

Linux 物理内存管理

Linux version: 2.4.18

By 金正操

version: 0.1 2005/6/14

目录

0 前言	2
1 物理页框管理.....	3
1.1 物理内存中的page (or page frame)	4
flags.....	5
1.2 Zone	5
free_area.....	7
zone_pgdat	8
1.3 Node.....	8
node和zone的关系	9
1.4 bootmem分配器.....	10
1.4.1 初始化.....	11
init_bootmem	13
init_bootmem_core	13
1.4.2 bootmem分配器分配接口	14
__alloc_bootmem_core	15
分配接口总结	18
1.4.3 bootmem分配器释放接口	19
free_bootmem_core.....	20
free_all_bootmem_core.....	21
1.5 buddy分配器.....	22
1.5.1 初始化.....	23
free_area_init	24
free_area_init_core.....	25
build_zonelists	29
1.5.2 __alloc_pages函数	30
__alloc_pages.....	31
rmqueue.....	34
expand	35
1.5.3 __free_pages_ok函数	36
2 Slab分配器	39
2.1 Slab	40
2.1.1 slab着色	43
2.2 Cache.....	43
2.3 Slab分配器初始化.....	45
kmem_cache_init	46
2.4 Slab分配器接口.....	47

2.4.1 创建cache.....	47
2.4.2 cache分配接口.....	51
kmem_cache_alloc_one_tail.....	53
kmem_cache_grow.....	54
2.4.3 cache释放接口.....	56
3 非连续内存分配.....	58
3.1 初始化.....	59
3.2 分配接口vmalloc/vmalloc_dma/vmalloc_32.....	59
__vmalloc.....	59
get_vm_area.....	60
vmalloc_area_pages.....	61
alloc_area_pmd.....	62
alloc_area_pte.....	62
3.3 释放接口vfree.....	63
vmfree_area_pages.....	63
free_area_pmd.....	64
free_area_pte.....	64
3.4 读写接口vread/vwrite.....	65
4 High Memory Mapping - kmapper.....	65
4.1 初始化.....	67
kmap_init.....	68
fixrange_init.....	68
4.2 kmap/kunmap接口.....	69
4.2.1 kmap.....	69
kmap_high.....	69
map_new_virtual.....	70
flush_all_zero_pkmaps.....	72
4.2.2 kunmap.....	73
kunmap_high.....	73
4.3 kmap_atomic/kunmap_atomic接口.....	74
4.3.1 kmap_atomic.....	74
4.3.2 kunmap_atomic.....	75
5 参考文档.....	75

0 前言

本文档只针对 **kernel 2.4.18** 物理内存管理。忽略有关 **DEBUG, SMP** 以及 **CONFIG_X86_PAE** 部分代码。

操作系统对于内存管理通常分成 2 个主要部分：物理内存管理，以及虚拟内存管理。物理内存管理的目的是有效管理物理内存，虚拟内存管理的目的是对用户进程的虚拟地址空间进行管理。

相对于成千上万的文件，系统的物理内存多大都显得不够大。如何有效管理，快速分配物理

内存是操作系统难以避免的问题。现代操作系统中，物理内存的分配释放的最小单位是页，访问粒度是字节，虽然现在的 cpu 架构的存取粒度是机器字长（32bit/64bit）。

系统在经过足够长时间运行后，会产生两个问题，页外碎片 external fragment 和页内碎片问题。为了解决这两个问题，linux 采用了 buddy allocator 和 slab allocator 算法。

但内核中，实现物理内存管理的并不仅仅只有这两个 allocator。还包括：

- | | |
|---------------|-------------------|
| • 连续物理内存分配 | buddy allocator |
| • 非连续物理内存分配 | vmalloc allocator |
| • 小内存分配 | slab allocator |
| • 高端物理内存管理 | kmapper |
| • 初始化阶段物理内存管理 | bootmem allocator |

下面将逐一介绍上述物理内存管理各模块。

1 物理页框管理

物理内存是有限的宝贵资源，如何有效管理，充分利用物理内存是操作系统的基本要求之一。在现有的物理内存分配算法中，通常是以 page frame（页框）为单位进行的。在 Linux 中也是如此。对于如何管理物理内存的几种方法，请参考^[4]。

采用页为单位进行分配，会带来 2 个问题：

- 如何解决页外碎片问题
- 如何解决页内碎片问题

这 2 个问题的目的是提高物理内存的利用率，同时又要达到高效的目的。

Linux 采用 buddy 系统来解决页外碎片，采用 slab 分配器来解决页内碎片。同时 Linux 采用了 Node, Zone 和 page 三级结构来描述物理内存的。buddy 系统是建立在这三级结构之上的。slab 同时又在 buddy 系统之上管理着物理页之内的内存请求（小内存分配）。

在 buddy 分配器初始化之前，使用 Bootmem 分配器进行简单的内存分配回收工作。

start_kernel 中的初始化工作就是要初始化所有内存管理，使得内存管理模块准备就绪。和内存管理相关的初始化工作如下：

```
start_kernel|--> setup_arch |--> init_bootmem
                |--> paging_init |--> pagetable_init
                |--> kmap_init
                |--> free_area_init
|--> kmem_cache_init
|--> mem_init
|--> kmem_cache_sizes_init
```

1.1 物理内存中的 page (or page frame)

Linux 中，页的大小默认是 4k。页也叫页框，注意和线性地址空间的页区分开来。为什么以 page frame 作为内存管理的最小单位，其原因是多方面的，其中最重要的原因是现行体系结构中的 mmu 单元就是以 page 为单位处理。

物理页描述符描述了一个物理页框：

```
include/linux/mm.h

typedef struct page {
    struct list_head list;          /* ->mapping has some page lists. */
    struct address_space *mapping; /* The inode (or ...) we belong to. */
    unsigned long index;           /* Our offset within mapping. */
    struct page *next_hash;        /* Next page sharing our hash bucket in
                                   the pagecache hash table. */
    atomic_t count;               /* Usage count, see below. */
    unsigned long flags;          /* atomic flags, some possibly
                                   updated asynchronously */
    struct list_head lru;         /* Pageout list, eg. active_list;
                                   protected by pagemap_lru_lock !! */
    wait_queue_head_t wait;       /* Page locked? Stand in line... */
    struct page **pprev_hash;      /* Complement to *next_hash. */
    struct buffer_head * buffers; /* Buffer maps us to a disk block. */
    void *virtual;                /* Kernel virtual address (NULL if
                                   not kmapped, ie. highmem) */
    struct zone_struct *zone;      /* Memory zone we are in. */
} mem_map_t;
```

成员变量说明：

list	双向链表指针。在不同的地方有不同的用处。:-)
mapping	指向映射的 inode
index	指向映射 inode 的偏移量
next_hash	页高速缓存散列表的下一项，指向 page_hash_table。
count	引用数
flags	页标志
lru	lru 双向链表指针，指向 inactive_list 或者 active_list。
wait	等待队列
pprev_hash	页高速缓存散列表的前一项，指向 page_hash_table。
buffers	当该页被用作磁盘块缓存时，指向缓存头部
virtual	该页的虚地址
zone	该页所属的 zone

所有的物理页描述符都存储在全局变量 mem_map 数组中。free_area_init 为描述符分配空间并初始化页描述符。

flags

对于页标志，所有的宏定义都在 `linux/mm.h` 中。宏定义都使用 `PG_xxx` 开头。

```
#define PG_locked      0  /* Page is locked. Don't touch. */
#define PG_error       1
#define PG_referenced  2
#define PG_uptodate    3
#define PG_dirty       4
#define PG_unused      5
#define PG_lru         6
#define PG_active      7
#define PG_slab        8
#define PG_skip        10
#define PG_highmem     11
#define PG_checked     12 /* kill me in 2.5.<early>. */
#define PG_arch_1      13
#define PG_reserved    14
#define PG_laundry     15 /* written out by VM pressure.. */
```

PG_locked	当某个 IO 过程启动时，如果此页参与 IO 操作，需要设置。当 IO 操作完成的时候，清除该标志。
PG_error	当 IO 操作在该页上失败，设置该标志。
PG_referenced	当 IO 操作刚访问过该页的时候，可以设置该标志位。当 <code>kswapd</code> 启动时，准备回收物理页的时候，将忽略该页，不会将其换出。但是 <code>kswapd</code> 会清除该标志。参考 <code>refill_inactive</code> 函数。
PG_uptodate	成功从磁盘读入该页，设置该标志。
PG_dirty	该页内容被修改，还没有更新到磁盘时，设置该标志。
PG_unused	保留，该标志真的没有用。
PG_lru	当该页在 <code>inactive_list</code> 或者 <code>active_list</code> 链表中时，设置该标志。
PG_active	当该页在 <code>active_list</code> 时，设置该标志位。
PG_slab	当该页被 <code>slab</code> 分配器使用时，设置该标志。
PG_skip	在 <code>sparc/sparc64</code> 平台上使用该标志。
PG_highmem	该页处在 <code>high memory</code> 区域， <code>zone highmem</code> 中的所有页框都要设置该标志。该标志一旦被设置，不能被修改。
PG_checked	只被 <code>ext2</code> 文件系统使用。
PG_arch_1	平台相关标志， <code>x86</code> 没有使用。
PG_reserved	具有该标志的物理页不能被交换到磁盘中。
PG_laundry	当 <code>shrink_cache</code> 涉及的 IO 操作中涉及该页，设置该标志。

1.2 Zone

Linux 中，将物理页框分成三种类型的区域，分别称为 `ZONE_DMA`，`ZONE_NORMAL`，`ZONE_HIGHMEM`。

其中 `zone dma` 的地址范围是 `0~16M`，是用来 `ISA DMA` 的内存区。当然，这个 `zone` 也可以存放内核数据和用户数据。`zone normal` 的地址范围是 `16~896M`，可以存放内核数据和用户



数据。zone highmem 的地址范围是高于 896M 的物理地址空间，只能用来存放用户数据。注意，虽然在 x86 平台下，是如此划分的，但是在其他平台，zone dma 和 highmem 可能都是空的，所有的物理页框都划分在 zone normal。

每个 zone 用一个 zone_struct 结构变量来描述：

```
include/linux/mmzone.h

/*
 * On machines where it is needed (eg PCs) we divide physical memory
 * into multiple physical zones. On a PC we have 3 zones:
 *
 * ZONE_DMA    < 16 MB    ISA DMA capable memory
 * ZONE_NORMAL 16-896 MB   direct mapped by the kernel
 * ZONE_HIGHMEM > 896 MB   only page cache and user processes
 */
typedef struct zone_struct {
    /*
     * Commonly accessed fields:
     */
    spinlock_t      lock;
    unsigned long    free_pages;
    unsigned long    pages_min, pages_low, pages_high;
    int              need_balance;

    /*
     * free areas of different sizes
     */
    free_area_t      free_area[MAX_ORDER];

    /*
     * Discontig memory support fields.
     */
    struct pglist_data *zone_pgdat;
    struct page       *zone_mem_map;
    unsigned long      zone_start_paddr;
    unsigned long      zone_start_mapnr;

    /*
     * rarely used fields:
     */
    char              *name;
    unsigned long      size;
} zone_t;
```

成员变量描述：

lock	zone 自旋锁
free_pages	当前 zone 中的空闲页数
pages_min	zone 中的空闲页数小于这个值时，只有 kernel 才能申请到 zone 中的空闲页
pages_low	zone 中的空闲页数小于这个值时，kernel 将立即启动 swaping 过程
pages_high	zone 中的空闲页数小于这个值时，kernel 将启动 swaping 过程
need_balances	启动该 zone 页平衡过程标志
free_area	buddy 算法使用的页 bitmap 数组
wait_table	等待散列表，存放每个页的等待散列表
wait_table_size	等待散列表大小，必须是 2 的 n 次方
wait_table_shift	等待散列表大小的左位移数，也就是 n
zone_pgdat	所属的 node
zone_mem_map	指向 zone 的页描述符数组
zone_start_paddr	zone 起始物理地址
zone_start_mapnr	zone 起始物理页号

dma 的最高地址定义在 include/asm-i386/dma.h 文件中：

```
#define MAX_DMA_ADDRESS (PAGE_OFFSET+0x1000000)
```

在 arch/i386/kernel/setup.c 中，定义了 zone normal 的最高页号。

```
#define MAXMEM_PFN PFN_DOWN(MAXMEM)
```

//MAXMEM 定义在 include/asm-i386/page.h 中

```
#define MAXMEM ((unsigned long)(-PAGE_OFFSET-VMALLOC_RESERVE))
```

在 setup_arch 函数中确定 zone highmem 的范围：

```
#ifdef CONFIG_HIGHMEM
    highstart_pfn = highend_pfn = max_pfn;
    if (max_pfn > MAXMEM_PFN) {
        highstart_pfn = MAXMEM_PFN;
        printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
               pages_to_mb(highend_pfn - highstart_pfn));
    }
#endif
```

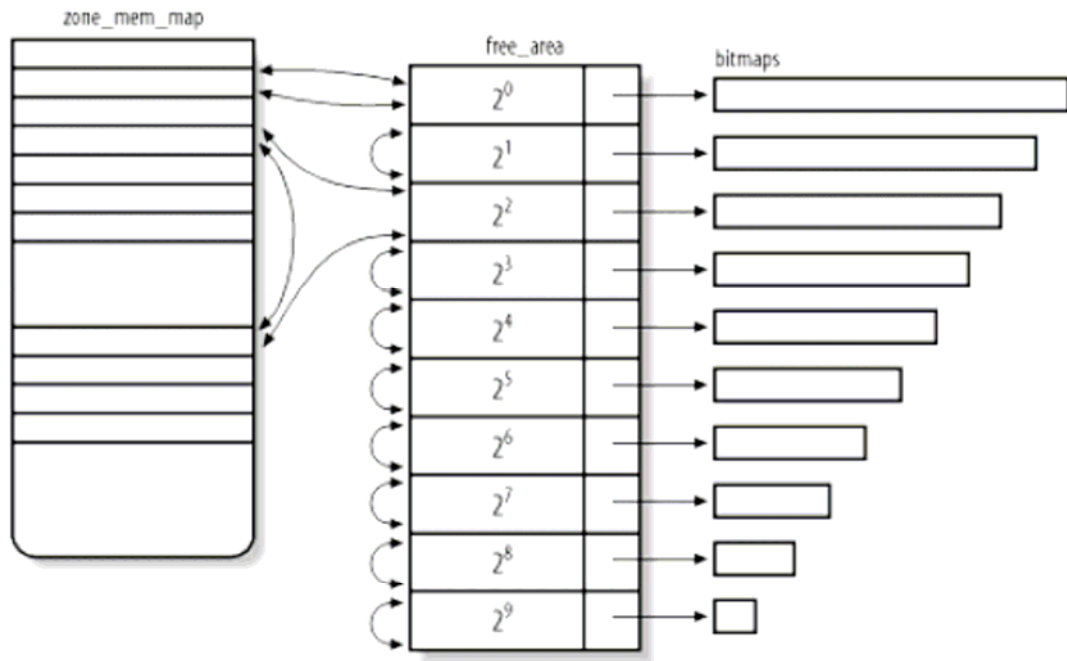
zone allocator 在 free_area_init 中被初始化。

free_area

```
typedef struct free_area_struct {
    struct list_head    free_list;
    unsigned long       *map;
```

```
} free_area_t;
```

成员变量 `free_area` 是 `free_area_t` 数组。这个数组指向了该 `zone` 中所包含的所有物理页框，并且尽可能将最大的连续页框块存放在 `free_area` 中不同的元素中。



zone_pgdat

`zone_pgdat` 指向该 `zone` 所在的 `node` 结构。

1.3 Node

由于 NUMA 体系结构中的每个 `cpu` 对不同地址的物理内存的存取时间是不一样的。为了优化对 NUMA 系统的支持，引进了 `Node` 来将 NUMA 物理内存进行划分。但是 Intel x86 系统不是 NUMA 系统。为了保持代码的一致性，在 x86 平台上，Linux 将所有物理内存都划分到同一个 `Node`。事实上，对于非 NUMA 体系结构，也是如此处理的。

```
/*
 * The pg_data_t structure is used in machines with CONFIG_DISCONTIGMEM
 * (mostly NUMA machines?) to denote a higher-level memory zone than the
 * zone_struct denotes.
 *
 * On NUMA machines, each NUMA node would have a pg_data_t to describe
 * it's memory layout.
 *
 * XXX: we need to move the global memory statistics (active_list, ...)
 *      into the pg_data_t to properly support NUMA.
 */
```



```

struct bootmem_data;
typedef struct pglist_data {
    zone_t node_zones[MAX_NR_ZONES];
    zonelist_t node_zonelists[GFP_ZONEMASK+1];
    int nr_zones;
    struct page *node_mem_map;
    unsigned long *valid_addr_bitmap;
    struct bootmem_data *bdata;
    unsigned long node_start_paddr;
    unsigned long node_start_mapnr;
    unsigned long node_size;
    int node_id;
    struct pglist_data *node_next;
} pg_data_t;

```

成员变量说明:

node_zones	zone 数组
node_zonelists	zone 指针数组, 隐含申请分配内存时 zone 的优先级。
nr_zones	zone 数目
node_mem_map	node 中 page 数组
valid_addr_bitmap	page 是否 usable 的位数组
bdata	bootmem 结构, bootmem allocator 管理信息
node_start_paddr	node 的起始物理地址
node_start_mapnr	node 的起始物理页号
node_size	node 页总数
node_id	标志符
node_next	下一个 node

