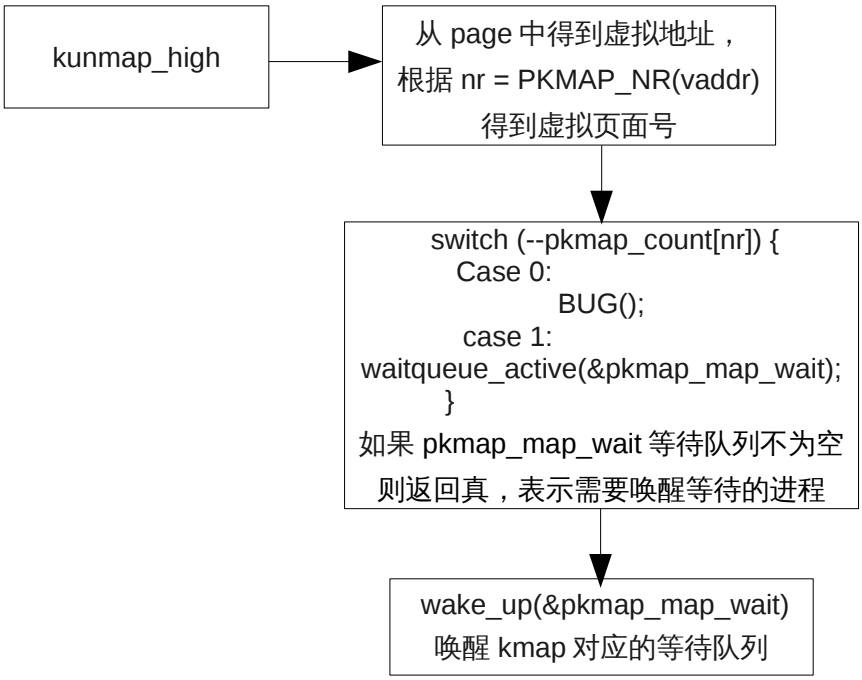



***4.解除长久内核映射***

```
void kunmap(struct page *page)
{
```

```
    if (in_interrupt())
        BUG();
    if (!PageHighMem(page))
        return;
    kunmap_high(page);
}
```

长久内核映射的过程中，可能因阻塞而睡眠，而中断过程不允许睡眠。




## 5.临时内核映射

### *1.数据结构*

临时内核映射是固定映射空间的一部份。用于为内核态不可睡眠执行路径使用高物理内存页提供支持。

```
enum fixed_addresses {
.....
  FIX_KMAP_BEGIN, /* reserved pte's for temporary kernel mappings */
    FIX_KMAP_END = FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1,
.....
}
```

枚举元素包括了临时内核映射空间中对应的虚拟页面号

```
enum km_type {
```

```
KMAP_D(0)   KM_BOUNCE_READ,
KMAP_D(1)   KM_SKB_SUNRPC_DATA,
KMAP_D(2)   KM_SKB_DATA_SOFTIRQ,
KMAP_D(3)   KM_USER0,
KMAP_D(4)   KM_USER1,
KMAP_D(5)   KM_BIO_SRC_IRQ,
KMAP_D(6)   KM_BIO_DST_IRQ,
KMAP_D(7)   KM_PTE0,
KMAP_D(8)   KM_PTE1,
KMAP_D(9)   KM_IRQ0,
KMAP_D(10)  KM_IRQ1,
KMAP_D(11)  KM_SOFTIRQ0,
KMAP_D(12)  KM_SOFTIRQ1,
KMAP_D(13)  KM_SYNC_ICACHE,
KMAP_D(14)  KM_SYNC_DCACHE,
/* UML specific, for copy_*_user - used in do_op_one_page */
KMAP_D(15)  KM_UML_USERCOPY,
KMAP_D(16)  KM_IRQ_PTE,
KMAP_D(17)  KM_NMI,
KMAP_D(18)  KM_NMI_PTE,
KMAP_D(19)  KM_KDB,
/*
 * Remember to update debug_kmap_atomic() when adding new kmap types!
 */
KMAP_D(20)  KM_TYPE_NR
};
```

### *2.设置临时内核映射*

arch/x86/mm/highmem_32.c

kmap_atomic(struct page *page, enum km_type type) 的具体实现，大致如下：

```
    pagefault_disable(); // 禁止内核抢占
    idx = type + KM_TYPE_NR*smp_processor_id();
    vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);
    set_pte(kmap_pte-idx, mk_pte(page, prot));
    return (void *)vaddr;
```

从上面的分析过程可以看出，建立临时内核映射的过程不会被阻塞，也就避免了采用此机制访问高物理内存页框的内核执行路径被阻塞。但是对普通的内核执行路径，绝对不能利用该机制来访问和使用高物理内存页框。

原因很简单，该机制不是为普通内核执行路径设计的。假如一个普通的内核执行路径利用该机制来访问和使用高物理内存页框，在该普通内核执行路径已经获得一个临时内核映射虚拟页面（此时禁止内核态抢占），而随后因某些操作而阻塞、需要睡眠时（睡眠时会激活调度器）会导致内核状态紊乱、系统停机。

## *3.高速缓存*

高速缓冲存储器的容量一般只有主存储器的几百分之一，但它的存取速度能与中央处理器相匹配。根据程序局部性原理，正在使用的主存储器某一单元邻近的那些单元将被用到的可能性很大。因而，当中央处理器存取主存储器某一单元时，计算机硬件就自动地将包括该单元在内的那一组单元内容调入高速缓冲存储器，中央处理 器即将存取的主存储器单元很可能就在刚刚调入到高速缓冲存储器的那一组单元内。于是，中央处理器就可以直接对高速缓冲存储器进行存取。在整个处理过程中， 如果中央处理器绝大多数存取主存储器的操作能为存取高速缓冲存储器所代替，计算机系统处理速度就能显著提高

高速缓冲存储器通常由高速存储器、联想存储器、替换逻辑电路和相应的控制线路组成。在有高速缓冲存储器的计算机系统中，中央处理器存取主存储器的地址划分为行号、列号和组内地址三个字段。于是，主存储器就在逻辑上划分为若干行；每行划分为若干的存储单元组；每组包含几个或几十个字。高速存储器也相应地划分为行和列的存储单元组。二者的列数相同，组的大小也相同，但高速存储器的行数却比主存储器的行数少得多。

联想存储器用于地址联想，有与高速存储器相同行数和列数的存储单元。当主存储器某一列某一行存储单元组调入高速存储器同一列某一空着的存储单元组时，与联想存储器对应位置的存储单元就记录调入的存储单元组在主存储器中的行号。

当中央处理器存取主存储器时，硬件首先自动对存取地址的列号字段进行译码，以便将联想存储器该列的全部行号与存取主存储器地址的行号字段进行比较：若有相同的，表明要存取的主存储器单元已在高速存储器中，称为命中，硬件就将存取主存储器的地址映射为高速存储器的地址并执行存取操作；若都不相同，表明该单元不在高速存储器中，称为脱靶，硬件将执行存取主存储器操作并自动将该单元所在的那一主存储器单元组调入高速存储器相同列中空着的存储单元组中，同时将该组在主存储器中的行号存入联想存储器对应位置的单元内。

当出现脱靶而高速存储器对应列中没有空的位置时，便淘汰该列中的某一组以腾出位置存放新调入的组，这称为替换。确定替换的规则叫替换算法，常用的替换算法有:最近最少使用法（LRU）、先进先出法（FIFO）和随机法（RAND）等。替换逻辑电路就是执行这个功能的。另外，当执行写主存储器操作时，为保持主存储器和高速存储器内容的一致性，对命中和脱靶须分别处理：

①写操作命中时，可采用写直达法（即同时写入主存储器和高速存储器）或写回法（即只写入高速存储器并标记该组修改过。淘汰该组时须将内容写回主存储器）；

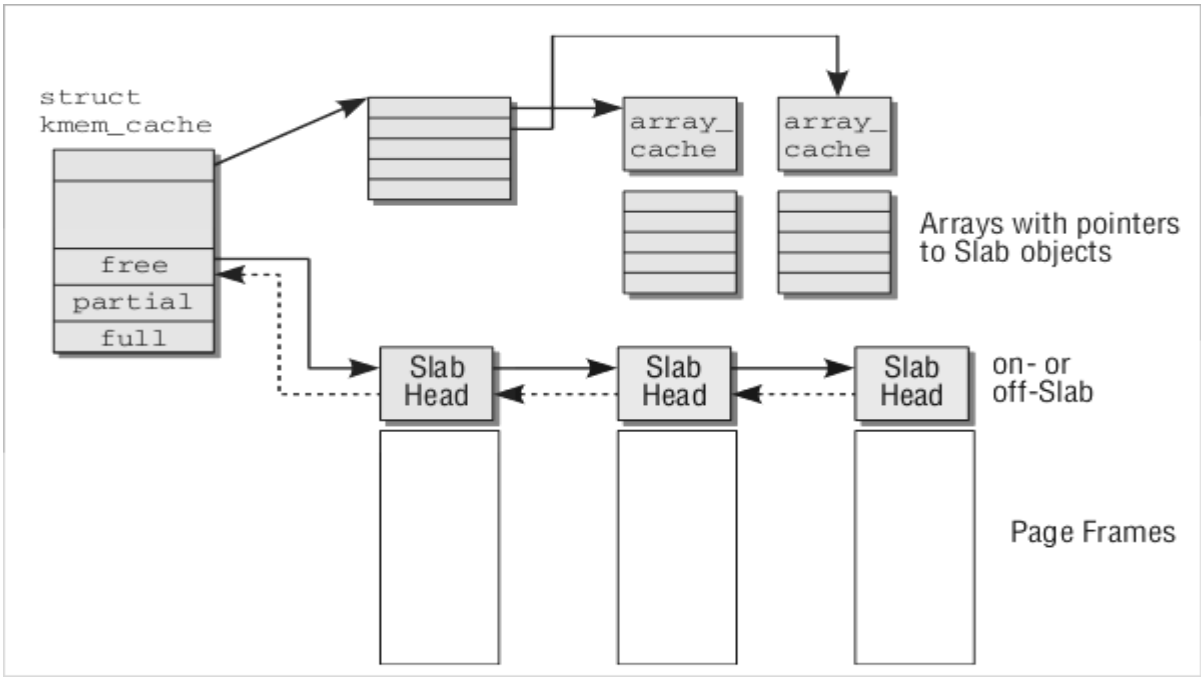
②写操作脱靶时，可采用写分配法（即写入主存储器并将该组调入高速存储器）或写不分配法（即只 写入主存储器但不将该组调入高速存储器）。高速缓冲存储器的性能常用命中率来衡量。影响命中率的因素是高速存储器的容量、存储单元组的大小、组数多少、地 址联想比较方法、替换算法、写操作处理方法和程序特性等。

## *4.slab 内存分配*

## 1.概述

每个缓存只负责一种对象类型(如：mm_struct , 具体可使用 cat /proc/slabinfo 查看)，或提供一般性的缓存区。系统中所有缓存都保存在一个双向链表里，让内核有机会遍历所有的缓存。

slab 的结构：



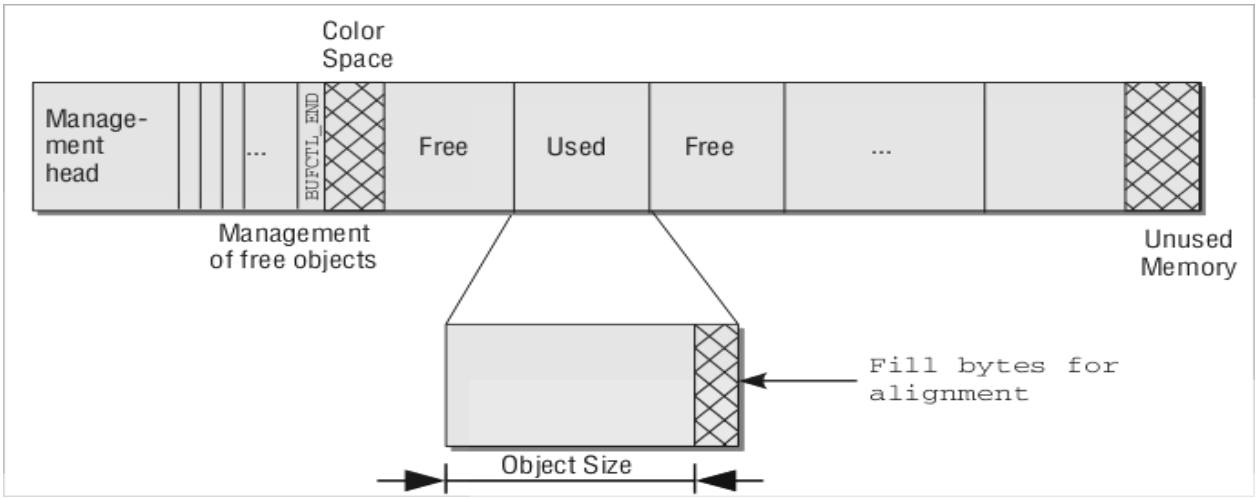

除了管理性数据，缓存结构包括两个重要成员：

1.指向一个数组的指针，其中保存了各个 cpu 最后释放的对象。

2.每个内存节点都对应 3 个表头，用于组织 slab 的链表。

为最好利用 cpu 高速缓存，这些 per-CPU 指针是很重要的。在分配和释放对象时，采用后进先出原理。内核假定刚释放的对象仍然处于 cpu 高速缓存中，会尽快再次分配它（以响应下一个请求）。仅当 per-CPU 缓存为空时，才会用 slab 中的空闲对象重新填充它们。

slab 的结构：



The size used for each object does not reflect its exact size. Instead, the size is rounded to fulfill certain alignment criteria. Two alternatives are possible:

1. Using the flag SLAB_HWCACHE_ALIGN at slab creation time, the slab user can request that objects are aligned to hardware cache lines. The alignment is then performed along the value returned by cache_line_size, which returns the processor-specific size of the L1 cache.
If objects are smaller than half of the cache line size, then more than one object is fit into one cache line.

2.If alignment along hardware cache lines is not requested, then the kernel ensures that objects are aligned with BYTES_PER_WORD — the number of bytes needed to represent a void pointer.

On 32-bit processors, 4 bytes are required for a void pointer. Consequently, for an object with 6 bytes, 8 = 2 × 4 bytes are needed, and objects with 15 bytes require 16 = 4 × 4 bytes. The superfluous bytes are referred to as fill bytes.

## 2.数据结构

slab_def.h

```
struct kmem_cache {
/* 1) per-cpu data, touched during every alloc/free */
    struct array_cache *array[NR_CPUS];
/* 2) Cache tunables. Protected by cache_chain_mutex */
    unsigned int batchcount;
    unsigned int limit;
    unsigned int shared;
    unsigned int buffer_size;
    u32 reciprocal_buffer_size;
/* 3) touched by every alloc & free from the backend */
   unsigned int flags;     /* constant flags */
    unsigned int num;       /* # of objs per slab */
/* 4) cache_grow/shrink */
    /* order of pgs per slab (2^n) */
    unsigned int gfporder;
  /* force GFP flags, e.g. GFP_DMA */
    gfp_t gfpflags;
   size_t colour;          /* cache colouring range */
    unsigned int colour_off;    /* colour offset */
```

```
    struct kmem_cache *slabp_cache;
    unsigned int slab_size;
    unsigned int dflags;        /* dynamic flags */
   /* constructor func */
    void (*ctor)(void *obj);
/* 5) cache creation/removal */
    const char *name;
    struct list_head next;
    struct kmem_list3 *nodelists[MAX_NUMNODES];
}
```

部分元素解释：

(1)array:是一个指向数组的指针，每个数组项都对应系统中的一个 cpu。数组项的类型是

```
struct array_cache {
    unsigned int avail;
    unsigned int limit;
    unsigned int batchcount;
    unsigned int touched;
    spinlock_t lock;
    void *entry[];
};
```

avail：保存了当前可用对象的数目。

touched：touched is set to 1 when an element is removed from the cache, whereas cache shrinking causes touched to be set to 0. This enables the kernel to establish whether a cache has been accessed since it was last shrunk and is an indicator of the importance of the cache.

(2)batchcount:指定了在 per-cpu 列表为空的情况下，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的数目。

(3)limit:指定了 per-cpu 列表中保存的对象的最大数目。如果超出该值，内核会将 batchcount 个对象返回到 slab(如果接下来内核缩减缓存，则释放的内存从 slab 返回到伙伴系统)。

(4)buffer_size:缓存中管理的对象的长度。

(5)flags:定义缓存的全局性质。如果管理结构存储在 slab 的外部，则置为 CFLGS_OFF_SLAB。

(6)num:保存了可以放入 slab 对象的最大数目。

(7)nodelists:是一个数组，每个数组项对应于系统中一个可能的内存节点。

The element must be placed at the end of the structure. While it formally always has MAX_NUMNODES entries, it is possible that fewer nodes are usable on NUMA machines. The array thus requires fewer entries, and

the kernel can achieve this at run time by simply allocating less memory than the array formally requires. This would not be possible if nodelists were placed in the middle of the structure. On UMA machines, this is not much of a concern because only a single node will ever be available.

该数组项的类型是：

```
struct kmem_list3 {
    struct list_head slabs_partial; /* partial list first, better asm code */
    struct list_head slabs_full;
    struct list_head slabs_free;
    unsigned long free_objects;
    unsigned int free_limit;
    unsigned int colour_next;   /* Per-node cache coloring */
    spinlock_t list_lock;
    struct array_cache *shared; /* shared per node */
    struct array_cache **alien; /* on other nodes */
    unsigned long next_reap;    /* updated without locking */
    int free_touched;       /* updated without locking */
};
```

free_touched：indicates whether the cache is active or not. When an object is taken from the cache, the kernel sets the value of this variable to 1; when the cache is shrunk, the value is reset to 0. However, the kernel only shrinks a cache if free_touched has been set to 0 beforehand, because the value 1 indicates that another part of the kernel has just taken objects from the cache and thus it is not advisable to shrink it.

next_reap:指定了内核在两次尝试收缩缓存之间，必须经过的时间间隔。

free_limit：指定了所有 slab 上容许未使用对象的最大数目。

(8)gfporder:指定了 slab 包含的页数目以 2 为底的对数。

(9)colour，colour_off， kmem_list3->colour_next: colour specifies the maximum number of colors and colour_next the color to use for the next slab created by the kernel. Note, however, that this value is specified as an element of kmem_list3. colour_off is the basic offset multiplied by a color value to obtain the absolute offset.

Example: If there are five possible colors (0, 1, 2, 3, 4) and the offset unit is 8 bytes, the kernel can

use the following offset values: 0 × 8 = 0, 1 × 8 = 8, 2 × 8 = 16, 3 × 8 = 24 and 4 × 8 = 32 bytes.

(10)slabp_cache: If the slab head is stored outside the slab, slabp_cache points to the general cache from which the required memory is taken. If the slab head is on-slab, slabp_cache contains a null pointer.

(11)slab_size: cachep->num * sizeof(kmem_bufctl_t) + sizeof(struct slab)

## 3.初始化

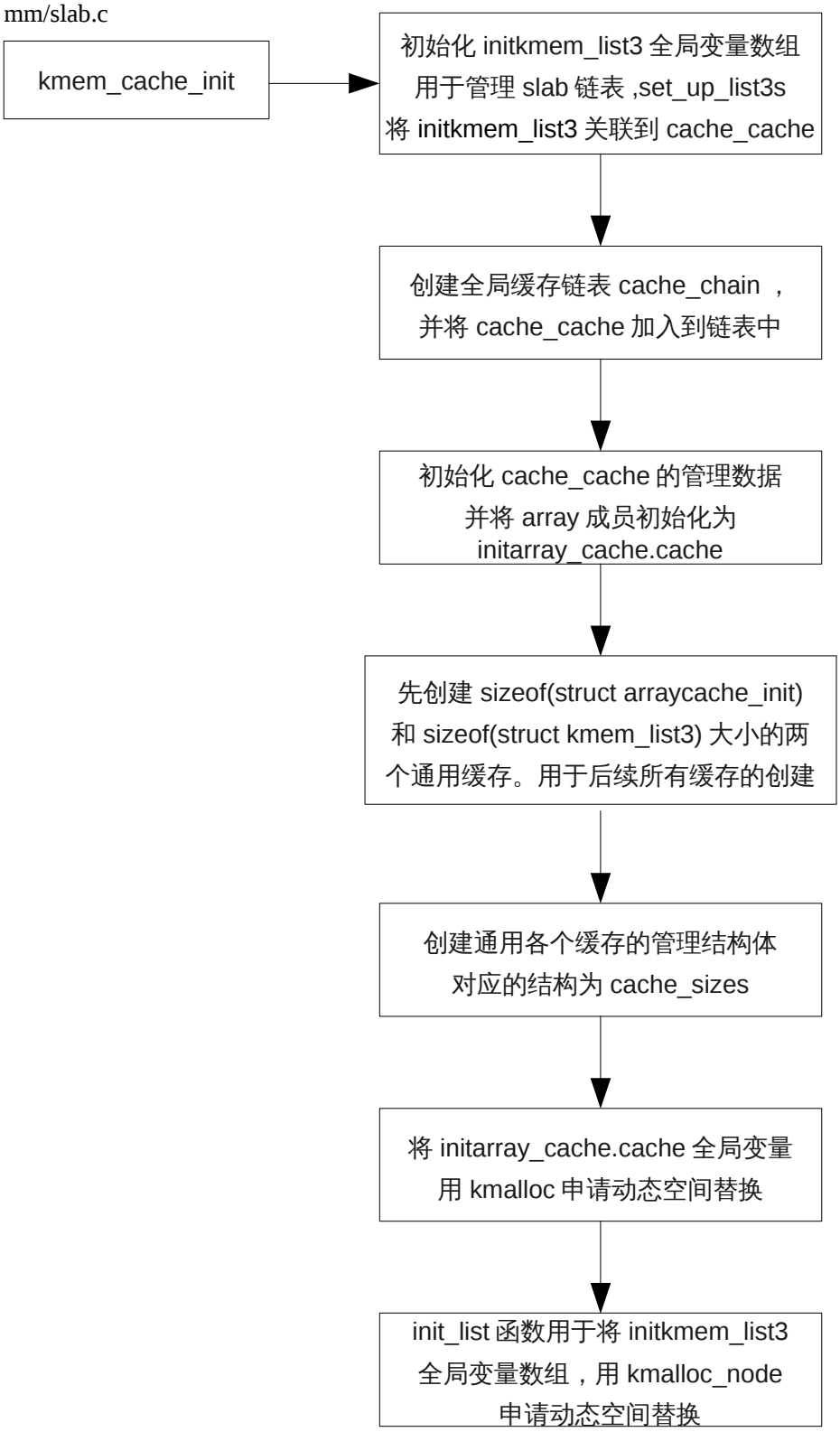
kmem_cache_init 函数用于初始化 slab 分配器。它在内核初始化阶段伙伴系统启用之后调用。

```
static struct kmem_cache cache_cache = {
    .batchcount = 1,
```

```
    .limit = BOOT_CPUCACHE_ENTRIES,
    .shared = 1,
    .buffer_size = sizeof(struct kmem_cache),
    .name = "kmem_cache",
};
```

cache_cache 为 kmem_cache 的缓存，即所有缓存的管理结构体空间都需要从该缓存中申请。

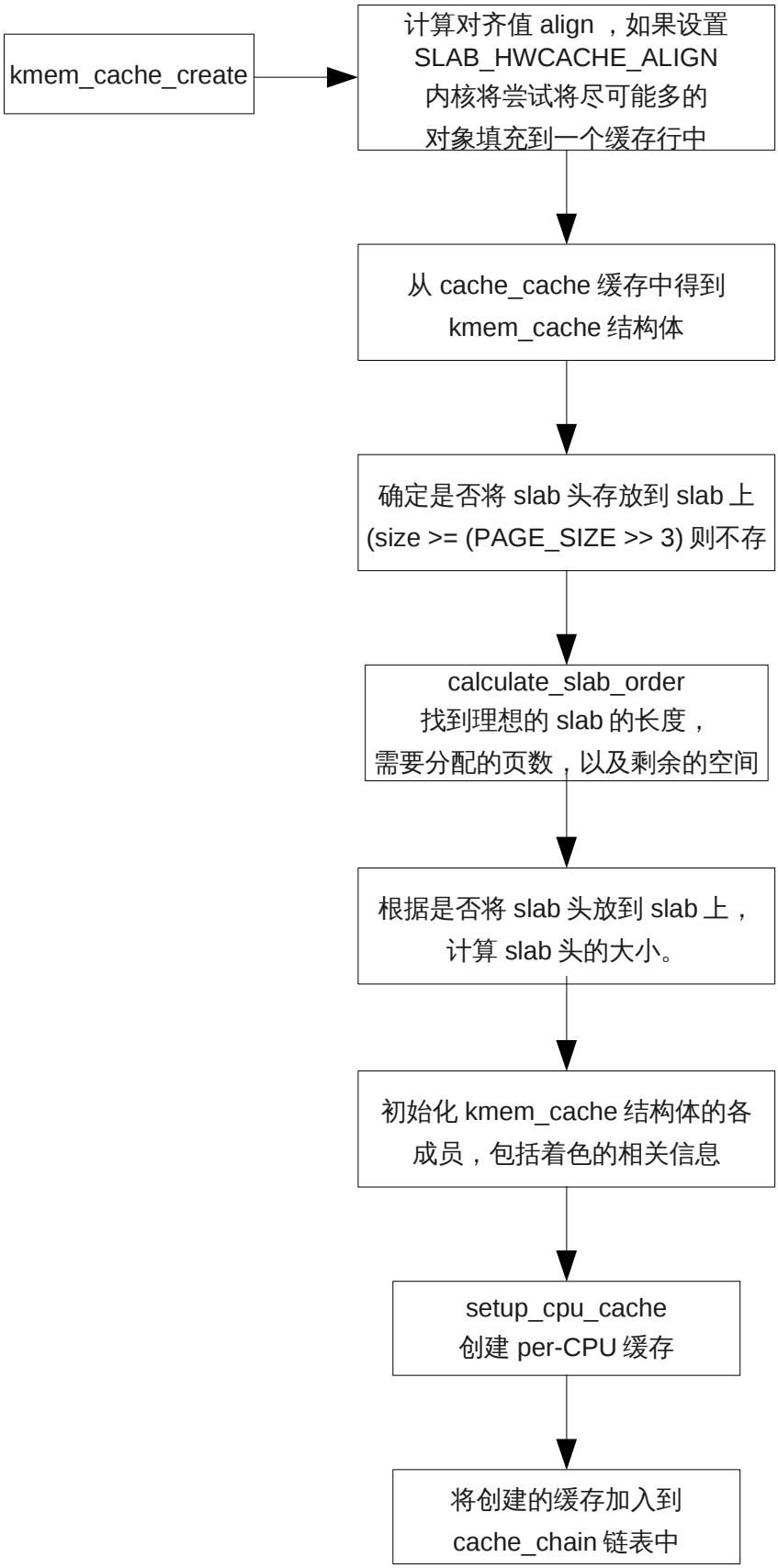
mm/slab.c

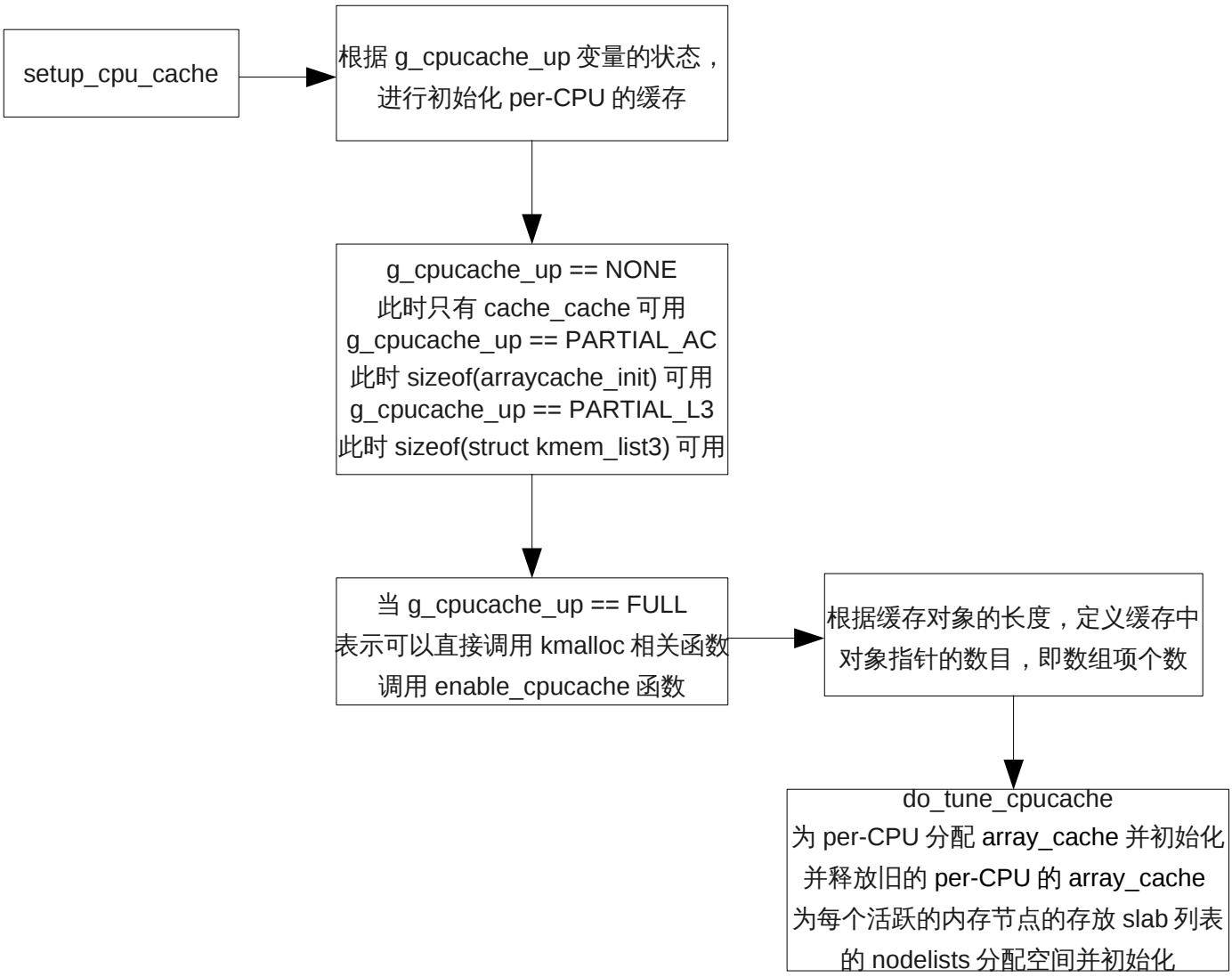


## 4.创建缓存

kmem_cache_create (const char *name, size_t size, size_t align, unsigned long flags, void (*ctor)(void *))

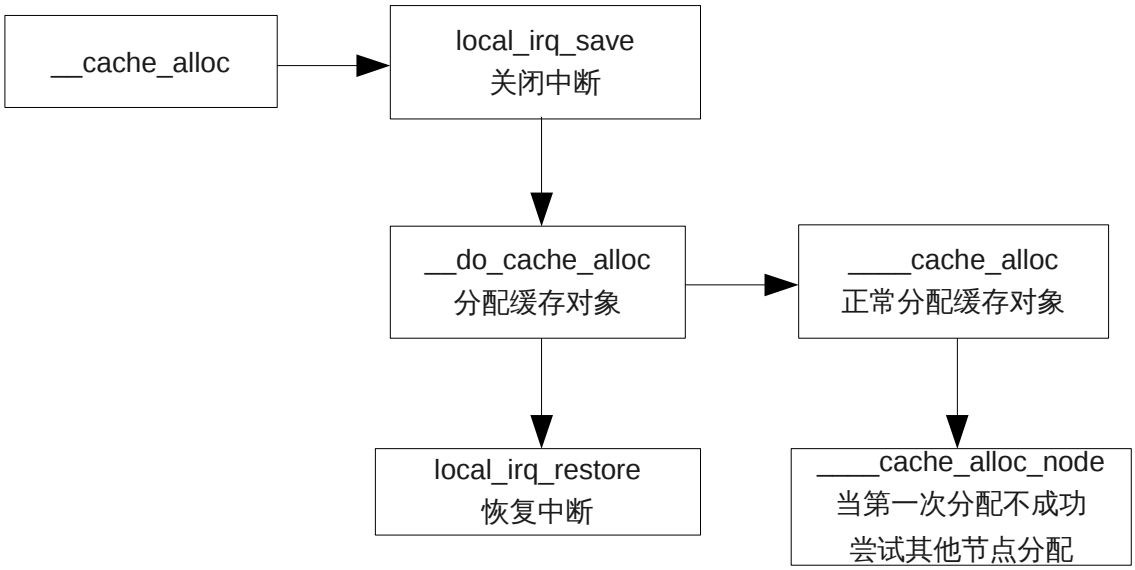
mm/slab.c

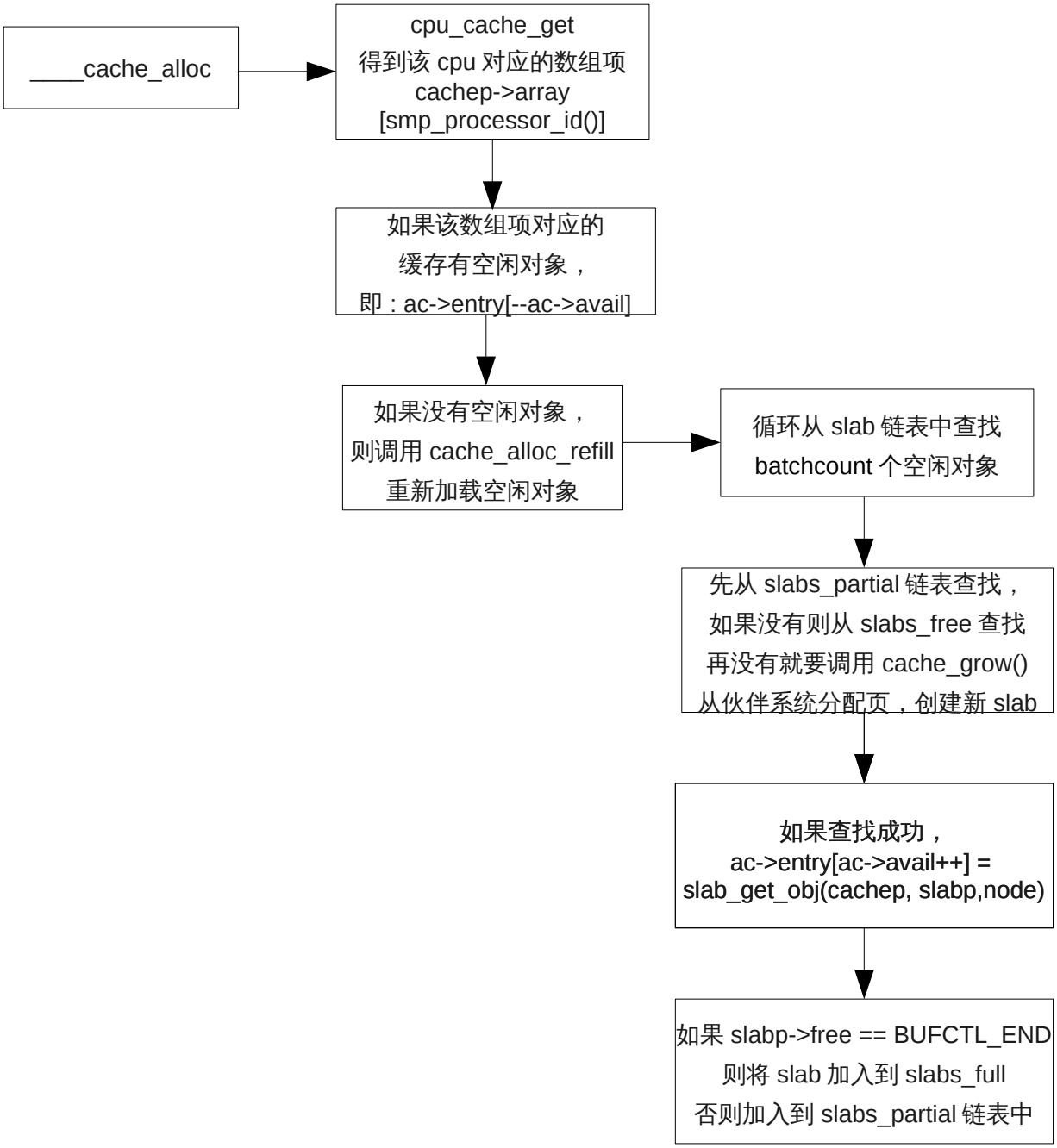




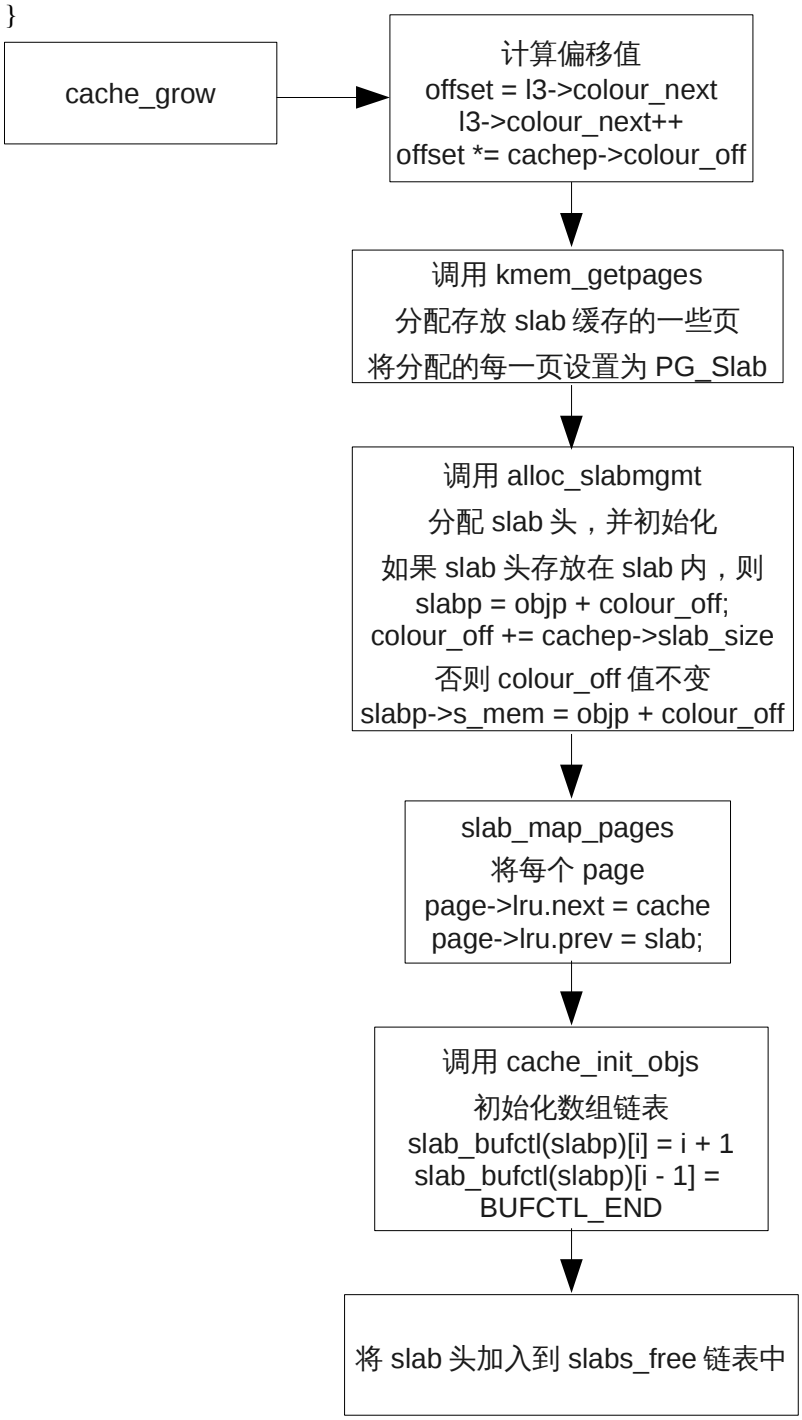

## 5.分配缓存对象

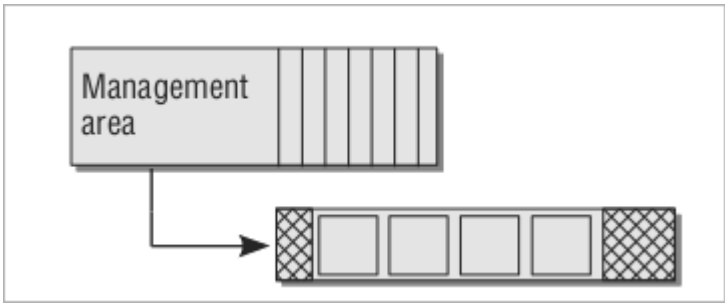
通用缓存或特定缓存的分配函数，都要归结到__cache_alloc。




____cache_alloc
cpu_cache_get
得到该 cpu 对应的数组项
cachep->array
[smp_processor_id()]
如果该数组项对应的
缓存有空闲对象，
即：ac->entry[--ac->avail]
如果没有空闲对象，
则调用 cache_alloc_refill
重新加载空闲对象
循环从 slab 链表中查找
batchcount 个空闲对象
先从 slabs_partial 链表查找，
如果没有则从 slabs_free 查找
再没有就要调用 cache_grow()
从伙伴系统分配页，创建新 slab
如果查找成功，
ac->entry[ac->avail++] =
slab_get_obj(cachep, slabp,node)
如果 slabp->free == BUFCTL_END
则将 slab 加入到 slabs_full
否则加入到 slabs_partial 链表中

```
static void *slab_get_obj(struct kmem_cache *cachep, struct slab *slabp,
                int nodeid)
{
    void *objp = index_to_obj(cachep, slabp, slabp->free);
    kmem_bufctl_t next;
    slabp->inuse++;
    next = slab_bufctl(slabp)[slabp->free];
    slabp->free = next;
    return objp;
}
```



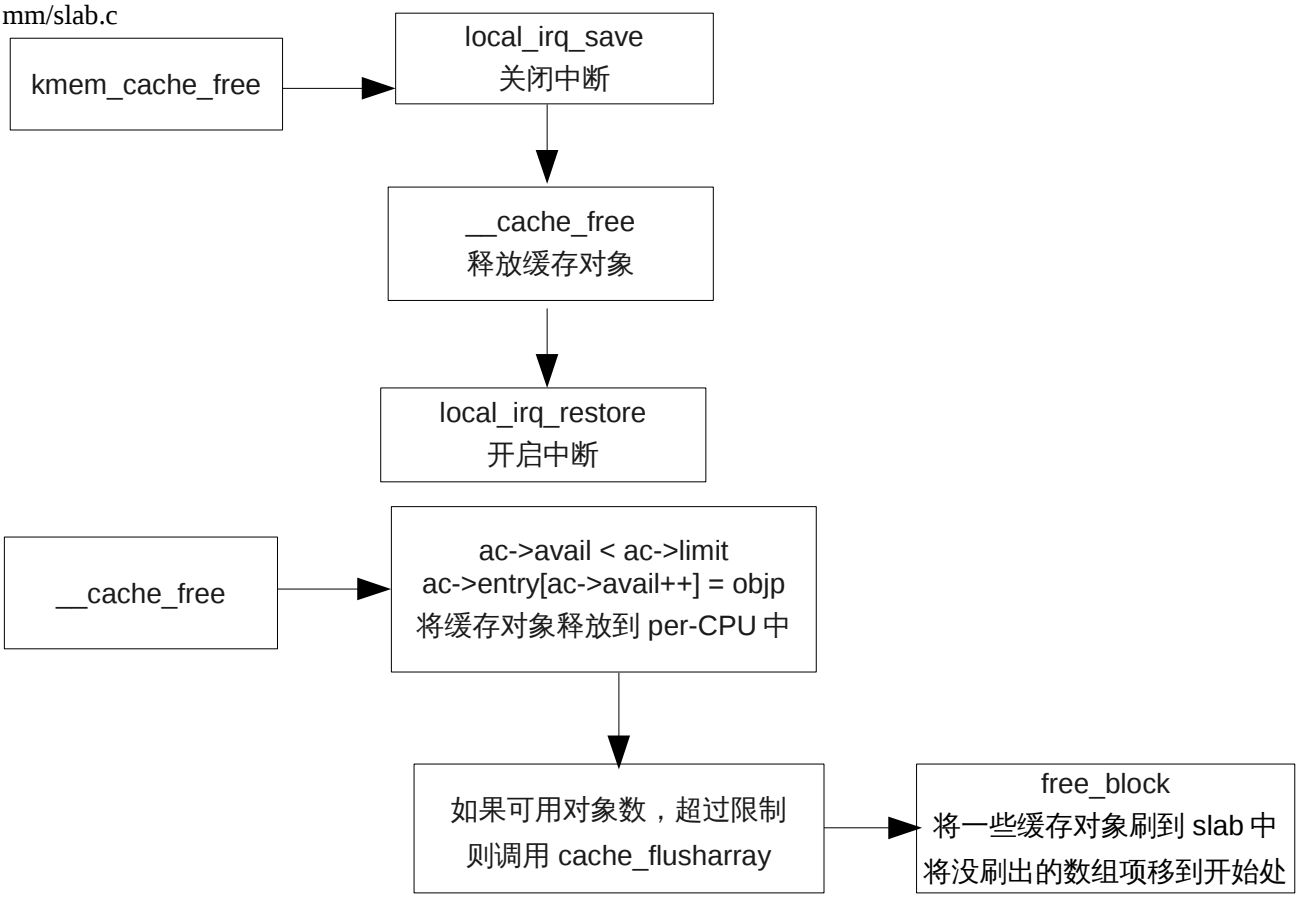


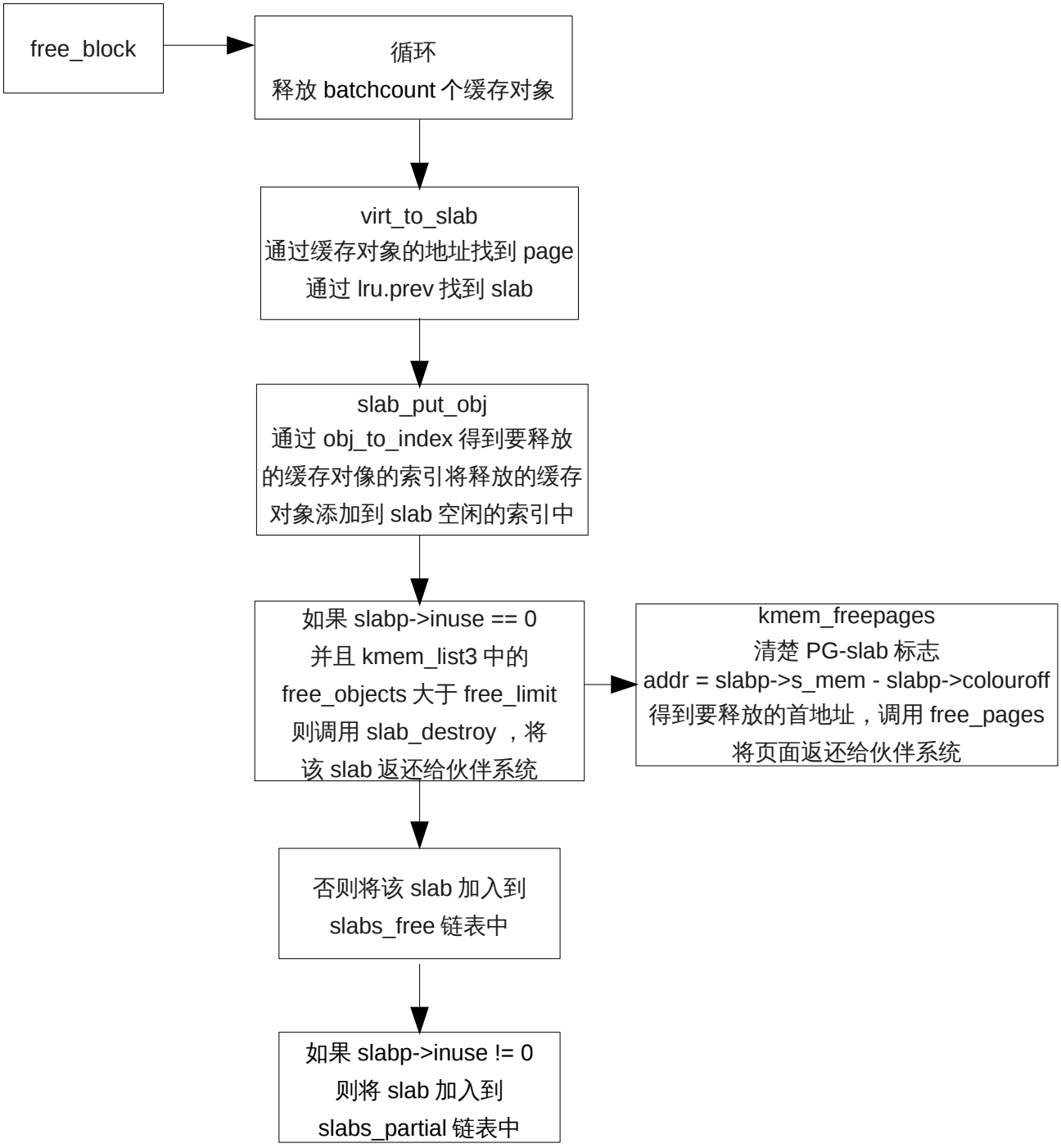

slab 管理数据在 slab 外的情况。

```
struct slab {
    struct list_head list;
    unsigned long colouroff;
    void *s_mem;        /* including colour offset */ 指向存储缓存对象的 slab 地址
    unsigned int inuse; /* num of objs active in slab */ 使用的缓存对象的数量
    kmem_bufctl_t free; //指向可用的缓存对象，当 free 为 BUFCTL_END 时，表示该slab 已经全部分配完
    unsigned short nodeid;
};
```

