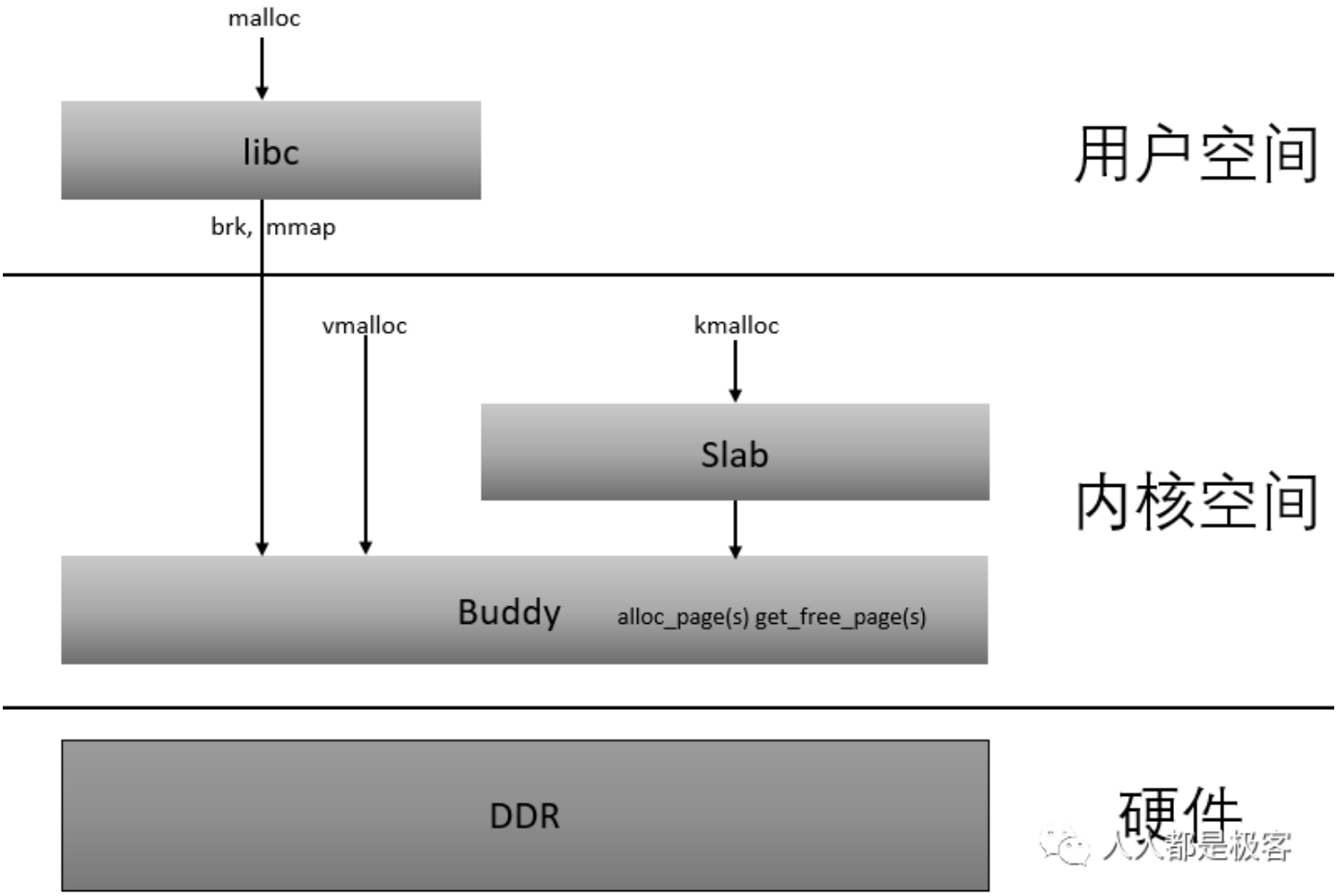
内存管理



一、物理内存管理

1.1 物理内存的组织形式

物理内存是以页为单位管理的，内核提供了page结构体与一页物理内存对应，而一页物理内存的作用或者使用情况由它所在的区域（zone）决定。

内核提供的与物理内存相关的变量(全局变量)，它们标示着系统的内存信息，如表8-1所示。

表8-1 物理内存变量表

名　称	描　述
max_pfn	pfn 已经介绍过了，页框号。 max_pfn 表示最大的有效的内存页框号
max_low_pfn	最大的有效的 Low Memory（见 9.2）页框号
highmem_pages	max_low_pfn 与 max_pfn 之间的空白，计算所得，或由用户设定 highmem_pages + max_low_pfn = max_pfn

节点（node）与NUMA（Non Uniform Memory Access Architecture，非统一内存访问）架构有密切联系，在传统的SMP架构中，所有CPU共享系统总线，限制了内存的访问能力，NUMA的引入一定程度上解决了该瓶颈。NUMA的节点通常由一组CPU和本地内存组成，系统中会存在多个节点，每一个节点都有自己的zone。它主要应用于服务器架构中。