Linux 静态分配一个 pg_data_t 变量, contig_page_data, 然后在 init_bootmem 中完成初始化。

```

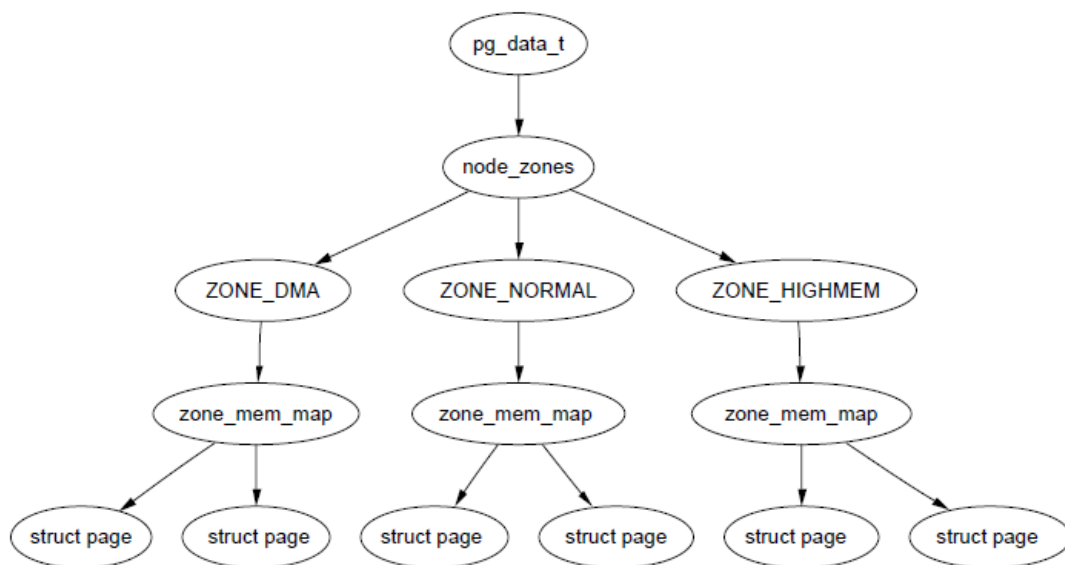
int numnodes = 1; /* Initialized for UMA platforms */
static bootmem_data_t contig_bootmem_data;
pg_data_t contig_page_data = { bdata: &contig_bootmem_data };

```

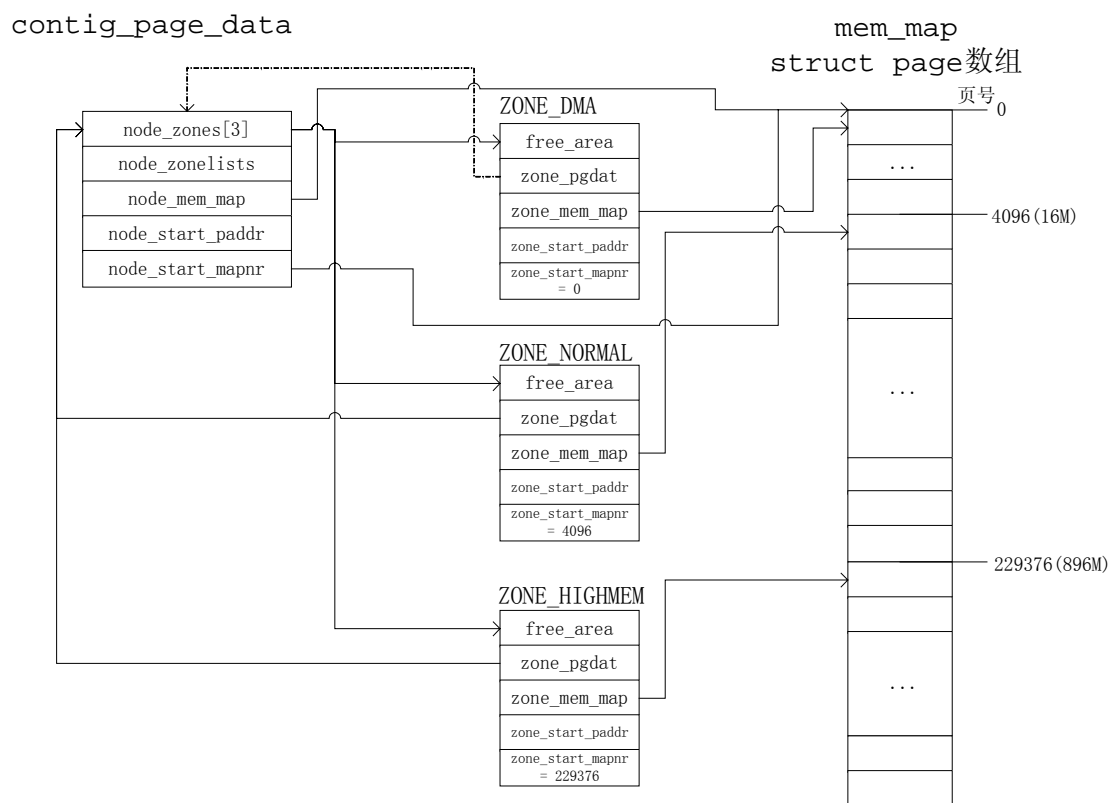
其中 bdata 初始化为 0。(注: 这是标准 c99 语法)

node 和 zone 的关系

综上, node, zone 与 page 数据结构之间的关系是:



实际运行中，x86 平台上，其结构是这样的（假设物理内存大于 1G）：



1.4 bootmem 分配器

在 buddy 和 slab 分配器就绪之前，Linux 使用了简单的 bootmem 分配器进行内存分配操作。只对低于 896M 的物理内存进行分配。

```

/*
 * node_bootmem_map is a map pointer - the bits represent all physical

```

```

* memory pages (including holes) on the node.
*/
typedef struct bootmem_data {
    unsigned long node_boot_start;
    unsigned long node_low_pfn;
    void *node_bootmem_map;
    unsigned long last_offset;
    unsigned long last_pos;
} bootmem_data_t;

```

成员变量说明：

node_boot_start 起始物理地址
node_low_pfn node 中 zone normal 的末页号
node_bootmem_map bootmem 对应 page 的 bitmap 数组，每个位对应一个 page
last_offset 最近一次分配在页内的偏移量，0 表示该页没有空余空间。
last_pos 最近一次分配的页号

bootmem allocator 用 bitmap 数组的方式来标志物理页的分配情况。分配时采用 first fit 算法。每个 Node 结构中都有相应的 bootmem_data_t 成员。在 contig_page_data 中，bootmem_data_t 指向静态变量 contig_bootmem_data。

1.4.1 初始化

初始化 bootmem 分配器实际上分成 3 步，都在 setup_arch 过程中完成。第一步调用 bootmem 分配器初始化函数 init_bootmem；第二步将所有的低端内存（low memory）中的可用物理页标志为“未分配”；第三步调用 bootmem 另一个接口函数 reserve_bootmem，将内核代码以及分配器自身所占用的物理内存页标志为“已分配”。下面是 setup_arch 三步走代码：

```

/*
 * Initialize the boot-time allocator (with low memory only):
 */
bootmap_size = init_bootmem(start_pfn, max_low_pfn);

```

调用 init_bootmem 初始化分配器，返回 bitmap 数组大小。

```

/*
 * Register fully available low RAM pages with the bootmem allocator.
 */
for (i = 0; i < e820.nr_map; i++) {
    unsigned long curr_pfn, last_pfn, size;
    /*
     * Reserve usable low memory
     */
    if (e820.map[i].type != E820_RAM)
        continue;
    /*

```

```

    * We are rounding up the start address of usable memory:
    */
    curr_pfn = PFN_UP(e820.map[i].addr);
    if (curr_pfn >= max_low_pfn)
        continue;
    /*
    * ... and at the end of the usable range downwards:
    */
    last_pfn = PFN_DOWN(e820.map[i].addr + e820.map[i].size);

    if (last_pfn > max_low_pfn)
        last_pfn = max_low_pfn;

    /*
    * .. finally, did all the rounding and playing
    * around just make the area go away?
    */
    if (last_pfn <= curr_pfn)
        continue;

    size = last_pfn - curr_pfn;
    free_bootmem(PFN_PHYS(curr_pfn), PFN_PHYS(size));
}

```

由于 `init_bootmem` 返回后，`bootmem` 的 `bitmap` 数组中的标志都标志为“已分配”，所以在这里要根据 `e820` 变量中指定的可用物理内存相应的标志修改成“未分配”。（这是因为在 `x86` 中，有些物理地址空间被硬件映射到 `rom` 等，所以这些物理地址不可用）

`e820` 中由 `setup_memory` 建立，该函数不详细描述。`e820` 的格式用结构 `e820map` 描述：

```

struct e820map {
    int nr_map;
    struct e820entry {
        unsigned long long addr;    /* start of memory segment */
        unsigned long long size;    /* size of memory segment */
        unsigned long type;        /* type of memory segment */
    } map[E820MAX];
};

```

`type` 成员中的标志说明如下：

```

#define E820_RAM        1
#define E820_RESERVED  2
#define E820_ACPI       3 /* usable as RAM once ACPI tables have been read */
#define E820_NVS        4

```

- 01h 可用内存
- 02h 保留
- 03h acpi 保留

- 04h acpi nvs 保留

上面的 `setup_arch` 代码中只将 `type` 为 01H 的内存区域的对应物理页标志为“未分配”。

```
/*
 * Reserve the bootmem bitmap itself as well. We do this in two
 * steps (first step was init_bootmem()) because this catches
 * the (very unlikely) case of us accidentally initializing the
 * bootmem allocator with an invalid RAM area.
 */
reserve_bootmem(HIGH_MEMORY, (PFN_PHYS(start_pfn) +
                             bootmap_size + PAGE_SIZE-1) - (HIGH_MEMORY));
```

然后根据 `mapsize`，将内核代码和 `bootmem` 分配器所占用的物理页标志为“已分配”。注意：`start_pfn` 是内核代码之后的第一个物理页号，而 `HIGH_MEMORY` 为内核代码起始地址，参考内核启动过程。

init_bootmem

当该函数被调用时，传递参数 `start_pfn` 和 `max_low_pfn`。这两个参数分别是内核代码之后的第一个物理页号，低端物理内存的最高页号。

```
unsigned long __init init_bootmem (unsigned long start, unsigned long pages)
{
    max_low_pfn = pages;
    min_low_pfn = start;
    return(init_bootmem_core(&contig_page_data, start, 0, pages));
}
```

设置 `bootmem.c` 中的全局变量 `max_low_pfn`，`min_low_pfn`。然后调用 `init_bootmem_core`，将该函数的返回值返回。

init_bootmem_core

```
/*
 * Called once to set up the allocator itself.
 */
static unsigned long __init init_bootmem_core (pg_data_t *pgdat,
                                              unsigned long mapstart, unsigned long start, unsigned long end)
{
    bootmem_data_t *bdata = pgdat->bdata;
    unsigned long mapsize = ((end - start)+7)/8;

    pgdat->node_next = pgdat_list;
    pgdat_list = pgdat;

    mapsize = (mapsize + (sizeof(long) - 1UL)) & ~(sizeof(long) - 1UL);
    bdata->node_bootmem_map = phys_to_virt(mapstart << PAGE_SHIFT);
```


```

bdata->node_boot_start = (start << PAGE_SHIFT);
bdata->node_low_pfn = end;

/*
 * Initially all pages are reserved - setup_arch() has to
 * register free RAM areas explicitly.
 */
memset(bdata->node_bootmem_map, 0xff, mapsize);

return mapsize;
}

```

将 bdata 指向 pgdat->bdata，即 contig_bootmem_data。计算 mapsize，并按照 8 对齐（每个字节的位数）。因为 x86 上只有一个 node，所以将 pgdat 指向自己。然后将 mapsize 按照 long 的字节数对齐。然后初始化 contig_bootmem_data 中的 node_bootmem_map，node_boot_start 和 node_low_pfn。这时 node_bootmem_map 为紧挨着_end（即内核代码末地址）之后的首个物理页地址。bitmap 数组所占用的空间就是 node_bootmem_map ~ node_bootmem_map+mapsize。node_boot_start 设置为 0，着将 node_bootmem_map 所指向的 bitmap 数组所有位设置为 1，即将所有物理页标志为“已分配”。

1.4.2 bootmem 分配器分配接口

在 include/linux/bootmem.h 文件中，定义了 x86 的 bootmem 分配器分配器接口。

```

extern void * __init __alloc_bootmem (unsigned long size, unsigned long align,
unsigned long goal);
#define alloc_bootmem(x) \
    __alloc_bootmem((x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))
#define alloc_bootmem_low(x) \
    __alloc_bootmem((x), SMP_CACHE_BYTES, 0)
#define alloc_bootmem_pages(x) \
    __alloc_bootmem((x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS))
#define alloc_bootmem_low_pages(x) \
    __alloc_bootmem((x), PAGE_SIZE, 0)

extern void * __init __alloc_bootmem_node (pg_data_t *pgdat, unsigned long size,
unsigned long align, unsigned long goal);
#define alloc_bootmem_node(pgdat, x) \
    __alloc_bootmem_node((pgdat), (x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))
#define alloc_bootmem_pages_node(pgdat, x) \
    __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS))
#define alloc_bootmem_low_pages_node(pgdat, x) \
    __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, 0)

```

其中，alloc_bootmem，alloc_bootmem_low，alloc_bootmem_pages，和 alloc_bootmem_low_pages 都调用 __alloc_bootmem；alloc_bootmem_node，

`alloc_bootmem_pages_node`, `alloc_bootmem_low_pages_node` 都调用 `__alloc_bootmem_node`。而 `__alloc_bootmem` 和 `__alloc_bootmem_node` 中, 调用 `__alloc_bootmem_core` 来完成分配任务。所以 `__alloc_bootmem_core` 是分配接口的核心函数。

`__alloc_bootmem_core`

```
static void * __init __alloc_bootmem_core (bootmem_data_t *bdata,
    unsigned long size, unsigned long align, unsigned long goal)
{
    unsigned long i, start = 0;
    void *ret;
    unsigned long offset, remaining_size;
    unsigned long areasize, preferred, incr;
    unsigned long eidx = bdata->node_low_pfn - (bdata->node_boot_start >>
        PAGE_SHIFT);
```

在 `__alloc_bootmem` 中, 针对每个 `node`, 调用 `__alloc_bootmem_core` 处理该 `node` 中的分配。因为在 x86 中, 只有一个 `node`, 所以 `__alloc_bootmem` 实际上就是调用 `__alloc_bootmem_core` 处理分配请求。

将 `bootmem` 中的总页数存储到 `eidx` 中。

```
if (!size) BUG();

if (align & (align-1))
    BUG();
```

检查 `size` 是否为零, 同时确保 `align` 是 2 的 n 次方。

```
offset = 0;
if (align &&
    (bdata->node_boot_start & (align - 1UL)) != 0)
    offset = (align - (bdata->node_boot_start & (align - 1UL)));
offset >>= PAGE_SHIFT;
```

如果 `node_boot_start` 不是 `align` 对齐, 计算 `node_boot_start` 按照 `align` 大小对齐的地址差异值, 然后将 `offset` 折算成页数。

```
/*
 * We try to allocate bootmem pages above 'goal'
 * first, then we try to allocate lower pages.
 */
if (goal && (goal >= bdata->node_boot_start) &&
    ((goal >> PAGE_SHIFT) < bdata->node_low_pfn)) {
    preferred = goal - bdata->node_boot_start;
} else
    preferred = 0;
```

```
preferred = ((preferred + align - 1) & ~(align - 1)) >> PAGE_SHIFT;
preferred += offset;
```

如果 `goal` 指定的地址落在 `bootmem` 的 `node_boot_start~node_low_pfn` 之内, 将 `preferred` 修改为 `goal` 相对 `node_boot_start` 的偏移量。否则 `preferred` 设置为 0。然后将 `preferred` 按照 `align` 对齐, 并调整成页偏移量。最后加上 `offset` 的偏移量。

`goal` 指定的地址, 表示调用者要求 `bootmem` 分配尽量将地址大于 `goal` 的物理页分配给调用者。

```
areasize = (size+PAGE_SIZE-1)/PAGE_SIZE;
incr = align >> PAGE_SHIFT ? : 1;
```

将 `size` 所占用的页数存储到 `areasize` 中。如果 `align >> PAGE_SHIFT` 为 0, 设置 `incr` 为 1, 否则将 `incr` 设置为 `align >> PAGE_SHIFT`。

```
restart_scan:
    for (i = preferred; i < eid; i += incr) {
        unsigned long j;
        if (test_bit(i, bdata->node_bootmem_map))
            continue;
        for (j = i + 1; j < i + areasize; ++j) {
            if (j >= eid)
                goto fail_block;
            if (test_bit(j, bdata->node_bootmem_map))
                goto fail_block;
        }
        start = i;
        goto found;
    fail_block:;
    }
    if (preferred) {
        preferred = offset;
        goto restart_scan;
    }
    return NULL;
```

从 `preferred` 开始到 `eid`, 寻找该范围内是否有符合请求大小的连续物理页。由于要求分配的内存页按照 `align` 对齐, 所以第一个 `for` 循环的步长是 `incr`。第二个 `for` 循环检查 `i+1` 开始到 `~i+areasize` 中的物理页是否已经被分配。如果是, 跳转到 `fail_block`, 该处是空语句, 只有一个分号, 因此会继续第一个 `for` 的循环, 继续寻找下一个合适的物理页。否则跳转到 `found`, 表示已经找到符合要求的物理页。

当第一个 `for` 完成, 而非跳转执行别处代码, 表示在 `preferred~eid` 的范围内找不到合适物理页, 所以当 `preferred` 非零时, 设置 `preferred` 为 `offset`, 跳转到 `restart_scan`, 重新进入 `for` 循环。(看出来吧, 这里代码有 bug, 如果 `offset` 非零, 而且在内存中没有超过 `areasize` 的连

续物理页，会进入死循环，⊗)，不过我们也可以看出，offset 一般情况下都是 0。这时候，第二次循环完成时，代码会判断 preferred 为 0，然后返回 NULL，过程结束。

```
found:
    if (start >= eidx)
        BUG();
```

进入 found 过程，表示在前面的 for 循环过程中，已经找到一个合适区域，起始物理页为 start。判断 start 是否大于 node 最大页号，如果是出错。

```
/*
 * Is the next page of the previous allocation-end the start
 * of this allocation's buffer? If yes then we can 'merge'
 * the previous partial page with this allocation.
 */
if (align <= PAGE_SIZE
    && bdata->last_offset && bdata->last_pos+1 == start) {
    offset = (bdata->last_offset+align-1) & ~(align-1);
    if (offset > PAGE_SIZE)
        BUG();
    remaining_size = PAGE_SIZE-offset;
```

if 语句判断以下三个条件是否都成立：

- align 不大于 PAGE_SIZE，如果大于 PAGE_SIZE，表明上次请求刚好结束在页边界。
- 上次请求分配的 offset 非零，如果为 0，表示上次请求刚好结束在页边界，所以上次分配没有内部碎片产生。
- 上次分配的内存末地址是否和这次相邻，如果是，说明可以将这两次分配 merge 在一起，减少碎片产生。