## 6.释放缓存对象

mm/slab.c

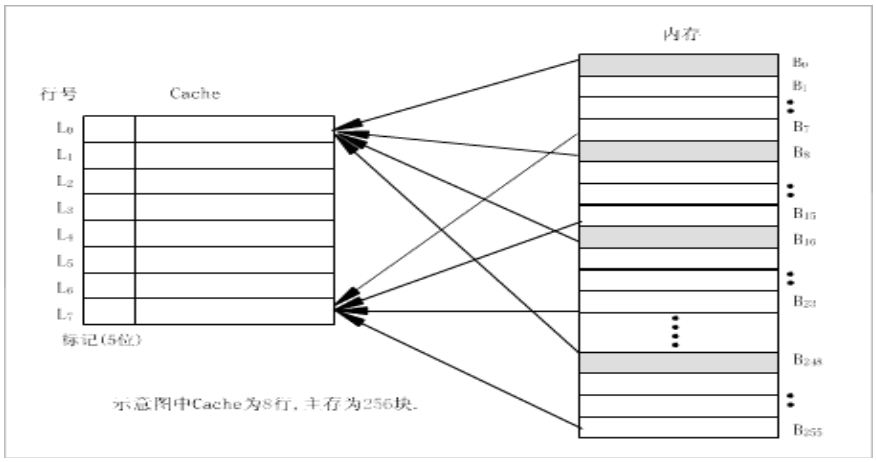


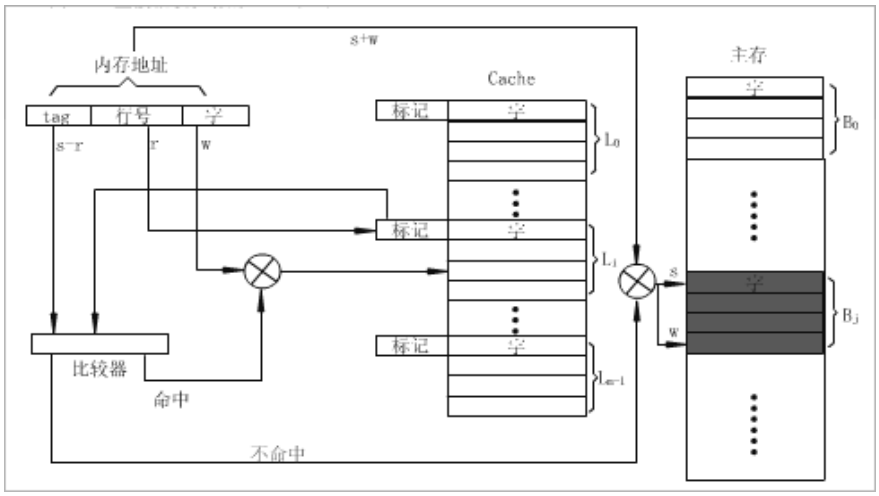



## *5.高速缓存的映射*

### 1.直接映射

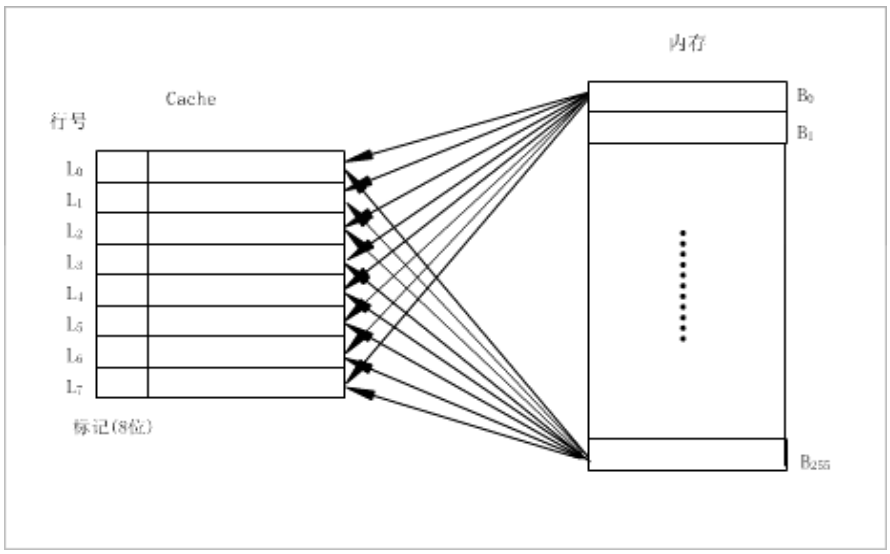
一种最简单而又直接的映射方法，指主存中每个块只能映射到 Cache 的一个特定的块。在该方法中，Cache 块地址 j 和主存块地址 i 的关系为： $j = i \bmod Cb$ 来计算，其中 *Cb* 是 Cache 的块数。这样，整个 Cache 地址与主存地址的低位部分完全相同。直接映射法的优点是所需硬件简单，只需要容量较小的按地址访问的区号标志表存储器和少 量比较电路；缺点是 Cache 块冲突概率较高，只要有两个或两个以上经常使用的块恰好被映射到 Cache 中的同一个块位置时，就会使 Cache 命中率急剧下降。

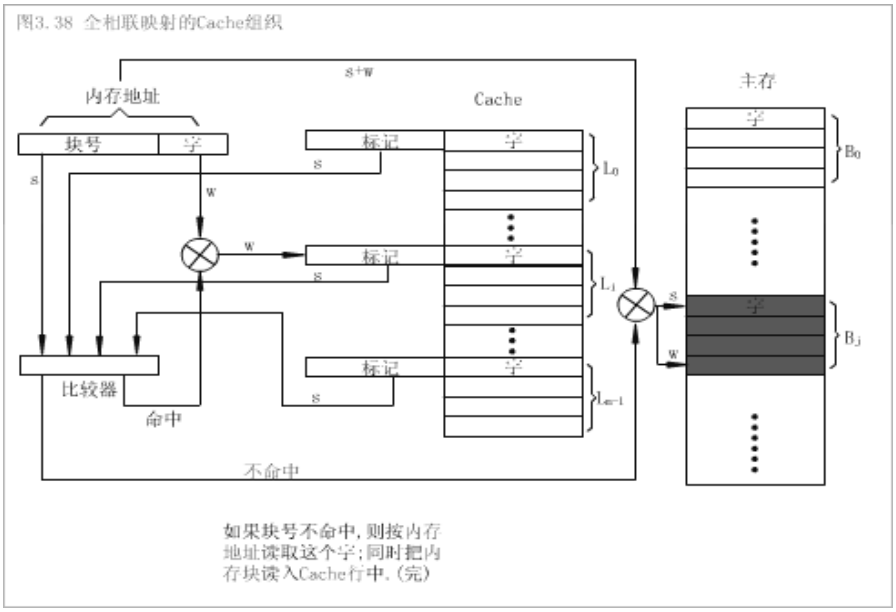



## 2.全相联映射

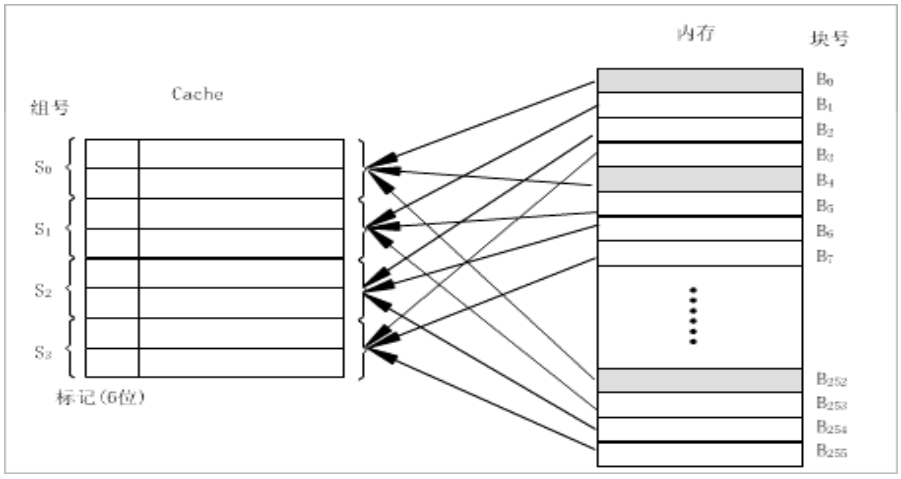
这种映射方式允许主存的每一块信息可以存到 Cache 的任何一个块空间，也允许从已被占满的 Cache 中替换掉任何一块信息。全相联映射的优点是块冲突概率低，其缺点是访问速度慢，并且成本太高。

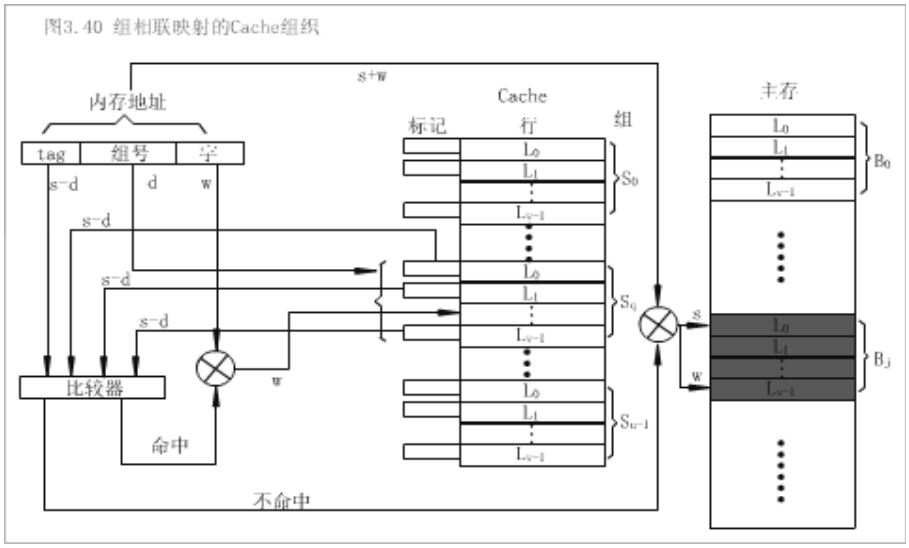





## 3.组相联映射

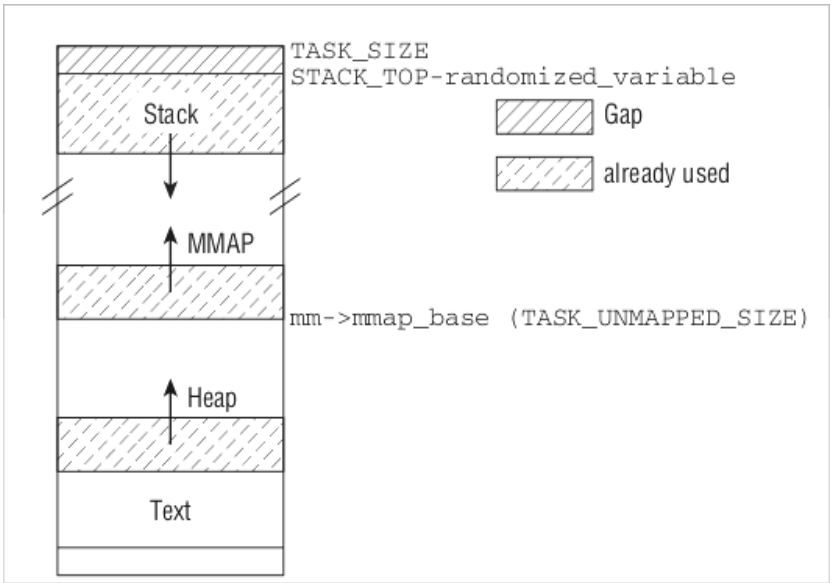
在这种映射方式中，把 Cache 分为 2e 组，每组包含 2r 个信息块，Cache 共有 2n =2 e+r 个信息块，主存共有 2 m =2 t+e+r 个信息块，是 Cache 的 2 t 倍。这种映射方式在组间是直接映射，而组内是全相联映射，其性能和复杂性介于直接映射和全相联映射之间。当 r =0 时，就成为直接映射方式；当 r =n 时，就是全相联映射。



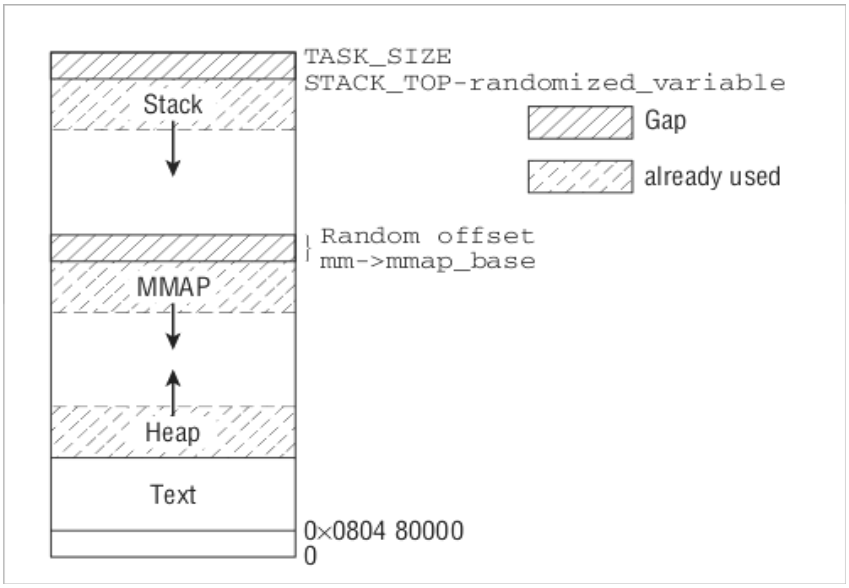



# 三.进程虚拟内存

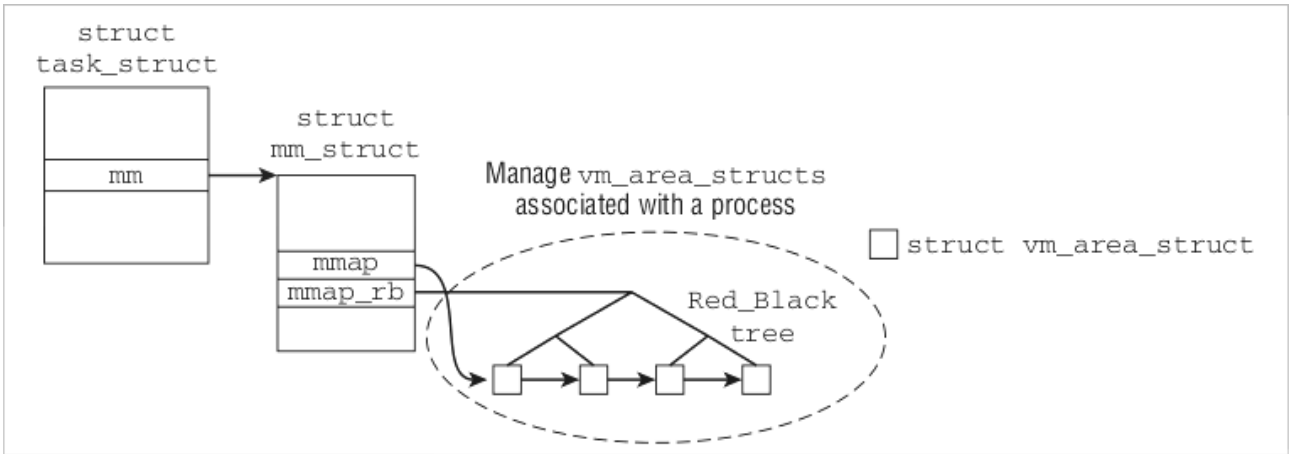
进程的线性地址空间：




mmap 区域自顶向下扩展时：




## *1.虚拟内存区域数据结构*



linux/mm_types.h

```
struct vm_area_struct {
    struct mm_struct * vm_mm;   /* The address space we belong to. */
    unsigned long vm_start;     /* Our start address within vm_mm. */
    unsigned long vm_end;
    struct vm_area_struct *vm_next, *vm_prev; /* linked list of VM areas per task, sorted by address */
    pgprot_t vm_page_prot;      /* Access permissions of this VMA. */
    unsigned long vm_flags;     /* Flags, see mm.h. */
    struct rb_node vm_rb;
   union {
        struct {
            struct list_head list;
            void *parent;   /* aligns with prio_tree_node parent */
            struct vm_area_struct *head;
        } vm_set;
     struct raw_prio_tree_node prio_tree_node;
    } shared;
    struct list_head anon_vma_chain;
    struct anon_vma *anon_vma;  /* Serialized by page_table_lock */
    const struct vm_operations_struct *vm_ops;
    unsigned long vm_pgoff;     /* Offset (within vm_file) in PAGE_SIZE units*/
    struct file * vm_file;      /* File we map to (can be NULL). */
    void * vm_private_data;     /* was vm_pte (shared mem) */
    unsigned long vm_truncate_count;/* truncate_count or restart_addr */
 #ifndef CONFIG_MMU
    struct vm_region *vm_region;    /* NOMMU mapping region */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;    /* NUMA policy for the VMA */
#endif
};
```

部分元素解释：

(1) vm_mm:指向区域所属的 mm_struct 实例。

(2) vm_start 和 vm_end:该区域在用户空间的起始和结束地址。

(3) vm_page_prot:存储该区域的访问权限

(4) vm_flags:描述该区域的一组标志：

❑ VM_READ, VM_WRITE, VM_EXEC, and VM_SHARED specify whether page contents can be read, written, executed, or shared by several processes.

❑ VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC, and VM_MAYSHARE determine whether the VM_* flags may be set. This is required for the mprotect system call.

❑ VM_GROWSDOWN and VM_GROWSUP indicate whether a region can be extended downward or upward (to lower/higher virtual addresses). Because the heap grows from bottom to top, VM_GROWSUP is set in its region; VM_GROWSDOWN is set for the stack, which grows from top to bottom.

❑ VM_SEQ_READ is set if it is likely that the region will be read sequentially from start to end;

VM_RAND_READ specifies that read access may be random. Both flags are intended as ‘‘prompts‘‘ for memory management and the block device layer to improve their optimizations (e.g., page readahead if access is primarily sequential. Chapter 8 takes a closer look at this technique).

❑ If VM_DONTCOPY is set, the relevant region is not copied when the fork system call is executed.

❑ VM_DONTEXPAND prohibits expansion of a region by the mremap system call.

❑ VM_HUGETLB is set if the region is based on huge pages as featured in some architectures.

❑ VM_ACCOUNT specifies whether the region is to be included in the calculations for the overcommit features. These features restrict memory allocations in various ways (refer to Section 4.5.3 for more details)

(5) shared:给出文件中的一个区间，内核有时需要知道该区间映射到的所有进程。这种映射称作共享映射，为提供所需的信息，所有的 vm_area_struct 实例都通过一个优先树管理。包含在 shared 中。等会具体讨论。

(6)anon_vma_chain 和 anon_vma:用于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中

(7)vm_ops:指向许多方法的集合，方法用于在区域上执行各种操作。

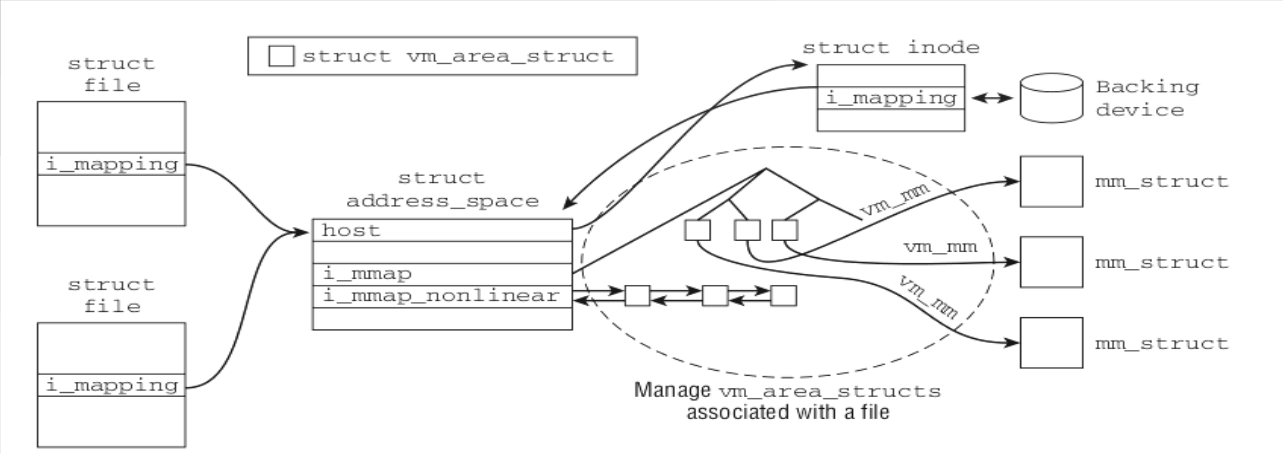
比如：int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);

如果地址空间中的某个虚拟内存页面不在物理内存中，自动触发的缺页异常处理程序会调用该函数。

(8)vm_pgoff:指定了文件映射的偏移量，单位是页。

(9)vm_file:描述一个被映射的文件。

优先树用于建立文件中的一个区域与该区域映射到的所有虚拟地址空间之间的关联。




每个文件和块设备都表示为 inode 的一实例。struct file 是通过 open 系统调用打开的文件的抽象，于此相反 inode 表示文件系统自身中的对象。inode 是特定于文件的数据结构，file 则是特定于给定进程的。

/include/linux/fs.h

struct address_space {

struct inode *host; /* owner: inode, block_device */

.....

struct prio_tree_root i_mmap; /* tree of private and shared mappings */

struct list_head i_mmap_nonlinear;/*list VM_NONLINEAR mappings */

spinlock_t i_mmap_lock; /* protect tree, count, list */

.....

}

include/linux/fs.h

struct file {

.....

struct address_space *f_mapping;

... .....

}

include/linux/fs.h

struct inode {

.....

struct address_space *i_mapping;

... .....

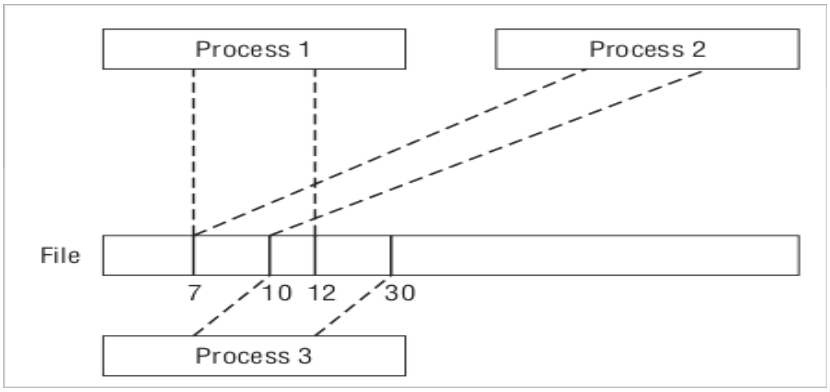
}

一个 vm_area_struct 实例，可以包含在两个数据结构中：一个建立进程虚拟地址空间中的区域与潜在的文件数据之间的关联，一个用于查找映射了给定文件区间的所有地址空间。

优先树的表示：

Priority trees allow for management of the vm_area_struct instances that represent a particular interval of the given file. This requires that the data structure cannot only deal with overlapping, but also with identical file intervals. The situation is illustrated in below: Two processes map the region [7, 12] of a file into their virtual address space, while a third process maps the interval [10, 30]. Managing overlapping intervals is not much of a

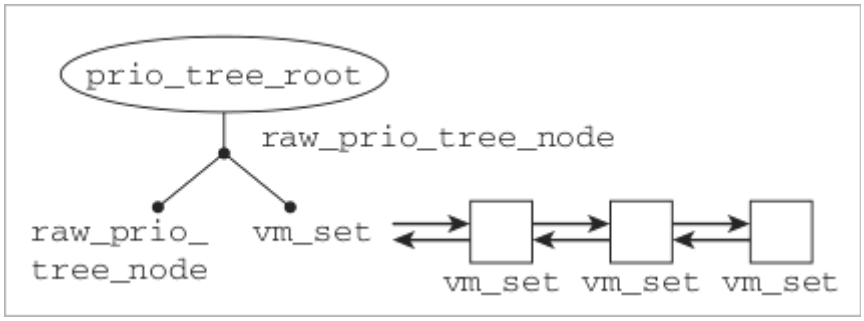
problem: The boundaries of the interval provide a unique index that allows for storing each interval in a unique tree node. It suffices to know that if intervals B, C, and D are completely contained in another interval A, then A will be the parent element of B, C, and D.



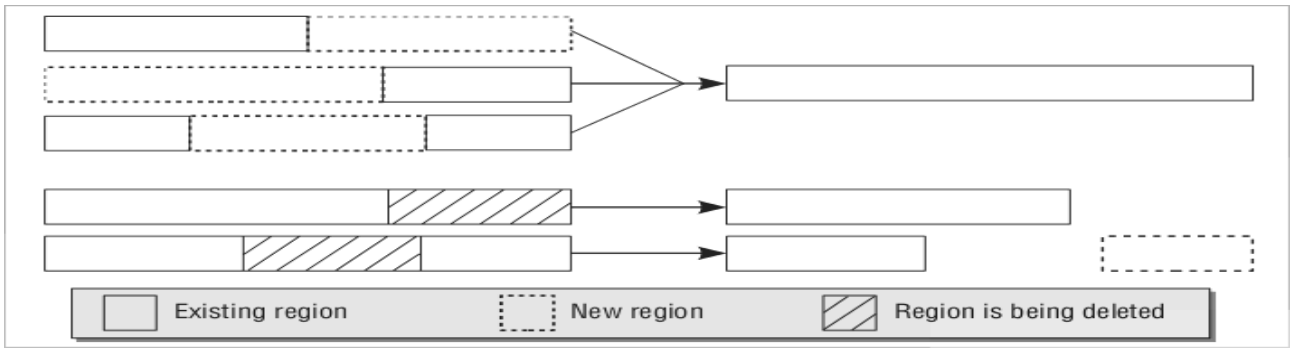

However, what happens if multiple identical intervals must be included in the prio tree? Each prio

tree node is represented by the raw_prio_tree_node instance, which is directly included in each vm_ area_struct instance. Recall, however, that it is in a union with a vm_set. This allows for associating

a list of vm_sets (and thus vm_area_structs) with a prio tree node.




## *2.内存区域操作*




1.查找地址关联到的区域：

mm/mmap.c

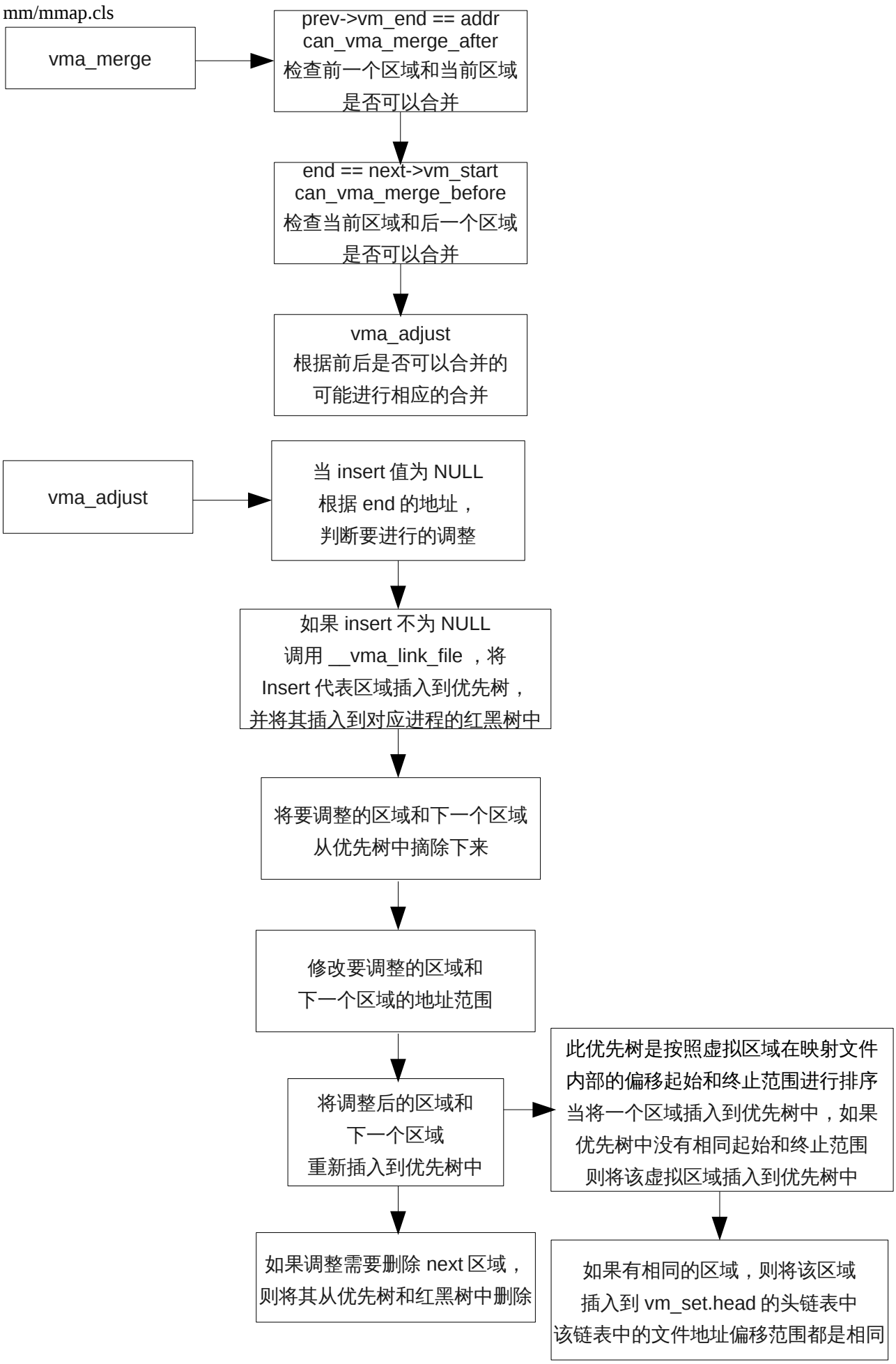
struct vm_area_struct *find_vma(struct mm_struct *mm, unsigned long addr)

在红黑树上查找包含 addr 的适当区域。

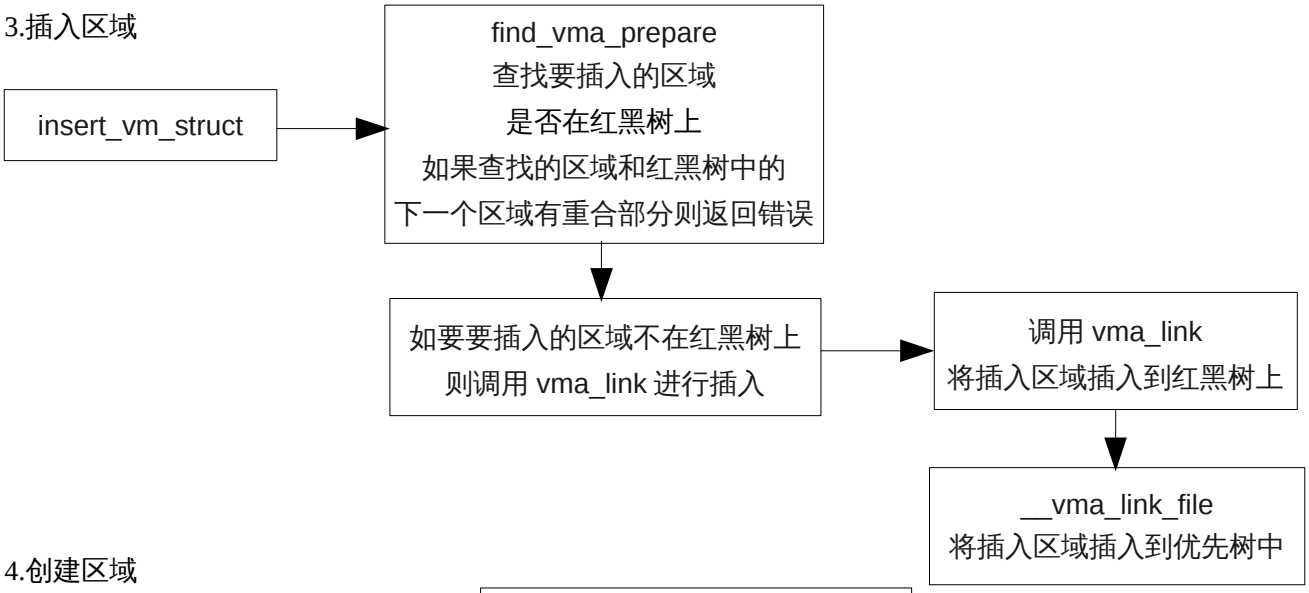
如果 addr 包含在红黑树的区域中，则返回查找到的区域。

否则返回离 addr 最近的下一个区域。

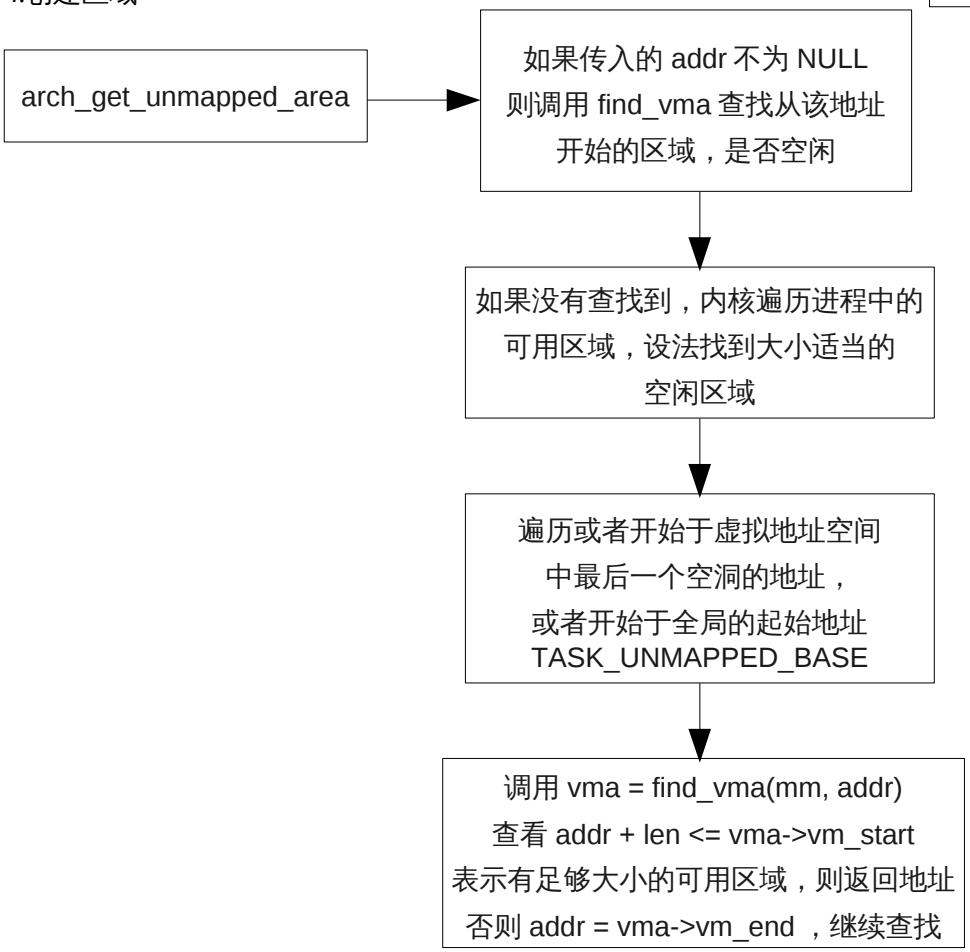
2.合并区域：


mm/mmap.cls
vma_merge
prev->vm_end == addr
can_vma_merge_after
检查前一个区域和当前区域
是否可以合并
end == next->vm_start
can_vma_merge_before
检查当前区域和后一个区域
是否可以合并
vma_adjust
根据前后是否可以合并的
可能进行相应的合并
vma_adjust
当 insert 值为 NULL
根据 end 的地址，
判断要进行的调整
如果 insert 不为 NULL
调用 __vma_link_file ，将
Insert 代表区域插入到优先树，
并将其插入到对应进程的红黑树中
将要调整的区域和下一个区域
从优先树中摘除下来
修改要调整的区域和
下一个区域的地址范围
将调整后的区域和
下一个区域
重新插入到优先树中
此优先树是按照虚拟区域在映射文件
内部的偏移起始和终止范围进行排序
当将一个区域插入到优先树中，如果
优先树中没有相同起始和终止范围
则将该虚拟区域插入到优先树中
如果调整需要删除 next 区域，
则将其从优先树和红黑树中删除
如果有相同的区域，则将该区域
插入到 vm_set.head 的头链表中
该链表中的文件地址偏移范围都是相同

3.插入区域

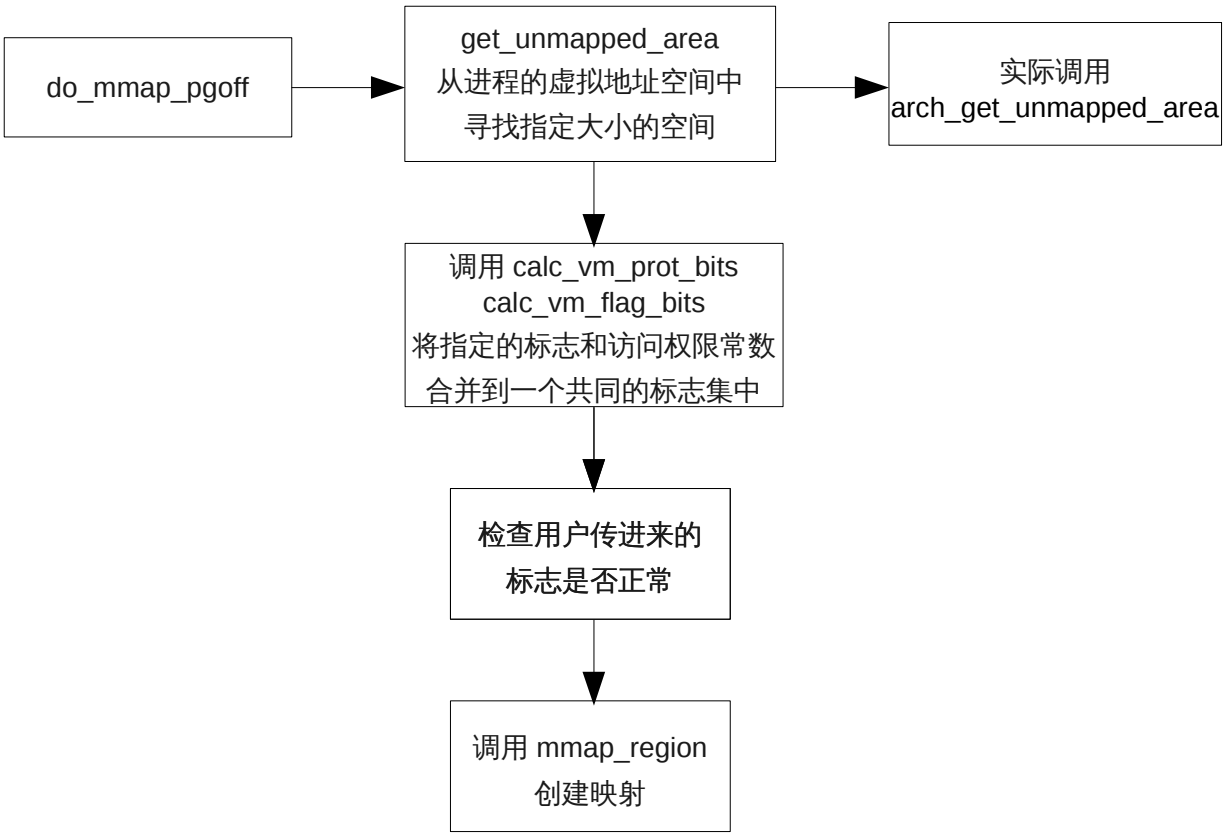


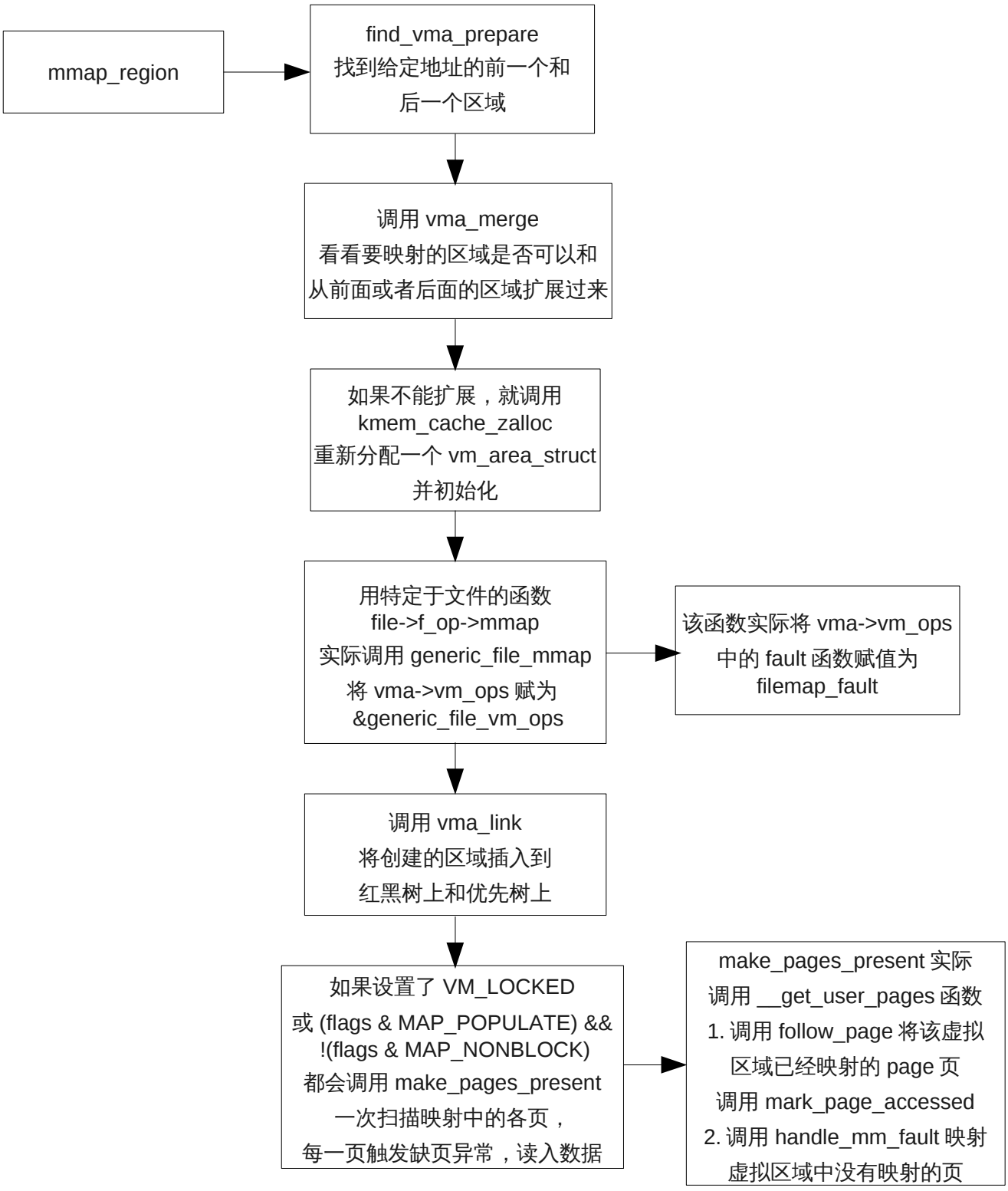

4.创建区域



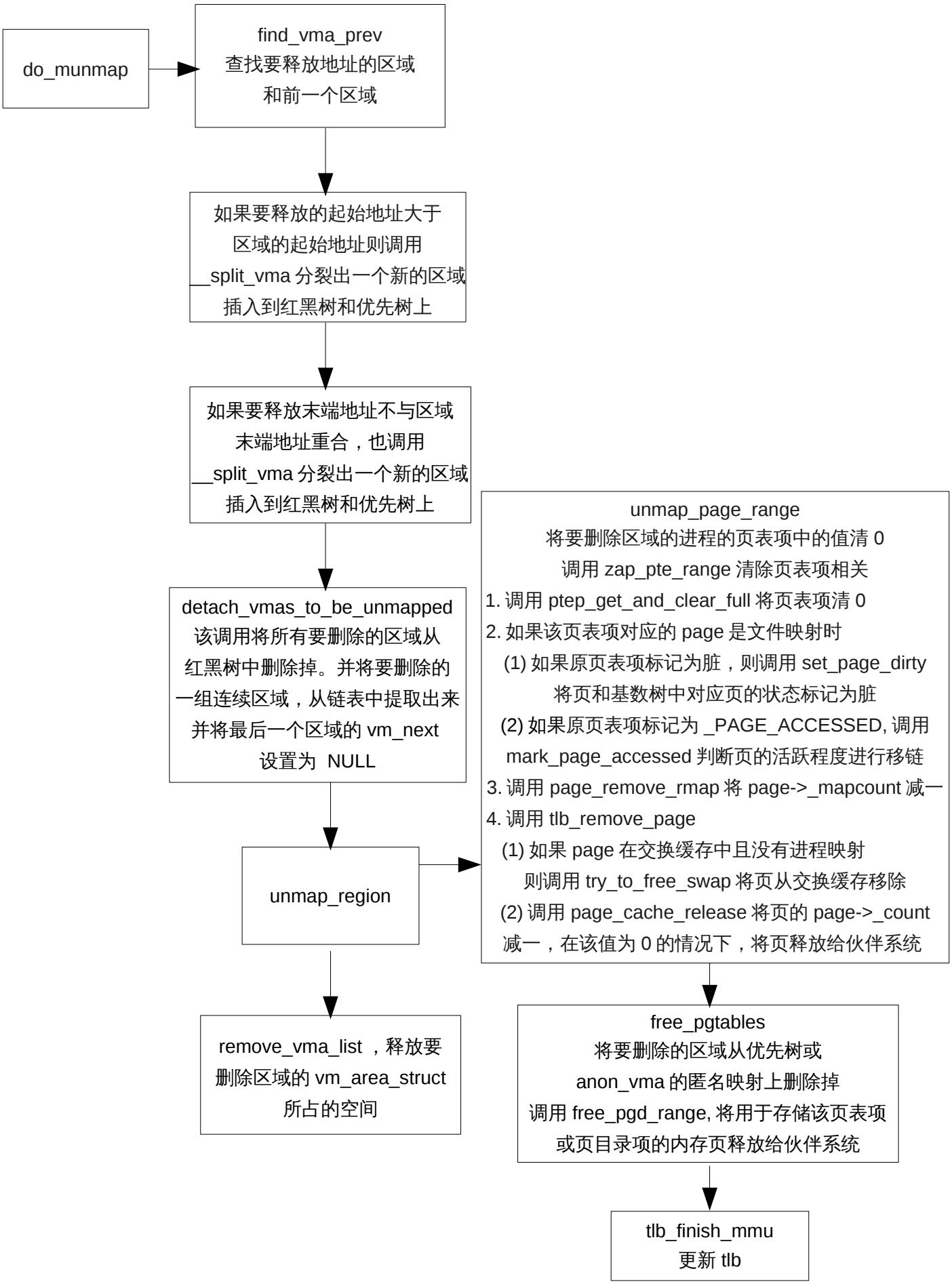

## *3.内存映射*

## 1.创建映射


do_mmap_pgoff
get_unmapped_area
从进程的虚拟地址空间中
寻找指定大小的空间
实际调用
arch_get_unmapped_area
调用 calc_vm_prot_bits
calc_vm_flag_bits
将指定的标志和访问权限常数
合并到一个共同的标志集中
检查用户传进来的
标志是否正常
调用 mmap_region
创建映射