当上述三条件都成立，表明在上次请求后，最后页还有空闲空间，那么这次请求尽量将这些空闲空间利用起来，减少页内碎片。

接着，将 last_offset 按照 align 对齐，存储在 offset 中，这是因为当前请求必须要按照 align 对齐；然后计算空闲空间，存储在 remaining_size 中。其间，再次确认 offset 不大于 PAGE_SIZE。

```
if (size < remaining_size) {
    areastore = 0;
    // last_pos unchanged
    bdata->last_offset = offset+size;
    ret = phys_to_virt(bdata->last_pos*PAGE_SIZE + offset +
        bdata->node_boot_start);
} else {
    remaining_size = size - remaining_size;
    areastore = (remaining_size+PAGE_SIZE-1)/PAGE_SIZE;
    ret = phys_to_virt(bdata->last_pos*PAGE_SIZE + offset +
        bdata->node_boot_start);
```

```

        bdata->last_pos = start+areasize-1;
        bdata->last_offset = remaining_size;
    }
    bdata->last_offset &= ~PAGE_MASK;

```

判断 size 是否小于 remaining_size，如果是：

- 更新 last_offset
- 计算该次请求的起始线性地址，并存储在 ret 变量中。

否则：

- 计算除去 remaining_size 空间后的请求大小，存储在 remaining_size 中。
- 重新剩余大小所占用的页数，存储在 area_size 中。
- 计算该次请求的起始线性地址，并存储在 ret 变量中。
- 更新 last_pos 为该次请求的末页号。
- 更新 last_offset 为 remaining_size，此时 remaining_size 不小于 PAGE_SIZE。

最后，将 last_offset 修改成页内的偏移量。

```

    } else {
        bdata->last_pos = start + areasize - 1;
        bdata->last_offset = size & ~PAGE_MASK;
        ret = phys_to_virt(start * PAGE_SIZE + bdata->node_boot_start);
    }

```

此时的 else 对应判断三个条件的 if 语句，即判断“是否需要合并”不成立，不需要进行合并。将 last_pos 更新为该次请求的末页号，直接按照 size 计算该次请求末页偏移量，最后计算该次请求的起始线性地址，并存储在 ret 变量中。

```

/*
 * Reserve the area now:
 */
for (i = start; i < start+areasize; i++)
    if (test_and_set_bit(i, bdata->node_bootmem_map))
        BUG();
memset(ret, 0, size);
return ret;

```

用 for 循环将该次请求所占用的新物理页标志为“已分配”，在标志这些物理页的过程中，同时检查是否已经被分配，如果是，出错。然后将刚分配的物理内存区域的内容初始化成 0，最后返回起始线性地址，ret。

分配接口总结

```

void * __init __alloc_bootmem (unsigned long size, unsigned long align, unsigned
long goal);

```

中间分配函数，处理多 node 的情况，在 x86 中相当于直接调用 __alloc_bootmem_core。

```

#define alloc_bootmem(x) \
    __alloc_bootmem((x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))

```

尽量从 zone normal 中请求 x 个字节的物理内存，并要求起始地址按照 SMP_CACHE_BYTES 对齐。

```
#define alloc_bootmem_low(x) \
    __alloc_bootmem((x), SMP_CACHE_BYTES, 0)
```

请求 x 个字节的物理内存，并要求起始地址按照 SMP_CACHE_BYTES 对齐。

```
#define alloc_bootmem_pages(x) \
    __alloc_bootmem((x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS))
```

尽量从 zone normal 中分配 x 个字节的物理内存，并要求起始地址按照 page 大小对齐。

```
#define alloc_bootmem_low_pages(x) \
    __alloc_bootmem((x), PAGE_SIZE, 0)
```

分配 x 个字节的物理内存，并且要求起始地址按照 page 大小对齐。

```
extern void * __init __alloc_bootmem_node (pg_data_t *pgdat, unsigned long size,
unsigned long align, unsigned long goal);
```

和 __alloc_bootmem 类似，只不过处理单个 node。

```
#define alloc_bootmem_node(pgdat, x) \
    __alloc_bootmem_node((pgdat), (x), SMP_CACHE_BYTES, __pa(MAX_DMA_ADDRESS))
```

从指定的 node 中，尽量从 zone normal 中请求 x 个字节的物理内存，并要求起始地址按照 SMP_CACHE_BYTES 对齐。

```
#define alloc_bootmem_pages_node(pgdat, x) \
    __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, __pa(MAX_DMA_ADDRESS))
```

从指定的 node 中，尽量从 zone normal 中分配 x 个字节的物理内存，并要求起始地址按照 page 大小对齐。

```
#define alloc_bootmem_low_pages_node(pgdat, x) \
    __alloc_bootmem_node((pgdat), (x), PAGE_SIZE, 0)
```

从指定的 node 中，分配 x 个字节的物理内存，并且要求起始地址按照 page 大小对齐。

1.4.3 bootmem 分配器释放接口

和分配接口类似，释放接口共有两组：free_bootmem/free_all_bootmem 和 free_bootmem_node/free_all_bootmem_node。这些接口同样定义在 mm/bootmem.c 文件中。从这些函数的定义，我们可以看出，这两组接口函数实际在做同样的事情，至少在 x86 上是如此。

```
void __init free_bootmem_node (pg_data_t *pgdat, unsigned long physaddr, unsigned
long size)
{
    return(free_bootmem_core(pgdat->bdata, physaddr, size));
}
```

```

unsigned long __init free_all_bootmem_node (pg_data_t *pgdat)
{
    return(free_all_bootmem_core(pgdat));
}

void __init free_bootmem (unsigned long addr, unsigned long size)
{
    return(free_bootmem_core(contig_page_data.bdata, addr, size));
}

unsigned long __init free_all_bootmem (void)
{
    return(free_all_bootmem_core(&contig_page_data));
}

```

同时我们也可以看出，释放接口的核心函数有两个，`free_bootmem_core` 和 `free_all_bootmem_core`。

free_bootmem_core

```

/*
 * round down end of usable mem, partially free pages are
 * considered reserved.
 */
unsigned long idx;
unsigned long endx = (addr + size - bdata->node_boot_start)/PAGE_SIZE;
unsigned long end = (addr + size)/PAGE_SIZE;

```

`idx` 初始化为要释放区域在 `bootmem` 内存区域的末页号；`end` 初始化为物理地址的末页号(用末页号可能描述的不太准确，实际上就是索引值)。

```

if (!size) BUG();
if (end > bdata->node_low_pfn)
    BUG();

```

检查合法性。

```

/*
 * Round up the beginning of the address.
 */
start = (addr + PAGE_SIZE-1) / PAGE_SIZE;
idx = start - (bdata->node_boot_start/PAGE_SIZE);

for (i = idx; i < endx; i++) {
    if (!test_and_clear_bit(i, bdata->node_bootmem_map))

```

```

        BUG();
    }

```

计算要释放内存区域的起始页号，存储在 `start` 中。将相对于 `node_boot_start` 的页号存储在 `sidx` 中。实际上，这时候 `sidx` 和 `edix` 就是要释放内存区域的在 `node_bootmem_map` 中的对应索引值。接下去，将这个区域相对应的位置成“未分配”。

不过要注意的是：由于分配的时候，可能和另外分配请求共用某个页（为了解决页内碎片引起的）也释放。

free_all_bootmem_core

```

    if (!bdata->node_bootmem_map) BUG();

```

检查 `node_bootmem_map`。

```

count = 0;
idx = bdata->node_low_pfn - (bdata->node_boot_start >> PAGE_SHIFT);
for (i = 0; i < idx; i++, page++) {
    if (!test_bit(i, bdata->node_bootmem_map)) {
        count++;
        ClearPageReserved(page);
        set_page_count(page, 1);
        __free_page(page);
    }
}
total += count;

```

将所有 `bootmem` 分配器中未分配的物理页用 `buddy` 分配器释放接口回收，这时候，这些物理内存自动被收纳入 `buddy` 分配器。所以在 `buddy` 被初始化之前，不能调用 `free_bootmem_core`。然后将空闲页总数存储在 `total` 变量中。

注意：其实这段代码也是 `buddy` 分配器初始化工作的最后部分，因为在这之前，`buddy` 分配器中没有任何空闲物理页。

```

/*
 * Now free the allocator bitmap itself, it's not
 * needed anymore:
 */
page = virt_to_page(bdata->node_bootmem_map);
count = 0;
for (i = 0; i < ((bdata->node_low_pfn - (bdata->node_boot_start >>
PAGE_SHIFT))/8 + PAGE_SIZE - 1)/PAGE_SIZE; i++, page++) {
    count++;
    ClearPageReserved(page);
    set_page_count(page, 1);
    __free_page(page);
}

```

```
total += count;
bdata->node_bootmem_map = NULL;

return total;
```

接着，将 bootmem 分配器中 node_bootmem_map 所占用的内存也回收，同样这些内存也收纳入 buddy 分配器。更新 total，然后返回总空闲页数。

1.5 buddy 分配器

buddy 算法实际上和 bootmem 分配器相比，并不复杂多少。其主要目的是为了减少 external fragment 问题。但是对于分配过程出现的内存不够情况的处理，却要复杂很多。

Buddy 算法将所有的空闲物理页分成 10 组，每组分别包含大小 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 个连续物理页，对于每组的情况，都有对应的 bitmap 数组表明相邻情况。这些都由 zone 中的 free_area 数组表示。当分配时，被分配的内存不能跨 zone，换句话说，所有的物理页都必须在单个 zone 中完成。

实际实现中，buddy 算法对于组中的空闲连续物理块还有一个限制条件：每个块的第一个页框的物理地址必须是该块大小的整数倍。所以，每个 zone 所管理的物理内存地址，其起始地址按照最大组的大小来对齐的。这个限制可以加快 buddy 分配回收速度，减少实现复杂度。在这样的限制下，我们可以看出，当执行回收操作时，根据给定的回收地址和大小，可以很容易得到相邻块的地址，也就是 buddy 的地址。假设被回收块地址是 0x10010000，且大小为 1 个物理页框，那么 buddy 的地址就应该是 0x10011000，而不是 0x1000f000。如果是后者的话，那么当合并成更大块的时候，其起始物理地址是 0x1000f000，而这个地址不符合上面所说的限制。

所以 buddy 的地址可以这样描述：x 是某物理块的首地址， 2^k 是该块的大小，那么相邻该物理块的 buddy 块的地址是：

$$buddy(x) = \begin{cases} x + 2^k, & \text{当 } x \bmod 2^{k+1} = 0 \\ x - 2^k, & \text{当 } x \bmod 2^{k+1} = 2^k \end{cases}$$

由于 mem_map 中包含了从物理地址 0 开始到最后一个物理页的描述符，可以借助物理块首页在 mem_map 数组中的下标，记为 k，大小是 2^{order} 个页。那么 buddy 的下标可以这么计算：

$$buddy(k) = \begin{cases} k + 2^{\text{order}}, & \text{当 } k \bmod 2^{\text{order}+1} = 0 \\ k - 2^{\text{order}}, & \text{当 } k \bmod 2^{\text{order}+1} = 2^{\text{order}} \end{cases}$$

k 一定也是 2^{order} 的整数倍。所以在计算 buddy 的时候，我们只需要判断 k 的第 order+1 位即可。如果该位为 1，则将该位设置为 0。如果该位为 0，则改为设置为 1。所以对于给定的 k 和 order。buddy 应该这样计算：

$$buddy(k) = k \oplus 2^{\text{order}}$$

让我们看看 __free_pages_ok 源码是如何确定 buddy 的地址的：

```
mask = (~0UL) << order;
```

```
page_idx = page - base;
buddy1 = base + (page_idx ^ -mask);
```

所以, $-mask = -(0xffffffff - (2^{\text{order}} - 1)) = -(0xffffffff - 2^{\text{order}} + 1) = 2^{\text{order}}$

buddy1 就指向 buddy。

在得到分配请求时,只要有足够的空闲页,先从最适合大小的组中查询有没有合适的连续内存页满足,如果没有,那么从大一级的组中寻找合适的内存块,如此反复,直到寻找到合适的。然后将该内存块分割,剩下的空闲页被分成 2 次方的大小,存放在 free_area 的相应组中。然后将请求的空闲页块返回给调用者。

当使用者释放页的时候,内核将检查是否有 buddy 块是否空闲。如果否,过程结束;否则将他们结合起来形成一个大的空闲块。然后再次检查是否可以进一步结合成更大的空闲块,直到不能再次结合。最后将这个空闲块插入 free_area 相应的组中。

关于 buddy 算法的详细讨论,参考《The Art of Computer Programming》Volume 1, chapter 2.5。

buddy 的接口 api 如下:

alloc_page	分配一个页, 返回页物理地址
alloc_pages	分配 2 的 n 次方个页, 返回页物理地址
get_free_page	分配一个页并初始化为 0, 返回页虚地址
__get_free_page	分配一个页, 返回页虚地址
__get_free_pages	分配 2 的 n 次方个页并初始化为 0, 返回页虚地址
__get_dma_page	从 zone dma 中分配 2 的 n 次方个页, 返回页物理地址
__free_pages	从指定的物理地址, 回收 2 的 n 次方个页
__free_page	从指定的物理地址, 回收一个页
free_page	从指定的虚地址, 回收一个页

不管使用哪个分配 api, 这些 api 最终都调用 __alloc_pages 来获取空闲物理页。__alloc_pages 不允许直接调用, 必须要通过上述 api 提出分配请求。

同样, free api 的核心是 __free_pages_ok 函数, 同样也不允许直接调用。

这些 api 定义在 mm/page_alloc.c 和 include/linux/mm.h 中。

1.5.1 初始化

buddy 分配器的初始化工作在 bootmem 分配器之后进行的, 因为它需要 bootmem 分配器的支持。其次, 由于 buddy 分配器的主要数据结构是建立在 node, zone 之上的。所以 buddy 分配器也需要先初始化 node, zone。

setup_arch 在获取可用物理内存之后, 初始化 bootmem 分配器, 实际上这里还有一个隐含的作用, 那就是用 bootmem 分配器纪录的可用物理内存情况。然后在 bootmem 分配器结束其

生命周期的时候，将这些可用的物理内存情况转存到 buddy 分配器中。（这里用转存这个词，可能不合适）

（在得到实际物理大小，初始化好 bootmem 分配器之后）setup_arch 调用 paging_init。

```
{
    unsigned long zones_size[MAX_NR_ZONES] = {0, 0, 0};
    unsigned int max_dma, high, low;

    max_dma = virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHIFT;
    low = max_low_pfn;
    high = highend_pfn;

    if (low < max_dma)
        zones_size[ZONE_DMA] = low;
    else {
        zones_size[ZONE_DMA] = max_dma;
        zones_size[ZONE_NORMAL] = low - max_dma;
#ifdef CONFIG_HIGHMEM
        zones_size[ZONE_HIGHMEM] = high - low;
#endif
    }
    free_area_init(zones_size);
}
```

根据实际物理内存大小，其中 max_dma 的大小是 16M；low 是 896M 以下的最高页号，如果实际物理大于 896M，low 就是 896M 处的末页号；high 就是实际物理地址的最高页号。

然后设置 zones_size 数组，传递给 free_area_init。

free_area_init

```
void __init free_area_init(unsigned long *zones_size)
{
    free_area_init_core(0, &contig_page_data, &mem_map, zones_size, 0, 0, 0);
}
```

free_area_init 直接调用了 free_area_init_core。传递的参数：

- contig_page_data node 结构，将被初始化
- mem_map mem_map，将被指向初始化后的 mem_map 数组。
- zones_size paging_init 中的 zones_size 数组

其中 mem_map 所指向的数组，其所用的物理内存由 bootmem 分配器分配。

注意：free_area_init 返回后，buddy 分配器中所有的物理页都是保留。哪些物理页是空闲的，此时这信息正纪录在 bootmem 分配器中。

free_area_init_core

```
void __init free_area_init_core(int nid, pg_data_t *pgdat, struct page **gmap,
    unsigned long *zones_size, unsigned long zone_start_paddr,
    unsigned long *zholes_size, struct page *lmem_map)
{
    struct page *p;
    unsigned long i, j;
    unsigned long map_size;
    unsigned long totalpages, offset, realtotalpages;
    const unsigned long zone_required_alignment = 1UL << (MAX_ORDER-1);
```

根据 MAX_ORDER, 计算 zone 起始地址的对齐量, 因为在 buddy 分配器中, 处理相邻情况时, 对这些块的起始物理地址有要求。(详见 __free_pages_ok 函数)

```
    if (zone_start_paddr & ~PAGE_MASK)
        BUG();
```

检查 zone 起始地址是否按照 page 大小对齐。

```
    totalpages = 0;
    for (i = 0; i < MAX_NR_ZONES; i++) {
        unsigned long size = zones_size[i];
        totalpages += size;
    }
    realtotalpages = totalpages;
    if (zholes_size)
        for (i = 0; i < MAX_NR_ZONES; i++)
            realtotalpages -= zholes_size[i];

    printk("On node %d totalpages: %lu\n", nid, realtotalpages);
```

计算实际物理页数, 然后输出总数。在 numa 架构下, zholes_size 可能不为 0, 也就是说物理地址空间存在着空洞。这些物理地址覆盖的物理页不应该被统计进去。

```
    INIT_LIST_HEAD(&active_list);
    INIT_LIST_HEAD(&inactive_list);
```

初始化 active_list 和 inactive_list。

```
    /*
     * Some architectures (with lots of mem and discontinous memory
     * maps) have to search for a good mem_map area:
     * For discontigmem, the conceptual mem map array starts from
     * PAGE_OFFSET, we need to align the actual array onto a mem map
     * boundary, so that MAP_NR works.
     */
    map_size = (totalpages + 1)*sizeof(struct page);
    if (lmem_map == (struct page *)0) {
```

```

lmem_map = (struct page *) alloc_bootmem_node(pgdat, map_size);
lmem_map = (struct page *) (PAGE_OFFSET +
    MAP_ALIGN((unsigned long)lmem_map - PAGE_OFFSET));
}
*gmap = pgdat->node_mem_map = lmem_map;

```

根据 `totalpages`，计算 `mem_map` 数组元素个数。接着给 `mem_map` 数组分配内存空间，然后将首地址按照结构 `page` 大小对齐。最后将全局指针 `mem_map` 指向这个数组。

```

pgdat->node_size = totalpages;
pgdat->node_start_paddr = zone_start_paddr;
pgdat->node_start_mapnr = (lmem_map - mem_map);
pgdat->nr_zones = 0;

```

初始化 `node`。

```

/*
 * Initially all pages are reserved - free ones are freed
 * up by free_all_bootmem() once the early boot process is
 * done.
 */
for (p = lmem_map; p < lmem_map + totalpages; p++) {
    set_page_count(p, 0);
    SetPageReserved(p);
    init_waitqueue_head(&p->wait);
    memlist_init(&p->list);
}

```

然后初始化所有的页描述符。所有的物理页都被描述为保留。也就是说现在 `buddy` 分配其中没有可用物理页。

```

offset = lmem_map - mem_map;

```

在多 `node` 的情况下，`offset` 不为 0，`offset` 表示该 `node` 所管理的起始物理页描述符在全局 `mem_map` 数组中的偏移量。在 x86，只有一个 `node` (`contig_page_data`)，所以 `offset` 为 0。

```

for (j = 0; j < MAX_NR_ZONES; j++) {
    zone_t *zone = pgdat->node_zones + j;
    unsigned long mask;
    unsigned long size, realsize;

```

开始初始化 `node` 中的 `zone` 结构。要处理的 `zone` 在局部变量 `zone` 中。

```

    realsize = size = zones_size[j];
    if (zholes_size)
        realsize -= zholes_size[j];

    printk("zone(%lu): %lu pages.\n", j, size);

```

计算并打印 `zone` 的实际物理页数。

```

zone->size = size;
zone->name = zone_names[j];
zone->lock = SPIN_LOCK_UNLOCKED;
zone->zone_pgdat = pgdat;
zone->free_pages = 0;
zone->need_balance = 0;

```

初始化 zone 结构。

```

if (!size)
    continue;

```

如果该 zone 的物理页大小为 0，接着处理下一个 zone。

```

pgdat->nr_zones = j+1;

```

该 zone 有效，将 node 中的 nr_zones 变量加 1。

```

mask = (realsize / zone_balance_ratio[j]);
if (mask < zone_balance_min[j])
    mask = zone_balance_min[j];
else if (mask > zone_balance_max[j])
    mask = zone_balance_max[j];
zone->pages_min = mask;
zone->pages_low = mask*2;
zone->pages_high = mask*3;

```

根据该 zone 的实际物理页数，设置 zone 中的“水位”，即 pages_min, pages_low 和 pages_high。

```

zone->zone_mem_map = mem_map + offset;
zone->zone_start_mapnr = offset;
zone->zone_start_paddr = zone_start_paddr;

```

初始化 zone_mem_map，zone_start_mapnr，和 zone_start_paddr。

```

if ((zone_start_paddr >> PAGE_SHIFT) & (zone_required_alignment-1))
    printk("BUG: wrong zone alignment, it will crash\n");

```

判断 zone 起始物理地址是否按照 zone_required_alignment 对齐。

```

for (i = 0; i < size; i++) {
    struct page *page = mem_map + offset + i;
    page->zone = zone;
    if (j != ZONE_HIGHMEM)
        page->virtual = __va(zone_start_paddr);
    zone_start_paddr += PAGE_SIZE;
}

```

初始化页描述符中的 zone，virtual。如果该物理内存是高端内存，那么其线性地址不修改，默认为 0。高端内存的线性地址如何获取，参考第四章 High Memory Mapping。

```
offset += size;
```

更新 offset。

```
for (i = 0; ; i++) {
    unsigned long bitmap_size;

    memlist_init(&zone->free_area[i].free_list);
    if (i == MAX_ORDER-1) {
        zone->free_area[i].map = NULL;
        break;
    }
}
```

开始设置 zone 中的 free_area 变量。free_area 是 buddy 分配器的核心数据。先将 free_area 中的 free_list 初始化为空，如果 free_area 数组中的最后一个元素，将 map 初始化空，并退出循环。

```
/*
 * Page buddy system uses "index >> (i+1)",
 * where "index" is at most "size-1".
 *
 * The extra "+3" is to round down to byte
 * size (8 bits per byte assumption). Thus
 * we get "(size-1) >> (i+4)" as the last byte
 * we can access.
 *
 * The "+1" is because we want to round the
 * byte allocation up rather than down. So
 * we should have had a "+7" before we shifted
 * down by three. Also, we have to add one as
 * we actually _use_ the last bit (it's [0,n]
 * inclusive, not [0,n]).
 *
 * So we actually had +7+1 before we shift
 * down by 3. But (n+8) >> 3 == (n >> 3) + 1
 * (modulo overflows, which we do not have).
 *
 * Finally, we LONG_ALIGN because all bitmap
 * operations are on longs.
 */
bitmap_size = (size-1) >> (i+4);
bitmap_size = LONG_ALIGN(bitmap_size+1);
zone->free_area[i].map =
    (unsigned long *) alloc_bootmem_node(pgdat, bitmap_size);
}
```

计算每个分组的 bitmap 大小，其所占用的内存由 bootmem 分配器分配。注意，bitmap 数组这时候的内容为 0。

bitmap 位数组的大小是这么决定的：每个位代表相邻情况，1 表示此 area 存在相邻物理块，0 表示不存在。

```
    }  
    build_zonelists(pgdat);  
}
```

free_area_init_core 最后调用 build_zonelists 初始化 node 结构中的 node_zonelists。

build_zonelists

```
static inline void build_zonelists(pg_data_t *pgdat)  
{  
    int i, j, k;  
  
    for (i = 0; i <= GFP_ZONEMASK; i++) {  
        zonelist_t *zonelist;  
        zone_t *zone;  
  
        zonelist = pgdat->node_zonelists + i;  
        memset(zonelist, 0, sizeof(*zonelist));  
    }
```

for 循环遍历 zonelists 数组中的每个元素。先将 zonelist 中每个元素都置为 0。

```
        j = 0;  
        k = ZONE_NORMAL;  
        if (i & __GFP_HIGHMEM)  
            k = ZONE_HIGHMEM;  
        if (i & __GFP_DMA)  
            k = ZONE_DMA;
```

根据 __GFP_HIMEM 和 __GFP_DMA，设置 k。

```
        switch (k) {  
            default:  
                BUG();  
            /*  
             * fallthrough:  
             */  
            case ZONE_HIGHMEM:  
                zone = pgdat->node_zones + ZONE_HIGHMEM;  
                if (zone->size) {  
#ifndef CONFIG_HIGHMEM  
                    BUG();  
                }
```

```

#endif

        zonelist->zones[j++] = zone;
    }
    case ZONE_NORMAL:
        zone = pgdat->node_zones + ZONE_NORMAL;
        if (zone->size)
            zonelist->zones[j++] = zone;
    case ZONE_DMA:
        zone = pgdat->node_zones + ZONE_DMA;
        if (zone->size)
            zonelist->zones[j++] = zone;
    }
    zonelist->zones[j++] = NULL;
}
}
}

```

以 `node_zonelists` 数组的下标作为内存请求时标志的分类依据（不知道该怎么描述）。如果下标和 `__GFP_HIGHMEM` 位与非 0，那么 `node_zonelists[i].zones` 数组中将依次指向：`ZONE_HIGHMEM`，`ZONE_NORMAL`，`ZONE_DMA`。因此这个 `zonelists` 隐含指定了内存分配时的 `zone` 顺序。

如果下标和 `__GFP_DMA` 位与非 0，那么 `node_zonelists[i].zones` 数组中将指向：`ZONE_DMA`。

如果下标和 `__GFP_HIGHMEM` 和 `__GFP_DMA` 位与为 0，那么 `node_zonelists[i].zones` 数组中将依次指向：`ZONE_NORMAL`，`ZONE_DMA`。

（上面的文字很晦涩，建议参考内核中的 `__alloc_pages` 的 `zonelists` 是如何传递的）

1.5.2 `__alloc_pages` 函数

分配接口函数定义（或声明）在 `include/linux/mm.h` 文件中。

```

static inline struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
{
    /*
     * Gets optimized away by the compiler.
     */
    if (order >= MAX_ORDER)
        return NULL;
    return __alloc_pages(gfp_mask, order);
}

#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)

extern unsigned long FASTCALL(__get_free_pages(unsigned int gfp_mask, unsigned

```

```
int order));
extern unsigned long FASTCALL(get_zeroed_page(unsigned int gfp_mask));

#define __get_free_page(gfp_mask) \
    __get_free_pages((gfp_mask), 0)

#define __get_dma_pages(gfp_mask, order) \
    __get_free_pages((gfp_mask) | GFP_DMA, (order))
```

`alloc_xxx` 函数返回的是 `page` 结构指针，而 `get_xxx` 返回的是内存区域的虚拟地址。这些接口函数和其他很多部分的接口函数一样，最终都调用一个核心分配函数。`buddy` 分配器的核心分配函数是 `__alloc_pages`。

`__alloc_pages`

```
/*
 * This is the 'heart' of the zoned buddy allocator:
 */
struct page * __alloc_pages(unsigned int gfp_mask, unsigned int order, zonelist_t
*zonelist)
{
    unsigned long min;
    zone_t **zone, * classzone;
    struct page * page;
    int freed;

    zone = zonelist->zones;
    classzone = *zone;
```

参数 `gfp_mask` 是分配请求标志，`order` 是大小，`__alloc_pages` 将依据 `order` 分配 2^{order} 个物理页。`zonelist` 指向 `node` 的 `node_zonelist` 某个元素，里面指定了寻找要分配物理页的 `zone` 次序。`zone` 指向 `zones` 数组，`classzone` 指向 `zones` 数组第一个元素。

```
min = 1UL << order;
```

根据 `order` 计算实际分配页数。

```
for (;;) {
    zone_t *z = *(zone++);
    if (!z)
        break;

    min += z->pages_low;
    if (z->free_pages > min) {
        page = rmqueue(z, order);
        if (page)
            return page;
```

```
    }
}
```

从第一个 zone 开始，将请求大小和 zone 水位比较，如果 zone 中的空闲页数大于水位 `pages_low`+请求大小，那么调用 `rmqueue` 分配物理页。若分配成功，返回第一个物理页的描述符地址。否则继续查询下一个 zone。

```
classzone->need_balance = 1;
mb();
if (waitqueue_active(&kswapd_wait))
    wake_up_interruptible(&kswapd_wait);
```

（上面分配未成功，即各 zone 都超过水位 `pages_low` 了）设置 `classzone` 的 `need_balance` 标志，同时唤醒 `kswapd`，期待 `kswapd` 能够将 `classzone` 中某些物理页交换出去。`kswapd` 将会对设置了 `need_balance` 的 zone 进行处理。（参考 `kswapd`）

```
zone = zonelist->zones;
min = 1UL << order;
for (;;) {
    unsigned long local_min;
    zone_t *z = *(zone++);
    if (!z)
        break;

    local_min = z->pages_min;
    if (!(gfp_mask & __GFP_WAIT))
        local_min >>= 2;
    min += local_min;
    if (z->free_pages > min) {
        page = rmqueue(z, order);
        if (page)
            return page;
    }
}
```

重新顺序遍历 `zonelist` 中的各个 zone，如果 zone 的空闲页数大于请求大小+水位 `pages_min`，那么调用 `rmqueue` 分配。其中，若请求标志为“不可等”，那么将水位暂时除以 4 再和空闲页数比较。

如果分配成功，则返回页描述符。

```
/* here we're in the low on memory slow path */

rebalance:
    if (current->flags & (PF_MEMALLOC | PF_MEMDIE)) {
        zone = zonelist->zones;
        for (;;) {
```



```

        zone_t *z = *(zone++);
        if (!z)
            break;

        page = rmqueue(z, order);
        if (page)
            return page;
    }
    return NULL;
}

```

（如果上述分配未成功，表示 zone 超过水位 pages_min 了，情况紧急）检查当前进程标志，如果具有 PF_MEMALLOC、PF_MEMDIE 中的一个，那么不管什么水位了，直接调用 rmqueue 分配，如果成功，返回。

通常情况下，进程标志为上述两个的，这个进程可能就是 kswapd、oom killer。

```

/* Atomic allocations - we can't balance anything */
if (!(gfp_mask & __GFP_WAIT))
    return NULL;

```

运行到这里，说明没有足够的空闲物理页满足请求。如果请求又是标志为“不可等”，那么直接返回 NULL。否则继续。

```

page = balance_classzone(classzone, gfp_mask, order, &freed);
if (page)
    return page;

```

调用 balance_classzone，尝试将 classzone 中的物理页释放。如果释放成功，直接返回。

```

zone = zonelist->zones;
min = 1UL << order;
for (;;) {
    zone_t *z = *(zone++);
    if (!z)
        break;

    min += z->pages_min;
    if (z->free_pages > min) {
        page = rmqueue(z, order);
        if (page)
            return page;
    }
}

```

再次察看水位 pages_min，如果空闲页数够了，那么调用 rmqueue 分配。若分配成功，返回。再次察看的原因是，上面操作可能释放了某些物理页。

```

/* Don't let big-order allocations loop */
if (order > 3)
    return NULL;

```

如果请求大小过大,最好还是返回 `NULL`。这可能是因为在这种情况下,下面的操作(`kswapd`)代价很高。

```

/* Yield for kswapd, and try again */
current->policy |= SCHED_YIELD;
__set_current_state(TASK_RUNNING);
schedule();
goto rebalance;

```

最后只好自动放弃 `cpu`, 等待 `cpu` 调度。跳转到 `rebalance` 执行代码。

rmqueue

```

static struct page * rmqueue(zone_t *zone, unsigned int order)
{
    free_area_t * area = zone->free_area + order;
    unsigned int curr_order = order;
    struct list_head *head, *curr;
    unsigned long flags;
    struct page *page;

```

根据 `order`, 将 `area` 指向相应的内存分组。

```

spin_lock_irqsave(&zone->lock, flags);
do {
    head = &area->free_list;
    curr = memlist_next(head);

```

锁住该 `zone`, 将 `head` 指向 `free_list` 头部, `curr` 指向第一个物理页。

```

    if (curr != head) {
        unsigned int index;

        page = memlist_entry(curr, struct page, list);

```

如果 `free_list` 非空, 从 `list` 中得到描述符。

```

    if (BAD_RANGE(zone, page))
        BUG();
    memlist_del(curr);

```

如果该物理页非法, 出错。否则将该物理页从 `free_list` 中删除 (也就是拿出)。这里不用检查后续的也不是空闲。因为在 `buddy` 分配器中, 能保证后续有足够的空闲物理页可供分配。

```

    index = page - zone->zone_mem_map;
    if (curr_order != MAX_ORDER-1)

```

```
MARK_USED(index, curr_order, area);
zone->free_pages -= 1UL << order;
```

如果不是在最大那组分配的, 将 area 的 map 数组中相对应位设置为 1。然后 zone->free_pages 减少被分配的页数。

```
page = expand(zone, page, index, order, curr_order, area);
spin_unlock_irqrestore(&zone->lock, flags);
```

调用 expand, 尝试着将剩余的连续物理页分布到较小的组中; 然后将锁释放。

```
set_page_count(page, 1);
if (BAD_RANGE(zone, page))
    BUG();
if (PageLRU(page))
    BUG();
if (PageActive(page))
    BUG();
return page;
}
```

设置 count 为 1, 然后检查不应该出现的标志, 否则出错。若检查通过, 返回页描述符。

```
curr_order++;
area++;
} while (curr_order < MAX_ORDER);
spin_unlock_irqrestore(&zone->lock, flags);

return NULL;
```

若该组中没有找到连续物理页来满足分配, 尝试着从更大的组中分配, 重复上面 do 语句, 直到遍历完最大组。

如果一直没有找到满足要求的物理页, 则返回 NULL。

expand

```
#define MARK_USED(index, order, area) \
    __change_bit((index) >> (1+(order)), (area)->map)

static inline struct page * expand (zone_t *zone, struct page *page,
    unsigned long index, int low, int high, free_area_t * area)
{
    unsigned long size = 1 << high;
```

zone 是要分配的 zone。index 指向该物理块第一个物理页描述符在 mem_map 数组中的下标。low 是请求大小, 而 high 是当前将要被分配的空闲物理块大小。area 是 zone 中的 free_area 数组首指针。

size 是空闲块大小。

```
while (high > low) {
    if (BAD_RANGE(zone,page))
        BUG();
    area--;
    high--;
    size >>= 1;
    memlist_add_head(&(page)->list, &(area)->free_list);
    MARK_USED(index, high, area);
    index += size;
    page += size;
}
```

如果空闲块大于请求大小，那么需要将剩余空闲块分存在比现在规模小的组中。所以当 high 大于 low 时，进入循环：

将 area 指向小一级的 free_area_t。

将 high 更新为 area 所指向组的大小

将 size 减半

将空闲块的低地址部分（一半空闲页）存放入 area 所指向的空闲链表中。

将空闲块的高地址部分标志为已分配。

index, page 指向高地址部分，

如果高地址部分仍然大于请求大小，那么继续循环，直到剩余空间都纳入到 free_area_t 相应的数组中。

```
if (BAD_RANGE(zone,page))
    BUG();
return page;
}
```

确认 page 所指向的被分配空闲块在 zone 的范围之内，然后返回 page。

1.5.3 __free_pages_ok 函数

```
static void __free_pages_ok (struct page *page, unsigned int order)
{
    unsigned long index, page_idx, mask, flags;
    free_area_t *area;
    struct page *base;
    zone_t *zone;

    /* Yes, think what happens when other parts of the kernel take
     * a reference to a page in order to pin it for io. -ben
     */
    if (PageLRU(page))
```

```
lru_cache_del(page);
```

将 `page` 从 `lru` 链表中删除。

```
if (page->buffers)
    BUG();
if (page->mapping)
    BUG();
if (!VALID_PAGE(page))
    BUG();
if (PageSwapCache(page))
    BUG();
if (PageLocked(page))
    BUG();
if (PageLRU(page))
    BUG();
if (PageActive(page))
    BUG();
page->flags &= ~(1<<PG_referenced) | (1<<PG_dirty));
```

确保该 `page` 在被释放前的标志都是正确的，然后将该 `page` 的 `PG_referenced` 和 `PG_dirty` 标志去掉。

```
if (current->flags & PF_FREE_PAGES)
    goto local_freelist;
```

如果当前进程标志为 `PF_FREE_PAGES`，则说明 `__free_pages_ok` 在 `balance_classzone` 调用。跳转到 `local_freelist` 处理。

`local_freelist` 部分代码和 `balance_classzone` 相对应，但是这两部分代码在此内核版本中并不匹配。《Memory Management in Linux》^[6]中猜测，这部分代码缺乏后续补丁导致的，下面是原话：

“At the moment `nr_local_pages` is being used as a flag to indicate if there is a free page block on the local free list of the current process. The following code is incomplete and might be clearer when newer patches of Andrea are merged in. There is a mismatch between what `__free_pages_ok` is actually doing, and what `balance_classzone` is expecting it to do. The following code believes that there are many free blocks of different orders on the local free list instead of one and tries to find the block of correct order and return that to the process while freeing the rest of the page blocks in reverse order. So we will skip over this piece of code until someone manages to complete it.”

```
back_local_freelist:

    zone = page->zone;

    mask = (~0UL) << order;
```

```

base = zone->zone_mem_map;
page_idx = page - base;
if (page_idx & ~mask)
    BUG();
index = page_idx >> (1 + order);

area = zone->free_area + order;

spin_lock_irqsave(&zone->lock, flags);

zone->free_pages -= mask;