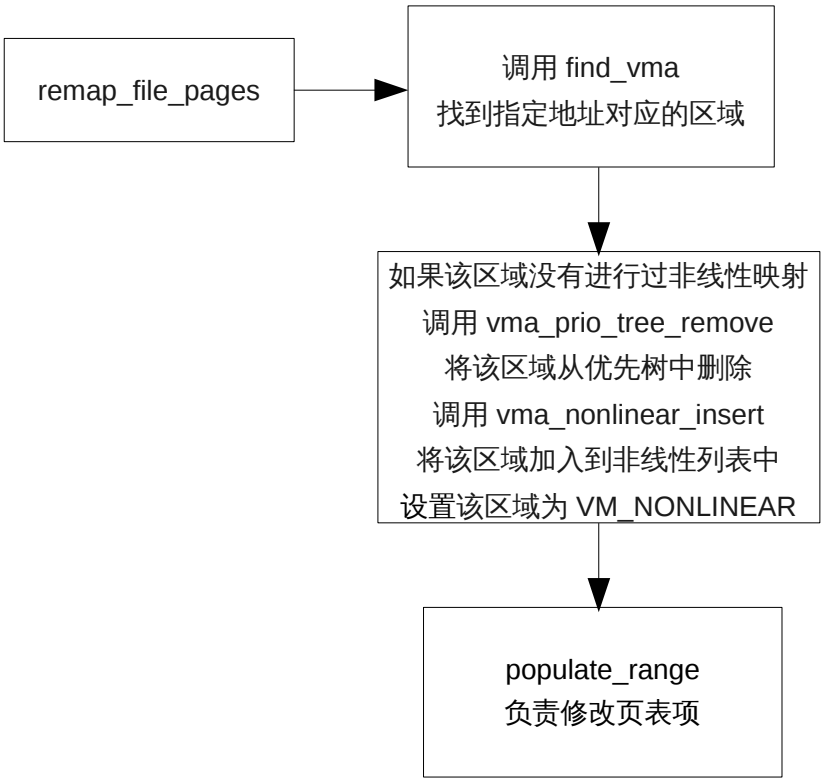

mmap_region
find_vma_prepare
找到给定地址的前一个和
后一个区域
调用 vma_merge
看看要映射的区域是否可以和
从前面或者后面的区域扩展过来
如果不能扩展，就调用
kmem_cache_zalloc
重新分配一个 vm_area_struct
并初始化
用特定于文件的函数
file->f_op->mmap
实际调用 generic_file_mmap
将 vma->vm_ops 赋为
&generic_file_vm_ops
该函数实际将 vma->vm_ops
中的 fault 函数赋值为
filemap_fault
调用 vma_link
将创建的区域插入到
红黑树上和优先树上
如果设置了 VM_LOCKED
或 (flags & MAP_POPULATE) &&
!(flags & MAP_NONBLOCK)
都会调用 make_pages_present
一次扫描映射中的各页，
每一页触发缺页异常，读入数据
make_pages_present 实际
调用 __get_user_pages 函数
1. 调用 follow_page 将该虚拟
区域已经映射的 page 页
调用 mark_page_accessed
2. 调用 handle_mm_fault 映射
虚拟区域中没有映射的页

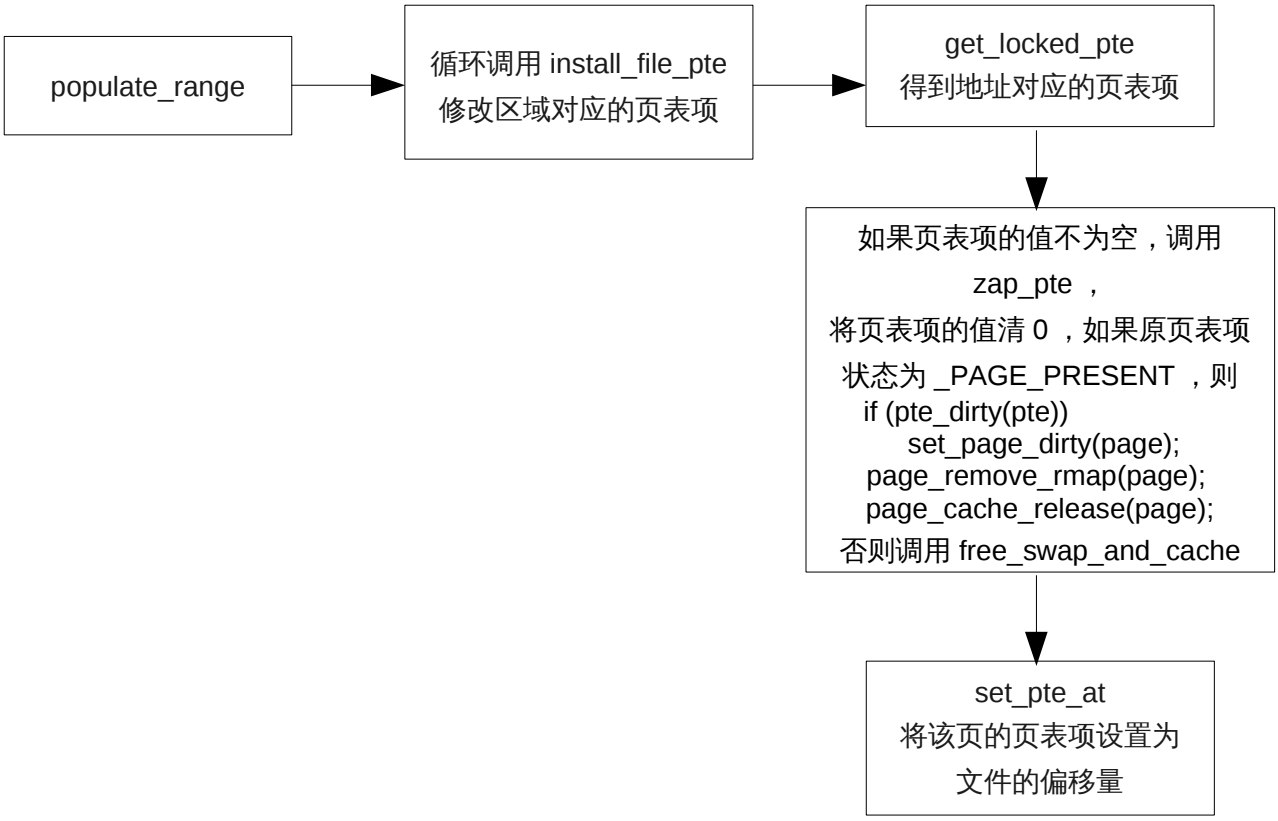
## 2.删除映射


do_munmap
find_vma_prev
查找要释放地址的区域
和前一个区域
如果要释放的起始地址大于
区域的起始地址则调用
__split_vma 分裂出一个新的区域
插入到红黑树和优先树上
如果要释放末端地址不与区域
末端地址重合，也调用
__split_vma 分裂出一个新的区域
插入到红黑树和优先树上
detach_vmas_to_be_unmapped
该调用将所有要删除的区域从
红黑树中删除掉。并将要删除的
一组连续区域，从链表中提取出来
并将最后一个区域的 vm_next
设置为 NULL
unmap_region
unmap_page_range
将要删除区域的进程的页表项中的值清 0
调用 zap_pte_range 清除页表项相关
1. 调用 ptep_get_and_clear_full 将页表项清 0
2. 如果该页表项对应的 page 是文件映射时
(1) 如果原页表项标记为脏，则调用 set_page_dirty
将页和基数树中对应页的状态标记为脏
(2) 如果原页表项标记为 _PAGE_ACCESSED, 调用
mark_page_accessed 判断页的活跃程度进行移链
3. 调用 page_remove_rmap 将 page->_mapcount 减一
4. 调用 tlb_remove_page
(1) 如果 page 在交换缓存中且没有进程映射
则调用 try_to_free_swap 将页从交换缓存移除
(2) 调用 page_cache_release 将页的 page->_count
减一，在该值为 0 的情况下，将页释放给伙伴系统
remove_vma_list ，释放要
删除区域的 vm_area_struct
所占的空间
free_pgtables
将要删除的区域从优先树或
anon_vma 的匿名映射上删除掉
调用 free_pgd_range, 将用于存储该页表项
或页目录项的内存页释放给伙伴系统
tlb_finish_mmu
更新 tlb

## 3.非线性映射

mm/fremap.c



## *4.逆向映射*

逆向映射用于通过物理页找到该页所属的进程

# 1.虚拟内存区域和逆向映射的关联

```
struct anon_vma {
    spinlock_t lock;    /* Serialize access to vma list */
    struct anon_vma *root;  /* Root of this anon_vma tree */
    struct list_head head;  /* Chain of private "related" vmas */
};

struct anon_vma_chain {
    struct vm_area_struct *vma;
    struct anon_vma *anon_vma;
    struct list_head same_vma;   /* locked by mmap_sem & page_table_lock */
    struct list_head same_anon_vma; /* locked by anon_vma->lock */
};
```

This structure allows us to find the anon_vmas associated with a VMA, or the VMAs associated with an anon_vma. The "same_vma" list contains the anon_vma_chains linking all the anon_vmas associated with this VMA. The "same_anon_vma" list contains the anon_vma_chains which link all the VMAs associated with this anon_vma.

```
struct vm_area_struct {
.....
    struct list_head anon_vma_chain; /* Serialized by mmap_sem & page_table_lock */
    struct anon_vma *anon_vma;  /* Serialized by page_table_lock */
.....
}

anon_vma_chain_link(struct vm_area_struct *vma,  struct anon_vma_chain *avc,  struct anon_vma *anon_vma)
{
    avc->vma = vma;
    avc->anon_vma = anon_vma;
    list_add(&avc->same_vma, &vma->anon_vma_chain);
    anon_vma_lock(anon_vma);
    list_add_tail(&avc->same_anon_vma, &anon_vma->head);
    anon_vma_unlock(anon_vma);   }
```

## 2.页面和逆向映射的关联

include/linux/mm_types.h

struct page {

atomic_t _mapcount;

struct address_space *mapping;  /* If low bit clear, points to inode address_space, or NULL.

* If page mapped as anonymous  memory, low bit is set, and  it points to

*anon_vma object: * see PAGE_MAPPING_ANON belowl*/

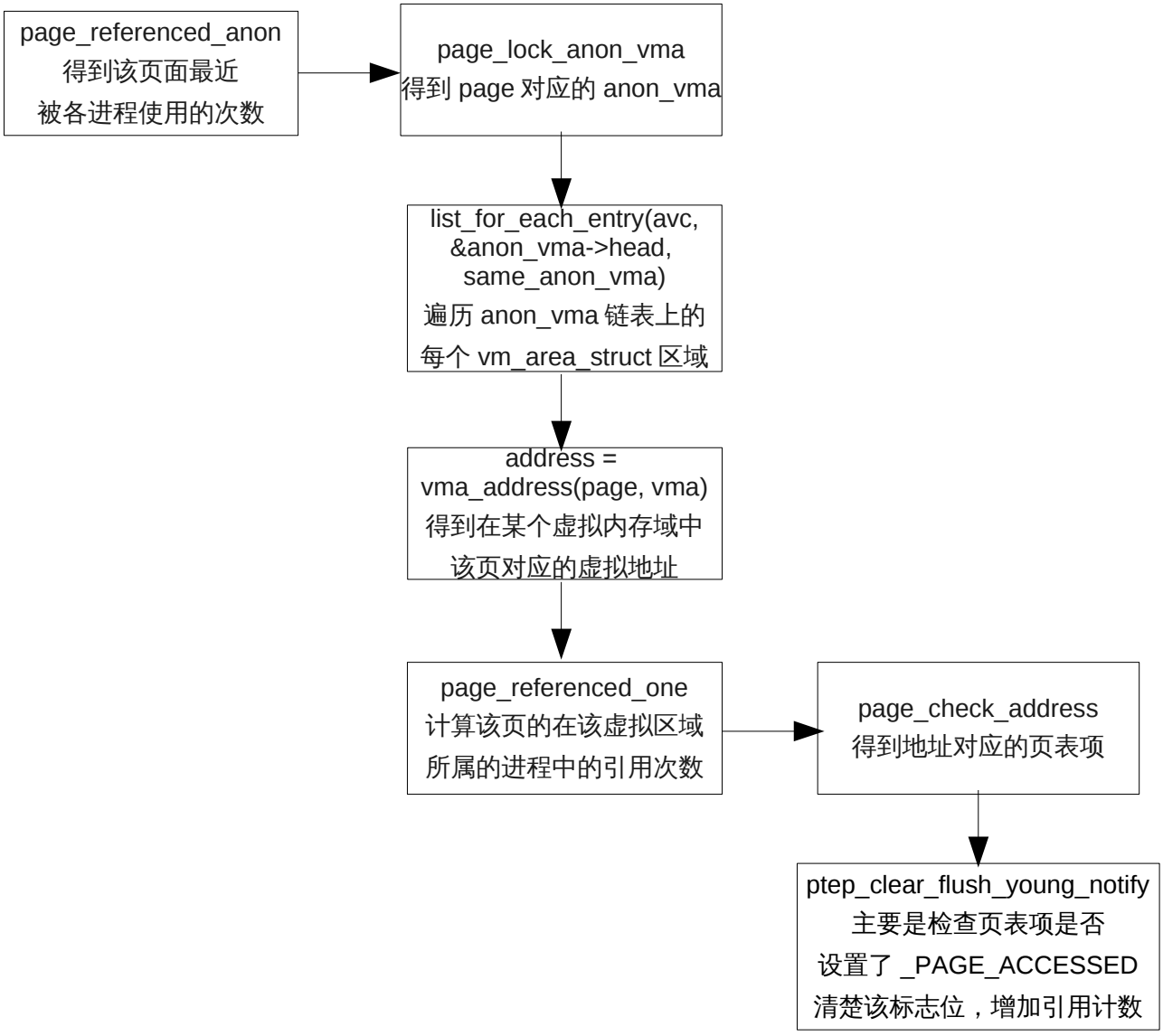
}

_mapcount 表明共享该页的位置的数目。计数器的初始值为-1。在页插入到逆向映射数据结构时，计数器赋值为 0.页每次增加一个使用者时，计数器加 1。这使得内核能够快速检查在所有者之外该页有多少使用者。

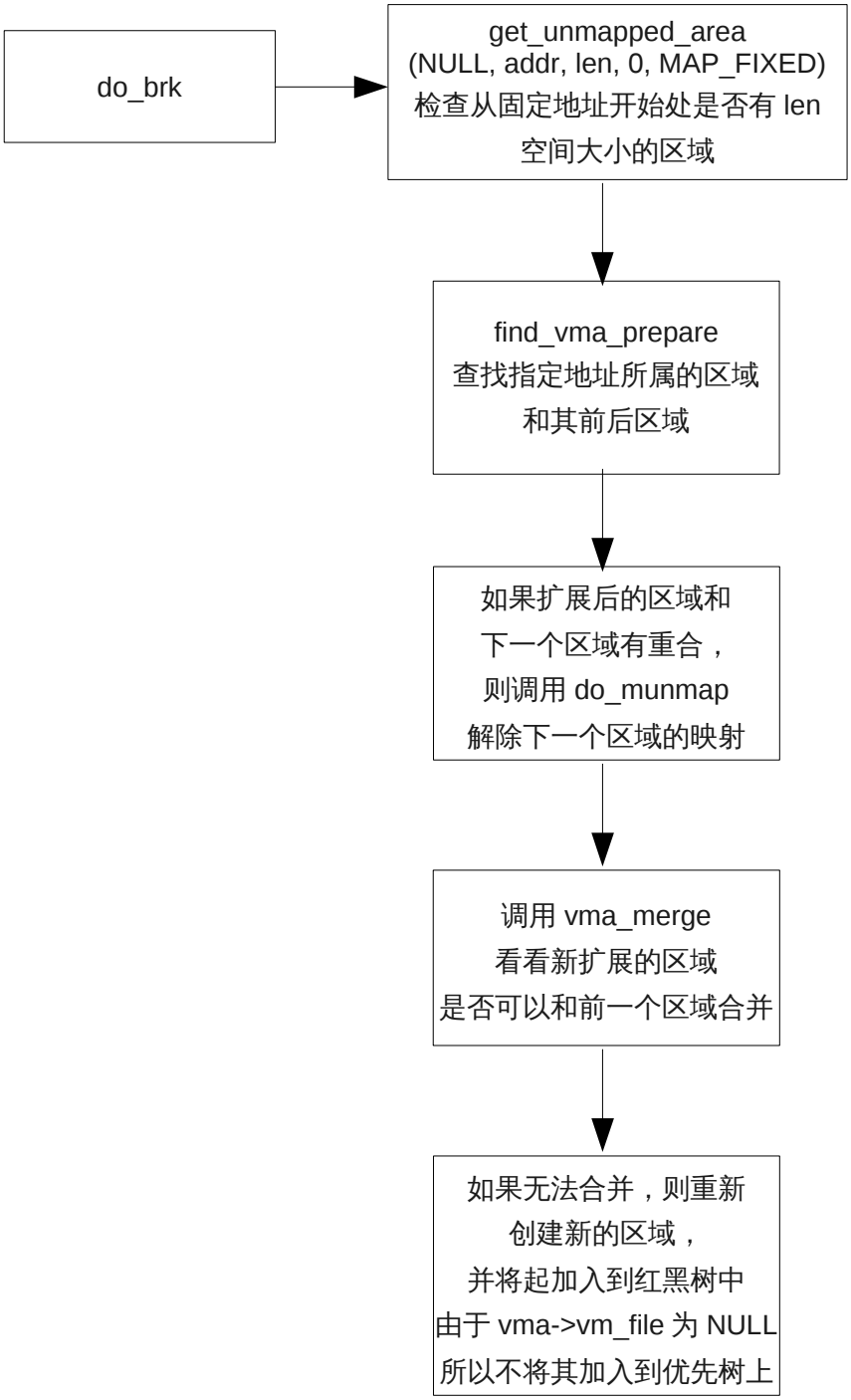
anon_vma = (void *) anon_vma + PAGE_MAPPING_ANON;

page->mapping = (struct address_space *) anon_vma;

通过检查 page 实例的 mapping 成员的最低位来区分是匿名页或普通映射的页。

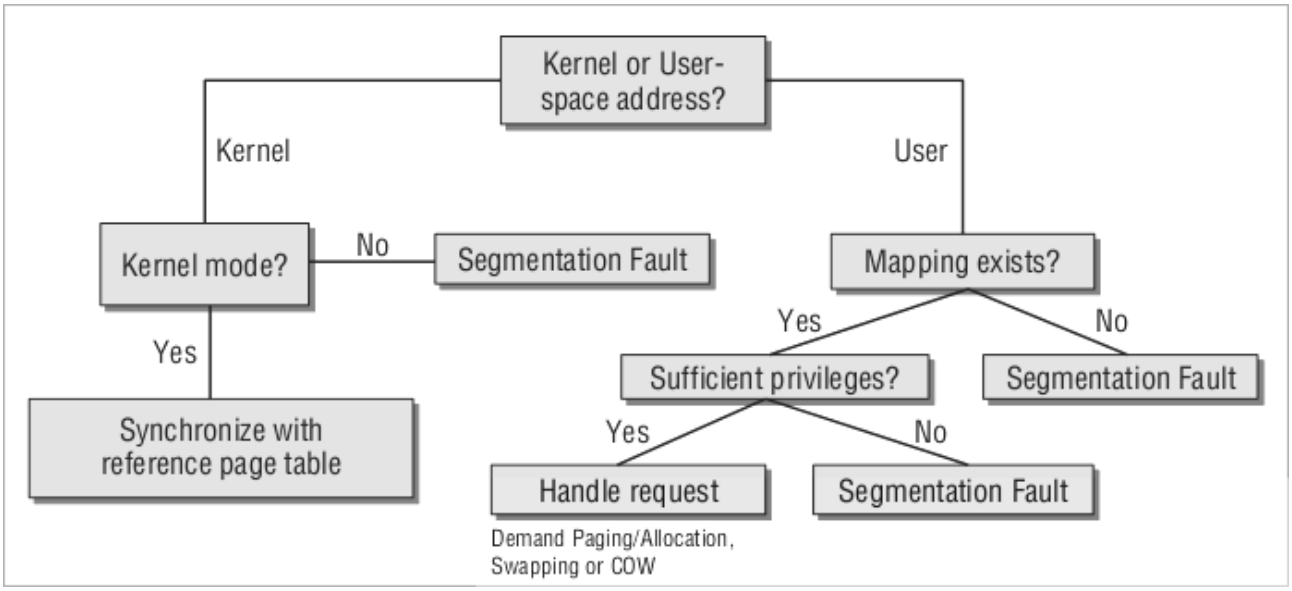


## *5.堆的管理*




## *6.缺页异常的处理*

在实际需要某个虚拟内存区域之前，虚拟和物理内存之间的关联不会建立。如果进程访问的虚拟地址空间部分尚未与页帧关联，处理器自动引发一个缺页异常。内核处理该异常，创建虚拟内存和物理页的关联。




arch/x86/mm/fault.c

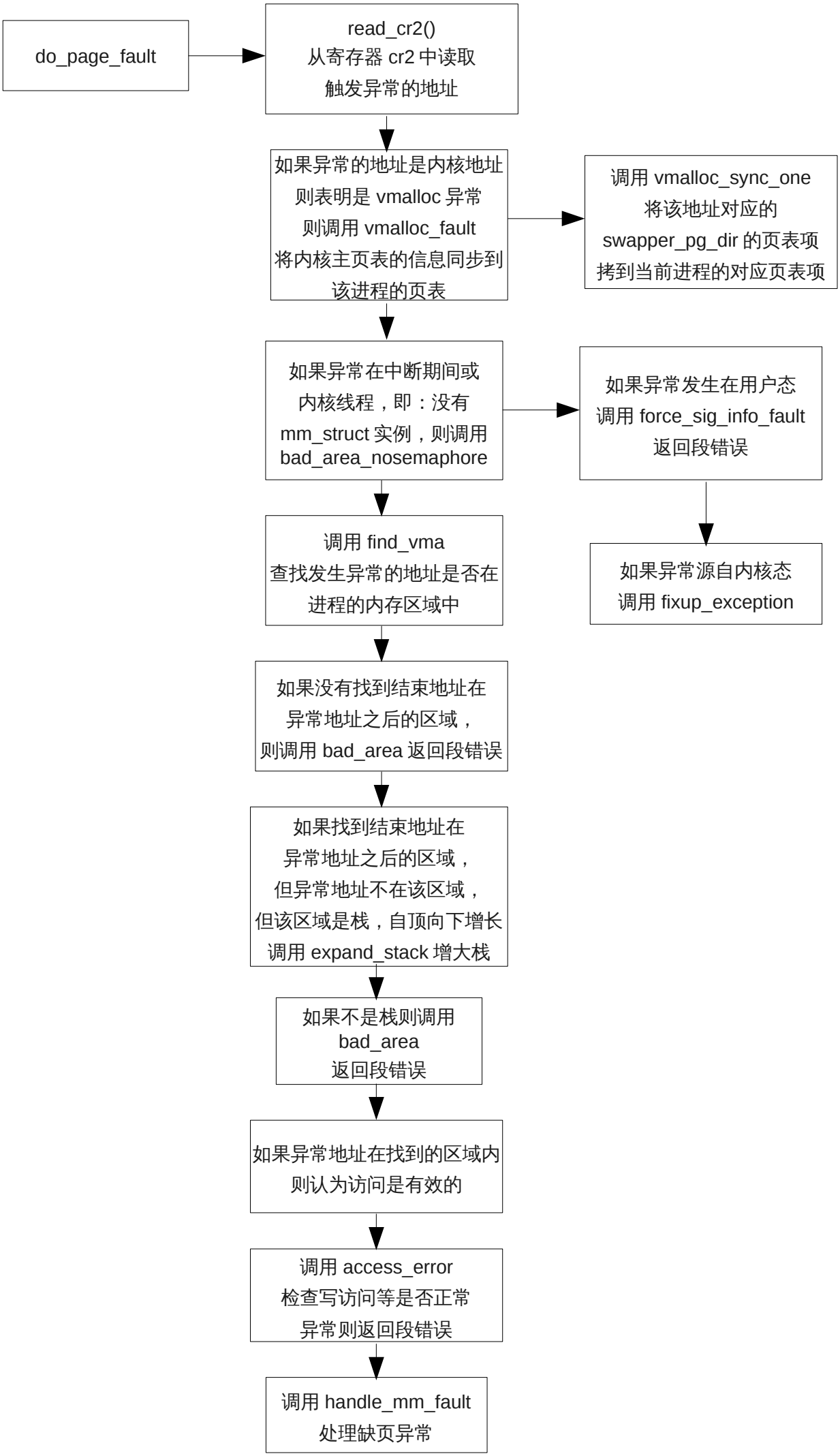
do_page_fault(struct pt_regs *regs, unsigned long error_code)

传入的错误码 error_code 的语义：

只使用了前 5 个比特位

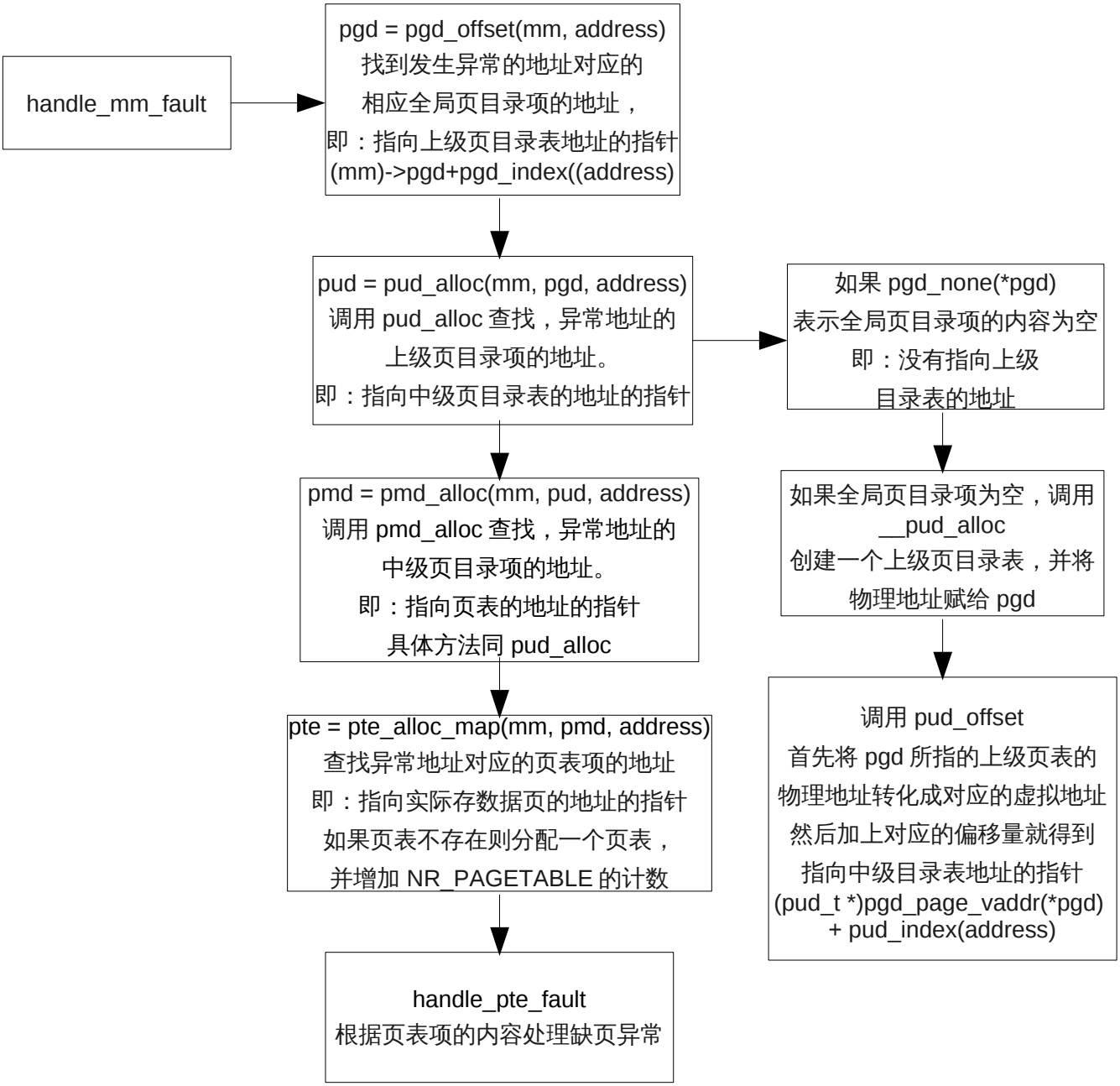
| 比特位 | 置为（1） | 置为（0） |
| --- | --- | --- |
| 0 | 保护异常(没有足够的访问权限) | 缺页 |
| 1 | 写访问 | 读访问 |
| 2 | 用户态 | 核心态 |
| 3 | 表示检测到使用了保留位 | |
| 4 | 表示缺页异常在取指令时出现的 | |

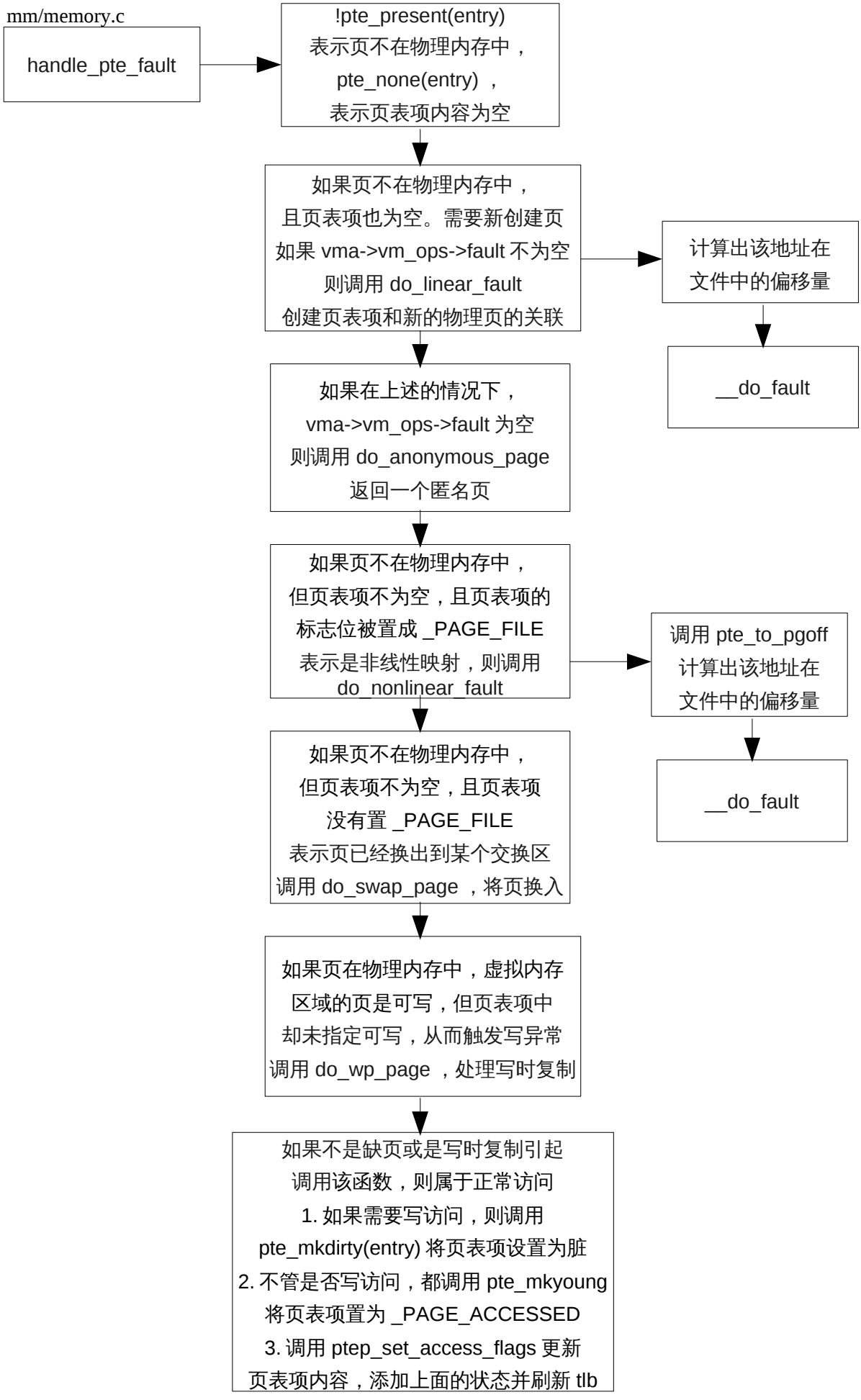
arch/x86/mm/fault.c

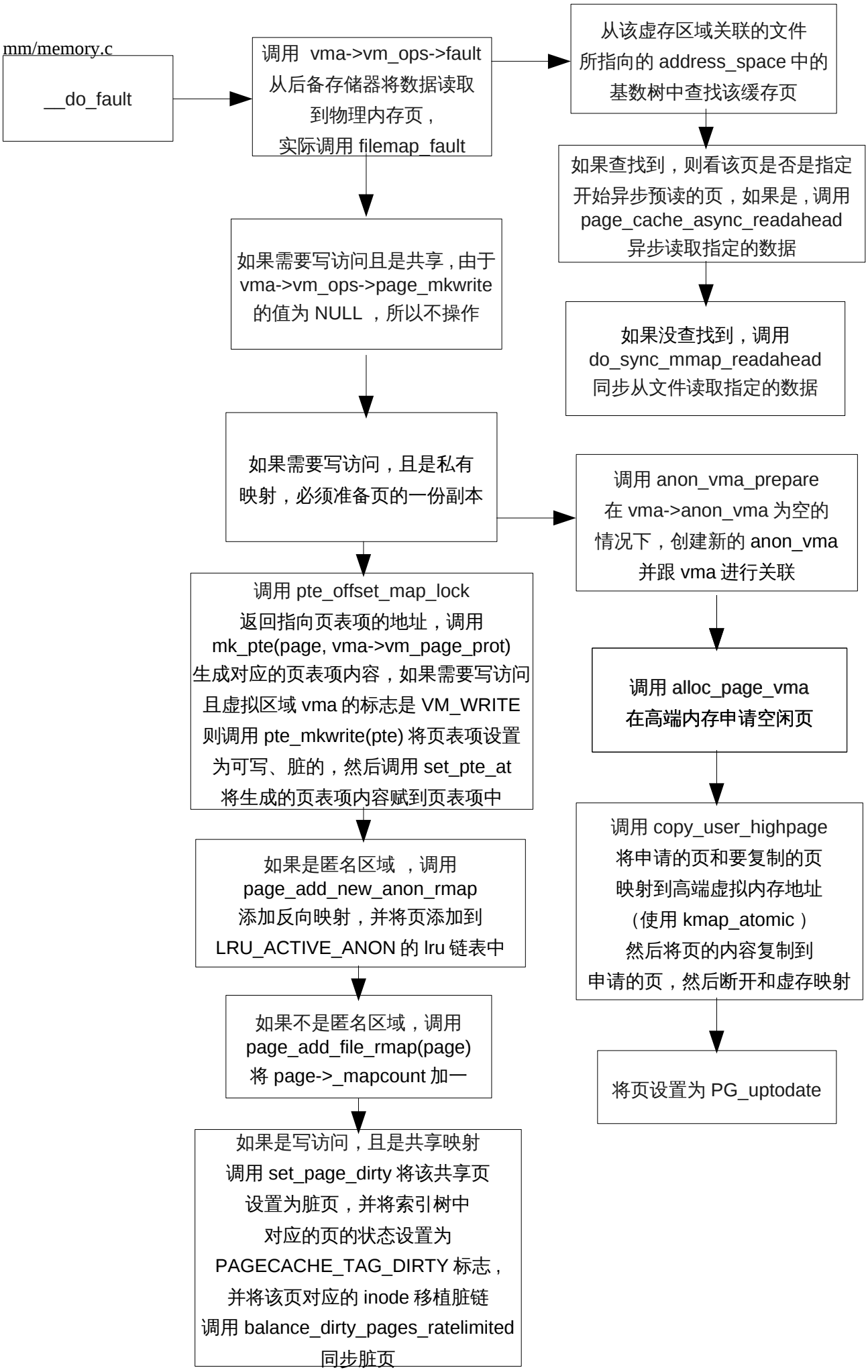

do_page_fault
read_cr2()
从寄存器 cr2 中读取
触发异常的地址
如果异常的地址是内核地址
则表明是 vmalloc 异常
则调用 vmalloc_fault
将内核主页表的信息同步到
该进程的页表
调用 vmalloc_sync_one
将该地址对应的
swapper_pg_dir 的页表项
拷到当前进程的对应页表项
如果异常在中断期间或
内核线程，即：没有
mm_struct 实例，则调用
bad_area_nosemaphore
如果异常发生在用户态
调用 force_sig_info_fault
返回段错误
调用 find_vma
查找发生异常的地址是否在
进程的内存区域中
如果异常源自内核态
调用 fixup_exception
如果没有找到结束地址在
异常地址之后的区域，
则调用 bad_area 返回段错误
如果找到结束地址在
异常地址之后的区域，
但异常地址不在该区域，
但该区域是栈，自顶向下增长
调用 expand_stack 增大栈
如果不是栈则调用
bad_area
返回段错误
如果异常地址在找到的区域内
则认为访问是有效的
调用 access_error
检查写访问等是否正常
异常则返回段错误
调用 handle_mm_fault
处理缺页异常

# 1.用户空间的缺页异常

mm/memory.c




mm/memory.c
handle_pte_fault
!pte_present(entry)
表示页不在物理内存中，
pte_none(entry) ，
表示页表项内容为空
如果页不在物理内存中，
且页表项也为空。需要新创建页
如果 vma->vm_ops->fault 不为空
则调用 do_linear_fault
创建页表项和新的物理页的关联
计算出该地址在
文件中的偏移量
__do_fault
如果在上述的情况下，
vma->vm_ops->fault 为空
则调用 do_anonymous_page
返回一个匿名页
如果页不在物理内存中，
但页表项不为空，且页表项的
标志位被置成 _PAGE_FILE
表示是非线性映射，则调用
do_nonlinear_fault
调用 pte_to_pgoff
计算出该地址在
文件中的偏移量
__do_fault
如果页不在物理内存中，
但页表项不为空，且页表项
没有置 _PAGE_FILE
表示页已经换出到某个交换区
调用 do_swap_page ，将页换入
如果页在物理内存中，虚拟内存
区域的页是可写，但页表项中
却未指定可写，从而触发写异常
调用 do_wp_page ，处理写时复制
如果不是缺页或是写时复制引起
调用该函数，则属于正常访问
1. 如果需要写访问，则调用
pte_mkdirty(entry) 将页表项设置为脏
2. 不管是否写访问，都调用 pte_mkyoung
将页表项置为 _PAGE_ACCESSED
3. 调用 ptep_set_access_flags 更新
页表项内容，添加上面的状态并刷新 tlb