```

正常情况下，`__free_pages_ok` 不会跳转到 `local_freelist` 执行，而是直接往下走。先通过 `page` 中的 `zone` 成员得到 `zone` 描述符。计算该物理块的首个页框在 `mem_map` 数组中的下标 `page_idx`。接着判断 `page_idx` 是否是块大小的整数倍。然后计算该物理块在相应大小 `free_area` 中 `bitmap` 数组中的位置 `index`。然后锁定该 `zone`，将 `zone` 的空闲页数加上块大小。最后一行语句写的不是很明白，这是因为： $\text{free_pages} = \text{free_pages} - \text{mask} = \text{free_pages} - (0\text{xfffffff} - 2^{\text{order}} + 1) = \text{free_pages} - 2^{\text{order}}$ ，这里 2^{order} 就是该物理块的页数。

```

while (mask + (1 << (MAX_ORDER-1))) {
    struct page *buddy1, *buddy2;

```

`while` 循环语句中的条件也比较容易迷糊人。循环中，`mask` 是递减的。事实上循环条件可以转换为这样：

```
while(0xfffffff - 2order + 1 + 2MAX_ORDER-1)
```

等同于：

```
while(2MAX_ORDER-1 - 2order)
```

所以当 `order` 逐渐增减时（也就是 `mask` 逐渐减半的时候），等 `order` 大于 `MAX_ORDER-1` 时，就跳出循环。

```

if (area >= zone->free_area + MAX_ORDER)
    BUG();
if (!__test_and_change_bit(index, area->map))
    /*
     * the buddy page is still allocated.
     */
    break;

```

检查 `area` 是否越界，然后判断是否 `buddy` 是否空闲。如果不空闲，则退出循环。否则下面将进行合并。

```

/*
 * Move the buddy up one level.
 */
buddy1 = base + (page_idx ^ ~mask);
buddy2 = base + page_idx;

```

```

    if (BAD_RANGE(zone,buddy1))
        BUG();
    if (BAD_RANGE(zone,buddy2))
        BUG();

```

计算 buddy 和该物理块的首个页框的描述符地址。然后检查他们是否越界。

```

    memlist_del(&buddy1->list);
    mask <= 1;
    area++;
    index >= 1;
    page_idx &= mask;
}

```

将 buddy 物理块删除。然后更新 mask, area, index, page_idx, 继续进入循环。直到不能合并。

```

    memlist_add_head(&(base + page_idx->list, &area->free_list);

    spin_unlock_irqrestore(&zone->lock, flags);
    return;

```

然后将新合并的物理块插入到当前 area 的 free_list 中。释放 zone 锁，返回。过程结束。

```

local_freelist:
    if (current->nr_local_pages)
        goto back_local_freelist;
    if (in_interrupt())
        goto back_local_freelist;

    list_add(&page->list, &current->local_pages);
    page->index = order;
    current->nr_local_pages++;
}

```

如果 nr_local_pages 非 0，转入正常释放部分。如果调用时在中断中，也转入正常释放部分。否则，将要释放的物理块插入到进程的 local_pages 链表中，更新 nr_local_pages 为 1。balance_classzone 调用此函数时，才有可能进入此部分执行。

到此，__free_pages_ok 过程结束。

2 Slab 分配器

buddy 算法解决了 external fragment 问题，然后没有解决 internal fragment。试想如果有某个程序反复申请只有几个字节的内存空间，难道内核为了这几个字节不断给申请者分配整个页吗？

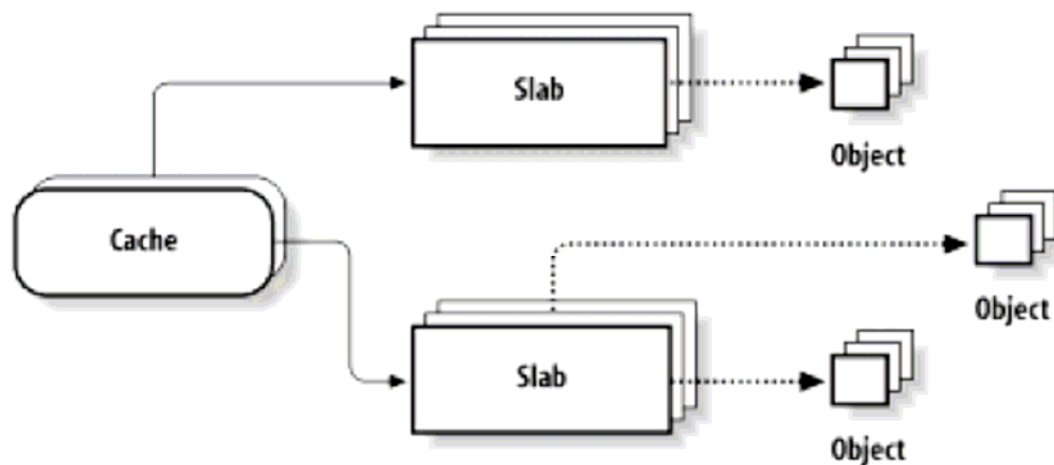
为了解决这个页内碎片，Linux 采用了 Slab 分配器，该分配器是从 Sun Solaris 中借鉴的，设计思想参考^[9]。

Slab 分配器的优点主要有以下三点：

- 解决 buddy 不能解决的页内碎片问题
- 缓存 object，以加快分配，初始化，释放操作
- 利用着色充分利用硬件 cache

Slab 中有三个逻辑单元，从大到小，cache，slab，object。每个 cache 中存放多个 slab，每个 slab 中又含有多个 object。

理论上针对任何一类固定大小的内存大小申请，都可以用 slab 分配器。但是用户的小内存申请无法事先预知，所以采用 2 次幂的 cache 来满足小内存申请。对于已预知的内核小内存申请，例如 fs_cache，dentry_cache 等等，都有相对应 cache。



2.1 Slab

每个 Slab 由若干个连续物理页框组成，这些物理页框是向 buddy 分配器请求得到。每个 Slab 中包含若干个 Object。Slab 描述符如下：

```
/*
 * slab_t
 *
 * Manages the objs in a slab. Placed either at the beginning of mem allocated
 * for a slab, or allocated from an general cache.
 * Slabs are chained into three list: fully used, partial, fully free slabs.
 */
typedef struct slab_s {
    struct list_head    list;
    unsigned long       colouroff;
    void                *s_mem;    /* including colour offset */
}
```



```

    unsigned int    inuse;        /* num of objs active in slab */
    kmem_bufctl_t   free;
} slab_t;

```

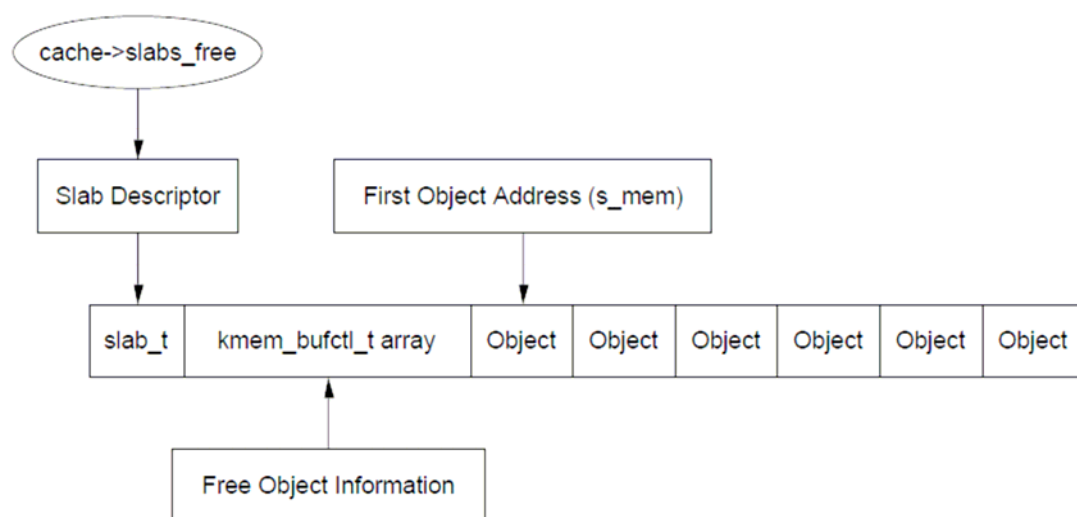
成员变量说明:

list 指向 cache 中的链表指针
 colouroff 着色偏移量
 s_mem 指向 slab 中的第一个 object
 inuse 非空闲的 object 个数
 free 指向第一个空闲 object

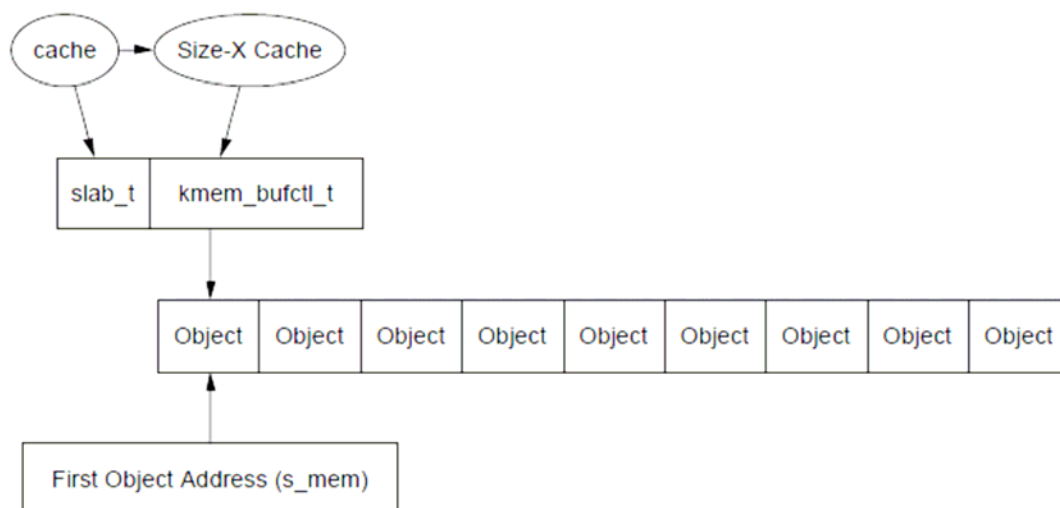
每个 Slab 中包含若干个 object，除这之外，还可能包含一个 slab_t 结构，还有和 object 个数相等的 kmem_bufctl_t 元素的数组。slab_t 和 kmem_bufctl_t，简称为 slab 管理信息。

slab 管理信息也可能不包含在 slab 中，也可能在 slab 中，这取决于运行时的情况。这两种情况下，slab 的组成是不一样的。

slab 管理信息在 slab 中的情况:



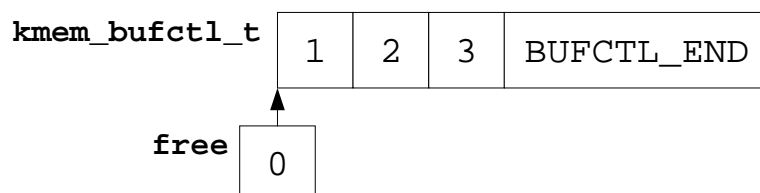
slab 管理信息不在 slab 中的情况:



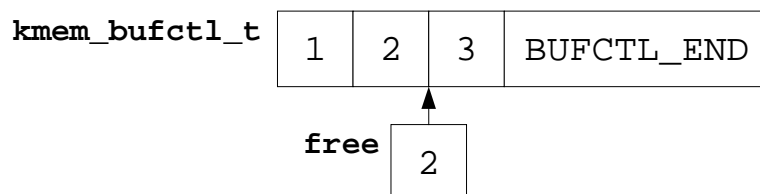
slab 管理信息中 `kmem_bufctl_t` 数组，我们在 `slab_t` 结构找不到相应的定义。但是我们如果熟悉 `kmem_cache_create` 过程的话，它实际上是个整型数组，里面记录着 slab 中所有 object 的空闲情况。结合 `slab_t` 中的 `free` 成员，数组随着 object 的分配，释放而随之改变。`kmem_bufctl_t` 数组的内容是这样安排的：

- `free` 指向第一个空闲 object，即 `kmem_bufctl_t` 数组的下标
- `free` 指向的数组元素内容指向下一个空闲 object 的下标
- 依此类推，直到最后一个 `kmem_bufctl_t` 数组中的元素指向结尾标志 `BUFCTL_END`。

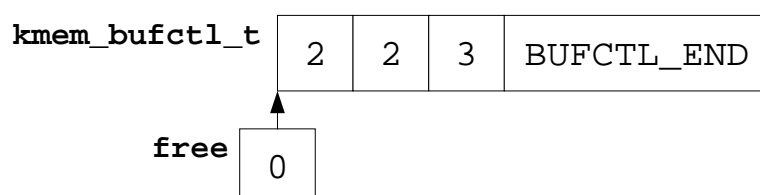
举个例子来说，假设我们 slab 中有 4 个 object。那么 slab 的组成情况就是：



当分配前 2 个 object 之后，组成情况变成这样：



虽然前 2 个元素的内容没有更改，却已经无效。有效的内容是 `free` 和后 2 个元素的内容。当使用者释放第 1 个 object 后，其内容更新为这样：



所以空闲 object 就是第 1, 3, 4 个 object。

到目前为止，slab 中基本组成已经了解清楚。不过考虑到要着色问题，其组成部分中的细节内容还没有提及到。事实上，一个 slab 中，除了 slab 管理信息，object 之外，还包含着色空间、浪费空间，以及 slab 描述符，kmem_bufctl_t 数组对齐所需空间，最后还有每个 object 对齐所需空间。一个复杂的组成应该是这样的：

着色空间	Slab描述符 bufctl数组	对齐	Object	对齐	Object	对齐	。 。 。	Object	对齐	剩余空间
------	---------------------	----	--------	----	--------	----	-------	--------	----	------

Linux 对于上述组成部分的实现有着细微的差别。它将上图中的 object 实际大小+对齐之和当作 object 大小来处理。所以上图可以简单地认为是这样的：

着色空间	Slab描述符 bufctl数组	对齐	Object	Object	。 。 。	Object	剩余空间
------	---------------------	----	--------	--------	-------	--------	------

2.1.1 slab 着色

着色的目的是为了来自同一个 cache 中的 object 尽可能避免在硬件缓存（L1 cache）中的位置错开，从而减少硬件 cache 内容的冲突情况。

kmem_cache_estimate 过程根据 slab 大小，减去 slab 管理信息和 object 所占用的空间，得到剩余的空闲空间 left_over。然后根据地址对齐情况（一般情况下，需要按照 L1 cache line 大小来对齐），得到该 cache 允许使用的颜色数。

slab 着色信息存放在 cache 结构中三个成员变量中：

```
size_t          colour;          /* cache colouring range */
unsigned int     colour_off;      /* colour offset */
unsigned int     colour_next;     /* cache colouring */
```

colour 指定该 cache 可使用的颜色数。colour_off 指定颜色的偏移量。colour_next 指定下一个 slab 被创建时所使用的颜色。

2.2 Cache

cache 是 slab 分配器中最高层次的逻辑单元。结构 kmem_cache_s 是用来描述一个 cache 的。它定义在 mm/slab.c 中。

```
struct kmem_cache_s {
/* 1) each alloc & free */
/* full, partial first, then free */
struct list_head  slabs_full;
struct list_head  slabs_partial;
struct list_head  slabs_free;
unsigned int      objsize;
unsigned int      flags;          /* constant flags */
unsigned int      num;           /* # of objs per slab */
spinlock_t        spinlock;
#ifdef CONFIG_SMP
unsigned int      batchcount;
#endif
};
```

```

#endif

/* 2) slab additions /removals */
/* order of pgs per slab (2^n) */
unsigned int      gfporder;

/* force GFP flags, e.g. GFP_DMA */
unsigned int      gfpflags;

size_t            colour;          /* cache colouring range */
unsigned int      colour_off;      /* colour offset */
unsigned int      colour_next;     /* cache colouring */
kmem_cache_t      *slabp_cache;
unsigned int      growing;
unsigned int      dflags;          /* dynamic flags */

/* constructor func */
void (*ctor)(void *, kmem_cache_t *, unsigned long);

/* de-constructor func */
void (*dtor)(void *, kmem_cache_t *, unsigned long);

unsigned long      failures;

/* 3) cache creation/removal */
char               name[CACHE_NAMELEN];
struct list_head   next;
#ifdef CONFIG_SMP
/* 4) per-cpu data */
cpucache_t         *cpudata[NR_CPUS];
#endif
#ifdef STATS
unsigned long       num_active;
unsigned long       num_allocations;
unsigned long       high_mark;
unsigned long       grown;
unsigned long       reaped;
unsigned long       errors;
#endif
#ifdef CONFIG_SMP
atomic_t            allochit;
atomic_t            allocmiss;
atomic_t            freehit;
atomic_t            freemiss;
#endif
#endif

```

```
#endif
};
```

成员变量说明，有关 `smp` 和统计信息的成员不说明：

<code>slabs_full</code>	指向 slab 描述符的双向链表，无空闲 object
<code>slabs_partial</code>	指向 slab 描述符的双向链表，部分空闲 object
<code>slabs_free</code>	指向 slab 描述符的双向链表，只有空闲 object
<code>objsize</code>	slab 中 object 大小
<code>flags</code>	描述 cache 的静态标志，在该 cache 生命周期中不改变
<code>num</code>	单个 slab 中的 object 数
<code>spinlock</code>	自旋锁
<code>gfporder</code>	单个 slab 所占用的页框数
<code>gfpflags</code>	请求页框时的页框标志
<code>colour</code>	slab 中可以使用颜色数
<code>colour_off</code>	每种颜色的跨度（字节数），即 slab 中每个 object 的偏移。
<code>colour_next</code>	下次要使用的颜色
<code>slabp_cache</code>	当 <code>flags</code> 标志为 <code>CFLGS_OFF_SLAB</code> 时，指向 slab 管理信息
<code>growing</code>	该 cache 是否正在 growing，避免和 shrink 操作冲突
<code>dflags</code>	动态标志，在该 cache 的生命周期中可能会被改变
<code>ctor</code>	初始化函数
<code>dtor</code>	析构函数
<code>failures</code>	未使用，永远为 0
<code>name</code>	名字
<code>next</code>	下一个 cache