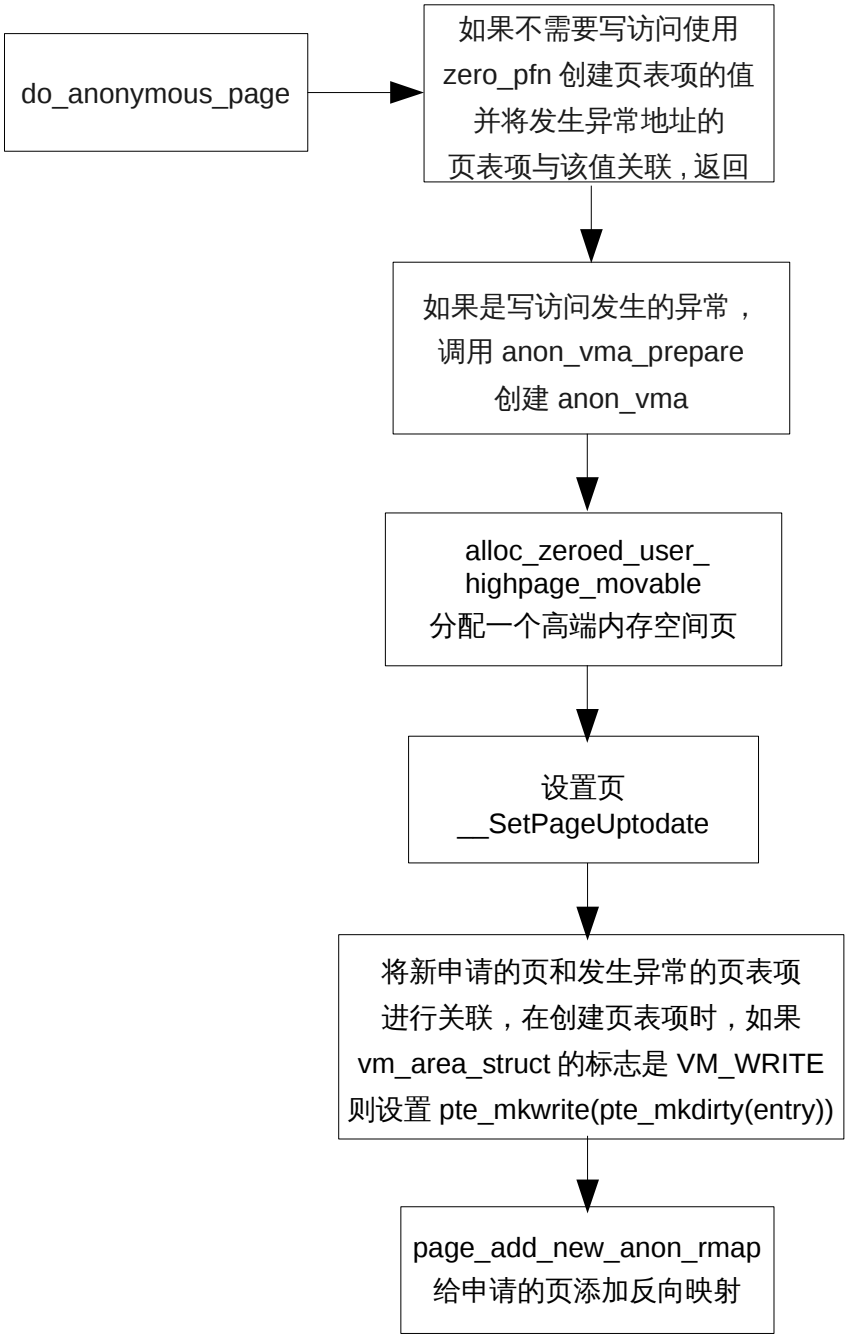

mm/memory.c
__do_fault
调用 vma->vm_ops->fault
从后备存储器将数据读取
到物理内存页，
实际调用 filemap_fault
从该虚存区域关联的文件
所指向的 address_space 中的
基数树中查找该缓存页
如果查找到，则看该页是否是指定
开始异步预读的页，如果是，调用
page_cache_async_readahead
异步读取指定的数据
如果没查找到，调用
do_sync_mmap_readahead
同步从文件读取指定的数据
如果需要写访问且是共享，由于
vma->vm_ops->page_mkwrite
的值为 NULL ，所以不操作
如果需要写访问，且是私有
映射，必须准备页的一份副本
调用 anon_vma_prepare
在 vma->anon_vma 为空的
情况下，创建新的 anon_vma
并跟 vma 进行关联
调用 pte_offset_map_lock
返回指向页表项的地址，调用
mk_pte(page, vma->vm_page_prot)
生成对应的页表项内容，如果需要写访问
且虚拟区域 vma 的标志是 VM_WRITE
则调用 pte_mkwrite(pte) 将页表项设置
为可写、脏的，然后调用 set_pte_at
将生成的页表项内容赋到页表项中
调用 alloc_page_vma
在高端内存申请空闲页
调用 copy_user_highpage
将申请的页和要复制的页
映射到高端虚拟内存地址
（使用 kmap_atomic ）
然后将页的内容复制到
申请的页，然后断开和虚存映射
如果是匿名区域 ，调用
page_add_new_anon_rmap
添加反向映射，并将页添加到
LRU_ACTIVE_ANON 的 lru 链表中
如果不是匿名区域，调用
page_add_file_rmap(page)
将 page->_mapcount 加一
将页设置为 PG_uptodate
如果是写访问，且是共享映射
调用 set_page_dirty 将该共享页
设置为脏页，并将索引树中
对应的页的状态设置为
PAGECACHE_TAG_DIRTY 标志，
并将该页对应的 inode 移植脏链
调用 balance_dirty_pages_ratelimited
同步脏页

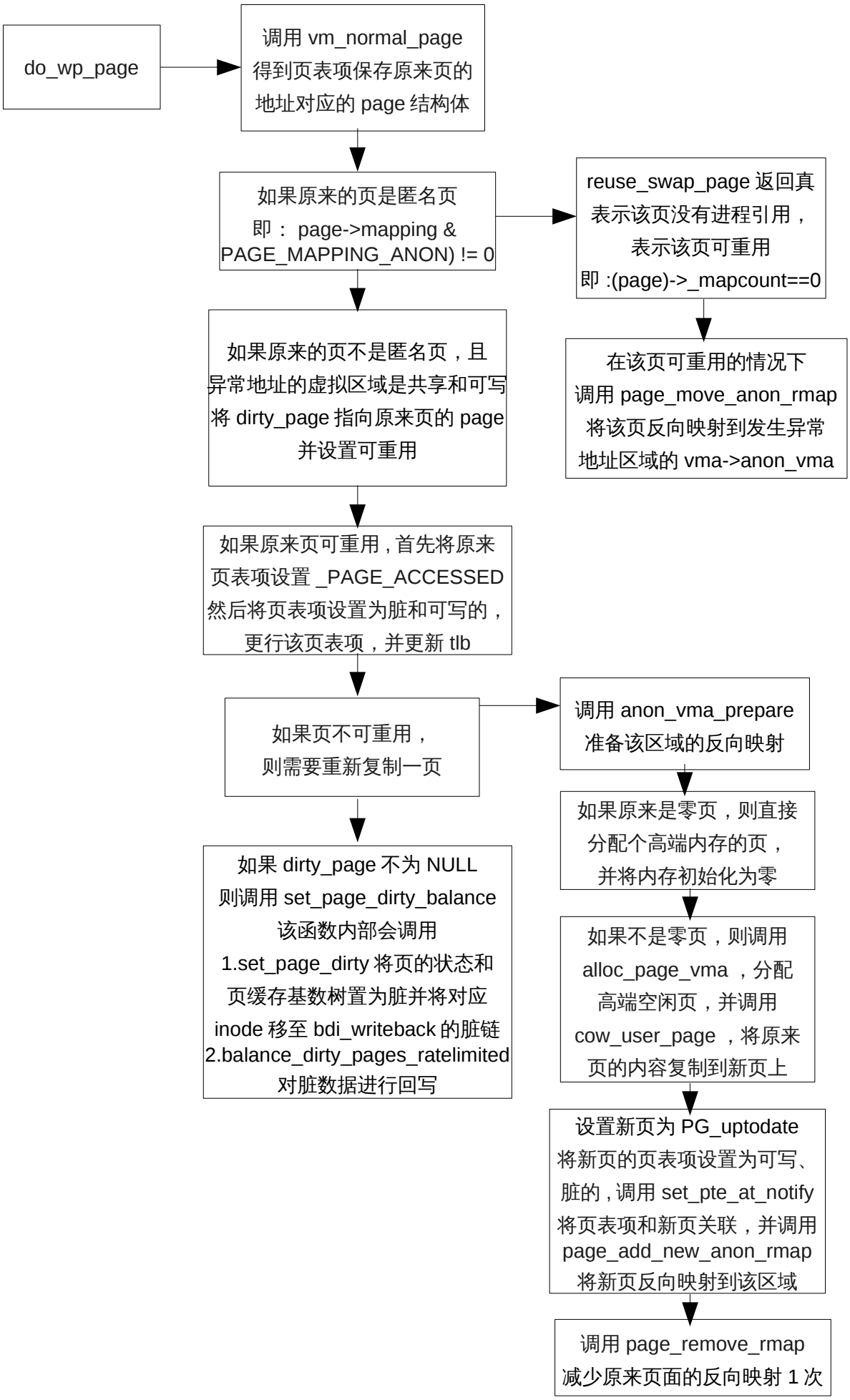
**内存页反向映射到匿名区域：**

```
void page_add_new_anon_rmap(struct page *page, struct vm_area_struct *vma, unsigned long address)
{
    VM_BUG_ON(address < vma->vm_start || address >= vma->vm_end);
    SetPageSwapBacked(page);
    atomic_set(&page->_mapcount, 0); /* increment count (starts at -1) */
    __inc_zone_page_state(page, NR_ANON_PAGES);
    __page_set_anon_rmap(page, vma, address, 1);
    if (page_evictable(page, vma))
        lru_cache_add_lru(page, LRU_ACTIVE_ANON);
    else
        add_page_to_unevictable_list(page);
}
static void __page_set_anon_rmap(struct page *page, struct vm_area_struct *vma, unsigned long address, int
exclusive)
{
    struct anon_vma *anon_vma = vma->anon_vma;
.....
    anon_vma = (void *) anon_vma + PAGE_MAPPING_ANON;
    page->mapping = (struct address_space *) anon_vma;
    page->index = linear_page_index(vma, address);
}
```

**内存页反向映射到文件映射区域：**

```
void page_add_file_rmap(struct page *page)
{
    if (atomic_inc_and_test(&page->_mapcount)) {
        __inc_zone_page_state(page, NR_FILE_MAPPED);
        mem_cgroup_update_file_mapped(page, 1);
    }
}
```


do_anonymous_page
如果不需要写访问使用
zero_pfn 创建页表项的值
并将发生异常地址的
页表项与该值关联，返回
如果是写访问发生的异常，
调用 anon_vma_prepare
创建 anon_vma
alloc_zeroed_user_
highpage_movable
分配一个高端内存空间页
设置页
__SetPageUptodate
将新申请的页和发生异常的页表项
进行关联，在创建页表项时，如果
vm_area_struct 的标志是 VM_WRITE
则设置 pte_mkwrite(pte_mkdirty(entry))
page_add_new_anon_rmap
给申请的页添加反向映射


do_wp_page
调用 vm_normal_page
得到页表项保存原来页的
地址对应的 page 结构体
如果原来的页是匿名页
即： page->mapping &
PAGE_MAPPING_ANON) != 0
reuse_swap_page 返回真
表示该页没有进程引用，
表示该页可重用
即 :(page)->_mapcount==0
在该页可重用的情况下
调用 page_move_anon_rmap
将该页反向映射到发生异常
地址区域的 vma->anon_vma
如果原来的页不是匿名页，且
异常地址的虚拟区域是共享和可写
将 dirty_page 指向原来页的 page
并设置可重用
如果原来页可重用，首先将原来
页表项设置 _PAGE_ACCESSED
然后将页表项设置为脏和可写的，
更行该页表项，并更新 tlb
如果页不可重用，
则需要重新复制一页
调用 anon_vma_prepare
准备该区域的反向映射
如果原来是零页，则直接
分配个高端内存的页，
并将内存初始化为零
如果不是零页，则调用
alloc_page_vma ，分配
高端空闲页，并调用
cow_user_page ，将原来
页的内容复制到新页上
设置新页为 PG_uptodate
将新页的页表项设置为可写、
脏的，调用 set_pte_at_notify
将页表项和新页关联，并调用
page_add_new_anon_rmap
将新页反向映射到该区域
调用 page_remove_rmap
减少原来页面的反向映射 1 次
如果 dirty_page 不为 NULL
则调用 set_page_dirty_balance
该函数内部会调用
1.set_page_dirty 将页的状态和
页缓存基数树置为脏并将对应
inode 移至 bdi_writeback 的脏链
2.balance_dirty_pages_ratelimited
对脏数据进行回写

## 2.内核缺页异常

Each time a page fault occurs, the cause of the fault and the address in the code currently executing are output. This enables the kernel to compile a list of all risky chunks of code that may carry out unauthorized memory access operations. This ‘‘exception table‘‘ is created when the kernel image is linked and is located between __start_exception_table and __end_exception_table in the binary file. Each entry corresponds to an instance of struct exception_table, which, although architecture-dependent, is almost always structured as follows:

arch/x86/include/asm/uaccess.h

struct exception_table_entry

{

unsigned long insn, fixup;

};

**insn**:specifies the position in virtual address space at which the kernel expects the fault;

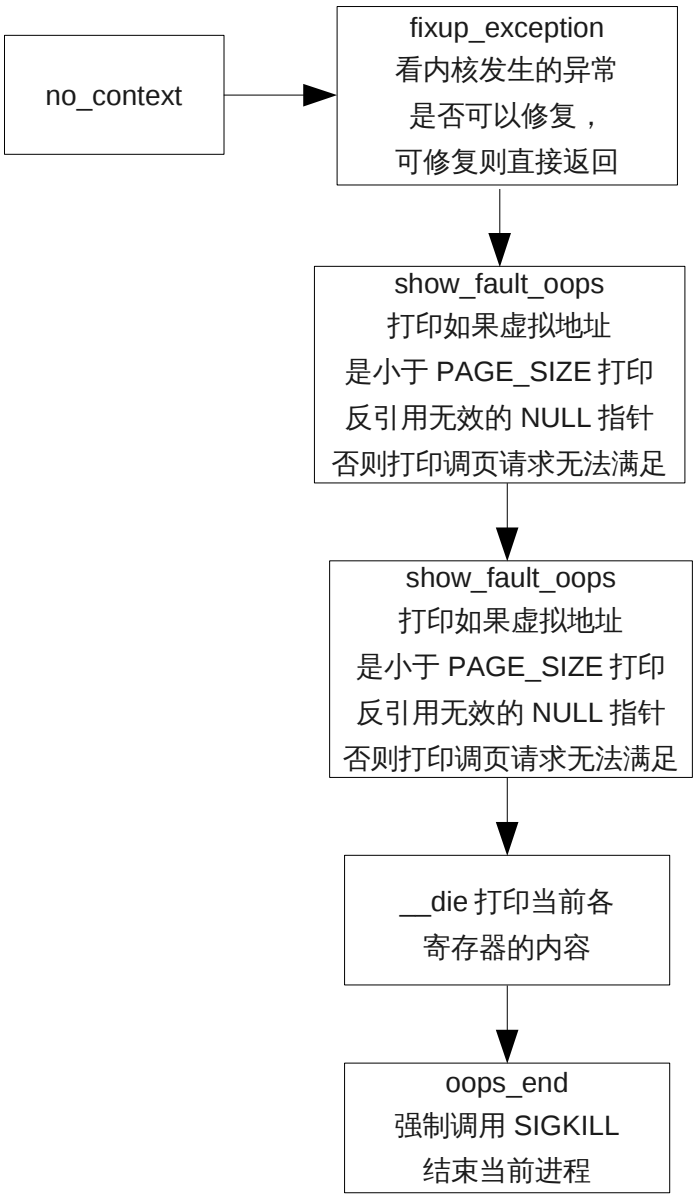
**fixup:** is the code address at which execution resumes when the fault occurs.

fixup_exception 用于搜索异常表在__start_exception_table 和 __end_exception_table 之间，当 regs->eip 找到对应的 insn,将 fixup 赋给 regs->eip 进行修复。

```
int fixup_exception(struct pt_regs *regs)
{
const struct exception_table_entry *fixup;
.....
fixup = search_exception_tables(regs->ip);
    if (fixup) {
        /* If fixup is less than 16, it means uaccess error */
        if (fixup->fixup < 16) {
            current_thread_info()->uaccess_err = -EFAULT;
            regs->ip += fixup->fixup;
            return 1;
        }
        regs->ip = fixup->fixup;
        return 1;
    }
    return 0;
}
}
```

内核缺页异常都会追溯到 no_context 函数。

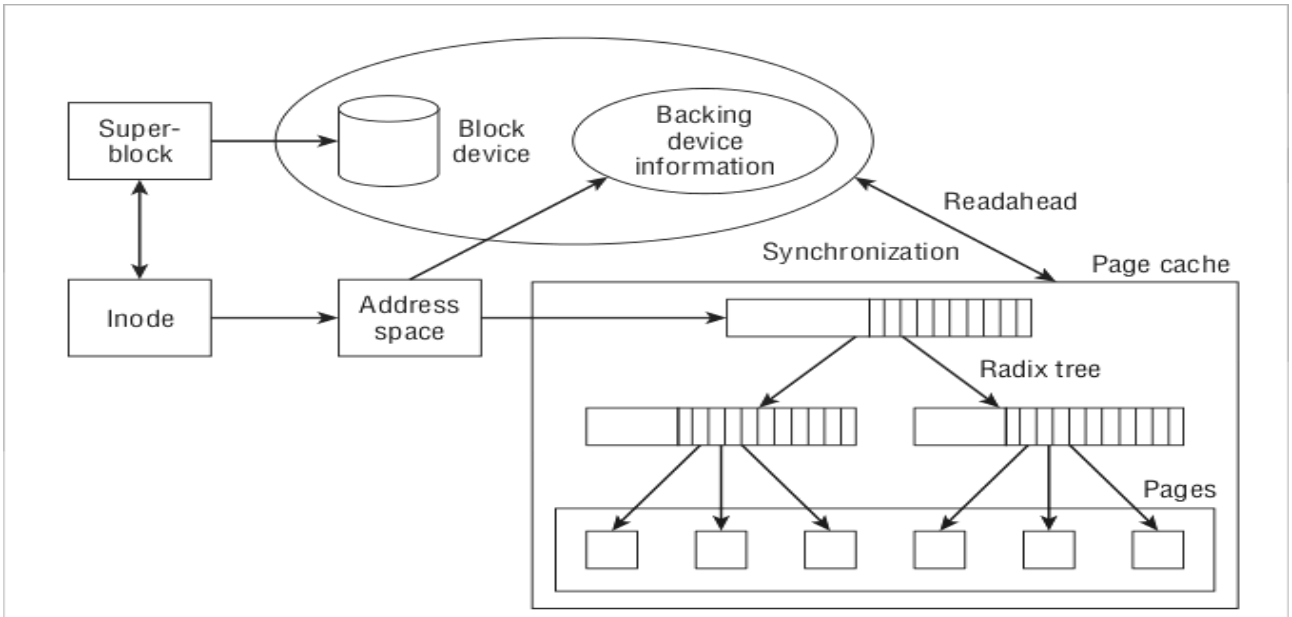
/arch/x86/mm/fault.c



# 四.页缓存和块缓存

## *1.页缓存*

## 1.地址空间的数据结构




linux/fs.h

```
struct address_space {
    struct inode        *host;      /* owner: inode, block_device */
    struct radix_tree_root  page_tree;  /* radix tree of all pages */
    spinlock_t      tree_lock;  /* and lock protecting it */
    unsigned int        i_mmap_writable;/* count VM_SHARED mappings */
    struct prio_tree_root   i_mmap;     /* tree of private and shared mappings */
    struct list_head    i_mmap_nonlinear;/*list VM_NONLINEAR mappings */
    spinlock_t      i_mmap_lock;    /* protect tree, count, list */
    unsigned int        truncate_count; /* Cover race condition with truncate */
    unsigned long       nrpages;    /* number of total pages */
    pgoff_t         writeback_index;/* writeback starts here */
    const struct address_space_operations *a_ops;   /* methods */
    unsigned long       flags;      /* error bits/gfp mask */
    struct backing_dev_info *backing_dev_info; /* device readahead, etc */
    spinlock_t      private_lock;   /* for use by the address_space */
```

```
    struct list_head    private_list;   /* ditto */
    struct address_space    *assoc_mapping; /* ditto */
} __attribute__((aligned(sizeof(long))));
```

部分元素解释：

(1)host: 指向 inode 实例的指针，指定了后备存储器。

(2)page_tree: 一个基数树的根列出了地址空间中所有的物理内存页。

(3)i_mmap:优先树的根包含了与该 inode 相关的所有普通内存映射，前面讨论过。

(4)i_mmap_writable:统计所有用 VM_SHARED 属性创建的映射。

(5)i_mmap_nonlinear:一个列表，包括所有包含在非线性映射中的页。

(6)nrpages:缓存页的总数。

(7)backing_dev_info:指向 backing_dev_info 结构，其中包含了与地址空间相关的后备存储器的相关信息。

(8)private_list：用于将包含文件系统元数据的 buffer_head 实例彼此连接起来。

(9)flags:The flag is used primarily to hold information on the GFP memory area from which the mapped pages originate. It can also hold errors that occur during asynchronous input/output and that cannot therefore be propagated directly. AS_EIO stands for a general I/O error, and AS_ENOSPC indicates that there is no longer sufficient space for an asynchronous write operation .

(10)a_ops:地址空间将后备存储器与内存区域关联起来，在两者之间传输数据，不仅需要数据结构，还需要相应的函数，该成员指向地址空间的操作。

```
struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    void (*sync_page)(struct page *);
    /* Write back some dirty pages from this mapping. */
    int (*writepages)(struct address_space *, struct writeback_control *);
    /* Set a page dirty.  Return true if this dirtied it */
    int (*set_page_dirty)(struct page *page);
    int (*readpages)(struct file *filp, struct address_space *mapping,  struct list_head *pages, unsigned nr_pages);
    /* Unfortunately this kludge is needed for FIBMAP. Don't use it */
    sector_t (*bmap)(struct address_space *, sector_t);
    void (*invalidatepage) (struct page *, unsigned long);
    int (*releasepage) (struct page *, gfp_t);
    ssize_t (*direct_IO)(int, struct kiocb *, const struct iovec *iov, loff_t offset, unsigned long nr_segs);
    /* migrate the contents of a page to the specified target */
    int (*migratepage) (struct address_space *, struct page *, struct page *);
```

int (*launder_page) (struct page *);

.....

};

❑ writepage and writepages： write one or more pages of the address space back to the underlying block device. This is done by delegating a corresponding request to the block layer.

❑ readpage and readpages： read one or more consecutive pages from the backing store into a page frame.

❑ sync_page： performs synchronization of data that have not yet been written back to the backing store. Unlike writepage, the function operates on block layer level and attempts to perform pending write requests still held in buffers in this layer. In contrast, writepage operates on the address space layer and simply forwards the data to the block layer without bothering about active buffering there.

❑ set_page_dirty： allows an address space to provide a specific method of marking a page as dirty. However, this option is rarely used. In this case, the kernel automatically uses code__set_page_dirty_buffers to simultaneously mark the page as dirty on the buffer level and to add it to the dirty_pages list of the current mapping.

❑ bmap： maps a logical block offset within an address space to a physical block number. This is usually straightforward for block devices, but since files are in general not represented by a linear number of blocks on a device, the required information cannot be determined otherwise.

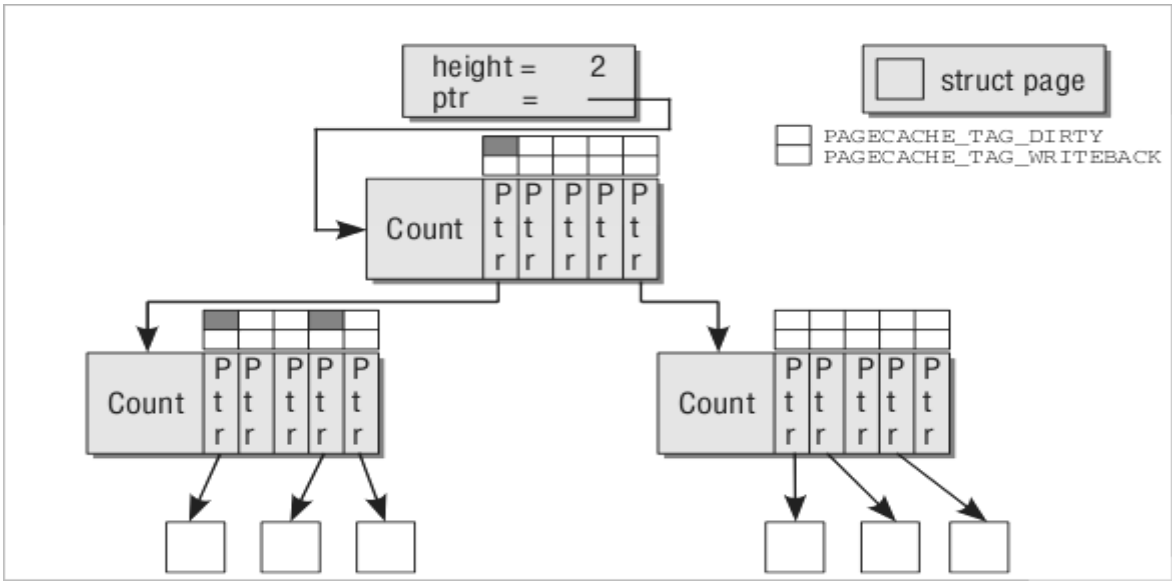
❑ invalidatepage： is called if a page is going to be removed from the address space and buffers are associated with it as signalized by the PG_Private flag.

❑ direct_IO is used to implement direct read and write access. This bypasses buffering in the block layer and allows an application to communicate very directly with a block device. Large databases make frequent use of this feature as they are better able to forecast future input and output than the generic mechanisms of the kernel and can therefore achieve better results by implementing their own caching mechanisms.

❑ migrate_page is used if the kernel wants to relocate a page, that is, move contents of one page onto another page. Since pages are often equipped with private data, it is not just sufficient to copy the raw information from the old to the new page. Moving pages is, for instance, required to support memory hotplugging.

❑ launder_page offers a last chance to write back a dirty page before it is freed.

## 2.基数树



linux/radix-tree.h

```
struct radix_tree_root {
    unsigned int        height;
    gfp_t           gfp_mask;
    struct radix_tree_node  *rnode;
};
```

(1)height:指定了树的高度。

(2)gfp_mask:指定了从哪个内存域分配内存。

(3) rnode:指向树的第一个节点。

lib/radix-tree.c

```
struct radix_tree_node {
    unsigned int    height;     /* Height from the bottom */
    unsigned int    count;
    struct rcu_head rcu_head;
    void        *slots[RADIX_TREE_MAP_SIZE];
    unsigned long   tags[RADIX_TREE_MAX_TAGS][RADIX_TREE_TAG_LONGS];
};
```

(1)count:保存了该结点中已经使用的数组项的数目。

(2)slots:指向数据或其他节点， RADIX_TREE_MAP_SIZE 的大小根据配置可以为 16 或 64.

(3) tags:二维数组，第一维用于区分不同的标记，第二维包含足够数量的 unsigned long,使得对该结点中可能组织的每一个页，都能分配到一个比特位。

当前支持三种标记：

```
#define PAGECACHE_TAG_DIRTY 0     //指定页是否是脏的
#define PAGECACHE_TAG_WRITEBACK 1 //表示该页当前正在回写
#define PAGECACHE_TAG_TOWRITE   2
```

为确保基数树的快速操作，内核使用一个独立的 slab 缓存来保存 radix_tree_node 的实例，以便快速分配此类型的结构实例。

每个基数树都还有一个 cpu 池，其中存放了分配的结点，以便进一步加速向树插入新数据项的操作。
radix_tree_preload 函数：

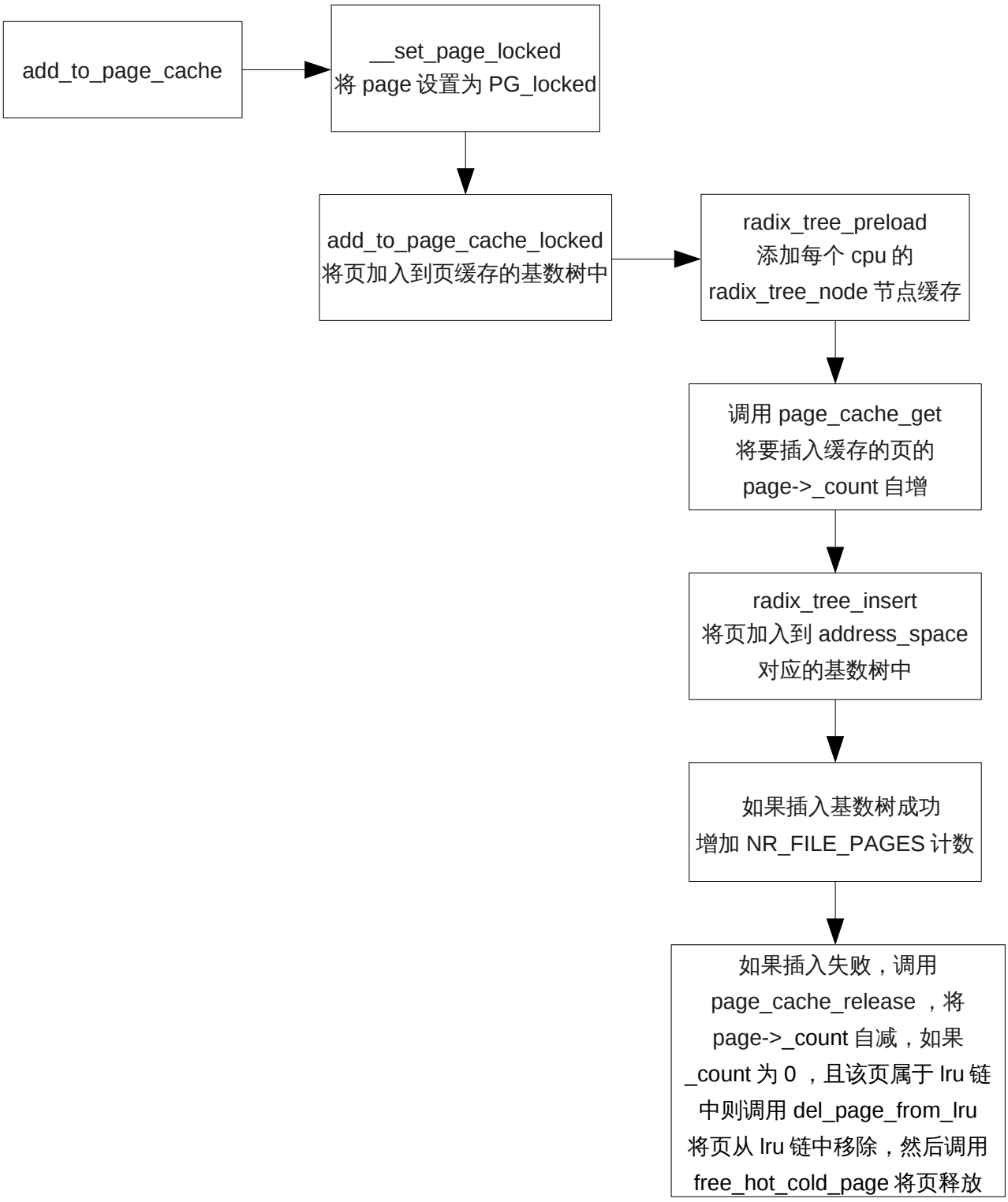
```
struct radix_tree_preload {
int nr;
struct radix_tree_node *nodes[RADIX_TREE_MAX_PATH];
};
```

```
int radix_tree_preload(gfp_t gfp_mask)
{
    struct radix_tree_preload *rtp;
    struct radix_tree_node *node;
    int ret = -ENOMEM;
    preempt_disable();
    rtp = &__get_cpu_var(radix_tree_preloads);
    while (rtp->nr < ARRAY_SIZE(rtp->nodes)) {
        preempt_enable();
        node = kmem_cache_alloc(radix_tree_node_cachep, gfp_mask);
        if (node == NULL)
            goto out;
        preempt_disable();
        rtp = &__get_cpu_var(radix_tree_preloads);
        if (rtp->nr < ARRAY_SIZE(rtp->nodes))
            rtp->nodes[rtp->nr++] = node;
        else
            kmem_cache_free(radix_tree_node_cachep, node);
    }
    ret = 0;
out:
    return ret;
}
```

# 3.页缓存的实现

### *1.将页加入到页缓存*

linux/pagemap.h




### *2.在缓存中查找页*

mm/filemap.c

find_get_page 用于判断给定页是否已经在缓存，页缓存的粒度是单个页。而文件中的位置是按字节偏移量指定的，而非页缓存中的偏移量。

Currently, the granularity of the page cache is a single page; that is, the leaf elements of the page cache radix tree are single pages. Future kernels might, however, increase the granularity, so assuming a page size granularity is not valid. Instead, the macro PAGE_CACHE_SHIFT is provided. The object size for a page cache element can be computed by 2PAGE_CACHE_SHIFT.

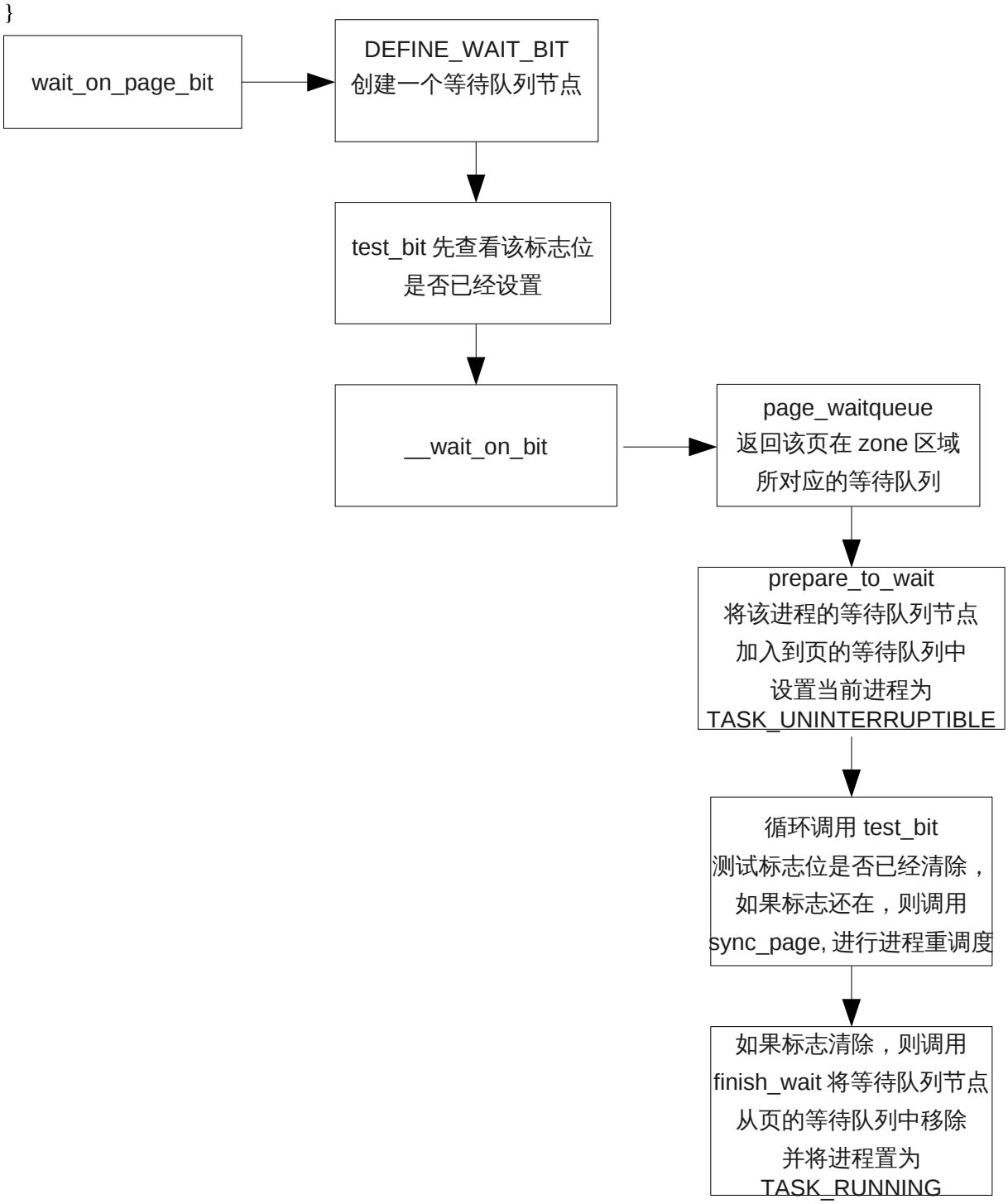
Converting between byte offsets in a file and page cache offsets is then a simple matter of dividing the index by PAGE_CACHE_SHIFT:

index = ppos >> PAGE_CACHE_SHIFT;

ppos is a byte offset into a file, and index contains the corresponding page cache offset.

### *3.在页上等待*

```
static inline void wait_on_page_writeback(struct page *page)
{
    if (PageWriteback(page))
        wait_on_page_bit(page, PG_writeback);
}
```

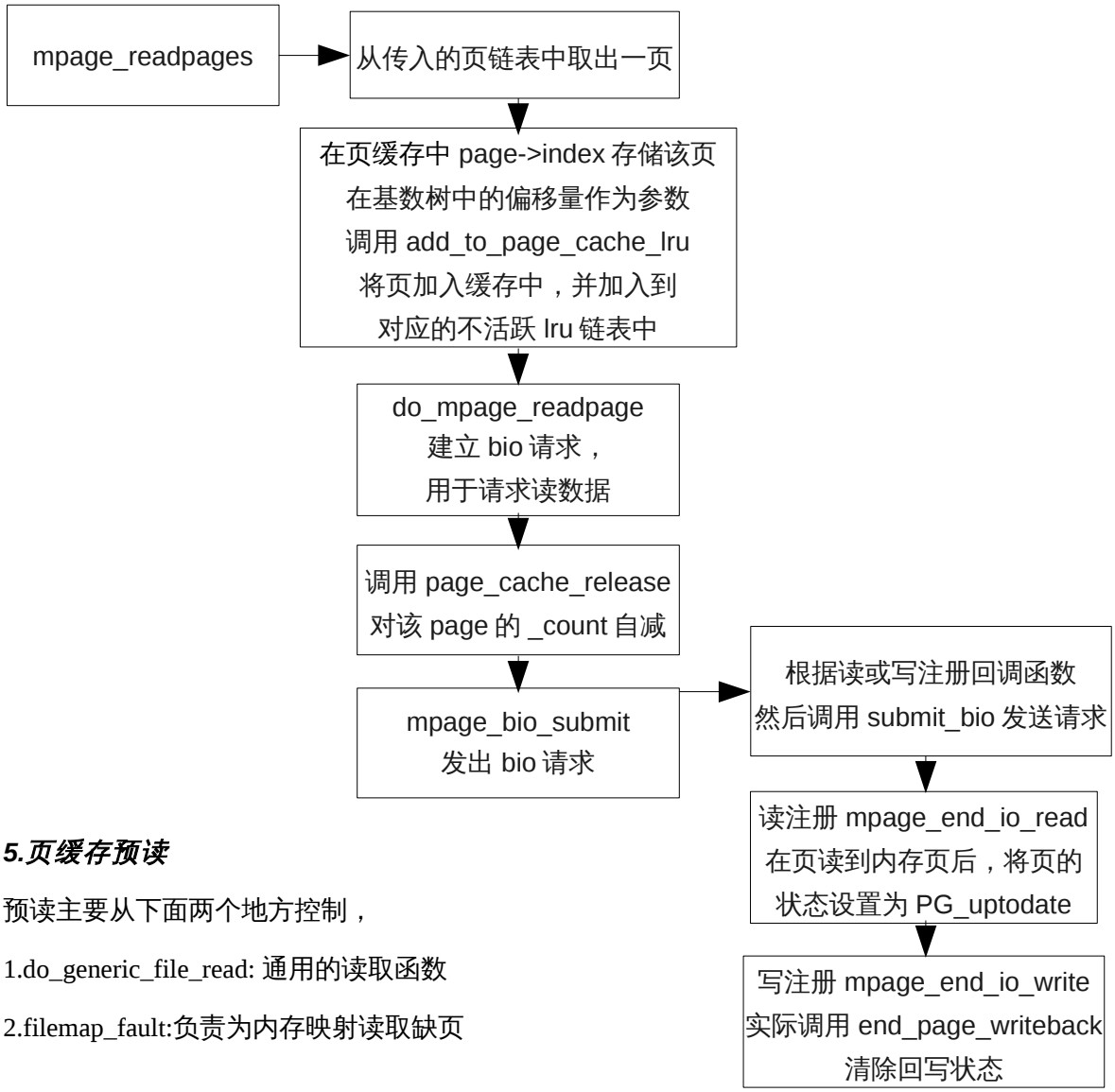


等待队列的数据结构：

```
struct __wait_queue {
    unsigned int flags;
#define WQ_FLAG_EXCLUSIVE   0x01
    void *private;
    wait_queue_func_t func;
    struct list_head task_list;
};

struct wait_bit_key {
    void *flags;
    int bit_nr;
};

struct wait_bit_queue {
    struct wait_bit_key key;
    wait_queue_t wait;
};

#define DEFINE_WAIT_BIT(name, word, bit)                \
    struct wait_bit_queue name = {                  \
        .key = __WAIT_BIT_KEY_INITIALIZER(word, bit),       \
        .wait   = {                     \
            .private    = current,          \
            .func       = wake_bit_function,        \
            .task_list  =               \
                LIST_HEAD_INIT((name).wait.task_list),  \
        },                      \
    }
```

### *4.读取页的操作*

address_space_operations 的 readpages 回调函数一般对应 mpage_readpages 函数，用于从后备设备中将数据读入到页缓存中。

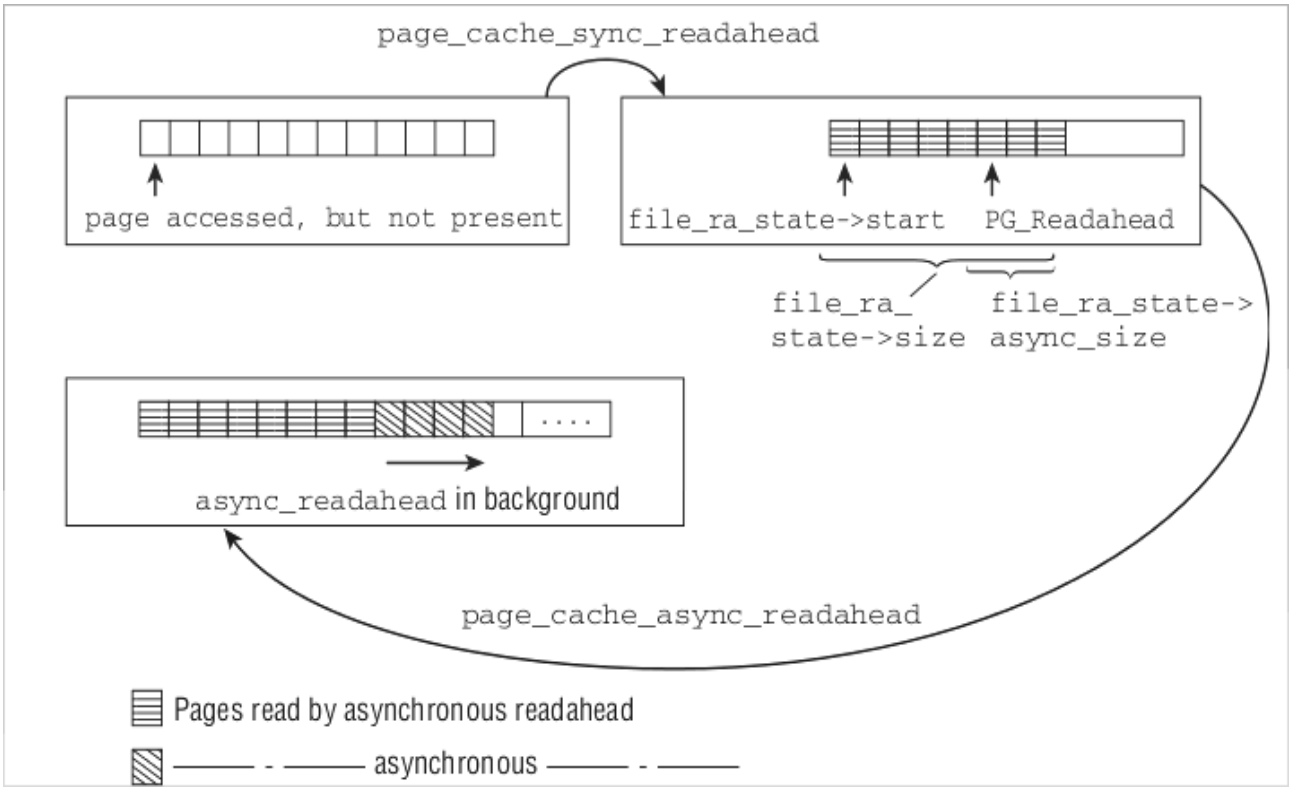



***5.页缓存预读***

预读主要从下面两个地方控制，

1.do_generic_file_read: 通用的读取函数

2.filemap_fault:负责为内存映射读取缺页



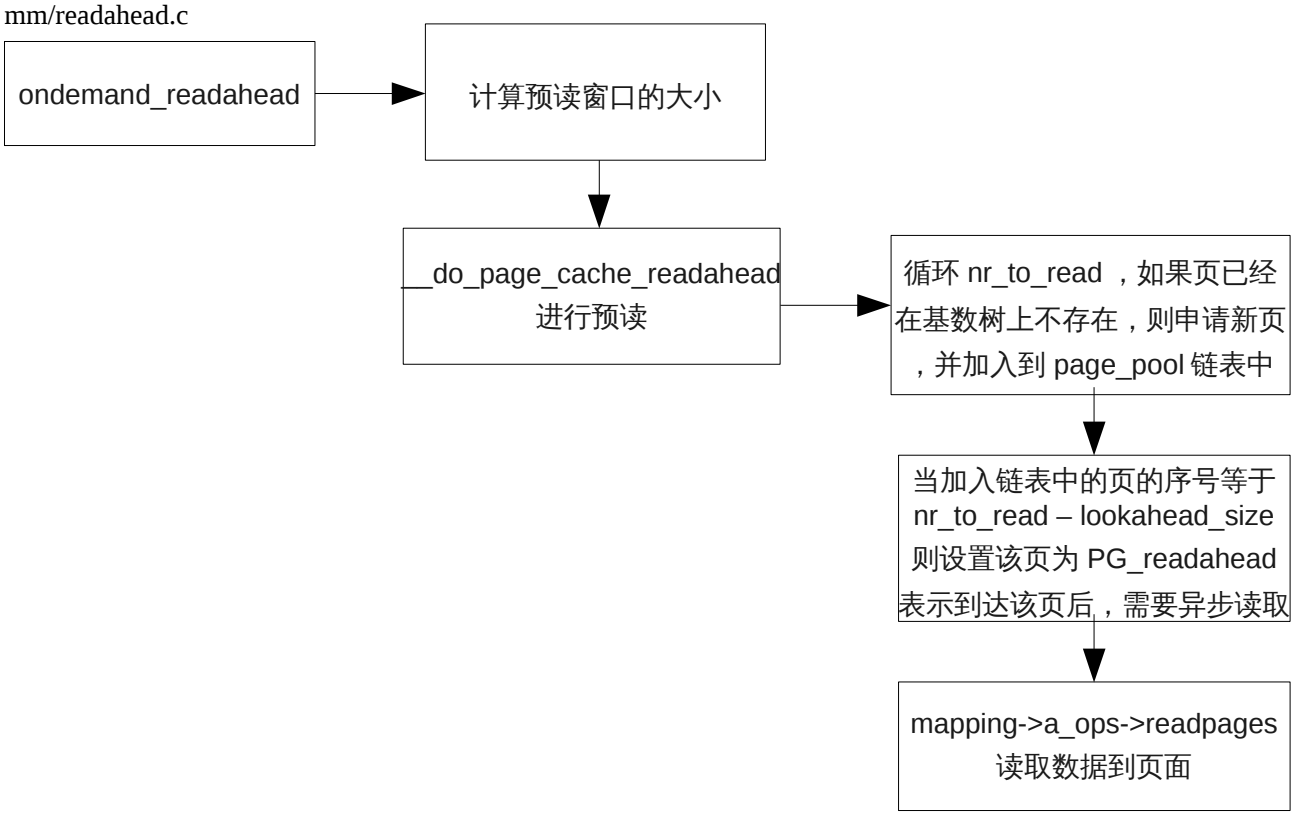
控制预读的结构体：

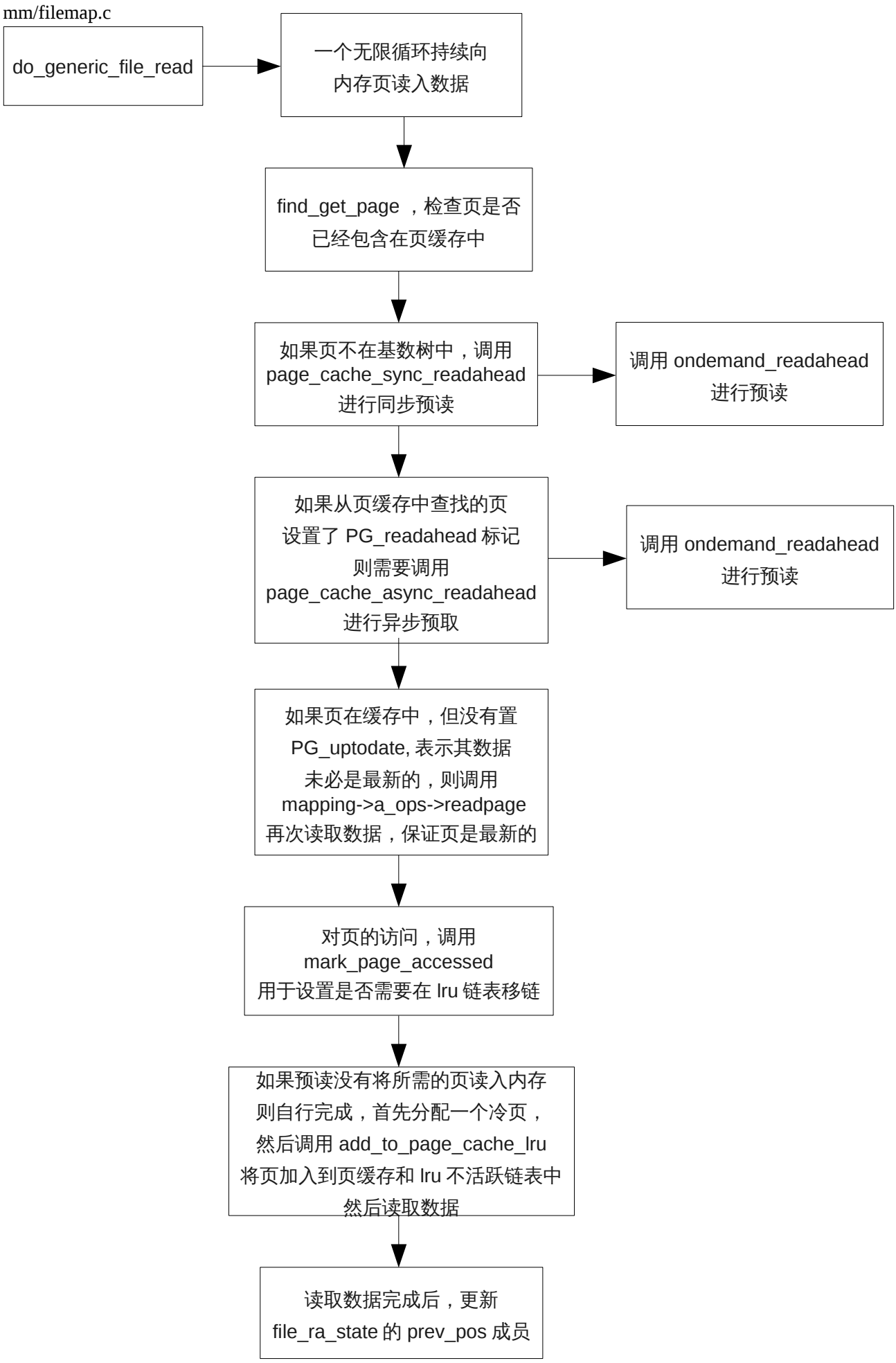
```
struct file_ra_state {
    pgoff_t start;          /* where readahead started */ 页缓存的开始位置
    unsigned int size;      /* # of readahead pages */ 预读窗口的长度
    unsigned int async_size;    /* do asynchronous readahead when there are only # of pages ahead */
                                当预读窗口剩余页数为该值时，启动异步预读
    unsigned int ra_pages;      /* Maximum readahead window */ 预读窗口的最大长度
    unsigned int mmap_miss;     /* Cache miss stat for mmap accesses */
    loff_t prev_pos;        /* Cache last read() position */ 上一次读取时，最后访问的位置
};
```