成员变量 `flags` 可以设置的标志有：

<code>CLGS_OFF_SLAB</code>	表示 slab 管理信息不存放在 slab 中
<code>CLGS_OPTIMIZE</code>	未使用

上述两个标志是在 `kmem_cache_create` 执行时确定。

<code>SLAB_HWCACHE_ALIGN</code>	L1 Cache 对齐
<code>SLAB_MUST_HWCACHE_ALIGN</code>	L1 Cache 对齐，不考虑付出的代价
<code>SLAB_NO_REAP</code>	当内存紧张时，此 cache 不参与回收
<code>SLAB_CACHE_DMA</code>	使用 zone dma 中的物理页

上述 4 个标志在调用 `kmem_cache_create` 时，由参数传递。换句话说，这些标志应该由调用者创建时指定。

成员变量 `dflags` 可以设置的标志只有一个，`DFLGS_GROWN`，当某个 cache 被设置成 `DFLGS_GROWN` 的时候，表示该 cache 刚刚完成 `kmem_cache_grow` 过程，这样 `kmem_cache_reap` 过程将忽略这个 cache，然后将这个标志清除。

2.3 Slab 分配器初始化

由于所有的 cache 描述符以及部分 slab 描述符都存放在 slab 分配器地 cache 中。所以概念上，我们可能会想：这是如何做到的？这不是典型的先有蛋还是先有鸡问题吗？

而事实上，Slab 分配器初始化代码很简单。由于 cache 描述符是存放在 slab 分配器的一个 cache 中的。那么只需要初始化好这个 cache 的描述符即可。全局变量 cache_cache 就是 cache 描述符的 cache，所有的 cache 描述符都缓冲在 cache_cache 中。cache_cache 在 slab.c 中静态声明。

```
/* internal cache of cache description objs */
static kmem_cache_t cache_cache = {
    slabs_full:      LIST_HEAD_INIT(cache_cache.slabs_full),
    slabs_partial:   LIST_HEAD_INIT(cache_cache.slabs_partial),
    slabs_free:      LIST_HEAD_INIT(cache_cache.slabs_free),
    objsize:         sizeof(kmem_cache_t),
    flags:           SLAB_NO_REAP,
    spinlock:        SPIN_LOCK_UNLOCKED,
    colour_off:      L1_CACHE_BYTES,
    name:            "kmem_cache",
};
```

cache_cache 其他成员变量由 kmem_cache_init 函数初始化。

同时，每个 cache 都链接入一个全局链表中，cache_chain，与之相对应的，还有一个全局信号量（semaphore）cache_chain_sem 来同步对 cache_chain 的存取操作。

除此之外，slab 初始化工作还有一个任务，就是建立起通用 cache，它由 kmem_cache_sizes_init 函数初始化。kmem_cache_sizes_init 的主要处理就是依据不同大小的通用 cache，调用 kmem_cache_alloc 建立这些不同大小的 cache。这里就不详述了。

kmem_cache_init

```
void __init kmem_cache_init(void)
{
    size_t left_over;

    init_MUTEX(&cache_chain_sem);
    INIT_LIST_HEAD(&cache_chain);

    kmem_cache_estimate(0, cache_cache.objsize, 0,
                       &left_over, &cache_cache.num);
    if (!cache_cache.num)
        BUG();

    cache_cache.colour = left_over/cache_cache.colour_off;
    cache_cache.colour_next = 0;
}
```

初始化 cache_chain_sem 和 cache_chain。根据 cache 描述符结构大小，调用 kmem_cache_estimate 函数来进行 slab 估算。然后设定 cache_cache 可使用的颜色数。

这样 Slab 分配器已经进入准备好状态，因为 slab 所使用的物理内存将会在 kmem_cache_alloc

中被分配。

`kmem_cache_estimate` 和 `kmem_cache_alloc` 参考下面。

2.4 Slab 分配器接口

在 `cache` 被创建之前，对该 `cache` 的分配请求是不能进行的。在创建好 `cache` 描述符之后，该 `cache` 就可以使用 `kmem_cache_alloc` 来请求分配一个 `obj`，使用 `kmem_cache_free` 释放一个 `obj`。

然而，对通用 `cache`，内核还定义了另外一组分配/释放接口，`kmalloc/kfree`。由于通用 `cache` 在函数 `kmem_cache_sizes_init` 中被创建，所以只要直接调用 `kmalloc/kfree` 请求分配，释放即可。

2.4.1 创建 `cache`

`kmem_cache_create` 负责创建一个指定名字的 `cache`。

```
kmem_cache_t *
kmem_cache_create (const char *name, size_t size,
size_t offset, unsigned long flags,
void (*ctor)(void*, kmem_cache_t *, unsigned long),
void (*dtor)(void*, kmem_cache_t *, unsigned long))
```

<code>name</code>	cache 的名字
<code>size</code>	object 的大小
<code>offset</code>	颜色偏移量
<code>flags</code>	cache 创建时标志
<code>ctor</code>	初始化函数指针
<code>dtor</code>	析构函数指针

运行过程如下：（忽略 `DEBUG` 部分）

```
/*
 * Sanity checks... these are all serious usage bugs.
 */
if ((!name) ||
    ((strlen(name) >= CACHE_NAMELEN - 1)) ||
    in_interrupt() ||
    (size < BYTES_PER_WORD) ||
    (size > (1<<MAX_OBJ_ORDER)*PAGE_SIZE) ||
    (dtor && !ctor) ||
    (offset < 0 || offset > size))
    BUG();

.....
/*
 * Always checks flags, a caller might be expecting debug
 * support which isn't available.
```

```

    */
    if (flags & ~CREATE_MASK)
        BUG();

```

完整性检查

```

/* Get cache's description obj. */
cachep = (kmem_cache_t *) kmem_cache_alloc(&cache_cache, SLAB_KERNEL);
if (!cachep)
    goto opps;
memset(cachep, 0, sizeof(kmem_cache_t));

```

调用 `kmem_cache_alloc`，从 `cache_cache`（也叫 `kmem_cache`）中获取一个 `kmem_cache_t` 结构的 object。然后初始化为 0。

```

/* Check that size is in terms of words. This is needed to avoid
 * unaligned accesses for some archs when redzoning is used, and makes
 * sure any on-slab bufctl's are also correctly aligned.
 */
if (size & (BYTES_PER_WORD-1)) {
    size += (BYTES_PER_WORD-1);
    size &= ~(BYTES_PER_WORD-1);
    printk("%sForcing size word alignment - %s\n", func_nm, name);
}

```

确认 `size` 是否 word 对齐，如果不是，强制 word 对齐。

```

align = BYTES_PER_WORD;
if (flags & SLAB_HWCACHE_ALIGN)
    align = L1_CACHE_BYTES;

```

检查 `SLAB_HWCACHE_ALIGN` 标志，如有，则 L1 cache 对齐。Intel CPU 中的 L1 cache line 长度参考“IA-32 Intel Architecture Software Developer’s Manual”。

```

/* Determine if the slab management is 'on' or 'off' slab. */
if (size >= (PAGE_SIZE>>3))
    /*
     * Size is large, assume best to place the slab management obj
     * off-slab (should allow better packing of objs).
     */
    flags |= CFLGS_OFF_SLAB;

```

如果 object size 大于 512，则标志为 `CFLGS_OFF_SLAB`。

```

if (flags & SLAB_HWCACHE_ALIGN) {
    /* Need to adjust size so that objs are cache aligned. */
    /* Small obj size, can get at least two per cache line. */
    /* FIXME: only power of 2 supported, was better */
    while (size < align/2)

```



```

        align /= 2;
        size = (size+align-1)&(~(align-1));
    }

```

比较 align 和 size 的大小，使得 align 尽可能地小，这样一个 cache line 尽可能包含更多的 object。然后将 size 按照 align 大小对齐。

```

    do {
        unsigned int break_flag = 0;
cal_wastage:
        kmem_cache_estimate(cachep->gfporder, size, flags,
                           &left_over, &cachep->num);
        if (break_flag)
            break;
        if (cachep->gfporder >= MAX_GFP_ORDER)
            break;
        if (!cachep->num)
            goto next;
        if (flags & CFLGS_OFF_SLAB && cachep->num > offslab_limit) {
            /* Oops, this num of objs will cause problems. */
            cachep->gfporder--;
            break_flag++;
            goto cal_wastage;
        }

        /*
         * Large num of objs is good, but v. large slabs are currently
         * bad for the gfp()s.
         */
        if (cachep->gfporder >= slab_break_gfp_order)
            break;

        if ((left_over*8) <= (PAGE_SIZE<<cachep->gfporder))
            break; /* Acceptable internal fragmentation. */
next:
        cachep->gfporder++;
    } while (1);

```

调用 kmem_cache_estimate 函数，计算每个 slab 中应该含有多少个 object，同时也得到 slab 的剩余空间。kmem_cache_estimate 需要的信息有：一个 slab 占用多少个物理页，object 大小，还有标志 flags。

根据 kmem_cache_estimate 返回的结果，判断 slab 大小是否合适，然后将 gfporder 调整到合适的值。

```

    if (!cachep->num) {

```

```

    printk("kmem_cache_create: couldn't create cache %s.\n", name);
    kmem_cache_free(&cache_cache, cachep);
    cachep = NULL;
    goto opps;
}

```

如果不成功，释放从 cache_cache 中申请到 object，跳转到 opps 退出。

```

slab_size = L1_CACHE_ALIGN(cachep->num*sizeof(kmem_bufctl_t)+sizeof(slab_t));
/*
 * If the slab has been placed off-slab, and we have enough space then
 * move it on-slab. This is at the expense of any extra colouring.
 */
if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
    flags &= ~CFLGS_OFF_SLAB;
    left_over -= slab_size;
}

```

判断在 CFLGS_OFF_SLAB 情况下，slab 中剩余空间能否放得下 slab 管理信息（slab_t 和 n*kmem_bufctl_t）。如果是去掉该标志，然后将管理信息大小计算到 slab_size 中。

```

/* Offset must be a multiple of the alignment. */
offset += (align-1);
offset &= ~(align-1);
if (!offset)
    offset = L1_CACHE_BYTES;
cachep->colour_off = offset;
cachep->colour = left_over/offset;

```

调整 offset 的大小，确保 offset 是 align 的倍数。然后初始化 cache 的 colour_off 和 colour 成员变量。

```

/* init remaining fields */
if (!cachep->gfporder && !(flags & CFLGS_OFF_SLAB))
    flags |= CFLGS_OPTIMIZE;

cachep->flags = flags;
cachep->gfpflags = 0;
if (flags & SLAB_CACHE_DMA)
    cachep->gfpflags |= GFP_DMA;
spin_lock_init(&cachep->spinlock);
cachep->objsize = size;
INIT_LIST_HEAD(&cachep->slabs_full);
INIT_LIST_HEAD(&cachep->slabs_partial);
INIT_LIST_HEAD(&cachep->slabs_free);

if (flags & CFLGS_OFF_SLAB)

```

```

    cachep->slabp_cache = kmem_find_general_cachep(slab_size,0);
    cachep->ctor = ctor;
    cachep->dtor = dtor;
    /* Copy name over so we don't have problems with unloaded modules */
    strcpy(cachep->name, name);

```

初始化 cache 的其他成员变量。如果是 CFLAG_OFF_SLAB，则调用 kmem_find_general_cachep 函数从通用缓存中获取大小合适的缓存，保存该 slab 管理信息。

```

/* Need the semaphore to access the chain. */
down(&cache_chain_sem);
{
    struct list_head *p;

    list_for_each(p, &cache_chain) {
        kmem_cache_t *pc = list_entry(p, kmem_cache_t, next);

        /* The name field is constant - no lock needed. */
        if (!strcmp(pc->name, name))
            BUG();
    }
}

/* There is no reason to lock our new cache before we
 * link it in - no one knows about it yet...
 */
list_add(&cachep->next, &cache_chain);
up(&cache_chain_sem);
oops:
    return cachep;

```

检查是否有相同名字的 cache 存在。然后将其加入 cache_chain 中。创建过程结束。

在这里我们可以看出，kmem_cache_create 仅仅建立了 cache 的管理信息，并没有给 slab 分配物理页。因为指向 slab 的 slabs_full，slabs_partial 和 slabs_free 都是空链表。

2.4.2 cache 分配接口

kmem_cache_alloc 是 cache 的分配接口，它直接调用了 __kmem_cache_alloc 函数。__kmem_cache_alloc 在同时满足下面情况下给该 cache 分配物理内存：

- 当前有要求分配 object 请求
- cache 中没有空闲 object

下面对 __kmem_cache_alloc 的说明不包括 SMP 情况。

```

static inline void * __kmem_cache_alloc (kmem_cache_t *cachep, int flags)

```

```

{
    unsigned long save_flags;
    void* objp;

    kmem_cache_alloc_head(cachep, flags);
try_again:
    local_irq_save(save_flags);

```

调用 `kmem_cache_alloc_head` 检查请求的标志 `SLAB_DMA` 和 `cache` 标志 `GFP_DMA`。如果不匹配，BUG 处理。

若检查通过，调用 `local_irq_save` 屏蔽中断，并保存 `save_flags`。

```
objp = kmem_cache_alloc_one(cachep);
```

调用 `kmem_cache_alloc_one`，事实上，它仅仅是个宏定义。定义如下：

```

/*
 * Returns a ptr to an obj in the given cache.
 * caller must guarantee synchronization
 * #define for the goto optimization 8-)
 */
#define kmem_cache_alloc_one(cachep) \
({ \
    struct list_head * slabs_partial, * entry; \
    slab_t *slabp; \
 \
    slabs_partial = &(cachep)->slabs_partial; \
    entry = slabs_partial->next; \
    if (unlikely(entry == slabs_partial)) { \
        struct list_head * slabs_free; \
        slabs_free = &(cachep)->slabs_free; \
        entry = slabs_free->next; \
        if (unlikely(entry == slabs_free)) \
            goto alloc_new_slab; \
        list_del(entry); \
        list_add(entry, slabs_partial); \
    } \
 \
    slabp = list_entry(entry, slab_t, list); \
    kmem_cache_alloc_one_tail(cachep, slabp); \
})

```

此宏定义也在 `slab.c` 文件中。先检查 `cache` 中的 `slabs_partial` 链表，如果 `slabs_partial` 不为空，则直接调用 `kmem_cache_alloc_one_tail` 得到一个 `obj`。否则，`slabs_partial` 为空，接着检查 `slabs_free` 是否为空。如果不为空，将 `slabs_free` 中的一个 `slab` 抽取出来，移到 `slabs_partial` 链表中。否则，`slabs_free` 为空，表示该 `cache` 中缺乏空闲 `slab`，跳转到 `alloc_new_slab` 处执

行。

`alloc_new_slab` 标签定义在 `__kmem_alloc_alloc` 中，因为 `kmem_cache_alloc_one`，所以此跳转是有效的。

```
local_irq_restore(save_flags);
return objp;
```

若 `kmem_cache_alloc_one` 分配 `obj` 成功，则开启中断，并还原 `save_flags`，然后返回 `obj` 地址。

```
alloc_new_slab:
    local_irq_restore(save_flags);
    if (kmem_cache_grow(cachep, flags))
        /* Someone may have stolen our objs. Doesn't matter, we'll
         * just come back here again.
         */
        goto try_again;
    return NULL;
}
```

如果执行此处代码，说明 `kmem_cache_alloc_one` 宏处理中，此 `cache` 没有可用的空闲空间。注意此时的中断还是屏蔽的，所以要先开启中断，并还原 `save_flags`。然后调用 `kmem_cache_grow` 分配一个 `slab`，接着跳转到 `try_again` 继续分配一个 `obj`。若 `kmem_cache_grow` 函数失败，返回 `Null`。

`kmem_cache_alloc_one_tail`

```
static inline void * kmem_cache_alloc_one_tail (kmem_cache_t *cachep,
                                                slab_t *slabp)
{
    void *objp;

    STATS_INC_ALLOCED(cachep);
    STATS_INC_ACTIVE(cachep);
    STATS_SET_HIGH(cachep);

    /* get obj pointer */
    slabp->inuse++;
    objp = slabp->s_mem + slabp->free*cachep->objsize;
    slabp->free=slab_bufctl(slabp)[slabp->free];
```

将 `slab` 描述符中的 `inuse` 加 1。通过 `s_mem` 和 `free` 得到可用 `obj` 的地址，更新 `free`，使其指向下一个可用 `obj`。

```
if (unlikely(slabp->free == BUFCTL_END)) {
    list_del(&slabp->list);
```

```

        list_add(&slabp->list, &cachep->slabs_full);
    }
    return objp;
}

```

如果 free 所指向的 obj 不可用，表明这个 slab 已满。所以要将这个 slab 转移到 slabs_full 链表中。然后返回 obj 地址。

kmem_cache_grow

kmem_cache_grow 给指定的 cache 分配一个 slab，一个 slab 通常是 2^n 个物理页。该函数只能由 kmem_cache_alloc 调用。

```

/*
 * Grow (by 1) the number of slabs within a cache. This is called by
 * kmem_cache_alloc() when there are no active objs left in a cache.
 */
static int kmem_cache_grow (kmem_cache_t * cachep, int flags)
{
    slab_t *slabp;
    struct page *page;
    void *objp;
    size_t offset;
    unsigned int i, local_flags;
    unsigned long ctor_flags;
    unsigned long save_flags;

    /* Be lazy and only check for valid flags here,
     * keeping it out of the critical path in kmem_cache_alloc().
     */
    if (flags & ~(SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW))
        BUG();
    if (flags & SLAB_NO_GROW)
        return 0;

```

检查是否有不合法的请求标志，如果有 BUG 处理。判断标志中是否表明该 cache 不能 grow，如不能 grow，返回 0。

```

/*
 * The test for missing atomic flag is performed here, rather than
 * the more obvious place, simply to reduce the critical path length
 * in kmem_cache_alloc(). If a caller is seriously mis-behaving they
 * will eventually be caught here (where it matters).
 */
if (in_interrupt() && (flags & SLAB_LEVEL_MASK) != SLAB_ATOMIC)
    BUG();

```

如果在中断过程中被调用，确保 SLAB_ATOMIC 标志被设置。

```

ctor_flags = SLAB_CTOR_CONSTRUCTOR;
local_flags = (flags & SLAB_LEVEL_MASK);
if (local_flags == SLAB_ATOMIC)
    /*
     * Not allowed to sleep. Need to tell a constructor about
     * this - it might need to know...
     */
    ctor_flags |= SLAB_CTOR_ATOMIC;

```

设置 constructor 函数标志，如果 SLAB_ATOMIC 被指定，则也需要传递给 constructor 函数。需要保证 constructor 不能在处理过程中 sleep。

```

/* About to mess with non-constant members - lock. */
spin_lock_irqsave(&cachep->spinlock, save_flags);