该结构体，包含在 struct file 结构体内

```
struct file {
......
const struct file_operations    *f_op;
struct file_ra_state    f_ra;
struct address_space    *f_mapping;
};
```

mm/readahead.c




mm/filemap.c
do_generic_file_read
一个无限循环持续向
内存页读入数据
find_get_page ，检查页是否
已经包含在页缓存中
如果页不在基数树中，调用
page_cache_sync_readahead
进行同步预读
调用 ondemand_readahead
进行预读
如果从页缓存中查找的页
设置了 PG_readahead 标记
则需要调用
page_cache_async_readahead
进行异步预取
调用 ondemand_readahead
进行预读
如果页在缓存中，但没有置
PG_uptodate, 表示其数据
未必是最新的，则调用
mapping->a_ops->readpage
再次读取数据，保证页是最新的
对页的访问，调用
mark_page_accessed
用于设置是否需要在 lru 链表移链
如果预读没有将所需的页读入内存
则自行完成，首先分配一个冷页，
然后调用 add_to_page_cache_lru
将页加入到页缓存和 lru 不活跃链表中
然后读取数据
读取数据完成后，更新
file_ra_state 的 prev_pos 成员

## 2.块缓存

### 1.缓存头的数据结构

```
struct buffer_head {
    unsigned long b_state;      /* buffer state bitmap (see above) */
    struct buffer_head *b_this_page;/* circular list of page's buffers */
    struct page *b_page;        /* the page this bh is mapped to */
    sector_t b_blocknr;     /* start block number */
    size_t b_size;          /* size of mapping */
    char *b_data;           /* pointer to data within the page */
    struct block_device *b_bdev;
    bh_end_io_t *b_end_io;      /* I/O completion */
    void *b_private;        /* reserved for b_end_io */
    struct list_head b_assoc_buffers; /* associated with another mapping */
    struct address_space *b_assoc_map;  /* mapping this buffer is associated with */
    atomic_t b_count;       /* users using this buffer_head */
}
```

(1)b_state: 表示缓冲头当前的状态

BH_Uptodate：如果缓冲区当前的数据与后备存储器一致，则置此位。

BH_Dirty：如果缓冲区中的数据已经修改，不在与后备存储器一致，则置此位。

BH_Lock:表示缓冲区被锁定，防止并发操作缓冲区。

BH_Mapped：表示存在一个缓冲区的内容到存储设备的映射。

BH_New：表示新创建的缓冲区。

(2)b_count: 实现访问计数器，以防内核释放仍然处于使用中的缓冲头。

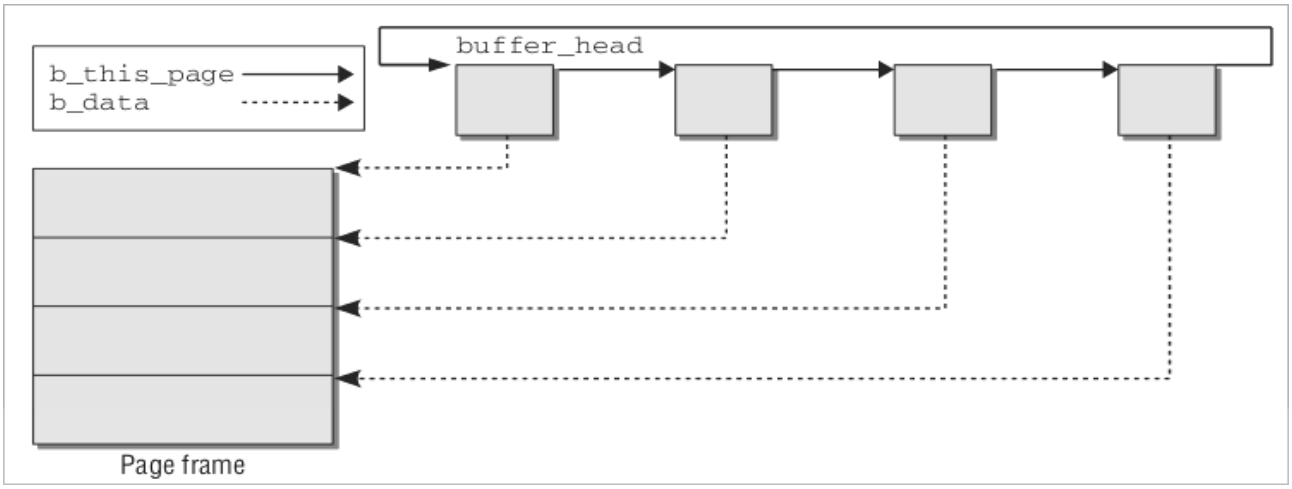
(3)b_page:  holds a pointer to a page instance with which the buffer head is associated when used in conjunction with the page cache. If the buffer is independent, b_page contains a null pointer 。

(4)b_this_page:在用几个缓冲区将一页划分为几个较小的单位，所有属于这些单位的缓冲头都保存在一个环形链表上。

(5)b_blocknr:保存了底层设备上的起始块号。

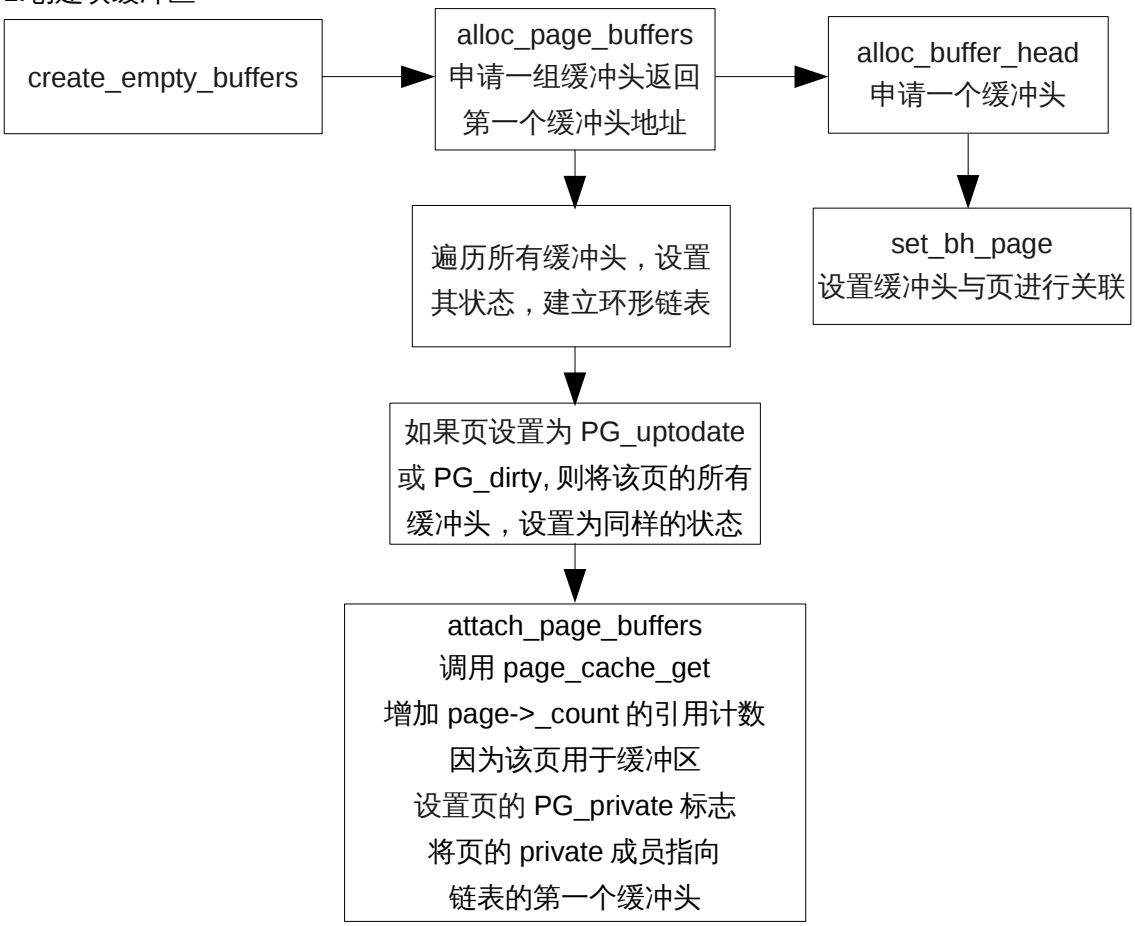
(6)b_data:指向内存中数据的地址。

### 2.页缓存和块缓存的交互




为支持页与缓冲区的交互，需要使用 struct page 的 private 成员。private 指向将页划分为更小单位的第一个缓冲头。各个缓冲头的 b_this_page 连接为一个环形的链表，这使得内核可以轻易的扫描与页关联的所有 buffer_head 实例。

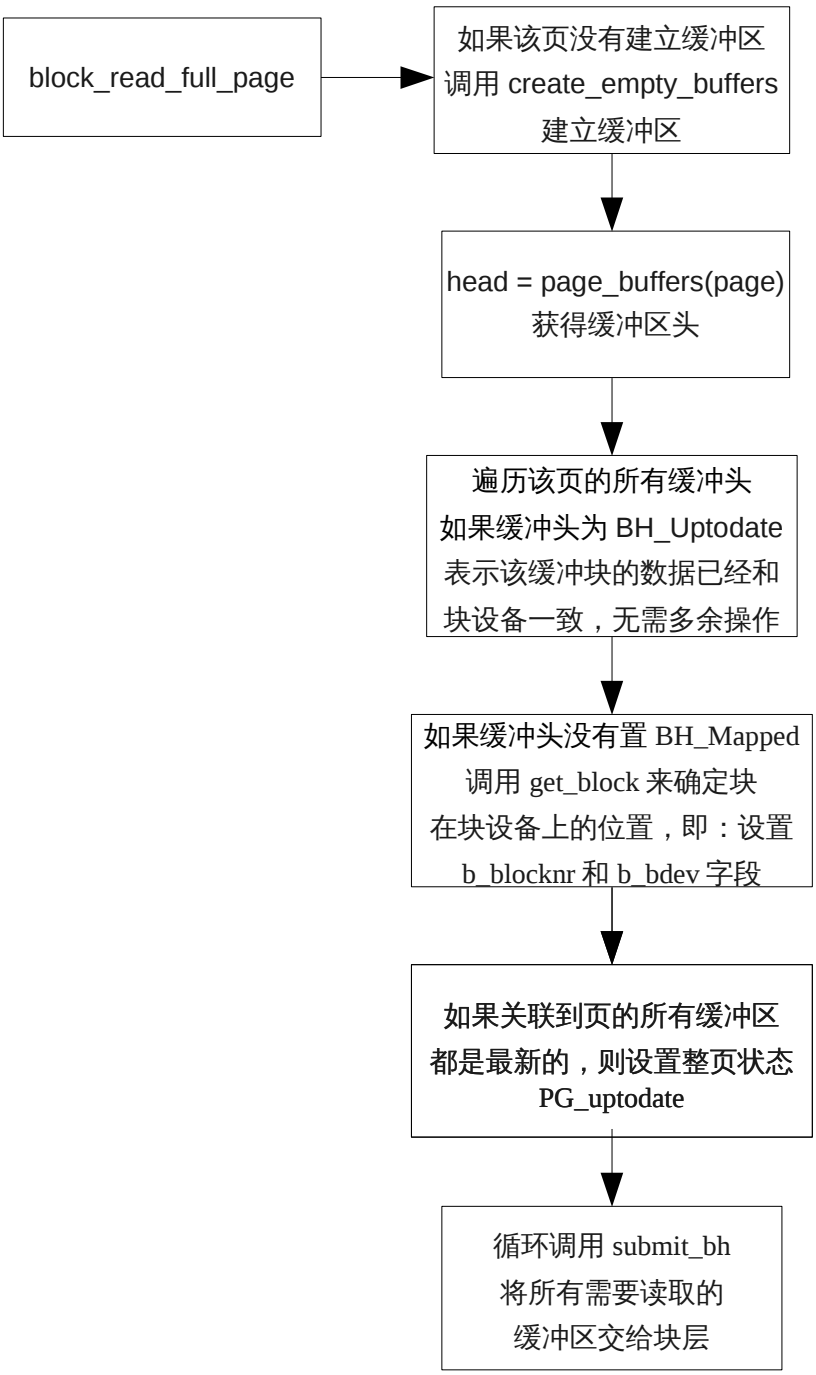
*1.*创建块缓冲区




*2.*缓冲区中读取整页

end_buffer_async_read，用于执行某个缓冲区的读操作结束时调用，检查页的所有其他缓冲区上的读操作是否也已经结束，如果结束，则将整页的状态设置为最新的。

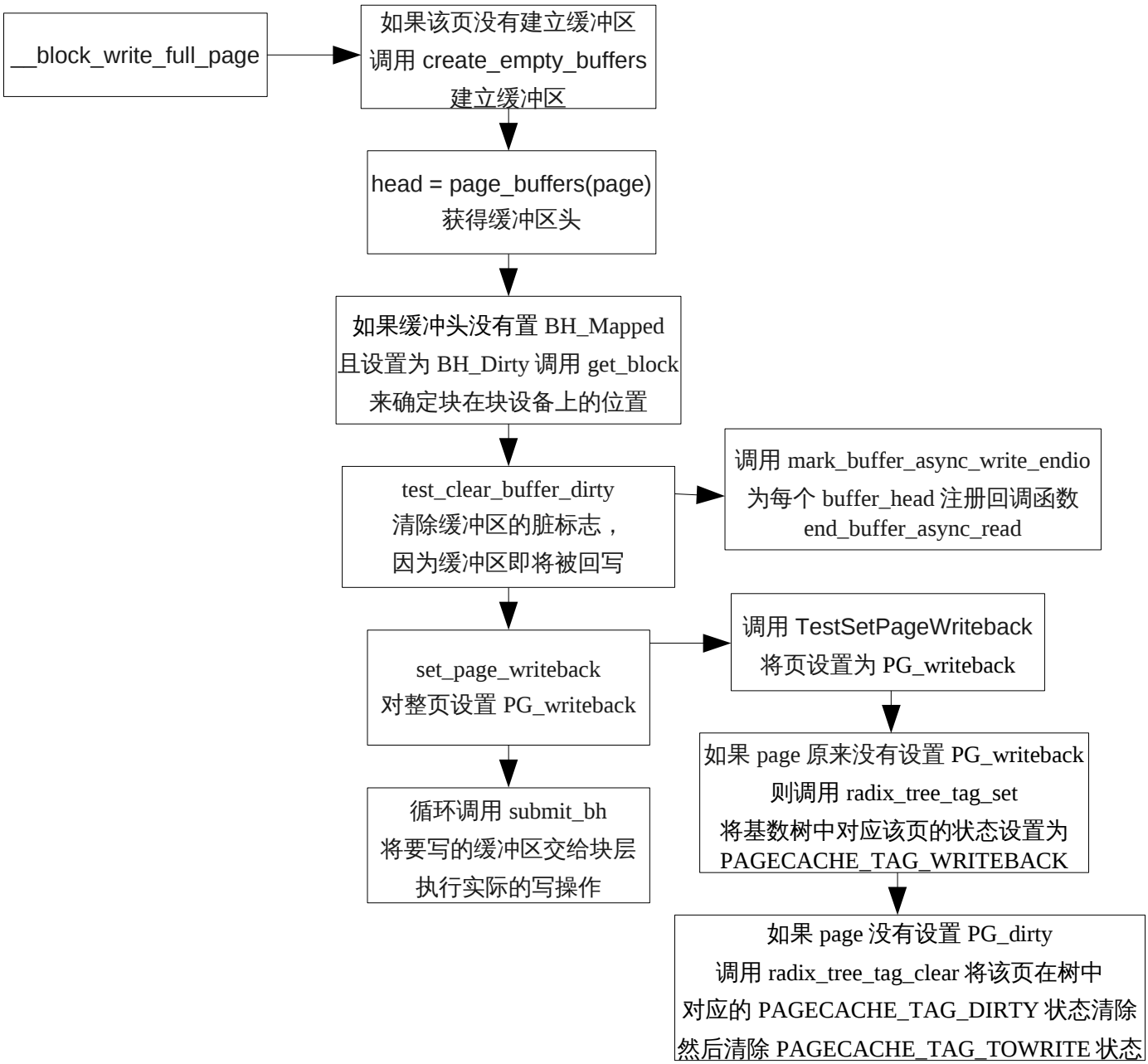
block_read_full_page 的优点在于，只需要读取页中不是最新的那些部分，但如果整页都不是最新的，则调用 mpage_readpage，避免缓冲区的多余开销。




*3.*将整页写到缓冲区

end_buffer_async_write，用于执行某个缓冲区的写操作结束时调用，检查页的所有其他缓冲区上的写操作是否也已经结束，如果结束，调用 end_page_writeback 清除回写标志，唤醒在于该页相关的队列上睡眠，等待此事件的所有进程。

该函数赋给 buffer_head 的 b_end_io 回调函数。




***3.独立的缓冲区(LRU 块缓存)***

在每次进行请求，需要查找一个独立缓存区时，为使查找操作更快速，内核首先自顶向下扫描所有缓存项。如果某个数据元素包含所需数据，则可以使用缓存的该数据实例。否则内核必须向块设备提交一个底层请求，来获取所需数据。

上一次使用的数据，将由内核自动放置到 LRU 列表的第一个位置上。如果缓冲区已经有数据元素，则只改变各个元素的位置。如果该数据元素是从块设备读取的，则将数组的最后一个元素退出缓存，从内存中释放。

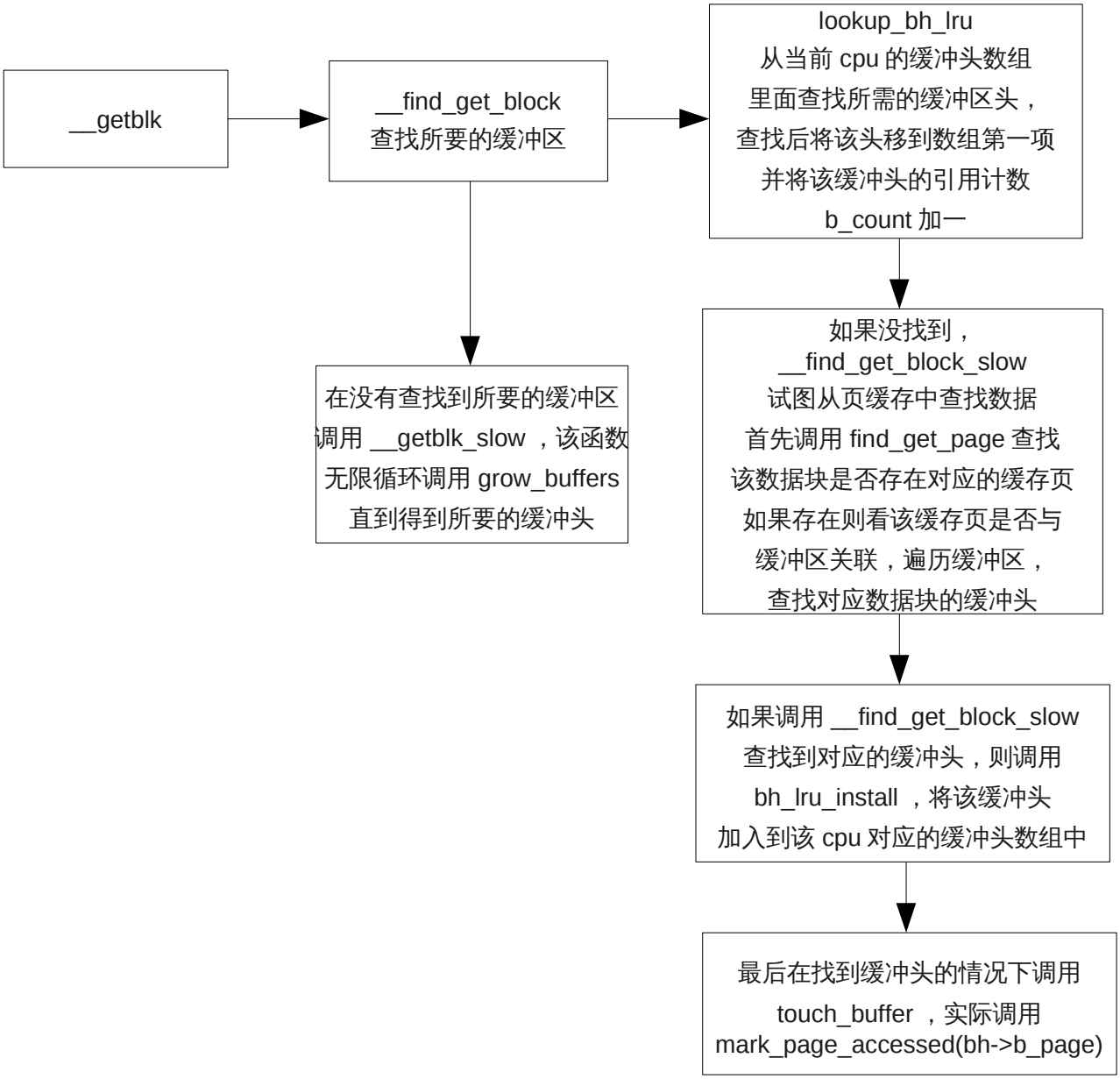
```
#define BH_LRU_SIZE 8
struct bh_lru {
    struct buffer_head *bhs[BH_LRU_SIZE];
};
```

static DEFINE_PER_CPU(struct bh_lru, bh_lrus) = {{ NULL }};

bhs 是一个缓冲头指针的数组，用作实现 LRU 算法的基础。内核使用 DEFINE_PER_CPU，为系统的每个 cpu 都建立一个实例，改进对 cpu 高速缓存的利用率。

__getblk，根据 block_device 实例，扇区编号(sector_t)和块长度来唯一标识一个数据块，返回该数据块的缓冲。

struct buffer_head * __getblk(struct block_device *bdev, sector_t block, unsigned size)



# 五.数据同步

## *1.相关的数据结构*

## 1.页的状态

全局 vm_stat,可用于查询

mm/vmstat.c

atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS];

各个内存域 zone 也提供每个内存域的页的状态信息

```
zone{
.........
atomic_long_t       vm_stat[NR_VM_ZONE_STAT_ITEMS];
}

enum zone_stat_item {
    NR_FREE_PAGES,
    NR_LRU_BASE,
    NR_INACTIVE_ANON = NR_LRU_BASE, /* must match order of LRU_[IN]ACTIVE */
    NR_ACTIVE_ANON,
    NR_INACTIVE_FILE,
    NR_ACTIVE_FILE,
    NR_UNEVICTABLE,
    NR_MLOCK,       /* mlock()ed pages found and moved off LRU */
    NR_ANON_PAGES,  /* Mapped anonymous pages */
    NR_FILE_MAPPED, /* pagecache pages mapped into pagetables. only modified from process context */
    NR_FILE_PAGES,
    NR_FILE_DIRTY,
    NR_WRITEBACK,
    NR_SLAB_RECLAIMABLE,
    NR_SLAB_UNRECLAIMABLE,
    NR_PAGETABLE,       /* used for pagetables */
    NR_KERNEL_STACK,
    NR_UNSTABLE_NFS,    /* NFS unstable pages */
```

```
    NR_BOUNCE,
    NR_VMSCAN_WRITE,
    NR_WRITEBACK_TEMP,  /* Writeback using temporary buffers */
    NR_ISOLATED_ANON,   /* Temporary isolated pages from anon lru */
    NR_ISOLATED_FILE,   /* Temporary isolated pages from file lru */
    NR_SHMEM,       /* shmem pages (included tmpfs/GEM pages) */
.........
}
```

## 2.回写控制

```
struct writeback_control {
    enum writeback_sync_modes sync_mode;
    unsigned long *older_than_this; /* If !NULL, only write back inodes older than this */
    unsigned long wb_start;             /* Time writeback_inodes_wb was  called. This is needed to avoid extra jobs
                                                                                                and livelock */
    long nr_to_write;       /* Write this many pages, and decrement  this for each page written */
    long pages_skipped;     /* Pages which were not written */
    loff_t range_start;
    loff_t range_end;
    unsigned nonblocking:1;     /* Don't get stuck on request queues */
    unsigned encountered_congestion:1; /* An output: a queue is full */
    unsigned for_kupdate:1;     /* A kupdate writeback */
    unsigned for_background:1;  /* A background writeback */
    unsigned for_reclaim:1;     /* Invoked from the page allocator */
    unsigned range_cyclic:1;    /* range_start is cyclic */
    unsigned more_io:1;     /* more io to be dispatched */
};
```

部分元素解释：

(1)sync_mode:对两种同步模式进行区分，

```
    enum writeback_sync_modes {
        WB_SYNC_NONE,   /* Don't wait on anything */
        WB_SYNC_ALL,    /* Wait on every mapping */
```

};

WB_SYNC_ALL，该模式要求进行等待回写完成，此模式下所有数据都已经与底层块设备同步。

WB_SYNC_NONE，内核发送请求，然后立即继续进行剩余的同步工作，不进行等待。

(2)older_than_this:在内核进行回写时，它必须确定哪些脏的缓存数据必须与后备存储器同步。因此使用了 older_than_this 和 nr_to_write 成员。如果数据变脏的时间已经超过 older_than_this 的值，那么将回写。

(3)nr_to_write:可以限制应该回写的页的最大数目。最大值 MAX_WRITEBACK_PAGES 为 1024.

(4)pages_skipped:在回写过程中，因为各种原因跳过的页的数目，通过该字段统计。

(5)nonblocking:标志位指定了回写队列在遇到拥塞时是否阻塞。

(6)encountered_congestion:标志位通知高层在数据回写期间发生了拥塞。

(7)for_kupdate:如果写请求由周期性机制发出，则置位。

(8)for_reclaim:写请求由内存回收发起的。

(9)for_background:后台回写发起的。

(10)range_cyclic:

如果该字段为 0，则回写机制限于对 range_start 和 range_end 指定的范围进行操作。该限制是对回写操作的目标映射设置的。

如果该字段为 1，内核可能多次遍历与映射相关的页。

## 3.后备设备相关的数据结构

### *1.设备信息*

```
struct backing_dev_info {
struct list_head bdi_list; //用于将各设备链接成链表
unsigned long state;    /* Always use atomic bitops on this */ 设备状态信息
......
struct bdi_writeback wb;  /* default writeback info for this bdi */
spinlock_t wb_lock;   /* protects work_list */
struct list_head work_list; //wb_writeback_work 的链表
};
```

### *2.设备的回写任务结构*

```
struct wb_writeback_work {
    long nr_pages;  //需要回写的页数
    struct super_block *sb;
    enum writeback_sync_modes sync_mode;
```

```
    unsigned int for_kupdate:1;
    unsigned int range_cyclic:1;
    unsigned int for_background:1;
    struct list_head list;      /* pending work list */
    struct completion *done;    /* set if the caller waits */
};
```

### *3.设备回写*

linux/backing-dev.h

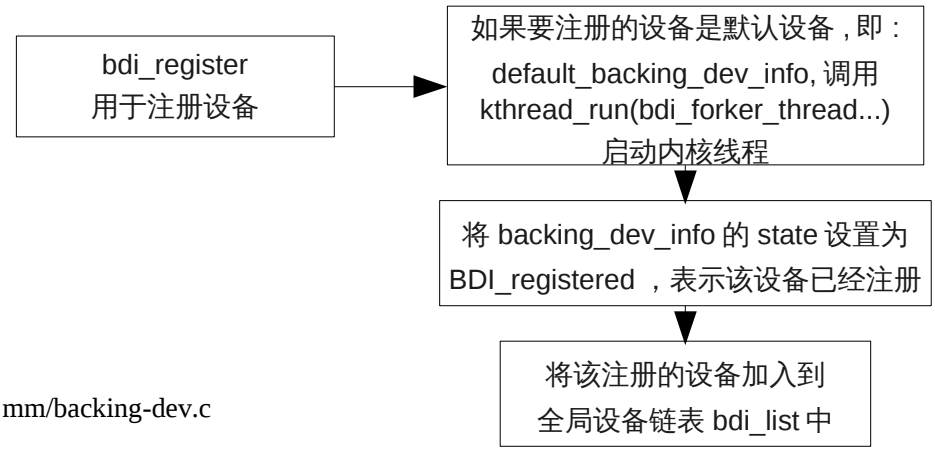
```
struct bdi_writeback {
    struct backing_dev_info *bdi;   /* our parent bdi */
    unsigned int nr;
    unsigned long last_old_flush;   /* last old data flush */
    unsigned long last_active;  /* last time bdi thread was active */
    struct task_struct *task;   /* writeback thread */
    struct timer_list wakeup_timer; /* used for delayed bdi thread wakeup */
    struct list_head b_dirty;   /* dirty inodes */
    struct list_head b_io;      /* parked for writeback */
    struct list_head b_more_io; /* parked for more writeback */
};
```

(1)b_dirty:该设备下的所有脏的 inode 都保存在 b_dirty 链表中。对于同步机制来说，所需数据都是现成的，文件系统会自动更新该链表。

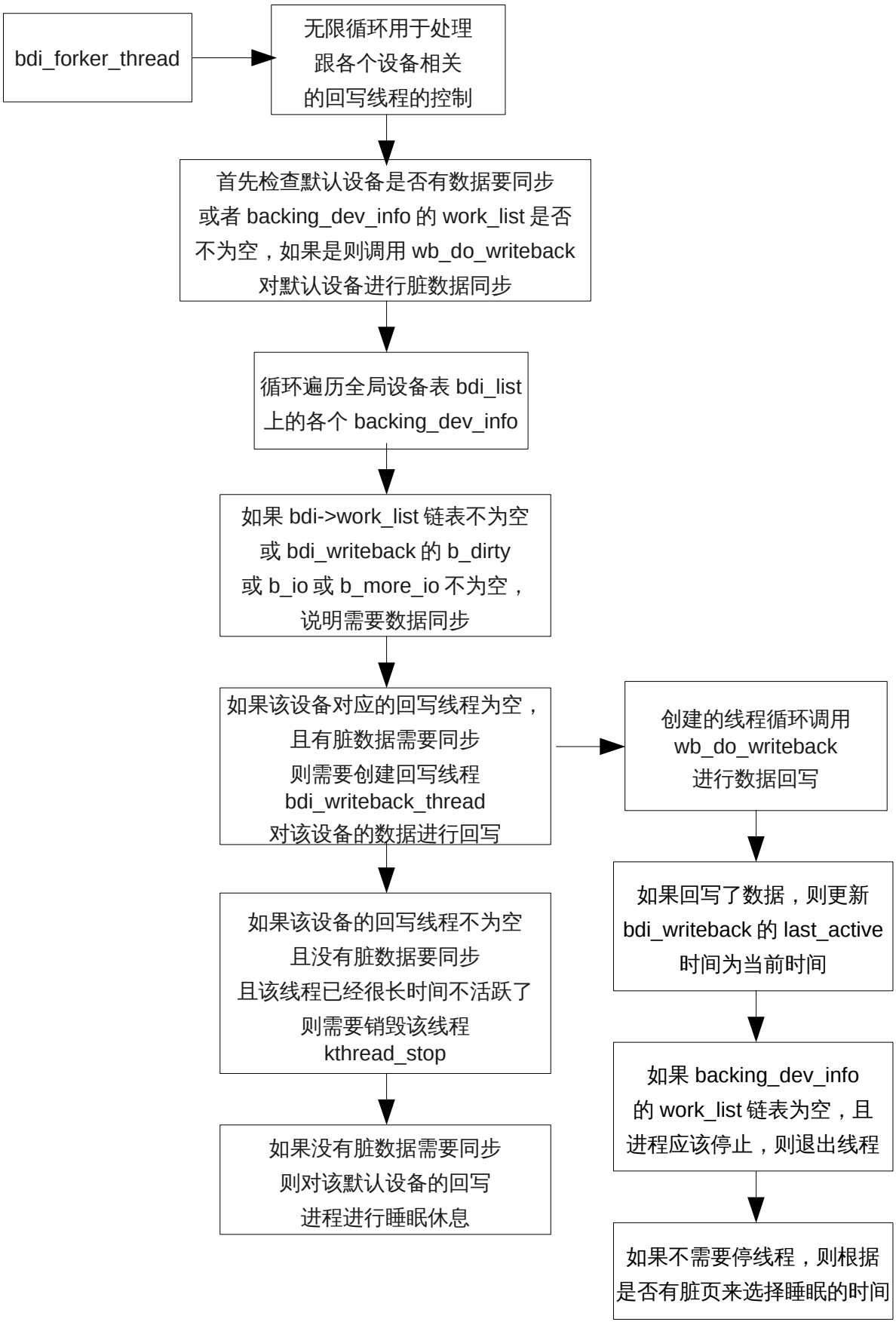
(2)b_io:该链表保存了同步代码当前考虑回写的所有 inode。

(3)b_more_io:已经选中进行同步的 inode(即:在 b_io 链表)，但一次没有处理完，将没处理完的放到该链表。

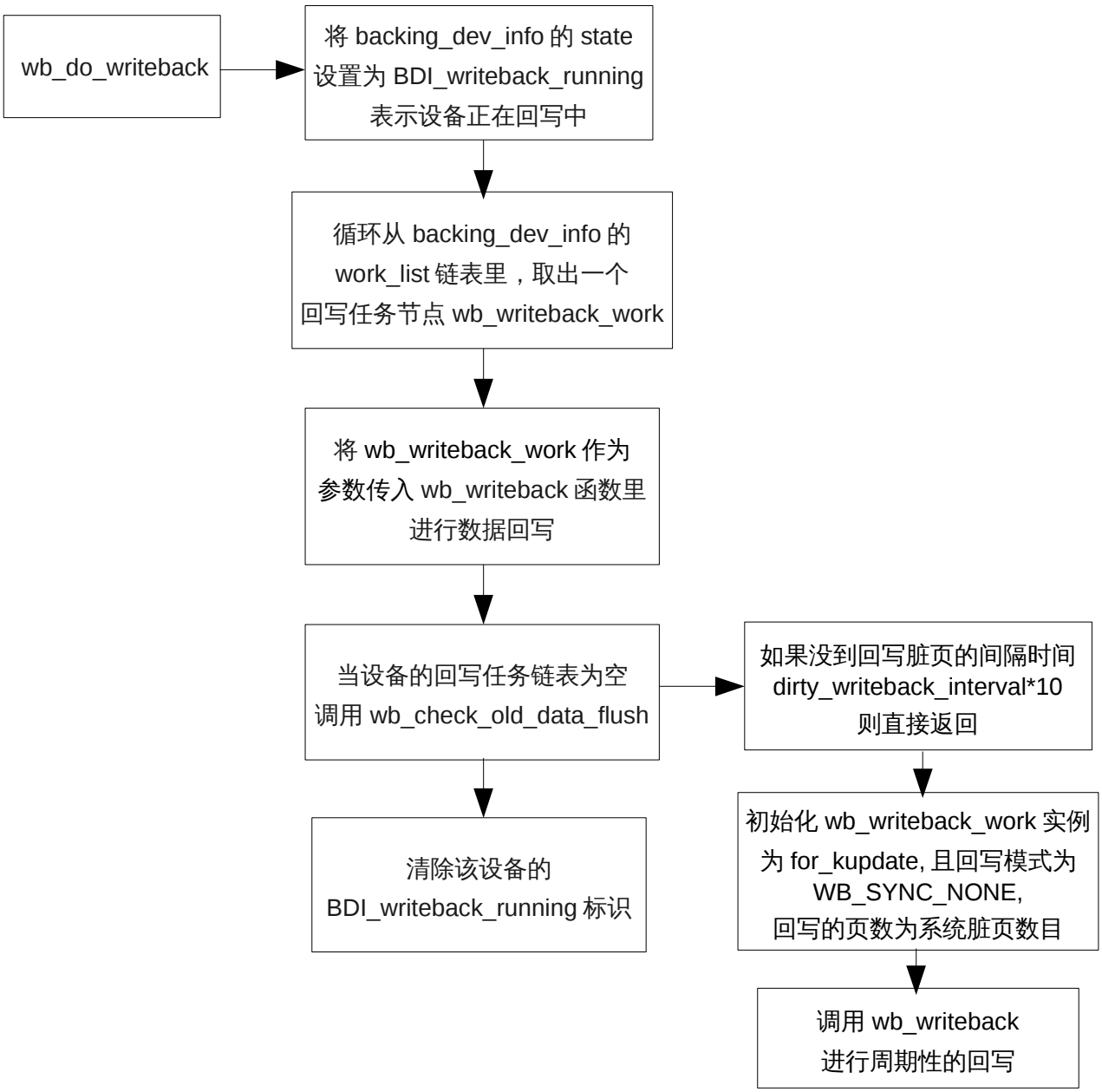
## *2.设备创建数据同步的内核线程*



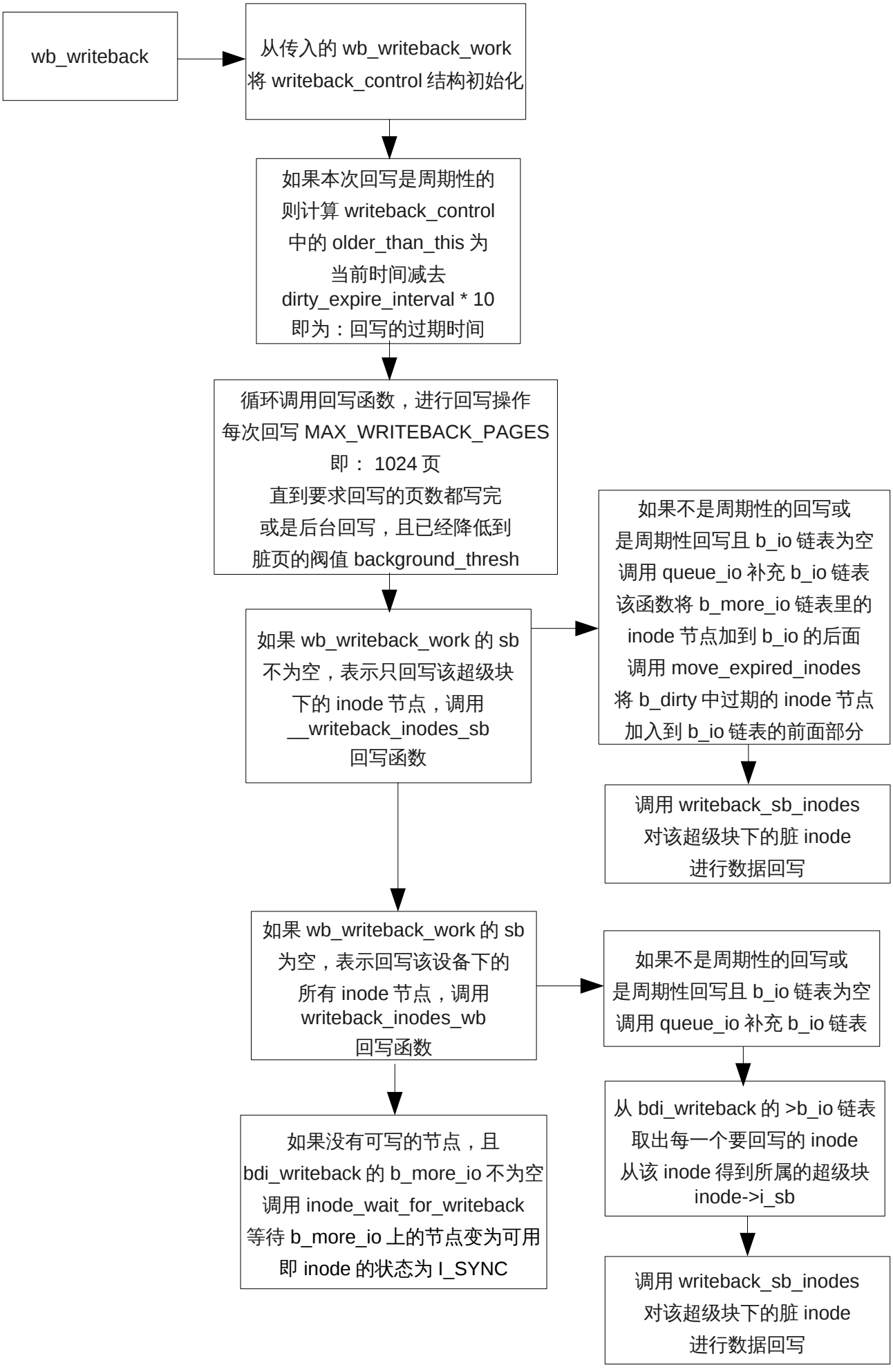

mm/backing-dev.c



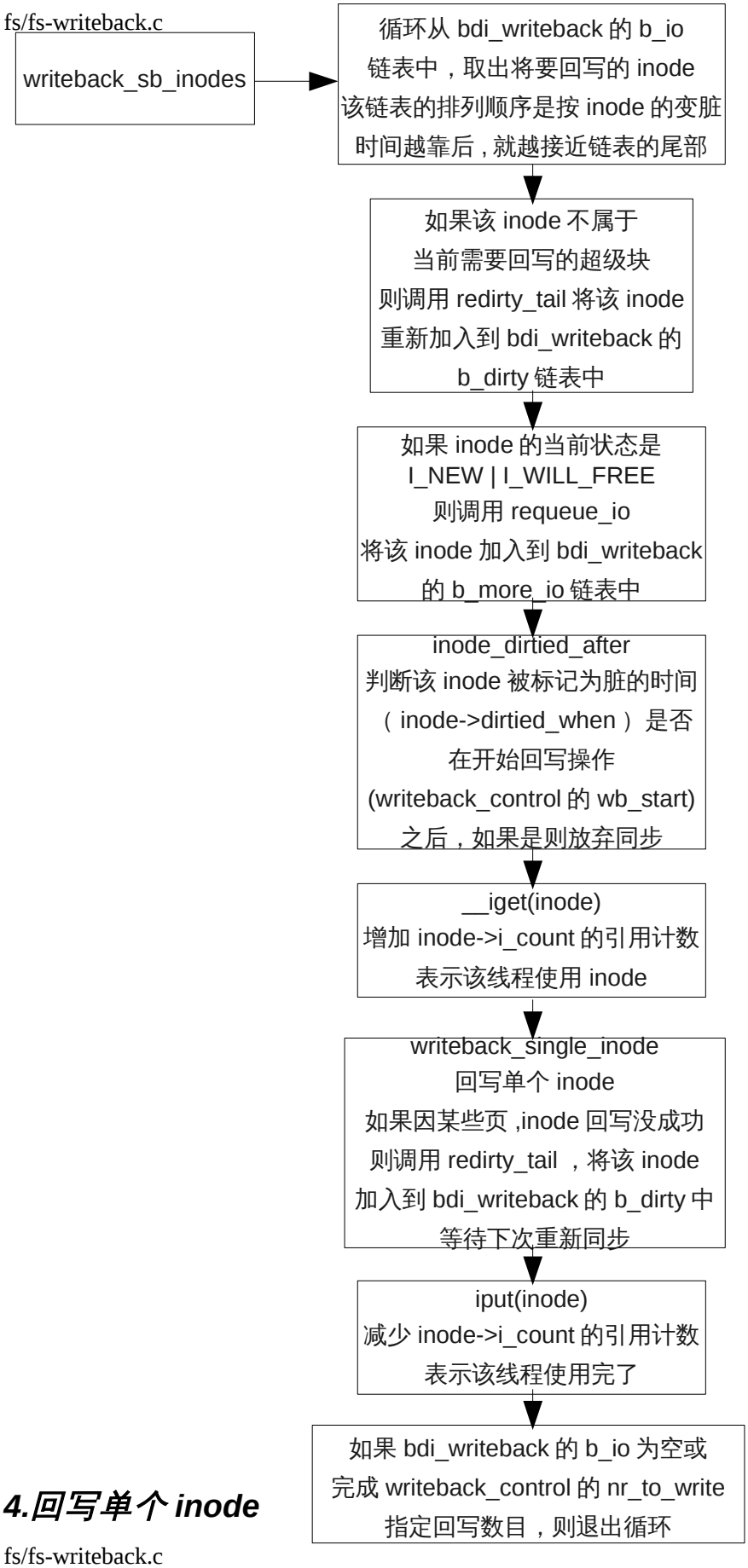

fs/fs-writeback.c




fs/fs-writeback.c

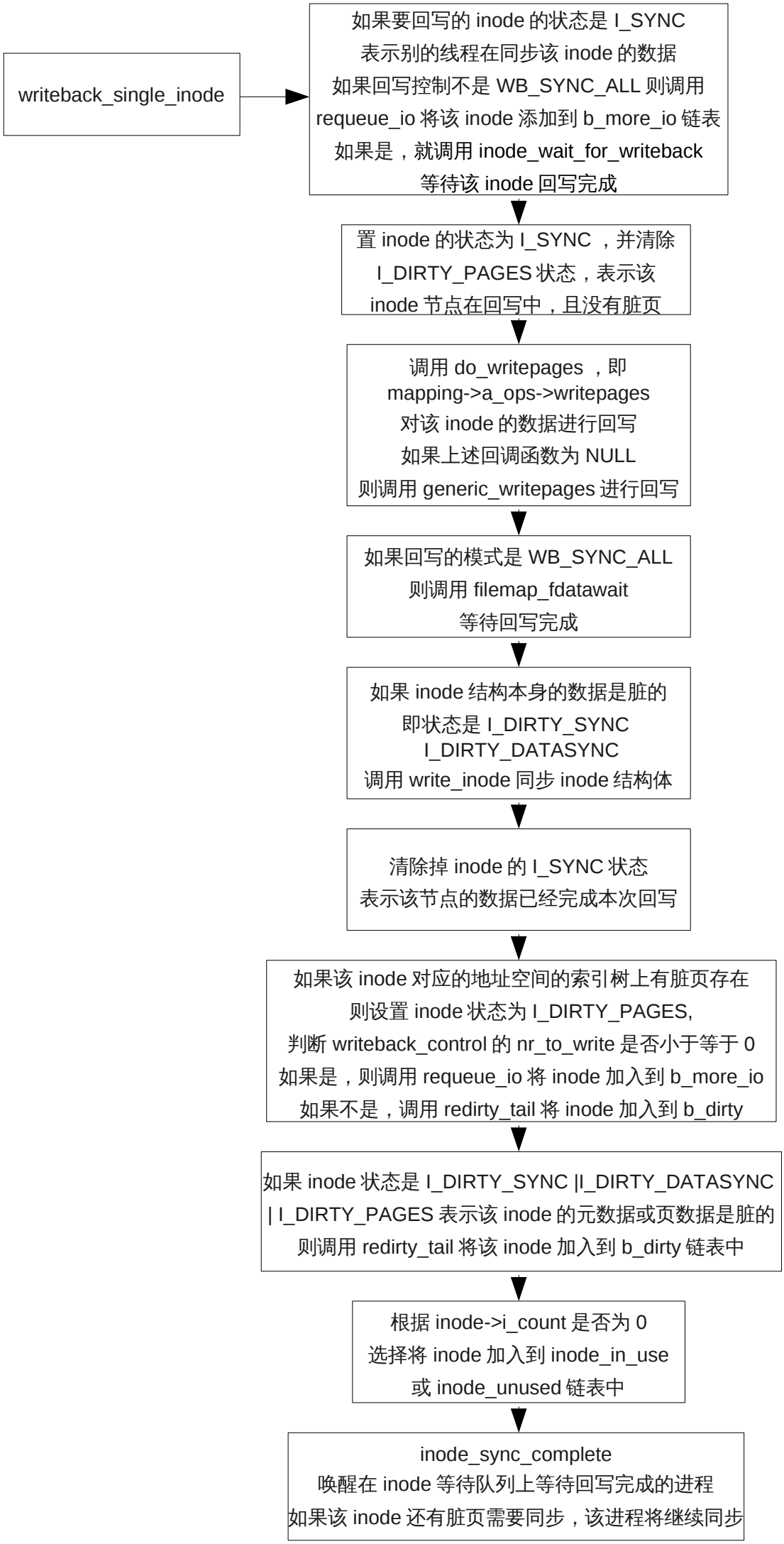

wb_writeback
从传入的 wb_writeback_work
将 writeback_control 结构初始化
如果本次回写是周期性的
则计算 writeback_control
中的 older_than_this 为
当前时间减去
dirty_expire_interval * 10
即为：回写的过期时间
循环调用回写函数，进行回写操作
每次回写 MAX_WRITEBACK_PAGES
即： 1024 页
直到要求回写的页数都写完
或是后台回写，且已经降低到
脏页的阀值 background_thresh
如果 wb_writeback_work 的 sb
不为空，表示只回写该超级块
下的 inode 节点，调用
__writeback_inodes_sb
回写函数
如果不是周期性的回写或
是周期性回写且 b_io 链表为空
调用 queue_io 补充 b_io 链表
该函数将 b_more_io 链表里的
inode 节点加到 b_io 的后面
调用 move_expired_inodes
将 b_dirty 中过期的 inode 节点
加入到 b_io 链表的前面部分
调用 writeback_sb_inodes
对该超级块下的脏 inode
进行数据回写
如果 wb_writeback_work 的 sb
为空，表示回写该设备下的
所有 inode 节点，调用
writeback_inodes_wb
回写函数
如果不是周期性的回写或
是周期性回写且 b_io 链表为空
调用 queue_io 补充 b_io 链表
从 bdi_writeback 的 >b_io 链表
取出每一个要回写的 inode
从该 inode 得到所属的超级块
inode->i_sb
调用 writeback_sb_inodes
对该超级块下的脏 inode
进行数据回写
如果没有可写的节点，且
bdi_writeback 的 b_more_io 不为空
调用 inode_wait_for_writeback
等待 b_more_io 上的节点变为可用
即 inode 的状态为 I_SYNC

## *3.超级块下的脏 inode 数据同步*




## *4.回写单个 inode*

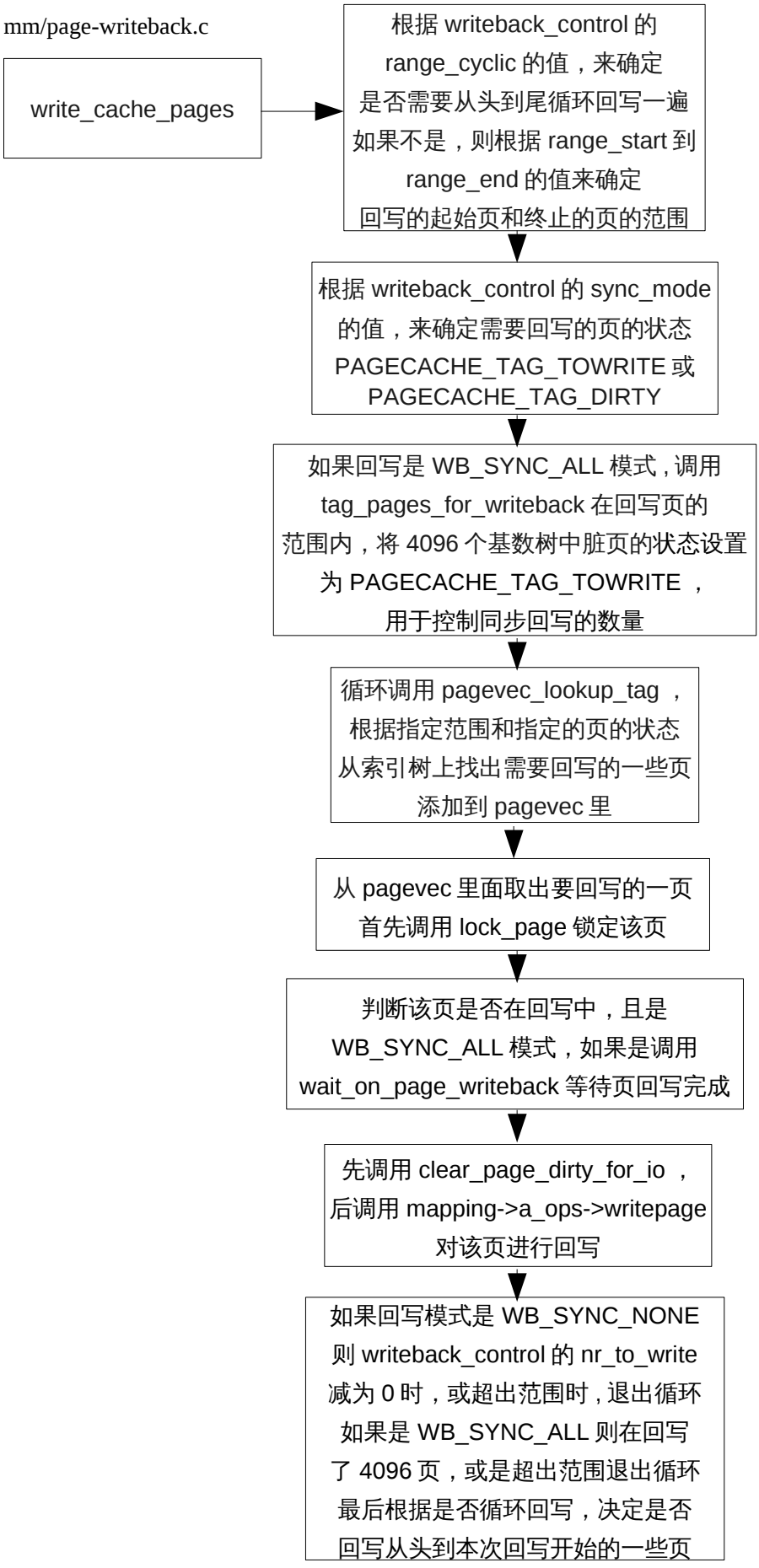
fs/fs-writeback.c


writeback_single_inode
如果要回写的 inode 的状态是 I_SYNC
表示别的线程在同步该 inode 的数据
如果回写控制不是 WB_SYNC_ALL 则调用
requeue_io 将该 inode 添加到 b_more_io 链表
如果是，就调用 inode_wait_for_writeback
等待该 inode 回写完成
置 inode 的状态为 I_SYNC ，并清除
I_DIRTY_PAGES 状态，表示该
inode 节点在回写中，且没有脏页
调用 do_writepages ，即
mapping->a_ops->writepages
对该 inode 的数据进行回写
如果上述回调函数为 NULL
则调用 generic_writepages 进行回写
如果回写的模式是 WB_SYNC_ALL
则调用 filemap_fdatawait
等待回写完成
如果 inode 结构本身的数据是脏的
即状态是 I_DIRTY_SYNC
I_DIRTY_DATASYNC
调用 write_inode 同步 inode 结构体
清除掉 inode 的 I_SYNC 状态
表示该节点的数据已经完成本次回写
如果该 inode 对应的地址空间的索引树上有脏页存在
则设置 inode 状态为 I_DIRTY_PAGES,
判断 writeback_control 的 nr_to_write 是否小于等于 0
如果是，则调用 requeue_io 将 inode 加入到 b_more_io
如果不是，调用 redirty_tail 将 inode 加入到 b_dirty
如果 inode 状态是 I_DIRTY_SYNC |I_DIRTY_DATASYNC
| I_DIRTY_PAGES 表示该 inode 的元数据或页数据是脏的
则调用 redirty_tail 将该 inode 加入到 b_dirty 链表中
根据 inode->i_count 是否为 0
选择将 inode 加入到 inode_in_use
或 inode_unused 链表中
inode_sync_complete
唤醒在 inode 等待队列上等待回写完成的进程
如果该 inode 还有脏页需要同步，该进程将继续同步

## *5.inode 页回写*

在调用 do_writepages 回写 inode 下的脏页时，在 mapping->a_ops->writepages 为 NULL 的情况下，调用 generic_writepages，该函数实际调用 write_cache_pages，下面详细介绍。

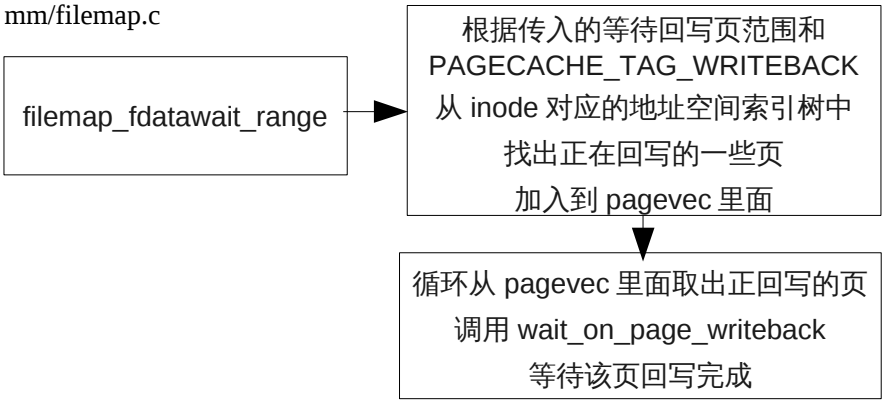
mm/page-writeback.c



### *6.等待 inode 回写完成*

filemap_fdatawait 函数等待 inode 下的脏页回写完成，实际调用 filemap_fdatawait_range。

mm/filemap.c




# 六.页面回收和页交换

可换出到交换分区的页

1.类别为 MAP_ANONYMOUS 的页，即匿名页，没有关联到文件。例如：进程的栈或是使用 mmap 匿名映射的内存区。

2.进程的私有映射用于映射后不向底层块设备回写的文件，内核使用 MAP_PRIVATE 来创建此类映射。

3.所有使用进程堆分配的页。

4.用于实现某种进程间通信机制的页，例如：进程之间交换数据共享的内存页。

内核本身使用的内存页是不会换出的。

### *1.交换区的数据结构*

mm/swapfile.c

```
static struct swap_info_struct *swap_info[MAX_SWAPFILES];
```

include/linux/swap.h

```
struct swap_info_struct {
    unsigned long   flags;      /* SWP_USED etc: see above */
    signed short    prio;       /* swap priority of this type */
    signed char type;       /* strange name for an index */
    signed char next;       /* next type on the swap list */
    unsigned int    max;        /* extent of the swap_map */
    unsigned char *swap_map;    /* vmalloc'ed array of usage counts */
    unsigned int lowest_bit;    /* index of first free in swap_map */
```

```
    unsigned int highest_bit;   /* index of last free in swap_map */
    unsigned int pages;     /* total of usable pages of swap */
    unsigned int inuse_pages;   /* number of those currently in use */
    unsigned int cluster_next;  /* likely index for next allocation */
    unsigned int cluster_nr;    /* countdown to next cluster search */
    unsigned int lowest_alloc;  /* while preparing discard cluster */
    unsigned int highest_alloc; /* while preparing discard cluster */
    struct swap_extent *curr_swap_extent;
    struct swap_extent first_swap_extent;
    struct block_device *bdev;  /* swap device or bdev of swap file */
    struct file *swap_file;     /* seldom referenced */
    unsigned int old_block_size;    /* seldom referenced */
};
```

部分元素解释：

(1)flag:表示交换区的状态。SWP_USED 表示当前项在交换数组中处于使用状态。否则，相应的数组项会使用 0 填充。SWP_WRITEOK 指定当前项对应的交换区可写。

(2)swap_file:指向与该交换区关联的 file 结构。对于交换分区，这是一个指向块设备上分区的设备文件的指针。对于交换文件，该指针指向相关文件的 file 实例。

(3)bdev:指向文件/分区所在底层块设备的 block_device 结构。

(4)prio:交换区的相对优先级。

(5)pages:保存了交换区中可用槽位的总数。

(6)max:除了有可用槽位数，还有损坏和管理目的的槽位。

(7)swap_map:指向一个数组，数组包含的项数与交换区槽位的数目相同。该数组用作一个访问计数器，每个数组项都表示共享对应换出页的进程的数目。

(8)next:内核将交换分区数组中的各个数组项按优先级连接起来，使用 next 成员建立一个相对顺序的，next 指向交换区在 swap_info 数组中的索引，使得内核能够根据优先级来跟踪各个数组项。

全局变量 swap_list 为了查找按优先级排列的交换区静态列表的第一项而定义的。

swap_list 的类型为：

```
struct swap_list_t {
    int head;   /* head of priority-ordered swapfile list */
    int next;   /* swapfile to be used next */
};
```

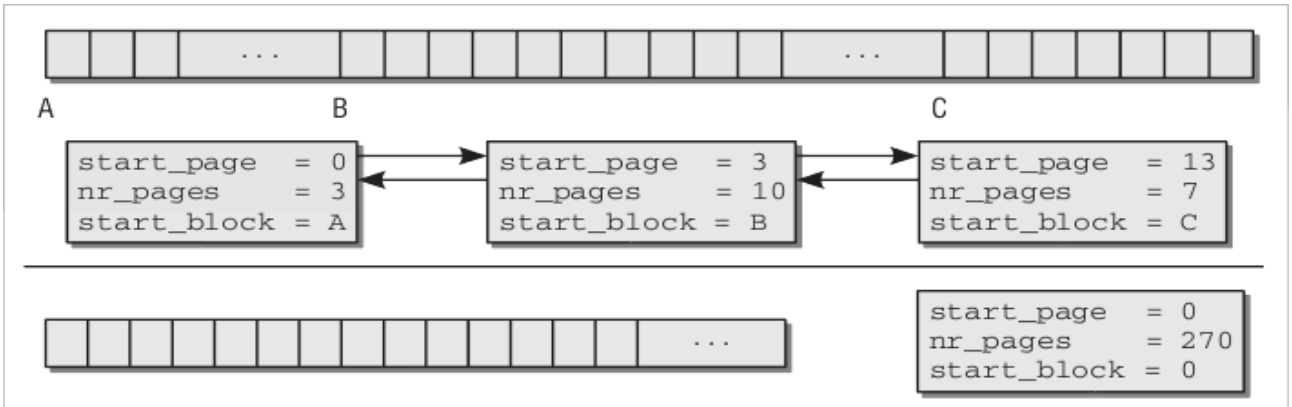
head 指向 swap_info 数组中的一个索引，用于选择优先级最高的交换区。

(9)lowest_bit 和 highest_bit:所有空闲的槽位于 lowest_bit 和 highest_bit 之间。这两个值分别是交换区中线性排布的各槽位的索引。

(10)cluster_next 和 cluster_nr:cluster_next 指定了在交换区中接下来使用的槽位(在某个聚集中)的索引，cluster_nr 表示当前聚集中仍然可用的槽位数。聚集用于容纳连续的回写内存页。

(11)curr_swap_extent 和 first_swap_extent:用于创建假定连续的交换区槽位与交换文件的磁盘块之间的映射。

该方法主要是针对用文件作为交换分区，因为无法保证文件的所有块在磁盘上是连续的。




```
struct swap_extent {
    struct list_head list;
    pgoff_t start_page;
    pgoff_t nr_pages;
    sector_t start_block;
}
```

first_swap_extent：指向链表的第一项。

curr_swap_extent：指向链表中上次访问的 swap_extent 实例，为了加快搜索。

## *2.交换缓存*

内核利用了一个缓存，称为交换缓存，该缓存在选择换出页的操作和实际执行页交换机制之间，充当协调者。

交换缓存的目的：

❑ When pages are swapped out, the selection logic first selects a suitable seldom-used page frame. This is buffered in the page cache, from where it is transferred to the swap cache.

❑ If a page used simultaneously by several processes is swapped out, the kernel must set the page entry in the directories of the process to point to the relevant position in the swap-out file. When one of the processes accesses the data on the page, the page is swapped in again, and the page table entry for this single process is set to the current memory address at which the page is now located. However, this causes a problem. The entries of all other processes still point to the entry in the swap-out file because, although it is possible to determine the number of

processes that share a page, it is not possible to identify which processes these are.

When shared pages are swapped in, they are therefore retained in the swap cache until all processes have requested the page from the swap area and are all thus aware of the new position of the page in memory.

## 1.交换缓存的页表项

换出页在页表中通过一种专门的页表项来标记，在换出页的页表项，所有 CPU 都会存储下列信息。

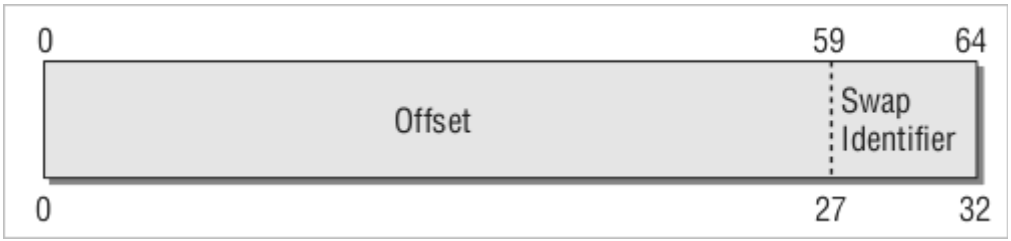
1.一个标志，表明页已经换出。

2.该页所在交换区的编号（即：类型）

3.对应槽位的偏移量，用于在交换区中查找该页所在的槽位。

交换分区的偏移量和交换分区的类型用以唯一确定一页，保存在 swp_entry_t 结构体里。

```
typedef struct {
    unsigned long val;
} swp_entry_t;
```




转换函数：

```
linux/swapops.h
#define MAX_SWAPFILES_SHIFT 5
#define SWP_TYPE_SHIFT(e)   (sizeof(e.val) * 8 - MAX_SWAPFILES_SHIFT)
#define SWP_OFFSET_MASK(e)  ((1UL << SWP_TYPE_SHIFT(e)) – 1)
static inline unsigned swp_type(swp_entry_t entry)
{
    return (entry.val >> SWP_TYPE_SHIFT(entry));
}
static inline pgoff_t swp_offset(swp_entry_t entry)
{
    return entry.val & SWP_OFFSET_MASK(entry);
}
static inline swp_entry_t swp_entry(unsigned long type, pgoff_t offset)
{
    swp_entry_t ret;
    ret.val = (type << SWP_TYPE_SHIFT(ret)) | ( offset & SWP_OFFSET_MASK(ret));
    return ret;
}
```

## 2.交换缓存的结构

```
struct address_space swapper_space = {
    .page_tree  = RADIX_TREE_INIT(GFP_ATOMIC|__GFP_NOWARN),
    .tree_lock  = __SPIN_LOCK_UNLOCKED(swapper_space.tree_lock),
    .a_ops      = &swap_aops,
    .i_mmap_nonlinear = LIST_HEAD_INIT(swapper_space.i_mmap_nonlinear),
    .backing_dev_info = &swap_backing_dev_info,
};

static const struct address_space_operations swap_aops = {
    .writepage  = swap_writepage,
    .sync_page  = block_sync_page,
    .set_page_dirty = __set_page_dirty_nobuffers,
    .migratepage    = migrate_page,
};
```

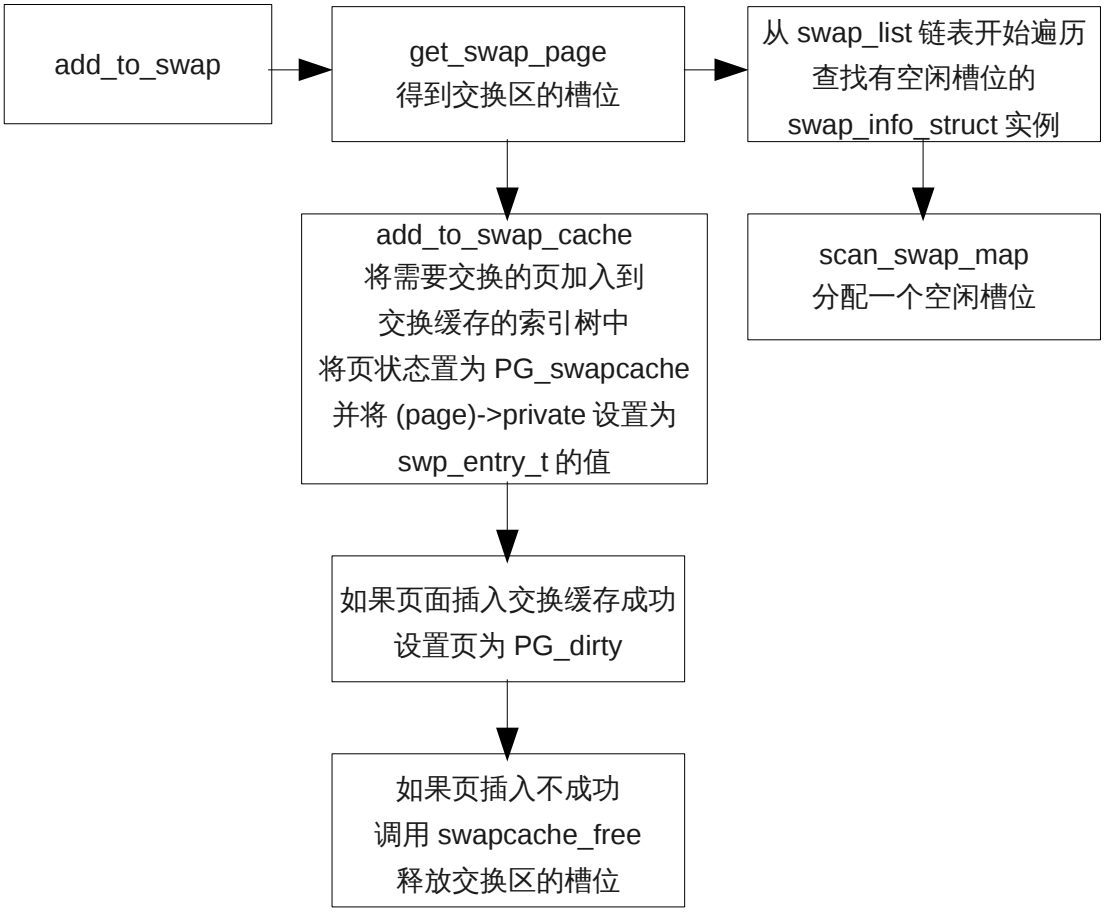
1. swap_writepage:  synchronizes dirty pages with the underlying block device. This is not done to maintain consistency between RAM memory and the block device, as is the case for all other page caches. Its purpose is to remove pages from the swap cache and to transfer their data to a swap area. The function is therefore responsible for implementing data transfer between RAM memory and the swap area on the disk.

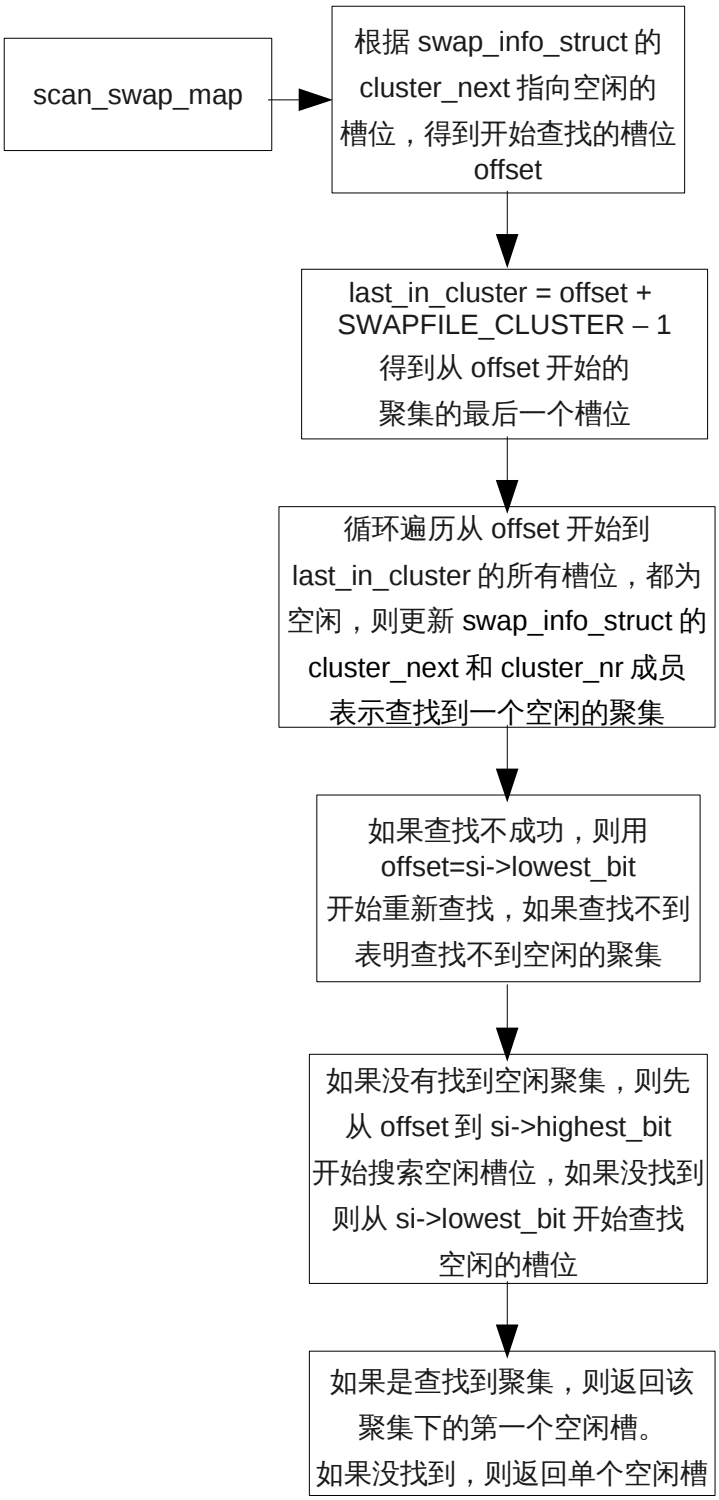
2.__set_page_dirty_nobuffers: Pages must be marked as ‘‘dirty’’ in the swap cache without having to allocate new memory — a resource that is scarce enough anyway when swap-out mechanisms are used.  one possible procedure to mark pages as dirty is to create buffers that enable the data to be written back chunk-by-chunk. However, additional memory is needed to hold the buffer_head instances that store the required management data. This is pointless as only complete pages in the swap cache are written back anyway. The __set_page_dirty_nobuffers function is therefore used to mark pages as dirty; it sets the PG_dirty flag but does not create buffers.

3.block_sync_page: As with most other page caches, the standard implementation of the kernel (block_sync_page) is used to synchronize pages in the swap area. This function does nothing more than unplug to corresponding block queues. As far as the swap cache is concerned, this means that all data transfer requests forwarded to the block layer are then executed.

## 3.向交换缓存添加新页

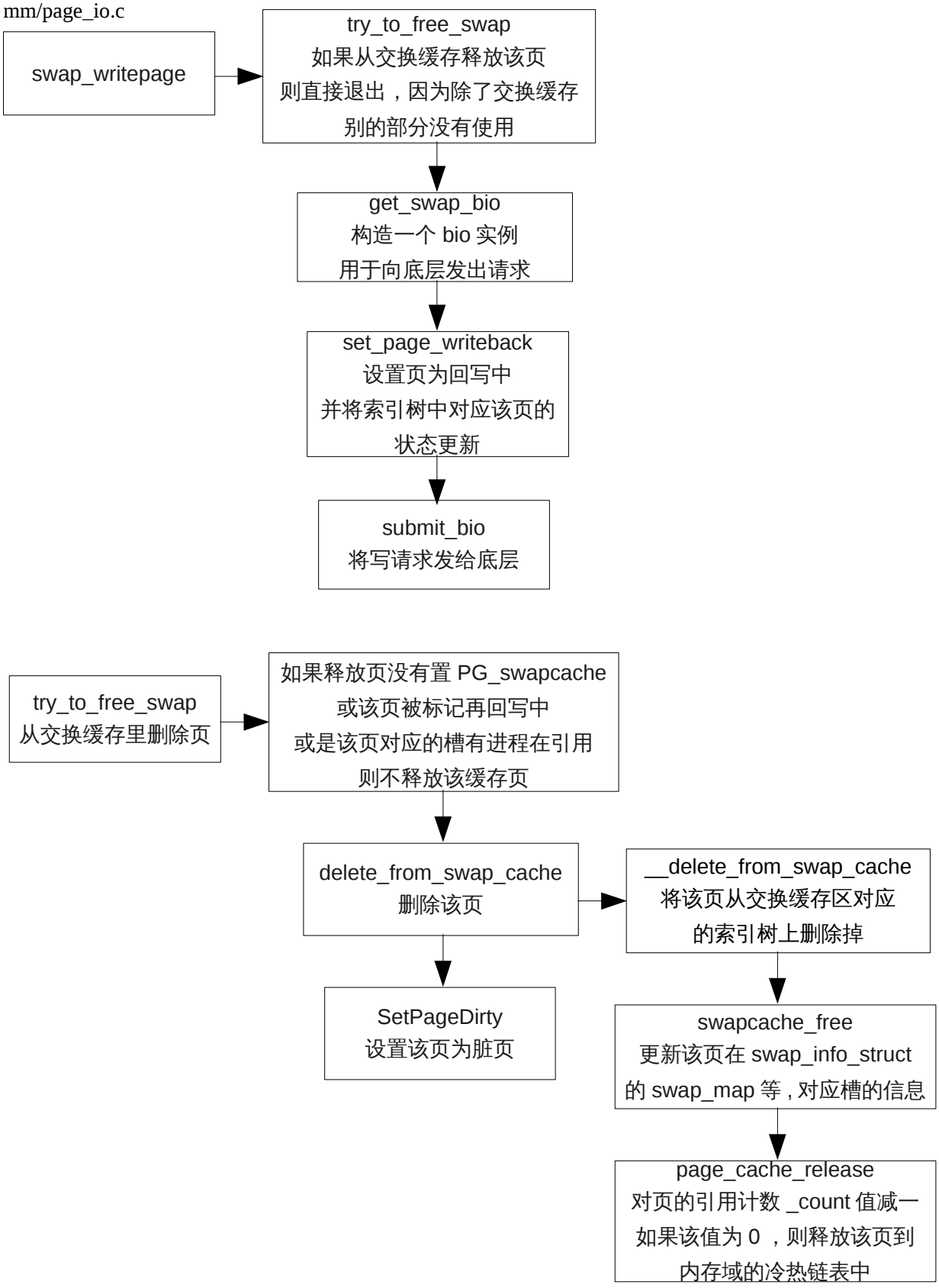
mm/swap_state.c


add_to_swap
get_swap_page
得到交换区的槽位
从 swap_list 链表开始遍历
查找有空闲槽位的
swap_info_struct 实例
add_to_swap_cache
将需要交换的页加入到
交换缓存的索引树中
将页状态置为 PG_swapcache
并将 (page)->private 设置为
swp_entry_t 的值
scan_swap_map
分配一个空闲槽位
如果页面插入交换缓存成功
设置页为 PG_dirty
如果页插入不成功
调用 swapcache_free
释放交换区的槽位


scan_swap_map
根据 swap_info_struct 的 cluster_next 指向空闲的槽位，得到开始查找的槽位 offset
last_in_cluster = offset + SWAPFILE_CLUSTER – 1
得到从 offset 开始的聚集的最后一个槽位
循环遍历从 offset 开始到 last_in_cluster 的所有槽位，都为空闲，则更新 swap_info_struct 的 cluster_next 和 cluster_nr 成员
表示查找到一个空闲的聚集
如果查找不成功，则用 offset=si->lowest_bit 开始重新查找，如果查找不到
表明查找不到空闲的聚集
如果没有找到空闲聚集，则先从 offset 到 si->highest_bit 开始搜索空闲槽位，如果没找到则从 si->lowest_bit 开始查找空闲的槽位
如果是查找到聚集，则返回该聚集下的第一个空闲槽。
如果没找到，则返回单个空闲槽

## 4.将交换缓存页回写到交换区

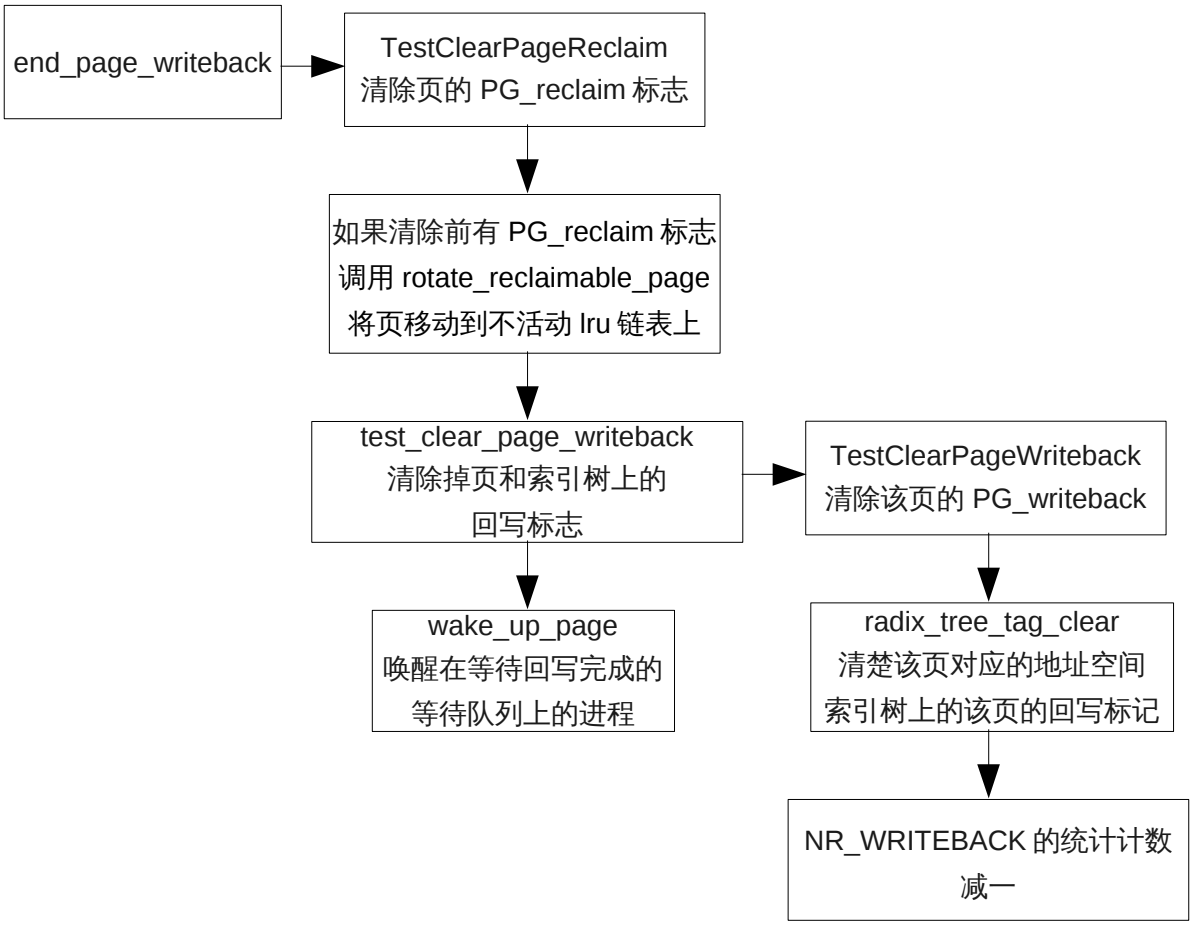
mm/page_io.c



## *3.回写完成后的调用*

底层回写完成后，调用 end_page_writeback，清除相应的 PG_writeback 标志

mm/filemap.c




## *4.页面回收*

页面回收以两种方式进行：

1.如果内核检测到在某个操作期间内存严重不足，将调用 try_to_free_pages，该函数检查当前内存域中所有页，并释放最不常用的那些。

2.一个后台守护进程，名为 kswapd，会定期检查内存使用情况，并将检查即将发生的内存不足，释放不常用的页。

内存域的 LRU 链表：

```
zone{
.....
    struct zone_lru {
```

```
        struct list_head list;
    } lru[NR_LRU_LISTS];
}

enum lru_list {
    LRU_INACTIVE_ANON = LRU_BASE,
    LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
    LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
    LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
    LRU_UNEVICTABLE,
    NR_LRU_LISTS
};
```

## 1.页向量（用于 LRU 缓存）

### *1.页向量*

```
#define PAGEVEC_SIZE    14
struct pagevec {
    unsigned long nr;
    unsigned long cold;
    struct page *pages[PAGEVEC_SIZE];
};
```

该结构用于将几个页群集到一个小的数组中，页向量使得可以对一组 page 结构整体执行操作。以一种“批处理模式”执行，这比分别对每个页执行同样的操作要快得多。

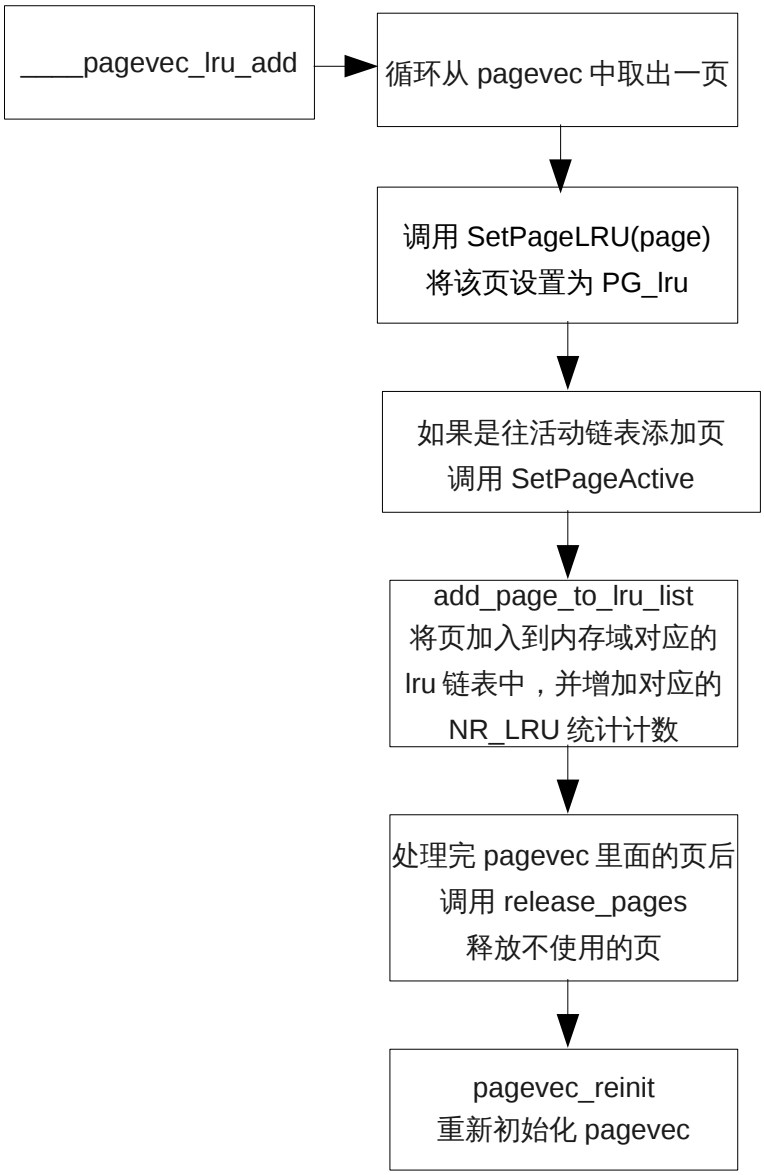
### *2.页向量的应用--LRU 缓存*

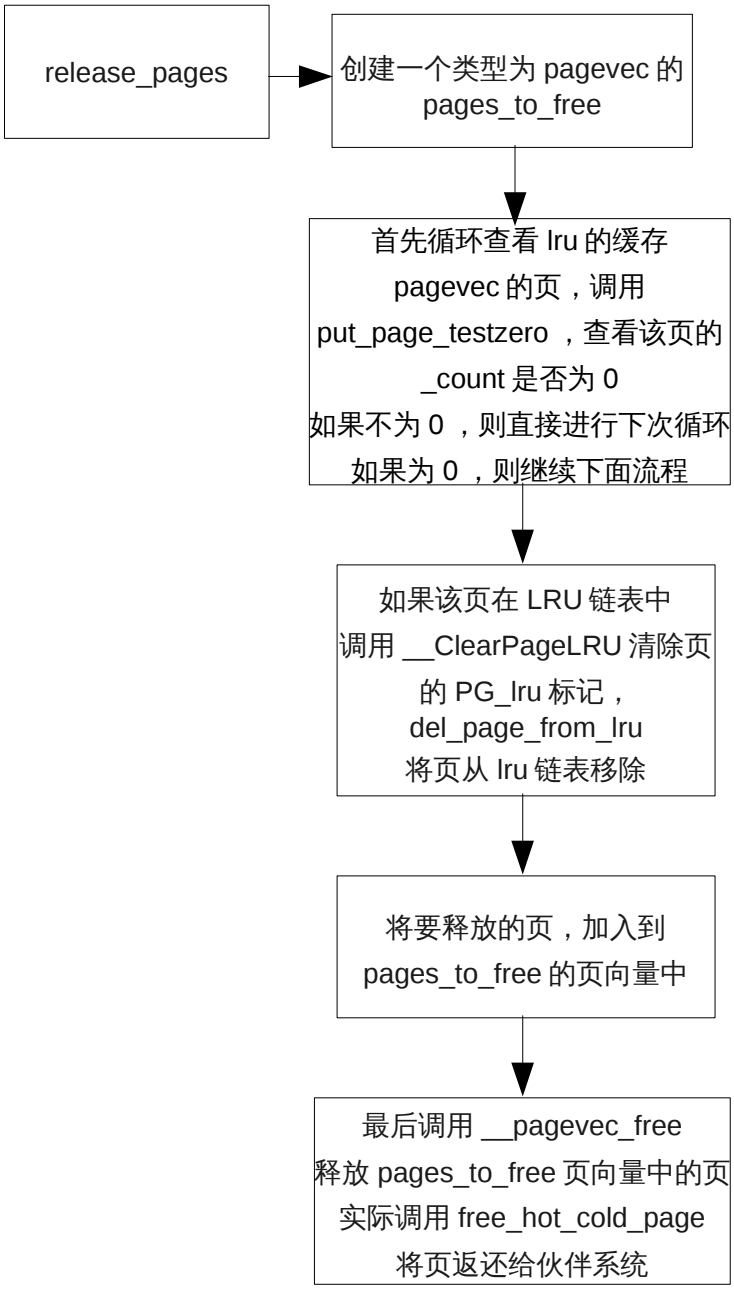
用于加速向系统的 LRU 链表添加页的操作。

向 LRU 链表添加页

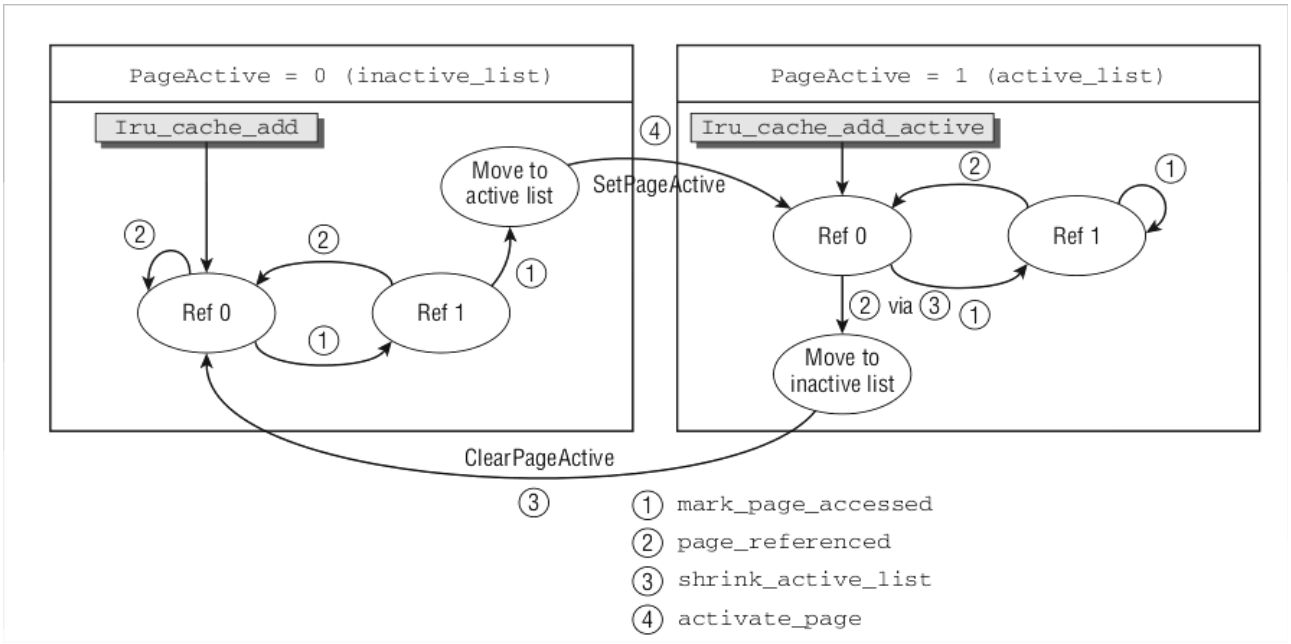
```
static DEFINE_PER_CPU(struct pagevec[NR_LRU_LISTS], lru_add_pvecs);

void __lru_cache_add(struct page *page, enum lru_list lru)
{
    struct pagevec *pvec = &get_cpu_var(lru_add_pvecs)[lru];
    page_cache_get(page);
```

```
    if (!pagevec_add(pvec, page))
        ____pagevec_lru_add(pvec, lru);
    put_cpu_var(lru_add_pvecs);
}
```




release_pages
创建一个类型为 pagevec 的 pages_to_free
首先循环查看 lru 的缓存 pagevec 的页，调用 put_page_testzero ，查看该页的 _count 是否为 0
如果不为 0 ，则直接进行下次循环
如果为 0 ，则继续下面流程
如果该页在 LRU 链表中 调用 __ClearPageLRU 清除页的 PG_lru 标记，
del_page_from_lru 将页从 lru 链表移除
将要释放的页，加入到 pages_to_free 的页向量中
最后调用 __pagevec_free 释放 pages_to_free 页向量中的页
实际调用 free_hot_cold_page 将页返还给伙伴系统

## 2.确定页的活跃程度


PageActive = 0 (inactive_list)
PageActive = 1 (active_list)
Iru_cache_add
Iru_cache_add_active
Move to active list
SetPageActive
Ref 0
Ref 1
Ref 0
Ref 1
via
Move to inactive list
ClearPageActive
① mark_page_accessed
② page_referenced
③ shrink_active_list
④ activate_page

1.If the page is deemed active, the PG_active flag is set; otherwise, not. The flag directly corresponds to the LRU list the page is on, namely, the (zone-specific) inactive or active list.