```

屏蔽中断，保存 save_flags，并锁定该 cache。

```

/* Get colour for the slab, and cal the next value. */
offset = cachep->colour_next;
cachep->colour_next++;
if (cachep->colour_next >= cachep->colour)
    cachep->colour_next = 0;
offset *= cachep->colour_off;
cachep->dflgs |= DFLGS_GROWN;

cachep->growing++;
spin_unlock_irqrestore(&cachep->spinlock, save_flags);

```

设置新 slab 的要使用的颜色及对应的偏移量，并计算下一个 slab 要使用的颜色。然后设置 GROWN 标志，并将 growing 增加 1（即表示该 cache 正在执行 growing 操作）。开启中断，还原 save_flags 标志，并释放该 cache 自旋锁。

```

/* Get mem for the objs. */
if (!(objp = kmem_getpages(cachep, flags)))
    goto failed;

/* Get slab management. */
if (!(slabp = kmem_cache_slabmgmt(cachep, objp, offset, local_flags)))
    goto opps1;

```

调用 kmem_getpages 给 slab 分配相应数目的物理页，接着调用 kmem_cache_slabmgmt 初始化 slab 描述符。

```

/* Nasty!!!!!! I hope this is OK. */
i = 1 << cachep->gfporder;
page = virt_to_page(objp);
do {

```

```

    SET_PAGE_CACHE(page, cachep);
    SET_PAGE_SLAB(page, slabp);
    PageSetSlab(page);
    page++;
} while (--i);

```

将页描述符中的 list 双向链表指针设置指向 cache 和 slab 描述符。这时候，页描述符中 list 并非形成一个双向链表。

```

kmem_cache_init_objs(cachep, slabp, ctor_flags);

```

调用 kmem_cache_init_objs 初始化 slab 中的 objs, bufctl 数组，以及 slab 描述符中的 free。

```

spin_lock_irqsave(&cachep->spinlock, save_flags);
cachep->growing--;

/* Make slab active. */
list_add_tail(&slabp->list, &cachep->slabs_free);
STATS_INC_GROWN(cachep);
cachep->failures = 0;

spin_unlock_irqrestore(&cachep->spinlock, save_flags);
return 1;

```

锁住 cache，并关闭中断。将 cache 的 growing 标志减 1，即置为 0。然后将 slab 链接入 cache 的 slabs_free 链表中，将 cache 的 failures 标志置为 0（实际上无任何用处）。最后是释放锁，开中断，返回 1。

```

oops1:
    kmem_freepages(cachep, objp);
failed:
    spin_lock_irqsave(&cachep->spinlock, save_flags);
    cachep->growing--;
    spin_unlock_irqrestore(&cachep->spinlock, save_flags);
    return 0;

```

如果上述过程失败，通常跳转到此处执行。释放新获得的物理页，锁定 cache，将 growing 置为 0，然后开启中断。返回 0。

2.4.3 cache 释放接口

kmem_cache_free 是 slab 分配器的释放接口。和 kmem_cache_alloc 类似，kmem_cache_free 中调用了 __kmem_cache_free 来处理释放实际工作。

kmem_cache_free 接口是要释放指定 cache 的一个 obj。

```

static inline void __kmem_cache_free (kmem_cache_t *cachep, void* objp)

```



```

{
#ifdef CONFIG_SMP
    ...
#else
    kmem_cache_free_one(cachep, objp);
#endif
}

```

`__kmem_cache_free` 又调用了 `kmem_cache_free_one` 来释放指定的 `objs`。

```

static inline void kmem_cache_free_one(kmem_cache_t *cachep, void *objp)
{
    slab_t* slabp;

    CHECK_PAGE(virt_to_page(objp));
    /* reduces memory footprint
     *
    if (OPTIMIZE(cachep))
        slabp = (void*)((unsigned long)objp & ~(PAGE_SIZE-1));
    else
        */
    slabp = GET_PAGE_SLAB(virt_to_page(objp));

```

`CHECK_PAGE` 检查 `obj` 所在的物理页是否标志为 `PF_slab`。由于在 `grow` 的时候, `obj` 所在的页描述符 `list` 成员并非形成链表, 而是在 `prev` 存放了 `slab` 描述符指针, `next` 存放了 `cache` 描述符指针。所以这里通过宏 `GET_PAGE_SLAB` 得到相应的 `slab` 描述符。

```

{
    unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;

    slab_bufctl(slabp)[objnr] = slabp->free;
    slabp->free = objnr;
}
STATS_DEC_ACTIVE(cachep);

```

更新 `slab` 描述符的 `free`, 以及 `bufctl` 数组中内容, 意味着这个 `obj` 就已经释放了。

```

/* fixup slab chains */
{
    int inuse = slabp->inuse;
    if (unlikely(!--slabp->inuse)) {
        /* Was partial or full, now empty. */
        list_del(&slabp->list);
        list_add(&slabp->list, &cachep->slabs_free);
    } else if (unlikely(inuse == cachep->num)) {
        /* Was full. */

```

```

        list_del(&slabp->list);
        list_add(&slabp->list, &cachep->slabs_partial);
    }
}
}

```

判断 slab 是否为空,或者部分为空,将 slab 链接入正确的链表中:slabs_free 或者 slabs_partial。

3 非连续内存分配

buddy 分配器并不能完全消除页外碎片问题,为了应付可能的情况(指总空闲空间大于请求大小,但空闲空间却不连续,这种情况 buddy 分配器没有办法处理),Linux 提供了一种机制,称 vmalloc 分配器。它允许将不连续的物理空间映射到连续的虚地址空间。这个虚地址空间的描述符:

```

struct vm_struct {
    unsigned long flags;
    void * addr;
    unsigned long size;
    struct vm_struct * next;
};

```

flags	标志
addr	起始虚地址
size	大小
next	下一个虚地址空间

由于该虚地址空间是非固定映射,所以 Linux 内核在 PAGE_OFFSET 之后的 1G 虚地址空间中留出了一部分虚地址空间。该部分地址空间大小取决于实际物理内存大小。起始地址由宏 VMALLOC_START 指定,宏 VMALLOC_END 指定结束地址。

```

#define VMALLOC_OFFSET (8*1024*1024)
#define VMALLOC_START  (((unsigned long) high_memory + 2*VMALLOC_OFFSET-1) & \
                        ~(VMALLOC_OFFSET-1))
#define VMALLOC_VMADDR(x) (((unsigned long)(x))
#if CONFIG_HIGHMEM
# define VMALLOC_END    (PKMAP_BASE-2*PAGE_SIZE)
#else
# define VMALLOC_END    (FIXADDR_START-2*PAGE_SIZE)
#endif

```

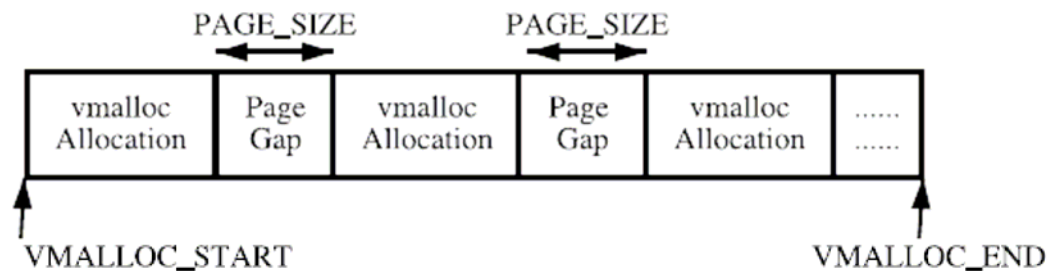
high_memory 是指向实际物理内存最高地址的虚地址(线性地址)。VMALLOC_START 在 high_memory 之后留出了至少 8M 的间隔,这是个安全区,目的是为了捕获越界访问(此说法来自^[5])。

我们假设在有 HIGHMEM 的情况下讨论, HIGHMEM 情况在下一章讨论。因此我们这里先

假设 `VMALLOC_END` 定义为 `PKMAP_BASE-2*PAGE_SIZE`。同时我们还假设这时候的物理内存大小超过 1G，小于 4G。在这种情况下，`high_memory` 指向物理地址 896M 处，值为 `PAGE_OFFSET+896M`，原因在下一章说明。

```
#define PKMAP_BASE (0xfe000000UL)
```

从 `PKMAP_BASE` 的值可以看出，`vmalloc` 所占用的虚地址空间略小于 100M。



在每个区域之间增加一个页大小的隔阂，同样也是为了防止越界访问。每个区域由 `vm_struct` 描述。在文件 `mm/vmalloc.c` 中，定义了全局变量 `vmlist`，当一个区域被创建的时候，这个 `vm_struct` 变量就会被链接入 `vmlist` 所指向的单向链表中，并按地址增序排列。

3.1 初始化

`vmalloc` 分配器的初始化是在 `slab` 分配器建立之后就隐性完成的。因为 `vmalloc` 分配器准备工作只需要满足 2 个条件：

- `high_memory` 值确定
- `kmalloc` 接口准备好

所以对于 `vmalloc` 分配器，不需要额外的初始化代码。

3.2 分配接口 `vmalloc/vmalloc_dma/vmalloc_32`

```
static inline void * vmalloc (unsigned long size)
{
    return __vmalloc(size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL);
}
```

`vmalloc` 是个内联函数，直接调用了 `__vmalloc` 函数。当调用者获得该区域的虚地址时，就可以使用这段内存了。典型的应用可以在内核驱动程序源码中找到。

`vmalloc_dma/vmalloc` 也调用 `__vmalloc`，唯一的区别是标志位不同。这些标志最终会影响 `alloc_page` 的行为。

`__vmalloc`

```
size = PAGE_ALIGN(size);
if (!size || (size >> PAGE_SHIFT) > num_physpages) {
    BUG();
}
```

```

        return NULL;
    }
    area = get_vm_area(size, VM_ALLOC);
    if (!area)
        return NULL;

```

将请求大小 `size` 按页大小对齐，然后检查是否等于 0 还是超出实际物理页总数。然后调用 `get_vm_area` 获取一个足够大的空闲区域 `area`。如果 `get_vm_area` 失败，返回空指针。

```

    addr = area->addr;
    if (vmalloc_area_pages(VMALLOC_VMADDR(addr), size, gfp_mask, prot)) {
        vfree(addr);
        return NULL;
    }
    return addr;

```

调用 `vmalloc_area_pages` 分配中间目录项，页表项，以及相应的物理页框。如果不成功，返回空地址，否则返回该区域起始虚地址。

get_vm_area

```

    area = (struct vm_struct *) kmalloc(sizeof(*area), GFP_KERNEL);
    if (!area)
        return NULL;

```

调用 slab 接口 `kmalloc` 从通用缓存中分配一个 `vm_struct`。

```

    size += PAGE_SIZE;
    addr = VMALLOC_START;

```

增加一个页的间隔，将临时变量 `addr` 指向 `vmalloc` 分配器的起始虚地址。

```

    write_lock(&vmlist_lock);
    for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
        if ((size + addr) < addr)
            goto out;
        if (size + addr <= (unsigned long) tmp->addr)
            break;
        addr = tmp->size + (unsigned long) tmp->addr;
        if (addr > VMALLOC_END-size)
            goto out;
    }

```

锁定 `vmlist` 链表。遍历 `vmlist` 链表，寻找现有的区域是否和要创建的区域在虚地址空间上有交集，直到找到一个合适的区域。如果没有找到合适的区域，则调转到 `out` 退出。

```

    area->flags = flags;
    area->addr = (void *)addr;

```

```

    area->size = size;
    area->next = *p;
    *p = area;
    write_unlock(&vmlist_lock);
    return area;

```

找到合适的区域之后，初始化 area 成员，然后将 area 插入到 vmlist 链表中。返回 area。

```

out:
    write_unlock(&vmlist_lock);
    kfree(area);
    return NULL;

```

找不到合适的区域，释放 area，返回空指针。

vmalloc_area_pages

```

pgd_t * dir;
unsigned long end = address + size;
int ret;

dir = pgd_offset_k(address);
spin_lock(&init_mm.page_table_lock);

```

得到将 address 地址所对应的全局目录项 pdg 地址，存放在临时变量 dir 中。然后锁定全局页表。end 初始化为区域末尾地址。

```

do {
    pmd_t *pmd;

    pmd = pmd_alloc(&init_mm, dir, address);
    ret = -ENOMEM;
    if (!pmd)
        break;

    ret = -ENOMEM;
    if (alloc_area_pmd(pmd, address, end - address, gfp_mask, prot))
        break;

    address = (address + PGDIR_SIZE) & PGDIR_MASK;
    dir++;

    ret = 0;
} while (address && (address < end));

```

调用 pmd_alloc 分配与 address 相对应的中间目录项 pmd（事实上，由于 pmd 不起作用），然后调用 alloc_area_pmd 分配物理内存，建立页表项。然后更新地址 address，继续进行下一个全局页目录以及相应页表项的建立。直到所有的地址相对应的页表项都建立。

```

spin_unlock(&init_mm.page_table_lock);
flush_cache_all();
return ret;

```

然后释放页表锁，使 TLB 硬件高速缓存失效。返回。

alloc_area_pmd

```

address &= ~PGDIR_MASK;
end = address + size;
if (end > PGDIR_SIZE)
    end = PGDIR_SIZE;

```

确保 address~address+size 地址范围落在一个 PGD 所指定的范围。

```

do {
    pte_t * pte = pte_alloc(&init_mm, pmd, address);
    if (!pte)
        return -ENOMEM;
    if (alloc_area_pte(pte, address, end - address, gfp_mask, prot))
        return -ENOMEM;
    address = (address + PMD_SIZE) & PMD_MASK;
    pmd++;
} while (address < end);
return 0;

```

为指定的 pmd 分配 pte，并且调用 alloc_area_pte 为该 pte 分配物理页框。如果过程中出错则直接返回。直到所有必需的虚地址空间建立好 pte 和得到物理页框。

alloc_area_pte

```

address &= ~PMD_MASK;
end = address + size;
if (end > PMD_SIZE)
    end = PMD_SIZE;

```

确保 address~address+size 地址范围落在一个 pmd 所指定的范围。

```

do {
    struct page * page;
    spin_unlock(&init_mm.page_table_lock);
    page = alloc_page(gfp_mask);
    spin_lock(&init_mm.page_table_lock);
    if (!pte_none(*pte))
        printk(KERN_ERR "alloc_area_pte: page already exists\n");
    if (!page)
        return -ENOMEM;
    set_pte(pte, mk_pte(page, prot));
}

```

```

        address += PAGE_SIZE;
        pte++;
    } while (address < end);
    return 0;

```

建立地址所对应的页表项 pte，并且调用 alloc_page 为每个 pte 分配相应的物理页框。

3.3 释放接口 vfree

```

if (!addr)
    return;
if ((PAGE_SIZE-1) & (unsigned long) addr) {
    printk(KERN_ERR "Trying to vfree() bad address (%p)\n", addr);
    return;
}
write_lock(&vmlist_lock);

```

检查地址的合法性。如果成功，锁定 vmlist 链表。

```

for (p = &vmlist ; (tmp = *p) ; p = &tmp->next) {
    if (tmp->addr == addr) {
        *p = tmp->next;
        vmfree_area_pages(VMALLOC_VMADDR(tmp->addr), tmp->size);
        write_unlock(&vmlist_lock);
        kfree(tmp);
        return;
    }
}

```

依据地址寻找 vm_struct 描述符。如找到，调用 vmfree_area_pages 释放该区域所占用的 pte 和物理页框。其后，调用 kfree 释放 vm_struct 描述符。

vmfree_area_pages

```

pgd_t * dir;
unsigned long end = address + size;

dir = pgd_offset_k(address);
flush_cache_all();

```

初始化 end，指向区域末尾。获得起始地址所对应的全局目录项。调用 flush_cache_all 使整个 cpu cache 失效。实际上 flush_cache_all 在 x86 平台上定义为空。

关于对于 tlb cache/cpu cache 的函数，以及这些函数应该如何使用，参考内核源码中文档 documents/cachetlb.txt。

```

do {
    free_area_pmd(dir, address, end - address);
} while (1);

```

```

        address = (address + PGDIR_SIZE) & PGDIR_MASK;
        dir++;
    } while (address && (address < end));
    flush_tlb_all();

```

遍历 address~end 地址空间的所有 pmd 项，并将 pmd 项与相应的物理页框都释放。然后 flush_tlb_all 使 tlb cache 失效。

vmfree_area_pages 函数结束。

free_area_pmd

```

    if (pgd_none(*dir))
        return;
    if (pgd_bad(*dir)) {
        pgd_ERROR(*dir);
        pgd_clear(dir);
        return;
    }

```

pgd 合法性检查。有可能不存在的原因是上一次 vmalloc 在中途失败了。仔细参考 vmalloc 过程就可以发现。

```

    pmd = pmd_offset(dir, address);
    address &= ~PGDIR_MASK;
    end = address + size;
    if (end > PGDIR_SIZE)
        end = PGDIR_SIZE;
    do {
        free_area_pte(pmd, address, end - address);
        address = (address + PMD_SIZE) & PMD_MASK;
        pmd++;
    } while (address < end);

```

设置 pmd 指向第一个 pmd 项，确保 end 是按照 PGDIR_SIZE 对齐。然后遍历所有的 pmd 项，针对每个 pmd 项，调用 free_area_pte 释放该 pmd 项中所包含的 pte 和物理页。

free_area_pte

```

    if (pmd_none(*pmd))
        return;
    if (pmd_bad(*pmd)) {
        pmd_ERROR(*pmd);
        pmd_clear(pmd);
        return;
    }

```

pmd 合法性检查。


```

pte = pte_offset(pmd, address);
address &= ~PMD_MASK;
end = address + size;
if (end > PMD_SIZE)
    end = PMD_SIZE;

```

pte 指向 pmd 中的起始地址 pte 项。然后确保 end 按照 PMD_SIZE 对齐。

```

do {
    pte_t page;
    page = ptep_get_and_clear(pte);
    address += PAGE_SIZE;
    pte++;
    if (pte_none(page))
        continue;
    if (pte_present(page)) {
        struct page *ptpage = pte_page(page);
        if (VALID_PAGE(ptpage) && (!PageReserved(ptpage)))
            __free_page(ptpage);
        continue;
    }
    printk(KERN_CRIT "Whee.. Swapped out page in kernel page table\n");
} while (address < end);

```

调用 ptep_get_and_clear 获得 pte 项内容（存放 page 中），并将页表中的 pte 项清除。然后检查 page 是否存在，若是，调用 __free_page 回收 pte 所指向的物理页框。遍历所有的 pte，直到遍历完成。

3.4 读写接口 vread/vwrite

事实上，在内核源代码中，极少使用 vread/vwrite 接口的。这两个接口只是提供给字符设备使用的。字符设备通过这两个接口，就像操作物理内存一样，操作 vmalloc 所获得内存区域。在这里不详细描述 vread/vwrite 过程。

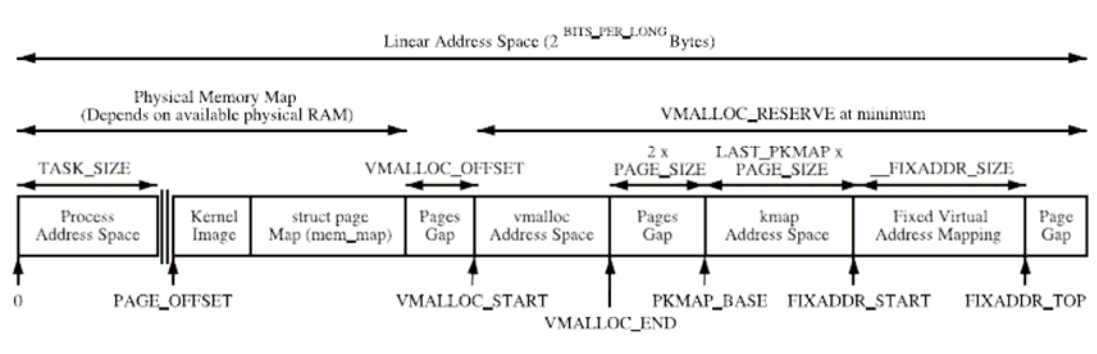
4 High Memory Mapping - kmapper

在非连续内存 vmalloc 分配器这一章中，我们曾经做过假设：系统中的物理内存大小超过 1G，小于 4G。这时候，我们可以看出一个问题：在内核线性空间中直接映射的物理只局限于 896M，那么我们该如何使用大于 896M 的物理空间？换句话说，在内核空间，所有的内存都被映射到 PAGE_OFFSET~4G 的空间，如果物理内存大于 1G 的话，那么高于 1G 的物理内存怎么被内核用？这个问题如何解决就是这一章所讨论的内容。

Linux 提供了一种方法来解决，由于函数名关系，暂称为 kmapper。-_-!