2. Each time the page is accessed, the flag PG_referenced is set. The function responsible for this is mark_page_accessed, and the kernel must make sure to call it appropriately.

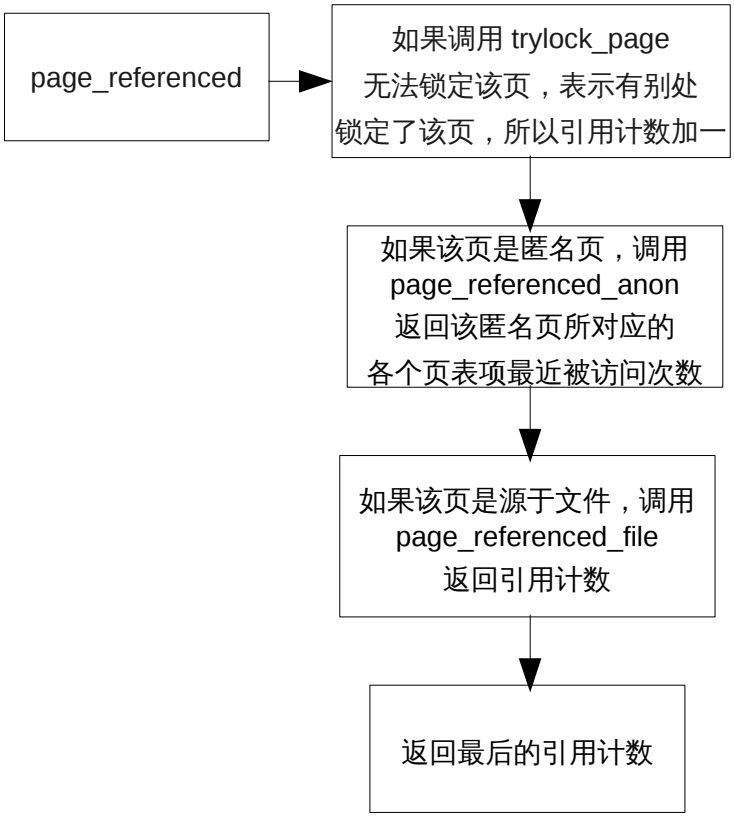
3. The PG_referenced flag and information provided by reverse mapping are used to determine page activity. The crucial point is that the PG_referenced flag is removed each time an activity check is performed. page_referenced is the function that implements this behavior.

4. Enter mark_page_accessed again. When it finds that the PG_accessed bit is already set when it checks the page, this means that no check was performed by page_referenced. The calls to mark_page_accessed have thus been more frequent than the calls to page_referenced, which implies that the page is often accessed. If the page is currently on the inactive list, it is moved to the active list. Additionally, the PG_active bit is set, and PG_referenced is removed.

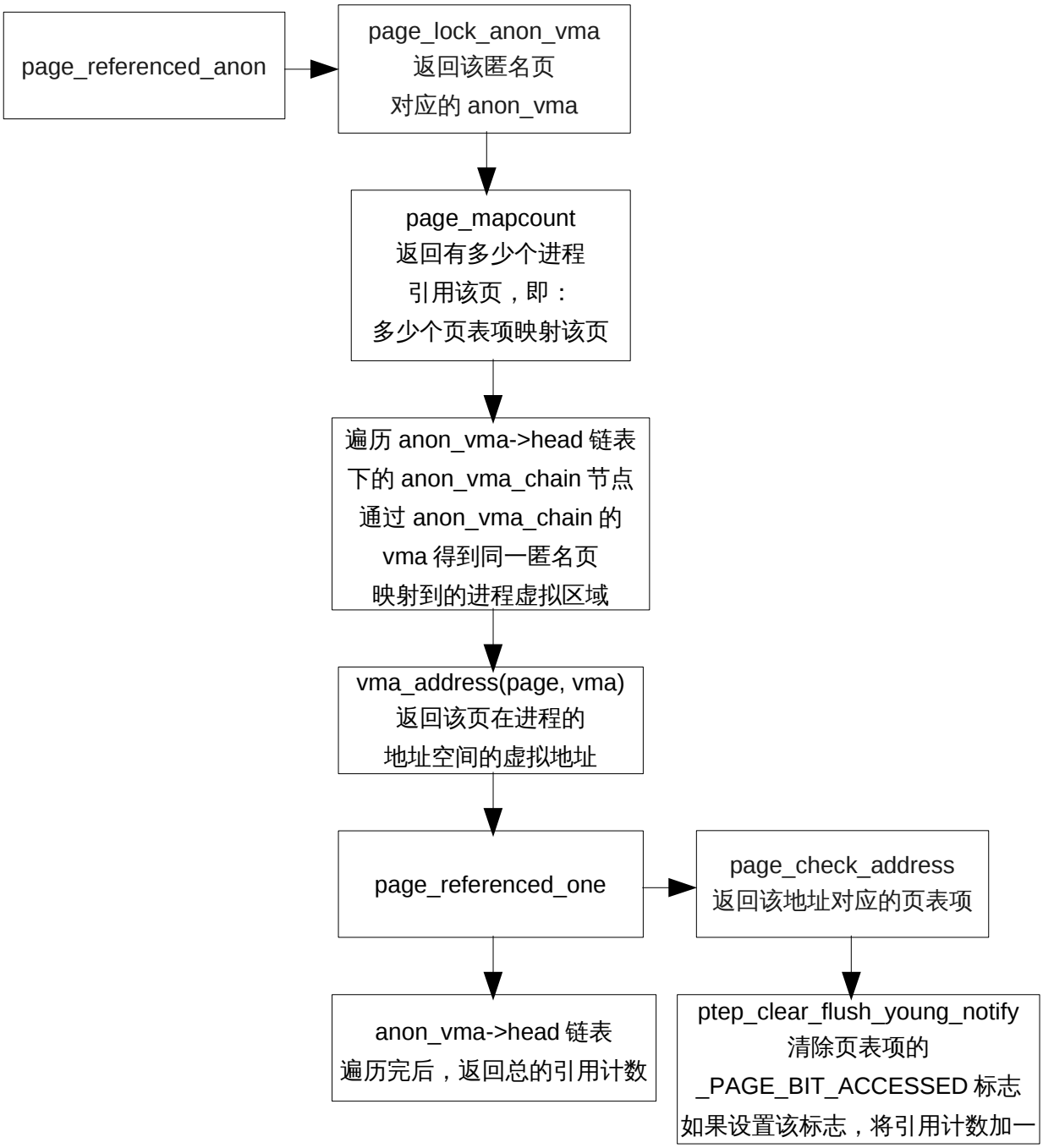
5.A downward promotion is also possible. If the page is on the active list and receives much attention, then PG_referenced is usually set. Once the page starts to experience less activity, then two calls of page_referenced are required without intervention of mark_page_accessed before it is put on the inactive list.

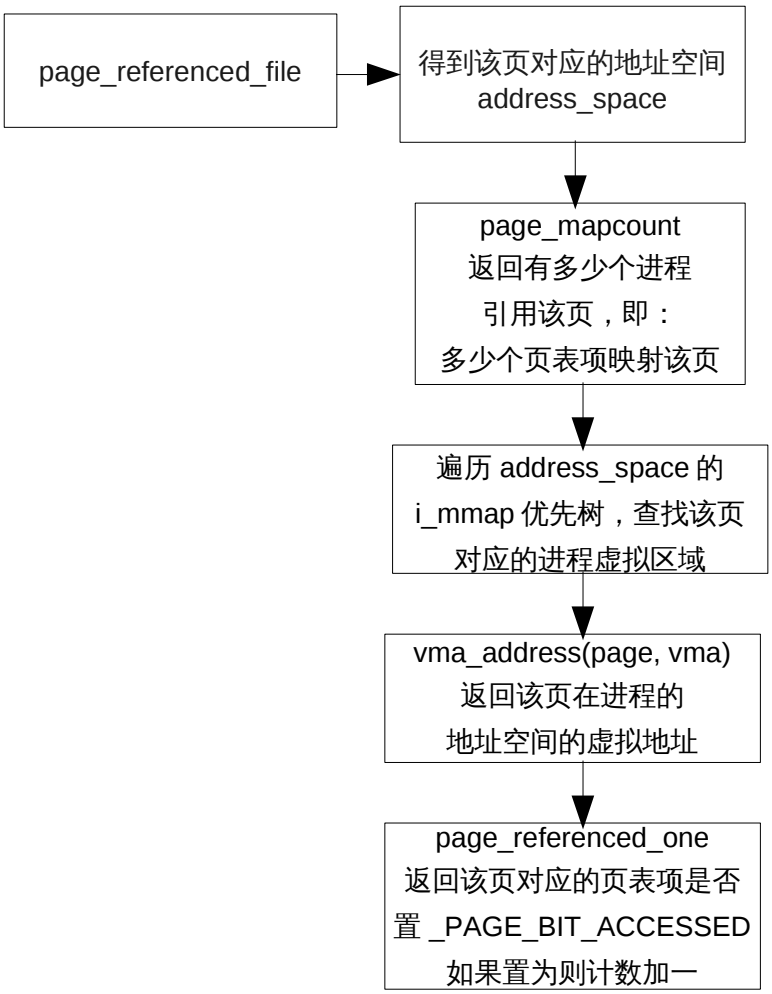
page_referenced 函数总计了最近引用该页的程度

mm/rmap.c



mm/rmap.c


page_referenced_anon
page_lock_anon_vma
返回该匿名页
对应的 anon_vma
page_mapcount
返回有多少个进程
引用该页，即：
多少个页表项映射该页
遍历 anon_vma->head 链表
下的 anon_vma_chain 节点
通过 anon_vma_chain 的
vma 得到同一匿名页
映射到的进程虚拟区域
vma_address(page, vma)
返回该页在进程的
地址空间的虚拟地址
page_referenced_one
page_check_address
返回该地址对应的页表项
anon_vma->head 链表
遍历完后，返回总的引用计数
ptep_clear_flush_young_notify
清除页表项的
_PAGE_BIT_ACCESSED 标志
如果设置该标志，将引用计数加一




mm/swap.c

```
void mark_page_accessed(struct page *page)
{
    if (!PageActive(page) && !PageUnevictable(page) &&
            PageReferenced(page) && PageLRU(page)) {
        activate_page(page); //将页加入到活动的 lru 链表中
        ClearPageReferenced(page);
    } else if (!PageReferenced(page)) {
        SetPageReferenced(page);
    }
}
```

该函数的作用：

```
 * inactive,unreferenced    ->  inactive,referenced
 * inactive,referenced      ->  active,unreferenced
 * active,unreferenced      ->  active,referenced
```

activate_page 函数用于将页从不活动链表移到对应的活动链表。

## 3.收缩内存域

### *1.数据结构*

```
struct scan_control {
    unsigned long nr_scanned;       /* Incremented by the number of inactive pages that were scanned */
    unsigned long nr_reclaimed;     /* Number of pages freed so far during a call to shrink_zones() */
    unsigned long nr_to_reclaim;    /* How many pages shrink_list() should reclaim */
    gfp_t gfp_mask;                  /* This context's GFP mask */
    int may_writepage;
    int may_unmap;                   /* Can mapped pages be reclaimed? */
    int may_swap;                    /* Can pages be swapped as part of reclaim? */
    int swappiness;
    int order;
   .....
     nodemask_t  *nodemask;
};
```

部分元素解释：

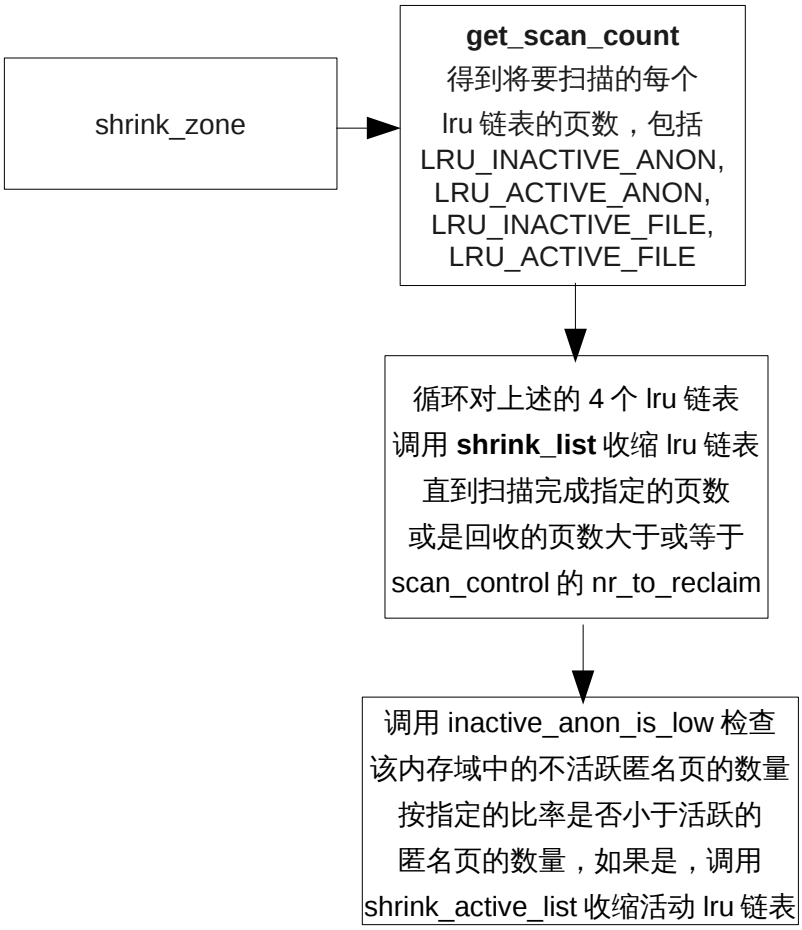
(1) nr_scanned:向调用者报告已经扫描到的不活动的页的数目。

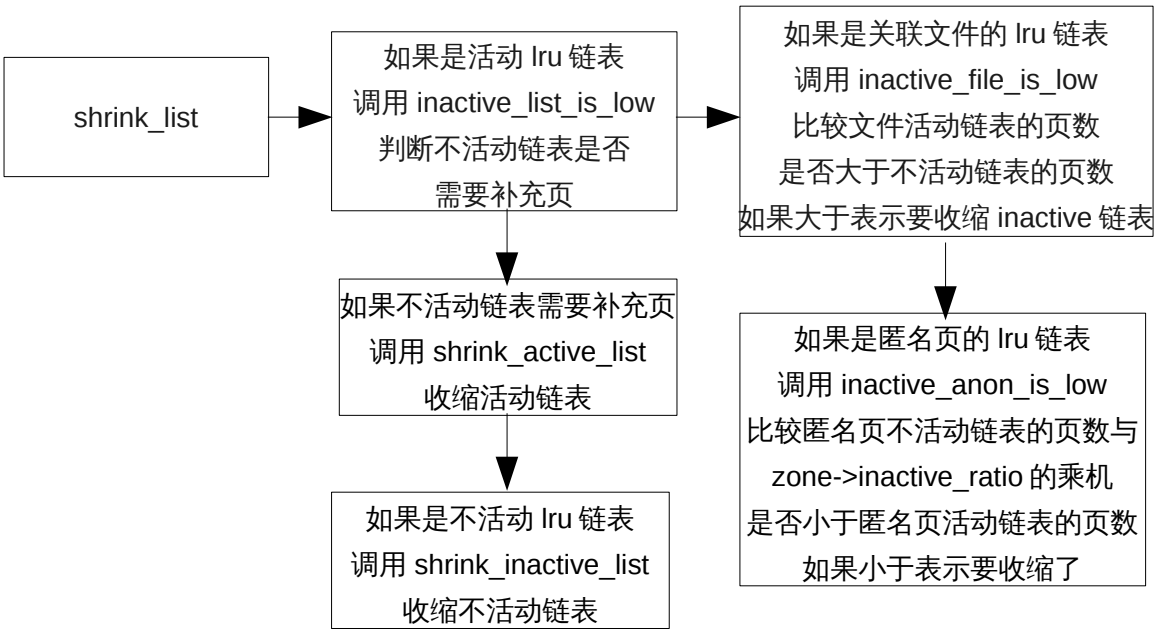
(2)may_writepage:指定了内核是否允许将页写出到后备存储器。

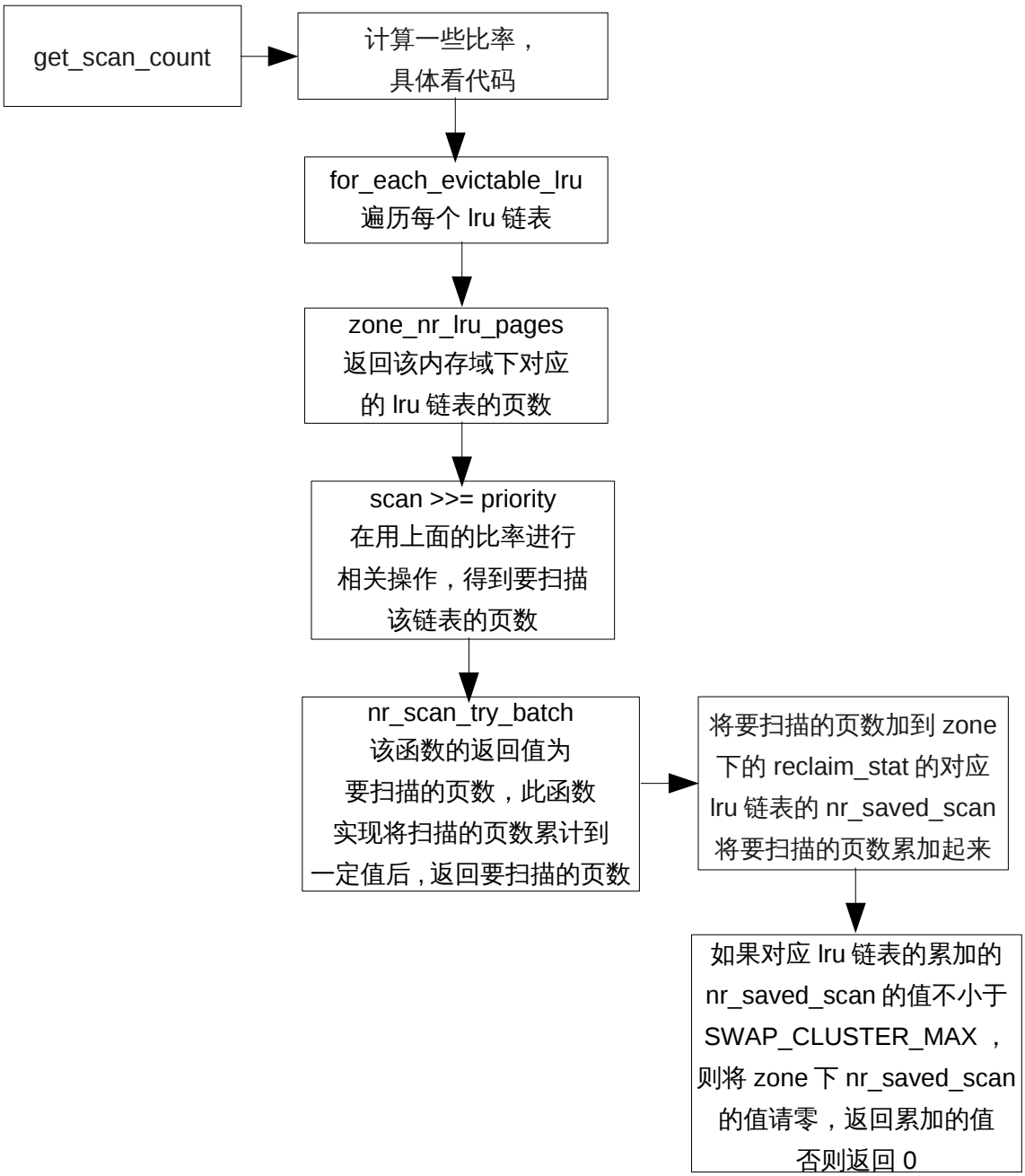
(3)swappiness:控制内核换出页的积极程度。

(4)order:内核按给定的阶来尝试回收一组内存页。

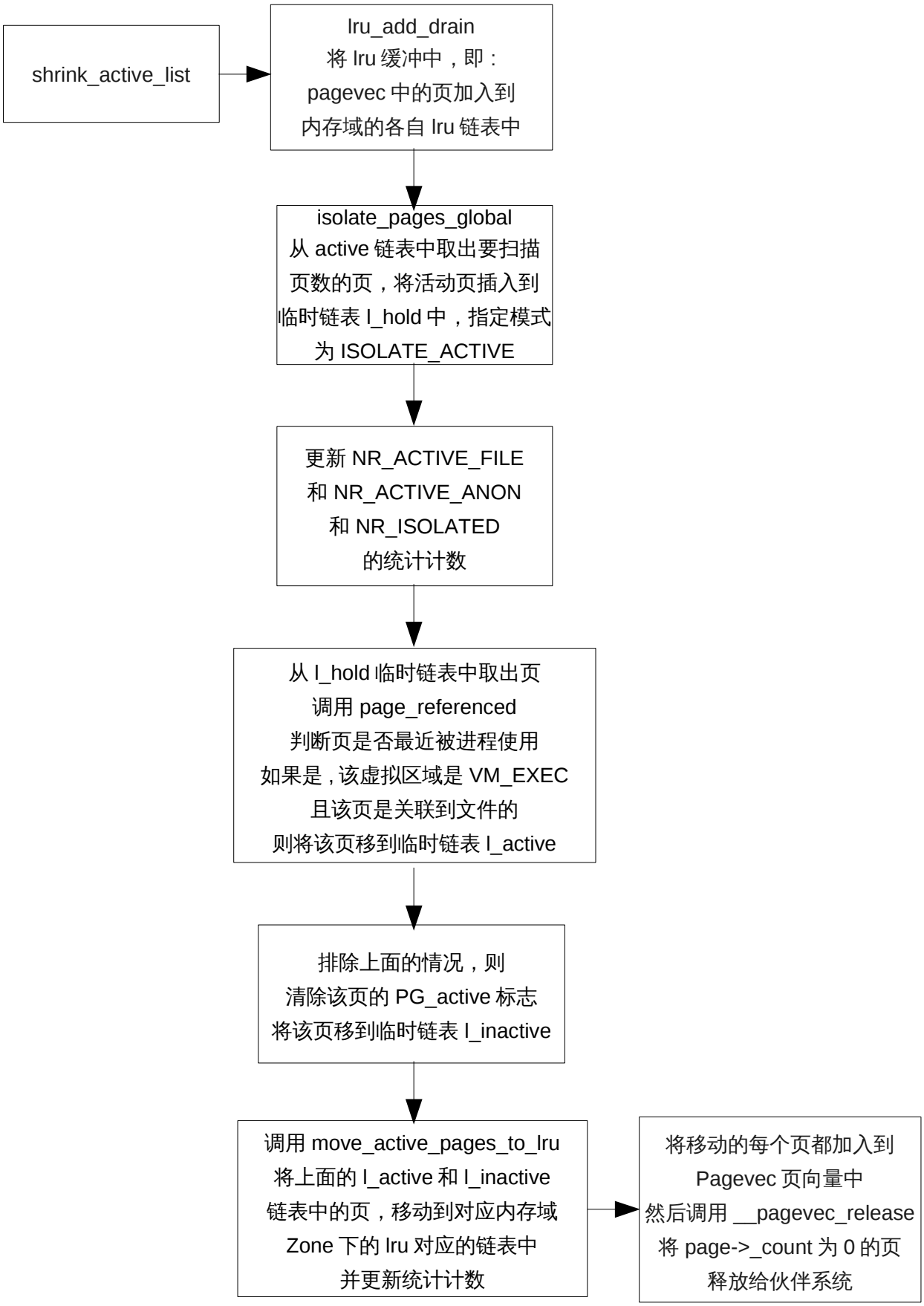
### *2.实现*


shrink_zone
get_scan_count
得到将要扫描的每个
lru 链表的页数，包括
LRU_INACTIVE_ANON,
LRU_ACTIVE_ANON,
LRU_INACTIVE_FILE,
LRU_ACTIVE_FILE
循环对上述的 4 个 lru 链表
调用 shrink_list 收缩 lru 链表
直到扫描完成指定的页数
或是回收的页数大于或等于
scan_control 的 nr_to_reclaim
调用 inactive_anon_is_low 检查
该内存域中的不活跃匿名页的数量
按指定的比率是否小于活跃的
匿名页的数量，如果是，调用
shrink_active_list 收缩活动 lru 链表


shrink_list
如果是活动 lru 链表
调用 inactive_list_is_low
判断不活动链表是否
需要补充页
如果是关联文件的 lru 链表
调用 inactive_file_is_low
比较文件活动链表的页数
是否大于不活动链表的页数
如果大于表示要收缩 inactive 链表
如果不活动链表需要补充页
调用 shrink_active_list
收缩活动链表
如果是匿名页的 lru 链表
调用 inactive_anon_is_low
比较匿名页不活动链表的页数与
zone->inactive_ratio 的乘机
是否小于匿名页活动链表的页数
如果小于表示要收缩了
如果是不活动 lru 链表
调用 shrink_inactive_list
收缩不活动链表


get_scan_count
计算一些比率，
具体看代码
for_each_evictable_lru
遍历每个 lru 链表
zone_nr_lru_pages
返回该内存域下对应
的 lru 链表的页数
scan >>= priority
在用上面的比率进行
相关操作，得到要扫描
该链表的页数
nr_scan_try_batch
该函数的返回值为
要扫描的页数，此函数
实现将扫描的页数累计到
一定值后，返回要扫描的页数
将要扫描的页数加到 zone
下的 reclaim_stat 的对应
lru 链表的 nr_saved_scan
将要扫描的页数累加起来
如果对应 lru 链表的累加的
nr_saved_scan 的值不小于
SWAP_CLUSTER_MAX ，
则将 zone 下 nr_saved_scan
的值请零，返回累加的值
否则返回 0

*1.*收缩活动链表

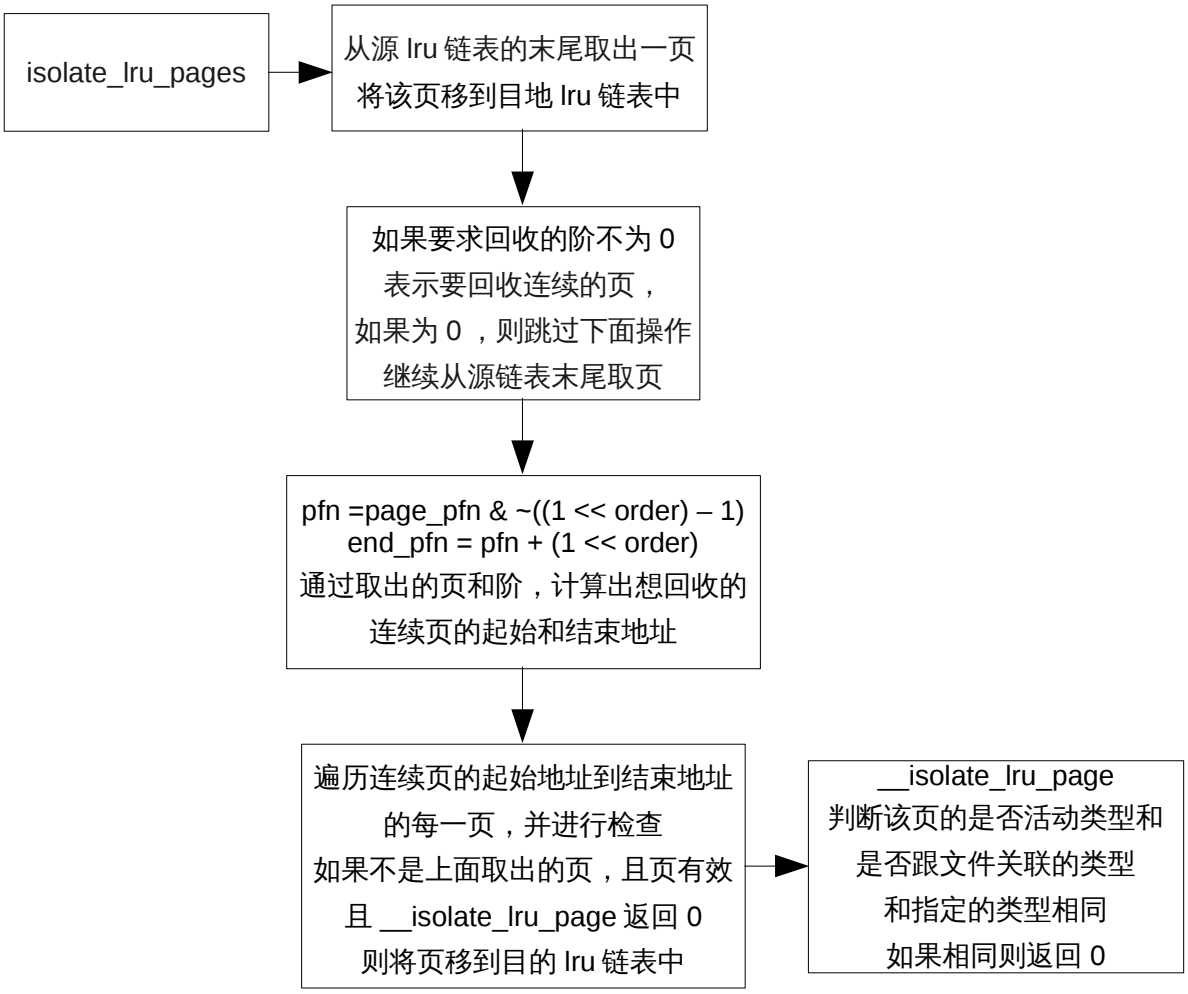

shrink_active_list
lru_add_drain
将 lru 缓冲中，即：
pagevec 中的页加入到
内存域的各自 lru 链表中
isolate_pages_global
从 active 链表中取出要扫描
页数的页，将活动页插入到
临时链表 l_hold 中，指定模式
为 ISOLATE_ACTIVE
更新 NR_ACTIVE_FILE
和 NR_ACTIVE_ANON
和 NR_ISOLATED
的统计计数
从 l_hold 临时链表中取出页
调用 page_referenced
判断页是否最近被进程使用
如果是，该虚拟区域是 VM_EXEC
且该页是关联到文件的
则将该页移到临时链表 l_active
排除上面的情况，则
清除该页的 PG_active 标志
将该页移到临时链表 l_inactive
调用 move_active_pages_to_lru
将上面的 l_active 和 l_inactive
链表中的页，移动到对应内存域
Zone 下的 lru 对应的链表中
并更新统计计数
将移动的每个页都加入到
Pagevec 页向量中
然后调用 __pagevec_release
将 page->_count 为 0 的页
释放给伙伴系统

使用临时链表可以降低锁的竞争性：

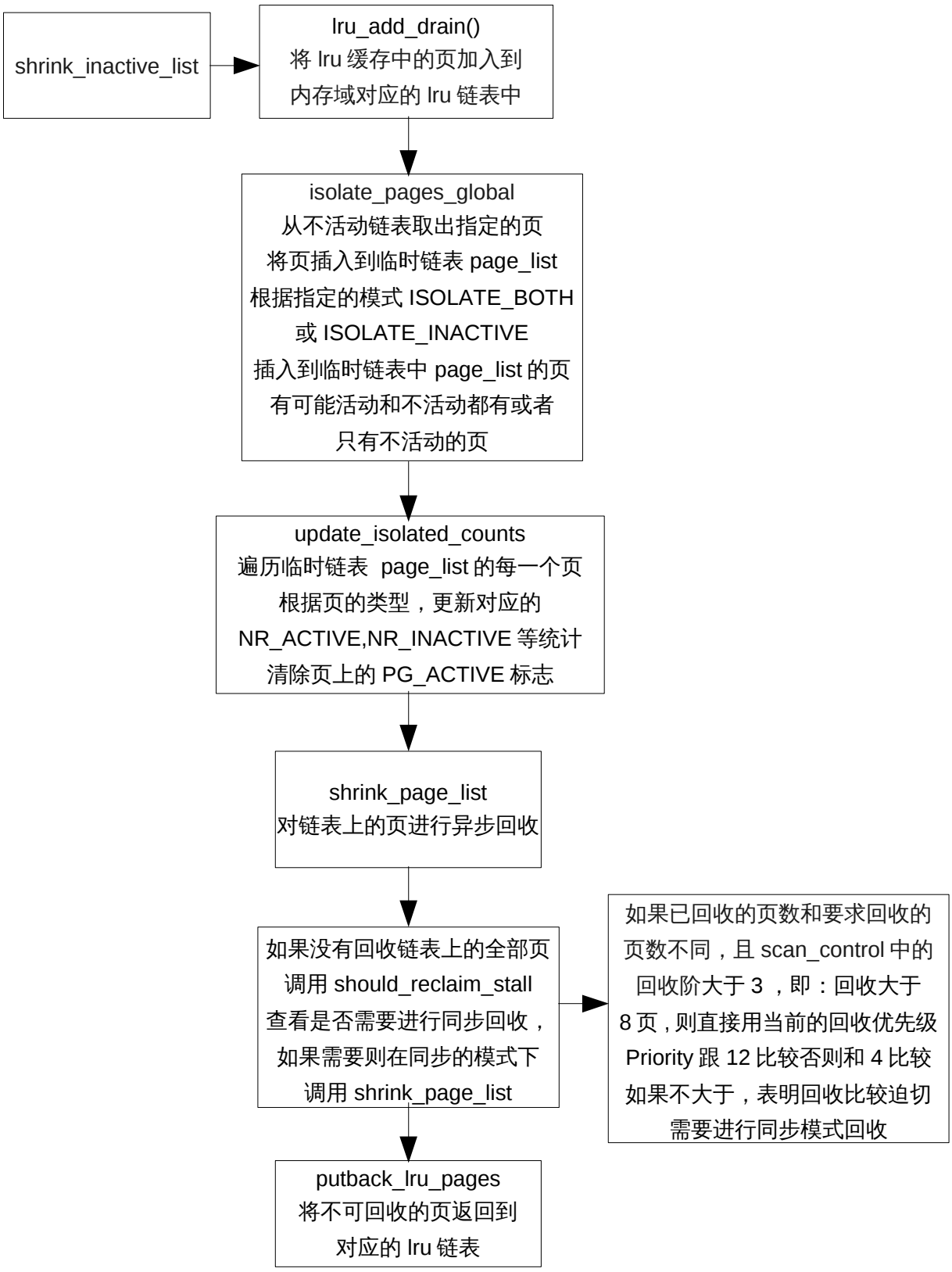
Both the active and inactive pages of a zone are kept on lists that need to be protected by a spinlock . When operations with the LRU lists are performed, they need to be locked, and one problem arises: The page reclaim code belongs to the hottest and most important paths in the kernel for many workloads, and lock contention is rather high. Therefore, the kernel need to work outside the lock as often as possible.

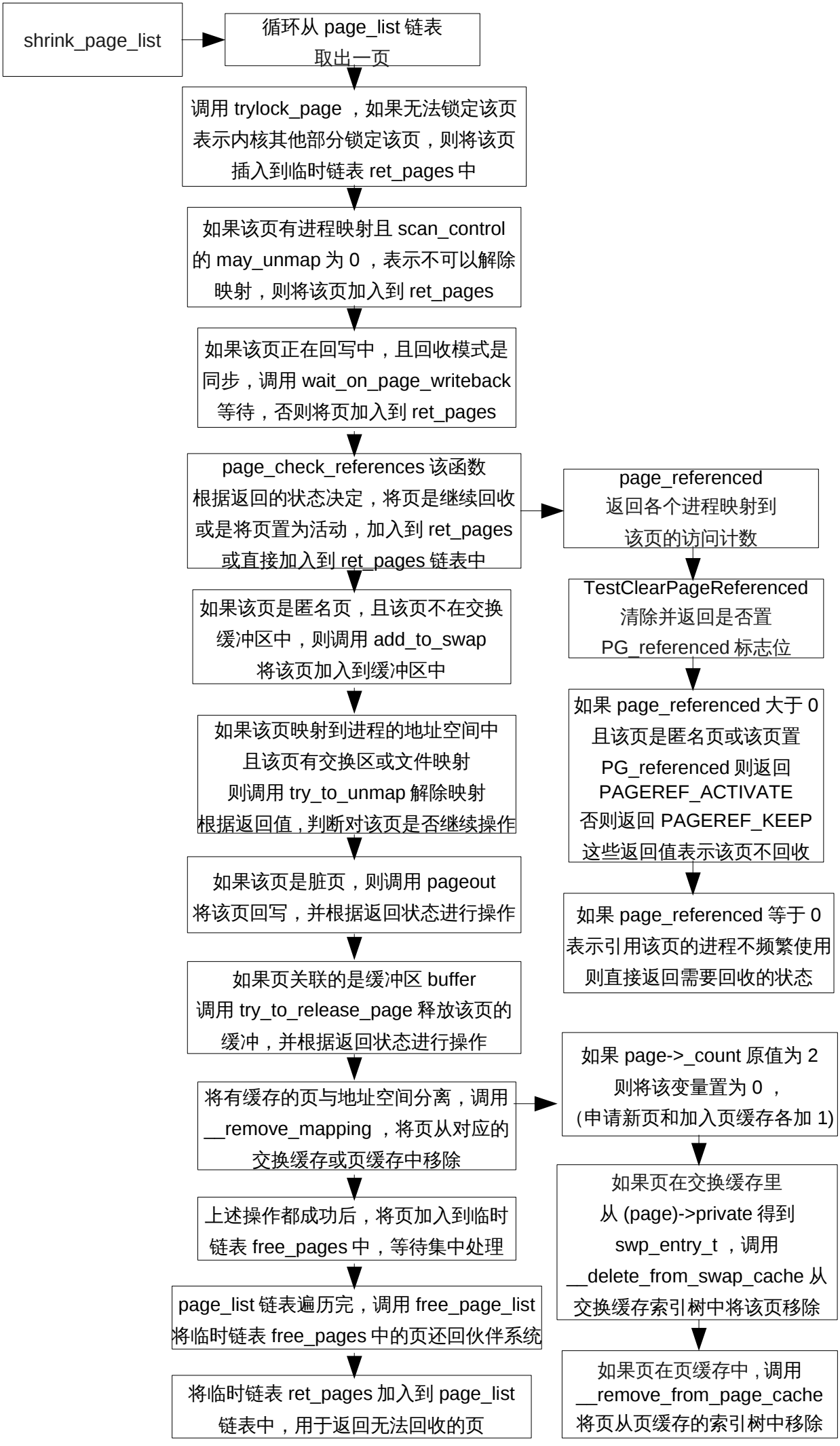
One optimization is to place all pages that are about to be analyzed in shrink_active_list and shrink_inactive_list on a local list, drop the lock, and proceed with the pages on the local list. Since they are not present on any global zone-specific list anymore, no other part of the kernel except the owner of the local list can touch them — the pages will not be affected by subsequent operations on the zone lists. Taking the zone list lock to work with the local list of pages is therefore not required.

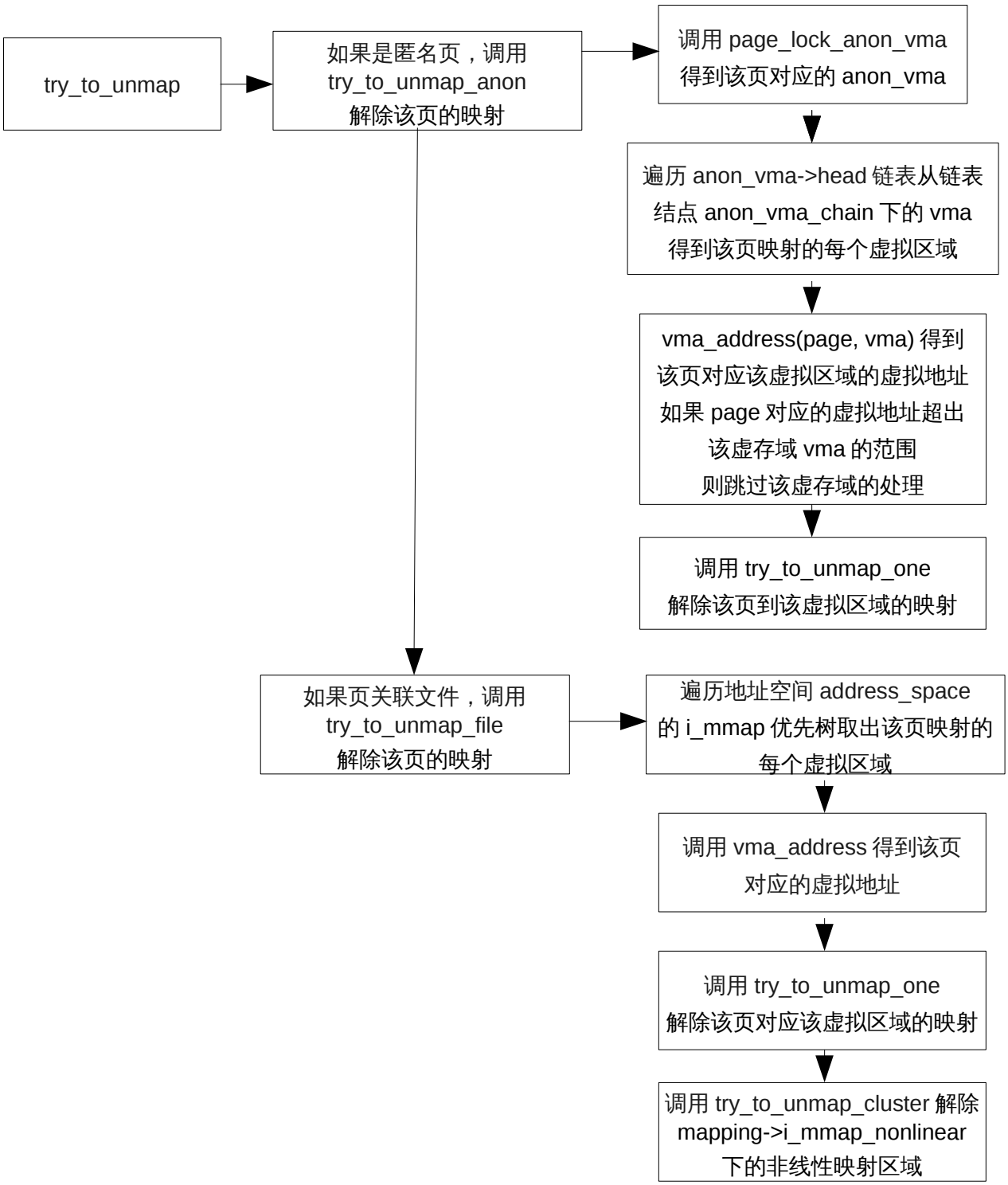
isolate_pages_global 函数实际调用 isolate_lru_pages

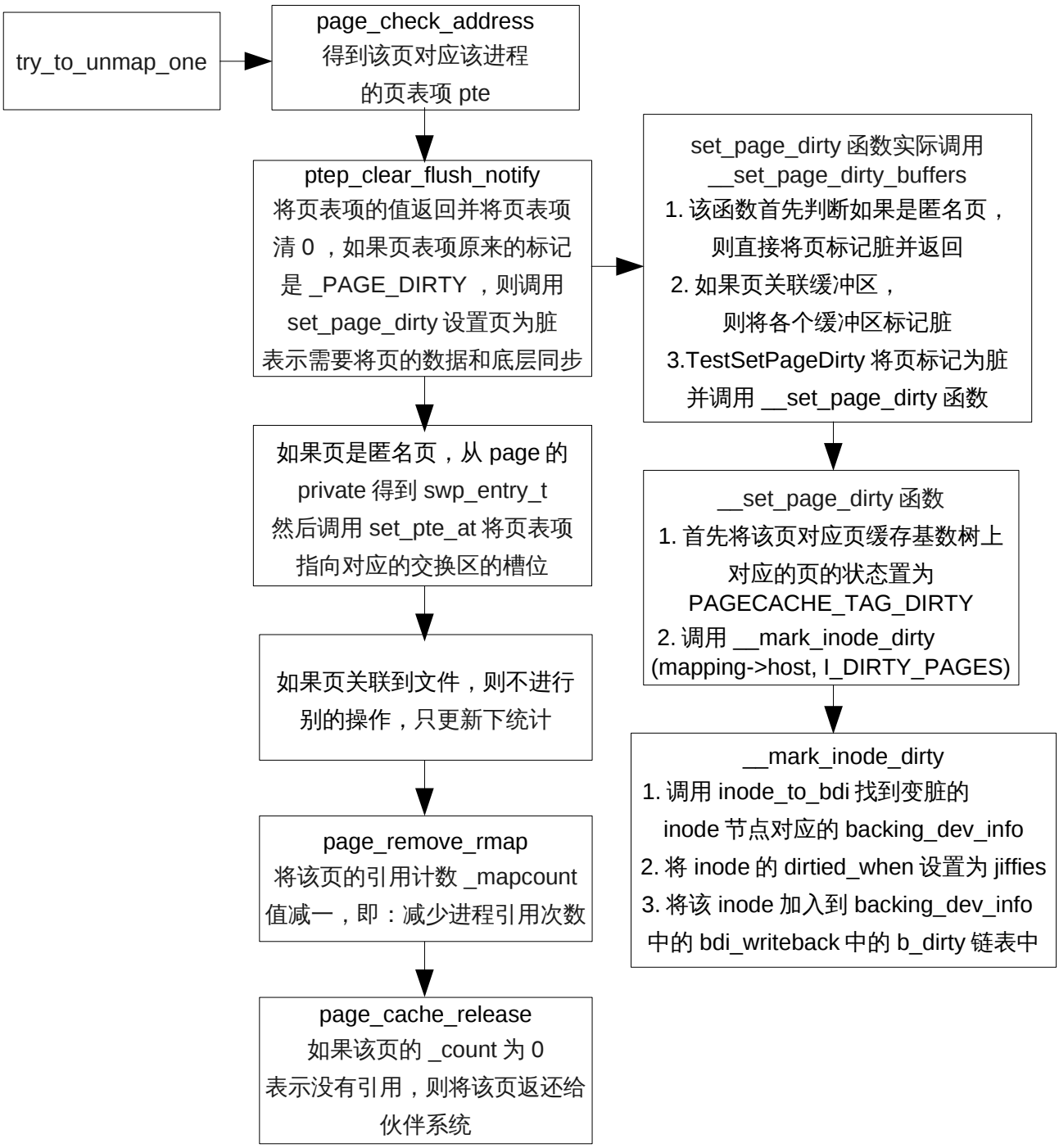


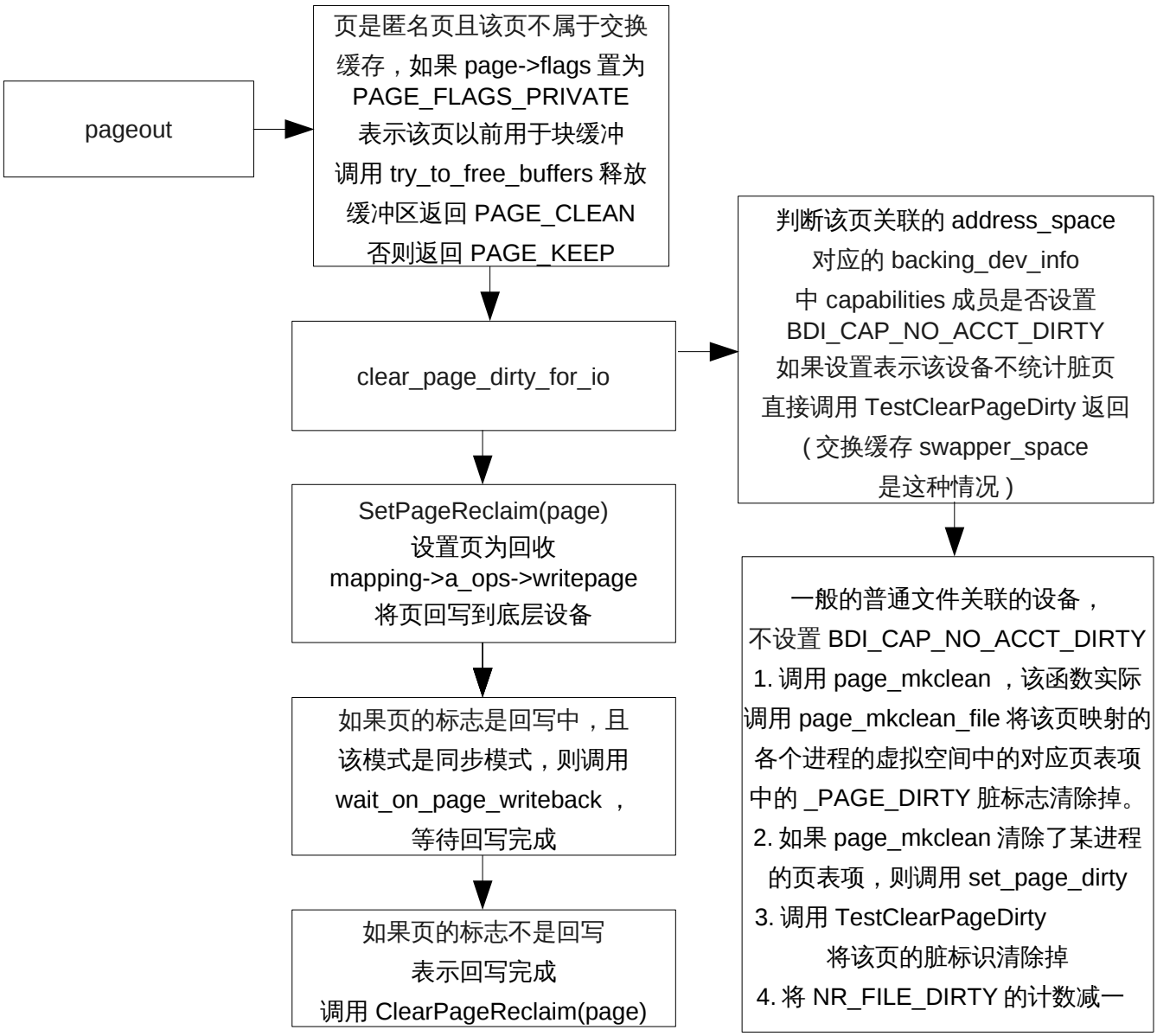

*2.*收缩不活动链表


shrink_inactive_list
lru_add_drain()
将 lru 缓存中的页加入到
内存域对应的 lru 链表中
isolate_pages_global
从不活动链表取出指定的页
将页插入到临时链表 page_list
根据指定的模式 ISOLATE_BOTH
或 ISOLATE_INACTIVE
插入到临时链表中 page_list 的页
有可能活动和不活动都有或者
只有不活动的页
update_isolated_counts
遍历临时链表 page_list 的每一个页
根据页的类型，更新对应的
NR_ACTIVE,NR_INACTIVE 等统计
清除页上的 PG_ACTIVE 标志
shrink_page_list
对链表上的页进行异步回收
如果没有回收链表上的全部页
调用 should_reclaim_stall
查看是否需要进行同步回收，
如果需要则在同步的模式下
调用 shrink_page_list
如果已回收的页数和要求回收的
页数不同，且 scan_control 中的
回收阶大于 3 ，即：回收大于
8 页，则直接用当前的回收优先级
Priority 跟 12 比较否则和 4 比较
如果不大于，表明回收比较迫切
需要进行同步模式回收
putback_lru_pages
将不可回收的页返回到
对应的 lru 链表


shrink_page_list
循环从 page_list 链表
取出一页
调用 trylock_page ，如果无法锁定该页
表示内核其他部分锁定该页，则将该页
插入到临时链表 ret_pages 中
如果该页有进程映射且 scan_control
的 may_unmap 为 0 ，表示不可以解除
映射，则将该页加入到 ret_pages
如果该页正在回写中，且回收模式是
同步，调用 wait_on_page_writeback
等待，否则将页加入到 ret_pages
page_check_references 该函数
根据返回的状态决定，将页是继续回收
或是将页置为活动，加入到 ret_pages
或直接加入到 ret_pages 链表中
page_referenced
返回各个进程映射到
该页的访问计数
TestClearPageReferenced
清除并返回是否置
PG_referenced 标志位
如果 page_referenced 大于 0
且该页是匿名页或该页置
PG_referenced 则返回
PAGEREF_ACTIVATE
否则返回 PAGEREF_KEEP
这些返回值表示该页不回收
如果 page_referenced 等于 0
表示引用该页的进程不频繁使用
则直接返回需要回收的状态
如果该页是匿名页，且该页不在交换
缓冲区中，则调用 add_to_swap
将该页加入到缓冲区中
如果该页映射到进程的地址空间中
且该页有交换区或文件映射
则调用 try_to_unmap 解除映射
根据返回值，判断对该页是否继续操作
如果该页是脏页，则调用 pageout
将该页回写，并根据返回状态进行操作
如果页关联的是缓冲区 buffer
调用 try_to_release_page 释放该页的
缓冲，并根据返回状态进行操作
将有缓存的页与地址空间分离，调用
__remove_mapping ，将页从对应的
交换缓存或页缓存中移除
如果 page->_count 原值为 2
则将该变量置为 0 ，
(申请新页和加入页缓存各加 1)
如果页在交换缓存里
从 (page)->private 得到
swp_entry_t ，调用
__delete_from_swap_cache 从
交换缓存索引树中将该页移除
如果页在页缓存中，调用
__remove_from_page_cache
将页从页缓存的索引树中移除
上述操作都成功后，将页加入到临时
链表 free_pages 中，等待集中处理
page_list 链表遍历完，调用 free_page_list
将临时链表 free_pages 中的页还回伙伴系统
将临时链表 ret_pages 加入到 page_list
链表中，用于返回无法回收的页

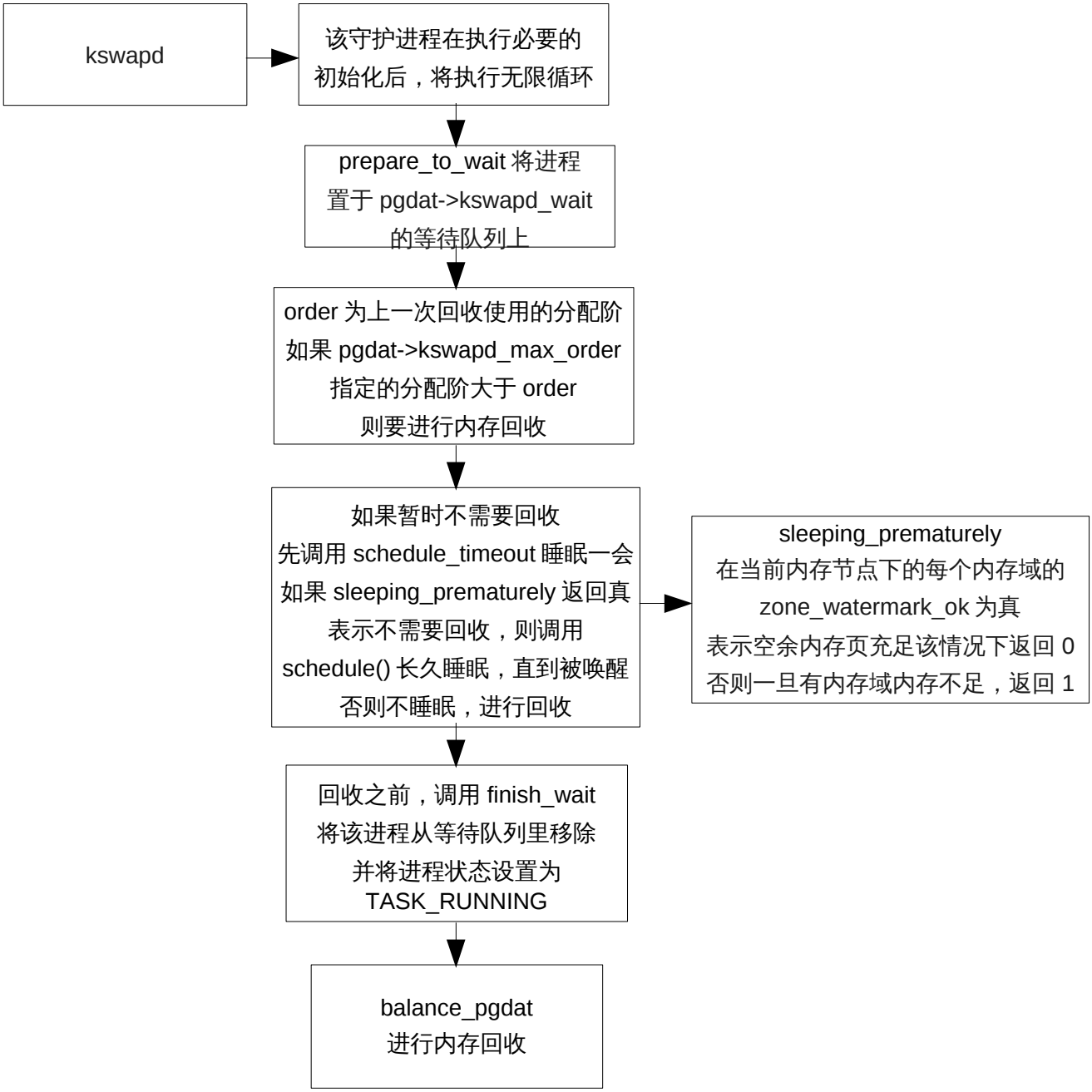

try_to_unmap
如果是匿名页，调用
try_to_unmap_anon
解除该页的映射
调用 page_lock_anon_vma
得到该页对应的 anon_vma
遍历 anon_vma->head 链表从链表
结点 anon_vma_chain 下的 vma
得到该页映射的每个虚拟区域
vma_address(page, vma) 得到
该页对应该虚拟区域的虚拟地址
如果 page 对应的虚拟地址超出
该虚存域 vma 的范围
则跳过该虚存域的处理
调用 try_to_unmap_one
解除该页到该虚拟区域的映射
如果页关联文件，调用
try_to_unmap_file
解除该页的映射
遍历地址空间 address_space
的 i_mmap 优先树取出该页映射的
每个虚拟区域
调用 vma_address 得到该页
对应的虚拟地址
调用 try_to_unmap_one
解除该页对应该虚拟区域的映射
调用 try_to_unmap_cluster 解除
mapping->i_mmap_nonlinear
下的非线性映射区域

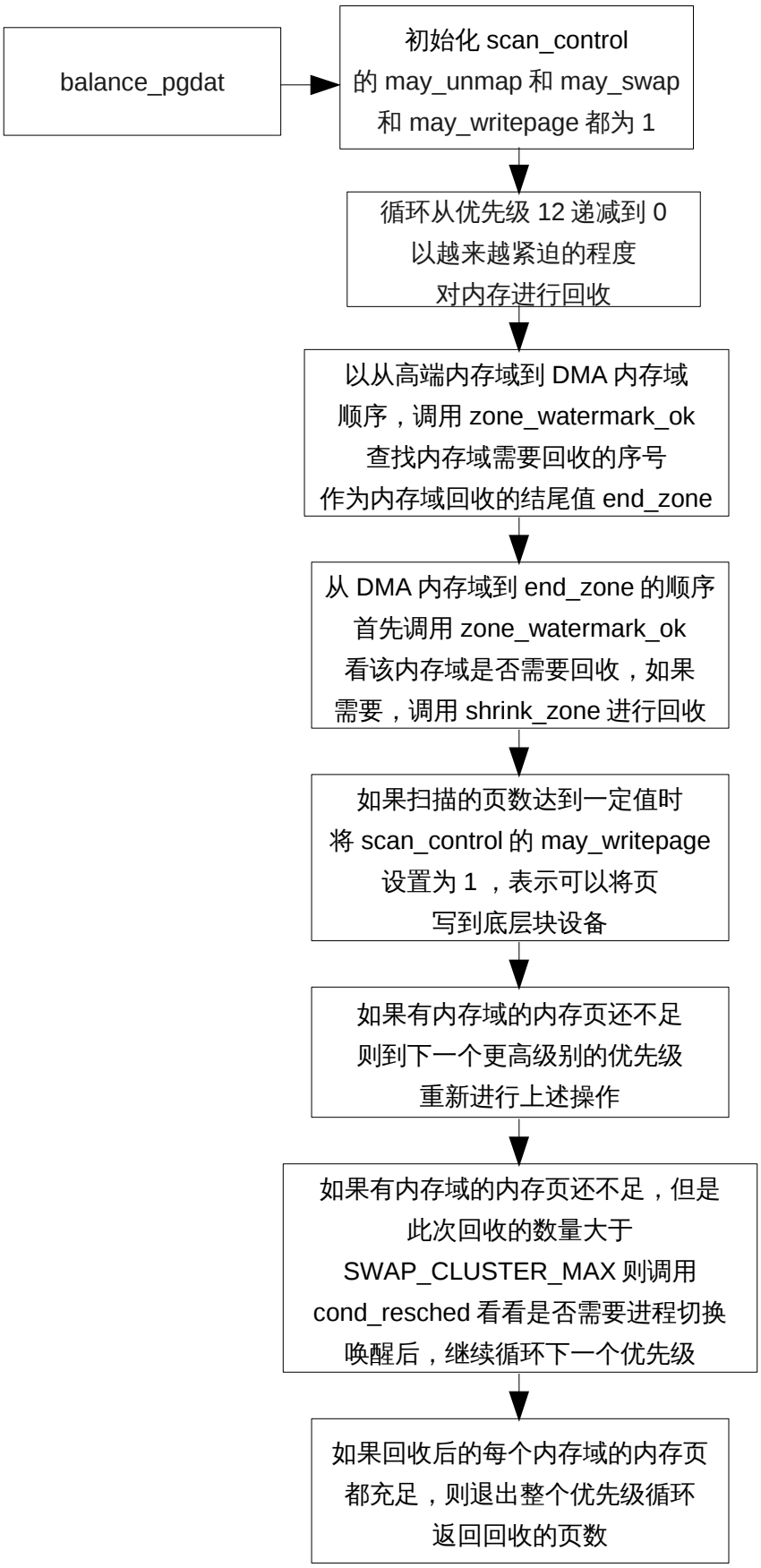

try_to_unmap_one
page_check_address
得到该页对应该进程
的页表项 pte
ptep_clear_flush_notify
将页表项的值返回并将页表项
清 0 ，如果页表项原来的标记
是 _PAGE_DIRTY ，则调用
set_page_dirty 设置页为脏
表示需要将页的数据和底层同步
如果页是匿名页，从 page 的
private 得到 swp_entry_t
然后调用 set_pte_at 将页表项
指向对应的交换区的槽位
如果页关联到文件，则不进行
别的操作，只更新下统计
page_remove_rmap
将该页的引用计数 _mapcount
值减一，即：减少进程引用次数
page_cache_release
如果该页的 _count 为 0
表示没有引用，则将该页返还给
伙伴系统
set_page_dirty 函数实际调用
__set_page_dirty_buffers
1. 该函数首先判断如果是匿名页，
则直接将页标记脏并返回
2. 如果页关联缓冲区，
则将各个缓冲区标记脏
3.TestSetPageDirty 将页标记为脏
并调用 __set_page_dirty 函数
__set_page_dirty 函数
1. 首先将该页对应页缓存基数树上
对应的页的状态置为
PAGECACHE_TAG_DIRTY
2. 调用 __mark_inode_dirty
(mapping->host, I_DIRTY_PAGES)
__mark_inode_dirty
1. 调用 inode_to_bdi 找到变脏的
inode 节点对应的 backing_dev_info
2. 将 inode 的 dirtied_when 设置为 jiffies
3. 将该 inode 加入到 backing_dev_info
中的 bdi_writeback 中的 b_dirty 链表中


pageout
页是匿名页且该页不属于交换缓存，如果 page->flags 置为 PAGE_FLAGS_PRIVATE 表示该页以前用于块缓冲 调用 try_to_free_buffers 释放缓冲区返回 PAGE_CLEAN 否则返回 PAGE_KEEP
clear_page_dirty_for_io
判断该页关联的 address_space 对应的 backing_dev_info 中 capabilities 成员是否设置 BDI_CAP_NO_ACCT_DIRTY 如果设置表示该设备不统计脏页 直接调用 TestClearPageDirty 返回 ( 交换缓存 swapper_space 是这种情况 )
SetPageReclaim(page) 设置页为回收 mapping->a_ops->writepage 将页回写到底层设备
一般的普通文件关联的设备， 不设置 BDI_CAP_NO_ACCT_DIRTY 1. 调用 page_mkclean ，该函数实际调用 page_mkclean_file 将该页映射的各个进程的虚拟空间中的对应页表项中的 _PAGE_DIRTY 脏标志清除掉。 2. 如果 page_mkclean 清除了某进程的页表项，则调用 set_page_dirty 3. 调用 TestClearPageDirty 将该页的脏标识清除掉 4. 将 NR_FILE_DIRTY 的计数减一
如果页的标志是回写中，且该模式是同步模式，则调用 wait_on_page_writeback ， 等待回写完成
如果页的标志不是回写 表示回写完成 调用 ClearPageReclaim(page)
free_page_list
定义页向量 freed_pvec 用于集中式释放要返还给伙伴系统中的页
循环取出临时链表 free_pages 中的每一页，将该页加入到 freed_pvec 中
如果页向量 freed_pvec 加满 则调用 __pagevec_free 一起释放页向量中的 所有页给伙伴系统 实际调用 free_hot_cold_page

# 4.发起内存回收

### *1.用 kswapd 进行周期行内存回收*

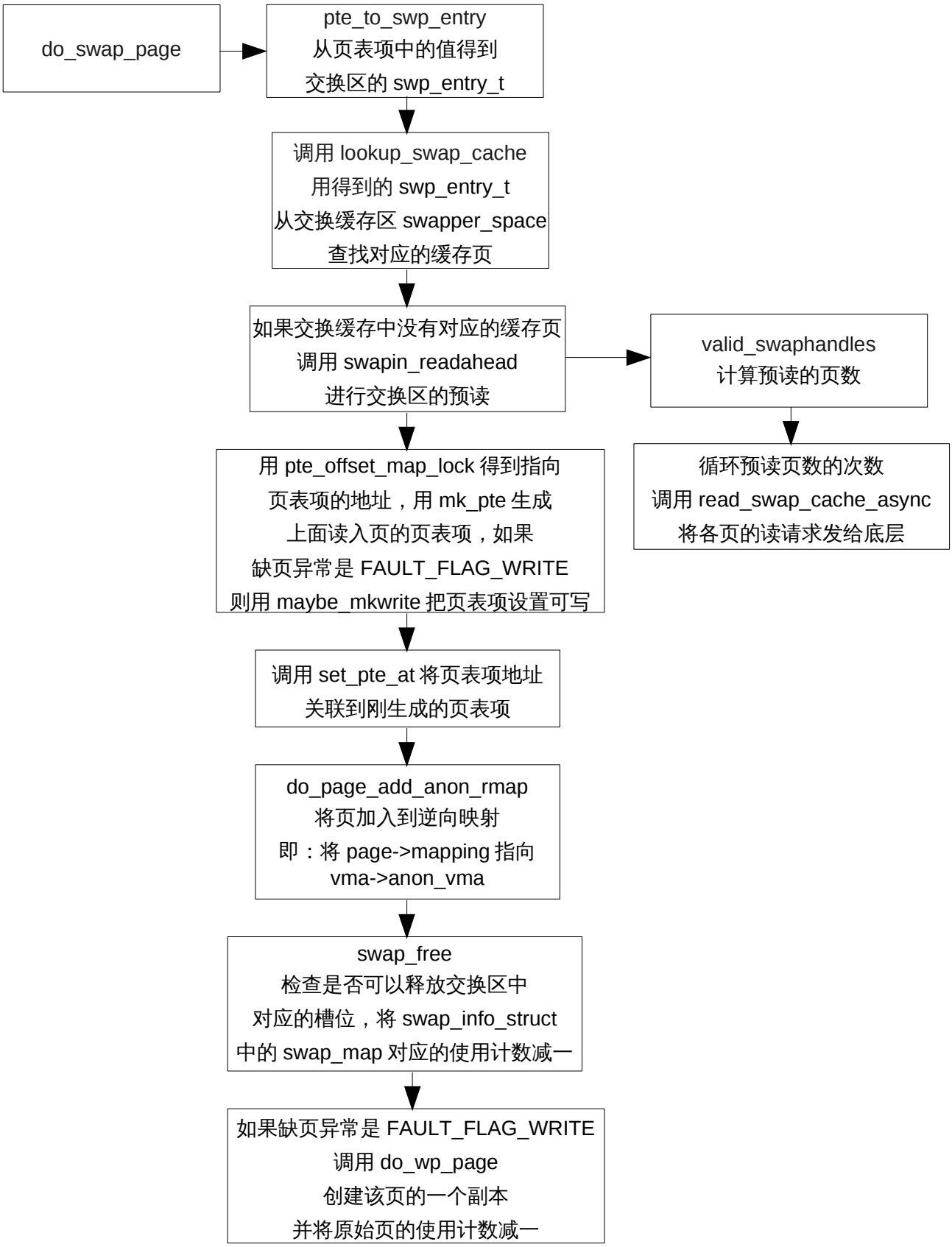




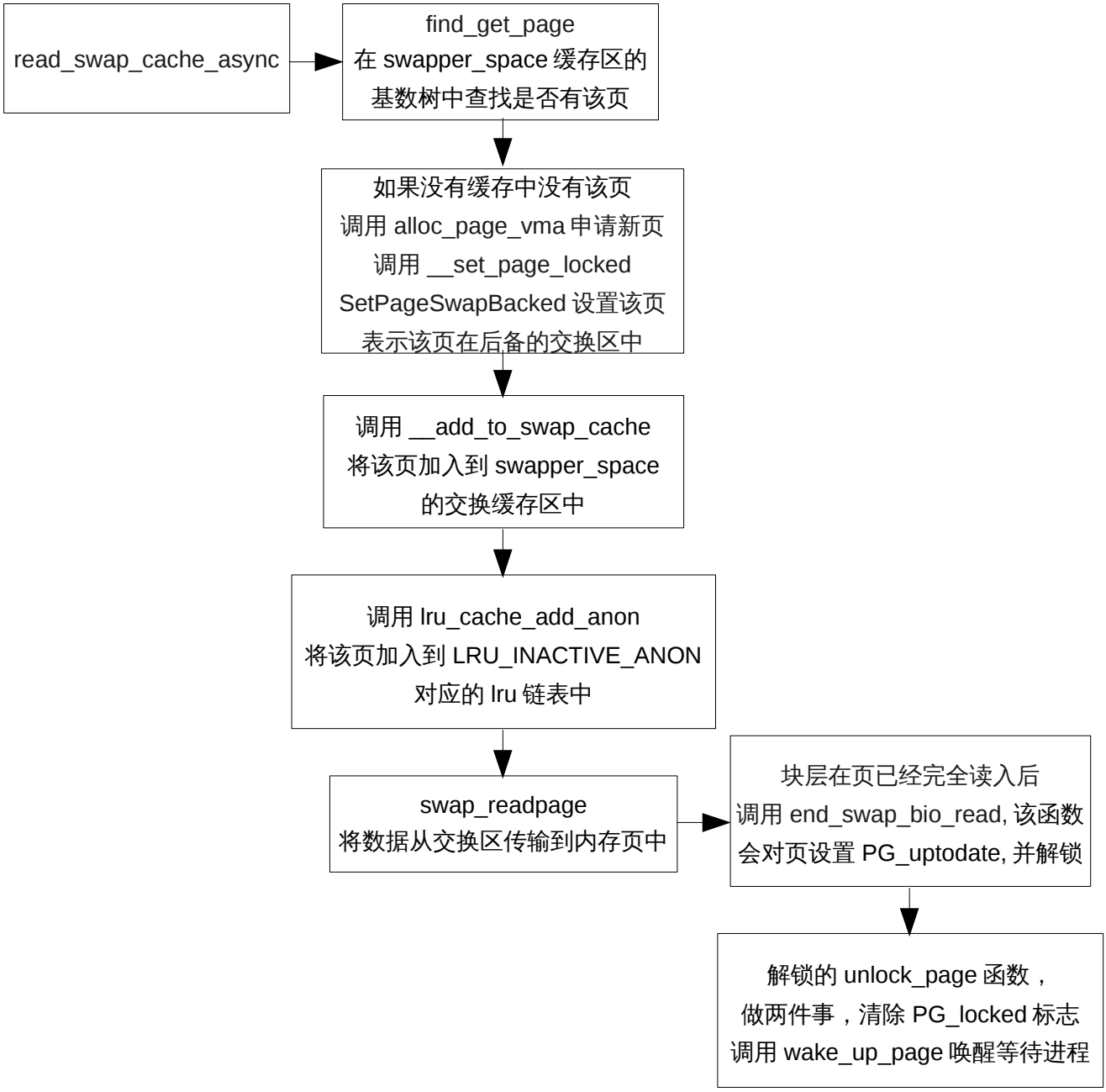

## 5.处理交换缺页异常

处理缺页异常中的 handle_pte_fault 函数，如果页不在物理内存中, 但页表项不为空,且页表项没有置 _PAGE_FILE ，表示页已经换出到某个交换区调用 do_swap_page 将页换入

mm/memory.c




mm/swap_state.c


read_swap_cache_async
find_get_page
在 swapper_space 缓存区的
基数树中查找是否有该页
如果没有缓存中没有该页
调用 alloc_page_vma 申请新页
调用 __set_page_locked
SetPageSwapBacked 设置该页
表示该页在后备的交换区中
调用 __add_to_swap_cache
将该页加入到 swapper_space
的交换缓存区中
调用 lru_cache_add_anon
将该页加入到 LRU_INACTIVE_ANON
对应的 lru 链表中
swap_readpage
将数据从交换区传输到内存页中
块层在页已经完全读入后
调用 end_swap_bio_read, 该函数
会对页设置 PG_uptodate, 并解锁
解锁的 unlock_page 函数，
做两件事，清除 PG_locked 标志
调用 wake_up_page 唤醒等待进程

# 最后的寄语：

也许十年后，看到自己所做的会觉得好笑。不过现在我觉得这个是我自己想做的。让未来来见证吧。

一个伟大人的演讲：

史蒂夫 乔布斯(Steve Jobs)在斯坦福大学 2005 年毕业典礼上的演讲

史蒂夫·乔布斯在斯坦福大学的演讲

我今天很荣幸能和你们一起参加毕业典礼，斯坦福大学是世界上最好的大学之一。我从来没有从大学中毕业。说实话，今天也许是在我的生命中离大学毕业最近的一天了。今天我想向你们讲述我生活中的三个故事。不是什么大不了的事情,只是三个故事而已。

第一个故事是关于如何把生命中的点点滴滴串连起来。

我在 Reed 大学读了六个月之后就退学了，但是在十八个月以后——我真正的作出退学决定之前，我还经常去学校。我为什么要退学呢？

故事从我出生的时候讲起。我的亲生母亲是一个年轻的，没有结婚的大学毕业生。她决定让别人收养我，她十分想让我被大学毕业生收养。所以在我出生的时候，她已经做好了一切的准备工作。所以我的养父母突然在半夜接到了一个电话:“我们现在这儿有一个不小心生出来的男婴,你们想要他吗？”他们回答道: “当然！”但是我亲生母亲随后发现，我的养母从来没有上过大学,我的养父 甚至从没有读过高中。她拒绝签这个收养合同。只是在几个月以后,我的父母答应她一定要让我上大学,那个时候她才勉强同意。

在十七岁那年,我真的上了大学。但是我很愚蠢的选择了一个几乎和你们斯坦福大学一样贵的学校, 我父母还处于蓝领阶层，他们几乎把所有积蓄都花在了我的学费上面。在六个月后, 我已经看不到其中的价值所在。我不知道我真正想要做什么,我也不知道大学能怎样帮助我找到答案。但是在这里，我几乎花光了我父母这一辈子的 全部积蓄。所以我决定要退学，我觉得这是个正确的决定。不能否认，我当时确实非常的害怕，但是现在回头看看，那的确是我这一生中最棒的一个决定。在我做出退学决定的那一刻，我终于可以不必去读那些令我提不起丝毫兴趣的课程了。然后我可以开始去修那些看起来有点意思的课程。

但是这并不是那么浪漫。我失去了我的宿舍,所以我只能在朋友房间的地板上面睡觉,我去捡可以换 5 美分的可乐罐，仅仅为了填饱肚子, 在星期天的晚上,我需要走七英里的路程，穿过这个城市到 Hare Krishna 神庙（注：位于纽约 Brooklyn 下城）,只是为了能吃上好饭——这个星期唯一一顿好一点的饭，我喜欢那里的饭菜。

我跟着我的直觉和好奇心走, 遇到的很多东西,此后被证明是无价之宝。让我给你们举一个例子吧：

Reed 大学在那时提供也许是全美最好的美术字课程。在这个大学里面的每个海报, 每个抽屉的标签上面全都是漂亮的美术字。因为我退学了, 不必去上正规的课程, 所以我决定去参加这个课程，去学学怎样写出漂亮的美术字。我学到了 san serif 和 serif 字体, 我学会了怎么样在不同的字母组合之中改变空白间距, 还有怎么样才能作出最棒的印刷式样。那种美好、历史感和艺术精妙，是科学永远不能捕捉到的, 我发现那实在是太迷人了。

当时看起来这些东西在我的生命中，好像都没有什么实际应用的可能。但是十年之后,当我们在设计第一台 Macintosh 电脑的时候,就不是那样了。我把当时我学的那些 东西全都设计进了 Mac。那是第一台使用了漂亮的印刷字体的电脑。如果我当时没有退学, 就不会有机会去参加这个我感兴趣的美术字课程,

Mac 就不会有这么多丰富的字体，以及赏心悦目的字体间距。因 为 Windows 只是照抄了 Mac，所以现在个人电脑才能有现在这么美妙的字型。

当然我在大学的时候，还不可能把从前的点点滴滴串连起来,但是当我十年后回顾这一切的时候,真的豁然开朗了。

再次说明的是,你在向前展望的时候不可能将这些片断串连起来；你只能在回顾的时候将点点滴滴串连起来。所以你必须相信这些片断会在你未来的某一天串连起来。你必须要相信某些东西：你的勇气、目的、生命、因缘.....这个过程从来没有令我失望, 只是让我的生命更加地与众不同。

我的第二个故事是关于爱和失去。

我非常幸运, 因为我在很早的时候就找到了我钟爱的东西。Woz 和我在二十岁的时候就在父母的车库里面开创了苹果公司。我们工作得很努力, 十年之后, 这个公司从那两个车库中的穷小子发展到了超过四千名的雇员、价值超过二十亿的大公司。在公司成立的第九年,我们刚刚发布了最好的产品,那就是 Macintosh。我也快要到三十岁了。在那一年, 我被炒了鱿鱼。你怎么可能被你自己创立的公司炒了鱿鱼呢? 嗯,在苹果快速成长的时候，我们雇用了一个很有天分的家伙和我一起管理这个公司, 在最初的几年,公司运转的很好。但是后来我们对未来的看法发生了分歧, 最终我们吵了起来。当争吵到不可开交的时候, 董事会站在了他的那一边。所以在三十岁的时候, 我被炒了。在这么多人目光下我被炒了。在而立之年，我生命的全部支柱离自己远去, 这真是毁灭性的打击。

在最初的几个月里，我真是不知道该做些什么。我觉得我很令上一代的创业家们很失望，我把他们交给我的接力棒弄丢了。我和创办惠普的 David Pack、创办 Intel 的 Bob Noyce 见面，并试图向他们道歉。我把事情弄得糟糕透顶了。但是我渐渐发现了曙光，我仍然喜爱我从事的这些东西。苹果公司发生的这些事情丝毫的没有改变这些, 一点也没有。我被驱逐了，但是我仍然钟爱我所做的事情。所以我决定从头再来。

我当时没有觉察, 但是事后证明，从苹果公司被炒是我这辈子发生的最棒的事情。因为，作为一个成功者的负重感被作为一个创业者的轻松感觉所重新代替, 没有比这更确定的事情了。这让我觉得如此自由, 进入了我生命中最有创造力的一个阶段。

在接下来的五年里，我创立了一个名叫 NeXT 的公司, 还有一个叫 Pixar 的公司, 然后和一个后来成为我妻子的优雅女人相识。Pixar 制作了世界上第一个用电脑制作的动画电影——“玩具总动员”，Pixar 现在也是世界上最成功的电脑制作工作室。 在后来的一系列运转中,Apple 收购了 NeXT，然后我又回到了 Apple 公司。我们在 NeXT 发展的技术在 Apple 的今天的复兴之中发挥了关键的作用。而且，我还和 Laurence 一起建立了一个幸福完美的家庭。

我可以非常肯定,如果我不被 Apple 开除的话，这些事情一件也不会发生的。这个良药的味道实在是太苦了，但是我想病人需要这个药。有些时候, 生活会拿起一块砖头向你的脑袋上猛拍一下。不要失去信仰。我很清楚唯一使我一直走下去的，就是我做的事情令我无比钟爱。你需要去找到你所爱的东西。对于工作是如此，对于你的爱人也是如此。你的工作将会占据生活中很大的一部分。你只有相信自己所做的是伟大的工作，你才能怡然自得。如果你现在还没有找到，那么继续找、不要停下来，只要全心全意的去找，在你找到的时候，你的心会告诉你的。就像任何真诚的关系，随着岁月的流逝只会越来越紧密。所以继续找，直到你找到它，不要停下来！我的第三个故事是关于死亡的。

当我十七岁的时候, 我读到了一句话：“如果你把每一天都当作生命中最后一天去生活的话,那么有一天你会发现你是正确的。”这句话给我留下了一个印象。从那时开始,过了 33 年，我在每天早晨都会对着镜子问自己:“如果今天是我生命中的最后一天, 你会不会完成你今天想做的事情呢？”当答案连续多天是

“No”的时候, 我知道自己需要改变某些事情了。

“记住你即将死去”是我一生中遇到的最重要箴言。它帮我指明了生命中重要的选择。因为几乎所有的事情, 包括所有的荣誉、所有的骄傲、所有对难堪和失败的恐惧,这些在死亡面前都会消失。我看到的是留下的真正重要的东西。你有时候会思考你将会失去某些东西, “记住你即将死去”是我知道的避免这些想法的最好办法。你已经赤身裸体了，你没有理由不去跟随自己内心的声音。

大概一年以前, 我被诊断出癌症。我在早晨七点半做了一个检查, 检查清楚的显示在我的胰腺有一个肿瘤。我当时都不知道胰腺是什么东西。医生告诉我那很可能是一种无法治愈的癌症, 我还有三到六个月的时间活在这个世界上。我的医生叫我回家, 然后整理好我的一切, 那是医生对临终病人的标准程序。那意味着你将要把未来十年对你小孩说的话在几个月里面说完.；那意味着把每件事情都安排好, 让你的家人会尽可能轻松的生活；那意味着你要说“再见了”。

我拿着那个诊断书过了一整天，那天晚上我作了一个活切片检查，医生将一个内窥镜从我的喉咙伸进去,通过我的胃, 然后进入我的肠子, 用一根针在我的胰腺上的肿瘤上取了几个细胞。我当时是被麻醉的，但是我的妻子在那里, 后来告诉我，当医生在显微镜下观察这些细胞的时候他们开始尖叫, 因为这些细胞最后竟然是一种非常罕见的可以用手术治愈的胰腺癌症细胞。我做了这个手术，现在我痊愈了。

那是我最接近死亡的时候, 我希望这也是以后的几十年最接近的一次。从死亡线上又活了过来, 我可以比以前把死亡只当成一 种想象中的概念的时候，更肯定一点地对你们说：

没有人愿意死, 即使人们想上天堂, 也不会为了去那里而死。但是死亡是我们每个人共同的终点。从来没有人能够逃脱它。也应该如此。因为死亡就是生命中最好的一个发明。它将旧的清除以便给新的让路。你们现在是新的, 但是从现在开始不久以后, 你们将会逐渐的变成旧的然后被送离人生舞台。我很抱歉这很戏剧性, 但是这十分的真实。

你们的时间很有限, 所以不要将他们浪费在重复其他人的生活上。不要被教条束缚,那意味着你和其他人思考的结果一起生活。不要被其他人喧嚣的观点掩盖你真正的内心的声音。还有最重要的是, 你要有勇气去听从你直觉和心灵的指示——它们在某种程度上知道你想要成为什么样子，所有其他的事情都是次要的。

当我年轻的时候，有一本叫做“整个地球的目录”振聋发聩的杂志，它是我们那一代人的圣经之一。它是一个叫 Stewart Brand 的家伙在离这里不远的 Menlo Park 编辑的，他象诗一般神奇地将这本书带到了这个世界。那是六十年代后期, 在个人电脑出现之前, 所以这本书全部是用打字机,、剪刀还有偏光镜制造的。有点像用软皮包装的 Google, 在 Google 出现三十五年之前：这是理想主义的，其中有许多灵巧的工具和伟大的想法。

Stewart 和他的伙伴出版了几期的“整个地球的目录”，当它完成了自己使命的时候，他们做出了最后一期的目录。那是在七十年代的中期, 我正是你们的年纪。在最后一期的封底上是清晨乡村公路的照片（如果你有冒险精神的话，你可以自己找到这条路的），在照片之下有这样一段话：“求知若饥，虚心若愚。”这是他们停止了发刊的告别语。“求知若饥，虚心若愚。（stay hungry,stay foolish）”我总是希望自己能够那样,现在，在你们即将毕业，开始新的旅程的时候，我也希望你们能这样：

求知若饥，虚心若愚。

# 作者及本文档

陈闯，邮箱：chenchuang0721@gmail.com  chenchuang0721@126.com

本文档，只是我浏览内核的内存方面的代码的总结，其中的截图和部分文字摘自相应的参考书籍。其中的每个函数解释的图，都为我浏览代码时画得，有误的地方我会在以后浏览代码时进行更改，你也可以跟我联系，告诉我错误，我会非常感激的。

浏览的代码版本为 2.6.36.2

# 遗留问题

1.第一次使用汇编在 wapper_pg_dir 中初始化的部分内核页表项和第二次使用 init_memory_mapping 的初始化的低内存区域的全部页表项，第二次初始化，是不是把第一次都覆盖了。

2.初始化阶段，用于存放初始的页表项的内存空间和第二次完全初始化内核页表项的内存空间和存放 bootmem 的位索引的空间，一旦被再次分配使用怎么办。这些区间在 early_node_map 中都为可用的空间，在 free_bootmem_with_active_regions 中，都会标记这些内存空间为可用的空闲内存。

3.__vunmap 函数

调用该函数用于解除 vmalloc 建立的虚存区域，调用 try_purge_vmap_area_lazy，使用惰性释放的方法，把该虚存对应的内核页表项值清 0. 我的问题是调用__vunmap ，已经把该块虚存对应的 page 页，释放给伙伴系统了，该物理页的状态已经是空闲了，而内核对应的页表项，没有立即清 0，此时访问该虚存，是不是又会访问该物理页。