在 vmalloc 分配器所占用的内核线性空间之后，还有约 32M 的线性地址空间。kmapper 所利

用的线性空间范围由 `PKMAP_BASE` 和 `PKMAP_BASE+LAST_PKMAP*PAGE_SIZE` 指定。



```
/*
 * Right now we initialize only a single pte table. It can be extended
 * easily, subsequent pte tables have to be allocated in one physical
 * chunk of RAM.
 */
#define PKMAP_BASE (0xfe000000UL)
#define LAST_PKMAP 1024
```

`kmapper` 的想法很简单：使用者在使用 `alloc_pages` 接口从 `ZONE HIGHMEM` 获得若干个物理页框之后，由于这些物理页框没有被内核固定映射到内核线性空间，所以没有相对应的虚地址（换句话说，这些物理页框没有相对应的页表项）。使用者可以利用 `kmapper` 的接口 `kmap` 在 `Low Memory` 的某个地方建立起这若干高端物理页框的页表项。由于这种映射是临时性的，因此要尽早调用释放接口 `kunmap` 释放这种映射。

将高端内存映射到内核线性空间，`kmapper` 有 2 种映射方式：

- permanent mapping
- temporary mapping（也称为 atomic mapping）

这两种映射分别对应接口 `kmap/kunmap`, `kmap_atomic/kunmap_atomic`。

这两组接口所使用的线性地址空间是不同的。前者使用的是 `PKMAP_BASE` 所指定的线性空间，而后者使用的是 `fixed mapping` 中的部分线性空间。

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

```
enum km_type {
    KM_BOUNCE_READ,
```

```

    KM_SKB_DATA,
    KM_SKB_DATA_SOFTIRQ,
    KM_USER0,
    KM_USER1,
    KM_TYPE_NR
};

```

从上述的定义可以看出，`kmap_atomic` 所使用的线性空间大小为 `KM_TYPE_NR*NR_CPUS*PAGE_SIZE`。起始线性地址为 `FIXADDR_TOP+FIX_KMAP_BEGIN*PAGE_SIZE`。

`kmap` 所使用的线性地址中，为了记录这些线性地址的使用情况，内核定义了 1024 大小的 `int` 数组

```

/*
 * Virtual_count is not a pure "count".
 * 0 means that it is not mapped, and has not been mapped
 *   since a TLB flush - it is usable.
 * 1 means that there are no users, but it has been mapped
 *   since the last TLB flush - so we can't use it.
 * n means that there are (n-1) current users of it.
 */
static int pkmap_count[LAST_PKMAP];

```

每个整数对应一个线性的映射情况：

- 0 该线性地址未被映射到高端物理页
- 1 该线性地址已经被映射到高端物理页，但未被使用
- $n > 1$ 该线性地址已经被映射到高端物理页，同时有 $n-1$ 个使用者。

4.1 初始化

`kmapper` 的初始化工作先在 `pagetable_init` 中展开。

```

/*
 * Fixed mappings, only the page table structure has to be
 * created - mappings will be set by set_fixmap():
 */
vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
fixrange_init(vaddr, 0, pgd_base);

#if CONFIG_HIGHMEM
/*
 * Permanent kmaps:
 */
vaddr = PKMAP_BASE;
fixrange_init(vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base);

pgd = swapper_pg_dir + __pgd_offset(vaddr);

```

```

    pmd = pmd_offset(pgd, vaddr);
    pte = pte_offset(pmd, vaddr);
    pkmap_page_table = pte;
#endif

```

第一个 `fixrange_init` 建立了 temporary mapping 部分的页表（可仔细看 `__end_of_fixed_addresses` 定义）。第二个 `fixrange_init` 建立了 permanent mapping 线性空间使用的页表项。从基地址 `PKMAP_BASE` 开始，约 $4K \times 1024 = 4M$ 大小的线性空间（页表项大小刚好一个物理页框）。调用 `fixrange_init` 为这段线性空间分配物理内存给页表。然后将 `pkmap_page_table` 指向这个页表的起始地址。注意这时候这些页表项是没有相对应的物理页。

```

#ifdef CONFIG_HIGHMEM
    kmap_init();
#endif

```

函数 `pagetable_init` 返回之后，紧接着调用 `kmap_init` 继续初始化 `kmapper`。

kmap_init

```

#ifdef CONFIG_HIGHMEM
pte_t *kmap_pte;
pgprot_t kmap_prot;

#define kmap_get_fixmap_pte(vaddr) \
    pte_offset(pmd_offset(pgd_offset_k(vaddr), (vaddr)), (vaddr))

void __init kmap_init(void)
{
    unsigned long kmap_vstart;

    /* cache the first kmap pte */
    kmap_vstart = __fix_to_virt(FIX_KMAP_BEGIN);
    kmap_pte = kmap_get_fixmap_pte(kmap_vstart);

    kmap_prot = PAGE_KERNEL;
}
#endif /* CONFIG_HIGHMEM */

```

`kmap_init` 将 temporary mapping 的页表首地址存放在全局变量 `kmap_pte` 中，将页表项的标志位存放在 `kmap_prot` 变量中。换句话说，`kmap_init` 只是做了 `kmap_atomic` 接口的“缓存”工作。

fixrange_init

```

vaddr = start;
i = __pgd_offset(vaddr);
j = __pmd_offset(vaddr);
pgd = pgd_base + i;

```

根据起始地址 start 获得 pgd 和 pmd 的偏移量。

(下面的代码中省略了 CONFIG_X86_PAE 部分)

```
for ( ; (i < PTRS_PER_PGD) && (vaddr != end); pgd++, i++) {
    pmd = (pmd_t *)pgd;
    for ( ; (j < PTRS_PER_PMD) && (vaddr != end); pmd++, j++) {
        if (pmd_none(*pmd)) {
            pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
            set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte)));
            if (pte != pte_offset(pmd, 0))
                BUG();
        }
        vaddr += PMD_SIZE;
    }
    j = 0;
}
```

遍历 start~end 地址空间所对应的所有 pgd 和 pmd，如果页表项 pte 不存在，利用 bootmem 分配器接口 alloc_bootmem_low_pages 获得物理页框，在这个物理页框中建立页表。这时候页表的内容被初始化为 0。

因为 kmap 接口使用的线性空间大小为 4M，因此这里在实际过程只分配了一个物理页，存储了全部 1024 个页表项内容。

4.2 kmap/kunmap 接口

kmap 是建立映射关系，kunmap 是释放映射关系。这种映射关系除非用 kunmap 撤销，否则这种映射关系将会被一直保留。

4.2.1 kmap

```
static inline void *kmap(struct page *page)
{
    if (in_interrupt())
        BUG();
    if (page < highmem_start_page)
        return page_address(page);
    return kmap_high(page);
}
```

首先检查是否在中断处理过程，kmap 不允许在中断中使用，因为 kmap 可能会等待，并放弃 cpu，进行调度。接着检查 page 是否落在 low memory 范围，如果是直接返回地址。否则调用 kmap_high。

kmap_high

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

```

```

unsigned long vaddr;

/*
 * For highmem pages, we can't trust "virtual" until
 * after we have the lock.
 *
 * We cannot call this from interrupts, as it may block
 */
spin_lock(&kmap_lock);
vaddr = (unsigned long) page->virtual;

```

锁定 **kmap**，得到该页描述符的线性地址。如果该页是刚分配得到的高端物理内存，其线性地址还是为 0。参考 **buddy** 分配器初始化 **free_area_init** 函数。

```

if (!vaddr)
    vaddr = map_new_virtual(page);

```

如果该页的线性地址为 0，调用 **map_new_virtual** 分配一个线性地址。

```

pkmap_count[PKMAP_NR(vaddr)]++;
if (pkmap_count[PKMAP_NR(vaddr)] < 2)
    BUG();
spin_unlock(&kmap_lock);
return (void*) vaddr;
}

```

如果分配成功返回，**pkmap_count** 数组相对应项加 1，（这时候值通常为 2），确认该线性地址引用数不小于 2。释放 **kmap** 锁，返回其线性地址。

map_new_virtual

```

static inline unsigned long map_new_virtual(struct page *page)
{
    unsigned long vaddr;
    int count;

start:
    count = LAST_PKMAP;
    /* Find an empty entry */
    for (;;) {

```

设置 **kmap** 所使用的页表项大小，准备从 **kmap** 所使用的线性空间中找到一个可用的页表项。

```

        last_pkmap_nr = (last_pkmap_nr + 1) & LAST_PKMAP_MASK;
        if (!last_pkmap_nr) {
            flush_all_zero_pkmaps();
            count = LAST_PKMAP;

```

```
}
```

`last_pkmap_nr` 是上一次执行 `map_new_virtual` 过程寻找到可用页表项的位置。从 `last_pkmap_nr` 开始到最后一个页表项，如果没有找不到可用页表项，调用 `flush_all_zero_pkmaps` 刷新，以回收可用页表项。然后设定 `count`，准备重新对所有页表项进行扫描。

```
if (!pkmap_count[last_pkmap_nr])
    break; /* Found a usable entry */
if (--count)
    continue;
```

如果找到一个可用页表项，调出循环。否则如果 `count` 计数器非零，继续执行循环。

```
/*
 * Sleep for somebody else to unmap their entries
 */
{
    DECLARE_WAITQUEUE(wait, current);

    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(&pkmap_map_wait, &wait);
    spin_unlock(&kmap_lock);
    schedule();
    remove_wait_queue(&pkmap_map_wait, &wait);
    spin_lock(&kmap_lock);

    /* Somebody else might have mapped it while we slept */
    if (page->virtual)
        return (unsigned long) page->virtual;

    /* Re-start */
    goto start;
}
}
```

如果遍历了所有页表项还没有找到可用页表项，那么执行此处代码。此时，进程将自己挂入 `pkmap_map_wait` 等待队列，调用 `schedule` 通知内核进行调度，等待其他进程释放某个页表项的映射。

当调度完成回来，此进程重新获得时间片之后，删除等待队列，然后判断该页的线性地址，因为在等待的过程中，此物理页可能被其他进程映射好。否则，跳转到 `start` 继续执行过程。

```
vaddr = PKMAP_ADDR(last_pkmap_nr);
set_pte(&(pkmap_page_table[last_pkmap_nr]), mk_pte(page, kmap_prot));

pkmap_count[last_pkmap_nr] = 1;
```

```

    page->virtual = (void *) vaddr;

    return vaddr;
}

```

如果发现一个可用页表项，执行此处代码。根据 `last_pkmap_nr`，计算其线性地址，设置相应的页表项。将这个新线性地址赋值给这个物理页。然后返回线性地址。

flush_all_zero_pkmaps

```

static void flush_all_zero_pkmaps(void)
{
    int i;

    flush_cache_all();
}

```

由于涉及页表操作，所以要先调用 `flush_cache_all` 使 CPU 缓存失效。

```

for (i = 0; i < LAST_PKMAP; i++) {
    struct page *page;

    /*
     * zero means we don't have anything to do,
     * >1 means that it is still in use. Only
     * a count of 1 means that it is free but
     * needs to be unmapped
     */
    if (pkmap_count[i] != 1)
        continue;
}

```

遍历 `pkmap_count` 数组，寻找引用数为 1 的对应项。

```

pkmap_count[i] = 0;

/* sanity check */
if (pte_none(pkmap_page_table[i]))
    BUG();

/*
 * Don't need an atomic fetch-and-clear op here;
 * no-one has the page mapped, and cannot get at
 * its virtual address (and hence PTE) without first
 * getting the kmap_lock (which is held here).
 * So no dangers, even with speculative execution.
 */
page = pte_page(pkmap_page_table[i]);
pte_clear(&pkmap_page_table[i]);

```



```

    page->virtual = NULL;
}

```

如果寻找到，那么先将引用数置为 0，得到被映射到物理地址描述符，清除 `pkmap_page_table` 中对应的页表项内容，然后将该物理页描述符中的 `virtual` 成员重新置为 0。

```

    flush_tlb_all();
}

```

当遍历完成，使 TLB 缓存失效。

4.2.2 kunmap

`kunmap` 是高端内存映射的释放接口。

```

static inline void kunmap(struct page *page)
{
    if (in_interrupt())
        BUG();
    if (page < highmem_start_page)
        return;
    kunmap_high(page);
}

```

同样，因为 `kunmap` 不允许在中断过程中调用。如果该物理页不再高端内存，直接返回，不做任何处理。否则调用 `kunmap_high` 撤销映射。

kunmap_high

```

void kunmap_high(struct page *page)
{
    unsigned long vaddr;
    unsigned long nr;
    int need_wakeup;

    spin_lock(&kmap_lock);
    vaddr = (unsigned long) page->virtual;
    if (!vaddr)
        BUG();
    nr = PKMAP_NR(vaddr);

```

先锁定 `kmap`，判断该物理页所映射的线性地址是否为 0，如果是，BUG 处理。否则根据该线性地址得到 `pkmap_count` 数组相对应的下标。

```

/*
 * A count must never go down to zero
 * without a TLB flush!
 */
need_wakeup = 0;

```

```

switch (--pkmap_count[nr]) {
case 0:
    BUG();
case 1:
    /*
     * Avoid an unnecessary wake_up() function call.
     * The common case is pkmap_count[] == 1, but
     * no waiters.
     * The tasks queued in the wait-queue are guarded
     * by both the lock in the wait-queue-head and by
     * the kmap_lock. As the kmap_lock is held here,
     * no need for the wait-queue-head's lock. Simply
     * test if the queue is empty.
     */
    need_wakeup = waitqueue_active(&pkmap_map_wait);
}
spin_unlock(&kmap_lock);

```

先将引用数减 1，根据引用数判断：

- 0 不可能出现的情况，BUG 处理
- 1 表示可用，映射已建立。判断有没有其他使用者正在等待线性地址被释放。

然后释放 kmap 锁。

```

/* do wake-up, if needed, race-free outside of the spin lock */
if (need_wakeup)
    wake_up(&pkmap_map_wait);
}

```

如果需要唤醒，那么唤醒其等待进程。

注：当引用数大于 1 的时候，表示该线性地址映射还有使用者，所以不能唤醒等待者。

4.3 kmap_atomic/kunmap_atomic 接口

虽说提供了这两个接口，但是不鼓励使用他们。尽量使用更通用的 kmap/kunmap。不过，这两个接口也有一个优点，就是他们能在中断处理过程中使用。

4.3.1 kmap_atomic

```

static inline void *kmap_atomic(struct page *page, enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;

    if (page < highmem_start_page)
        return page_address(page);
}

```

和 `kmap` 相比, `kmap_atomic` 接口参数多了一个类型参数, 其值必须为 `enum km_type` 枚举类型值之一。

首先判断, 该物理页是否高端内存, 如果不是, 直接返回其固定线性地址。

```
idx = type + KM_TYPE_NR*smp_processor_id();
vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);

set_pte(kmap_pte+idx, mk_pte(page, kmap_prot));
__flush_tlb_one(vaddr);

return (void*) vaddr;
```

根据参数 `type` 和 `cpu` 序号计算出其对应的线性地址, 设置页表, 将该线性地址映射到 `page` 制定的高端页框。然后使该线性地址的 TLB 缓存失效。返回线性地址。

从处理过程中我们可以看出, `kmap_atomic` 并不考虑这个线性地址的使用情况, 它不关心该线性地址是否已经被映射, 如果调用此过程, 那么就直接覆盖。⊗

4.3.2 kunmap_atomic

事实上出去 `DEBUG` 部分的代码, 该函数是空的。

```
static inline void kunmap_atomic(void *kvaddr, enum km_type type)
{
#ifdef HIGHMEM_DEBUG
    ...
#endif
}
```

5 参考文档

- [1] IA-32 Intel Architecture Software Developer's Manual Volume 3, Intel Corp.
- [2] linux-mm.org website
- [3] Understanding the linux virtual memory manager, Mel Gorman
- [4] Operating System Concepts 6ed, Abraham Silberschatz
- [5] Understanding Linux Kernel second edition, Daniel P. Bovet
- [6] Memory Management in Linux, Abhishek Nayani
- [7] Too little, Too small, Lecture by Rik van Riel
- [8] Linux的NUMA技术, IBM Developer Works Linux
- [9] The Slab Allocator: An Object-Caching Kernel Memory Allocator, Jeff Bonwick