内核提供了pglist_data结构体与节点对应，主要的字段如表8-2所示。

表8-2 pglist_data字段表

字　段	类　型	描　述
node_zones	zone[MAX_NR_ZONES]	节点包含的 zone 的集合，数组
node_mem_map	page *	节点包含的物理内存页对应的 page 结构体集合
node_start_pfn	unsigned long	节点包含的物理内存的起始页页框号
node_present_pages	unsigned long	节点包含的物理内存页数
node_spanned_pages	unsigned long	节点包含的物理内存区间，包含中间的 hole

```
/* 内存结点描述符，所有的结点描述符保存在 struct pglist_data *node_data[MAX_NUMNODES]
typedef struct pglist_data {
    /* 管理区描述符的数组 */
    struct zone node_zones[MAX_NR_ZONES];
    /* 页分配器使用的zonelist数据结构的数组，将所有结点的管理区按一定的关联链接成一个链表，
    struct zonelist node_zonelists[MAX_ZONELISTS];
    /* 结点中管理区的个数 */
    int nr_zones;
#ifdef CONFIG_FLAT_NODE_MEM_MAP    /* means !SPARSEMEM */
    /* 结点中页描述符的数组，包含了此结点中所有页框描述符，实际分配是是一个指针数组 */
    struct page *node_mem_map;
#ifdef CONFIG_MEMCG
    /* 用于资源限制机制 */
    struct page_cgroup *node_page_cgroup;
#endif
#endif
#ifndef CONFIG_NO_BOOTMEM
    /* 用在内核初始化阶段 */
    struct bootmem_data *bdata;
#endif
#ifdef CONFIG_MEMORY_HOTPLUG
    /* 自旋锁 */
    spinlock_t node_size_lock;
#endif
    /* 结点中第一个页框的下标，在numa系统中，页框会有两个序号，所有页框的一个序号，还有就是
     * 比如结点2中的页框1，它在结点2中的序号是1，但是在所有页框中的序号是1001，这个变量就是
     */
    unsigned long node_start_pfn;
```

```
    /* 内存结点的大小，不包括洞(以页框为单位) */
    unsigned long node_present_pages;
    /* 结点的大小，包括洞(以页框为单位) */
    unsigned long node_spanned_pages;

    /* 结点标识符 */
    int node_id;
    /* kswaped页换出守护进程使用的等待队列 */
    wait_queue_head_t kswapd_wait;
    wait_queue_head_t pfmemalloc_wait;
    /* 指针指向kswapd内核线程的进程描述符 */
    struct task_struct *kswapd;    /* Protected by
                       mem_hotplug_begin/end() */
    /* kswapd将要创建的空闲块大小取对数的值 */
    int kswapd_max_order;
    enum zone_type classzone_idx;
#ifdef CONFIG_NUMA_BALANCING
    /* 以下用于NUMA的负载均衡 */
    /* Lock serializing the migrate rate limiting window */
    spinlock_t numabalancing_migrate_lock;

    /* Rate limiting time interval */
    unsigned long numabalancing_migrate_next_window;

    /* Number of pages migrated during the rate limiting time interval */
    unsigned long numabalancing_migrate_nr_pages;
#endif
} pg_data_t;
```

为了更好地阐述各个字段，我们先看一下内核看到的内存是什么样子的。

内核得到的内存不一定是连续的，也不一定全部由内核使用。系统初始化阶段，内核会使用某些机制（如e820）获得系统当前的内存配置情况。比如，我的32位2G内存系统上的配置如下（e820打印的log，也可以通过/sys/firmware/memmap查看，原log中每个数字都是“%#018Lx”格式的，已将数字开头多余的0去掉）。

```
e820: BIOS-provided physical RAM map:
BIOS-e820: [mem 0x0-0x9dbff] usable
BIOS-e820: [mem 0x9dc00-0x9ffff] reserved
BIOS-e820: [mem 0xdc000-0xdffff] reserved
BIOS-e820: [mem 0xe4000-0xfffff] reserved
BIOS-e820: [mem 0x100000-0x7f5bffff] usable
BIOS-e820: [mem 0x7f5c0000-0x7f5ccfff] ACPI data
BIOS-e820: [mem 0x7f5cd000-0x7f5d0fff] ACPI NVS
BIOS-e820: [mem 0x7f5d1000-0x7fffffff] reserved
BIOS-e820: [mem 0xe0000000-0xefffffff] reserved
BIOS-e820: [mem 0xfec00000-0xfec0ffff] reserved
BIOS-e820: [mem 0xfee00000-0xfee00fff] reserved
BIOS-e820: [mem 0xff000000-0xffffffff] reserved
```

e820描述的内存段共有6种类型，但只有E820_RAM和E820_RESERVED_KERN两种属于log中的usable，其他的几种并不归内核管理。所以只有0x0-0x9dbff和0x100000-0x7f5bffff两段内存是内核管辖范围的，其他各段都有特殊的用途。

在线性空间布局的讨论中，我们提到过变量max_pfn，它表示系统可使用内存的最大的页框号，那么在我的系统里max_pfn就等于0x7f5c0。另外，这两段内存中间是有洞（hole）的，所以实际可用的页数是小于0x7f5c0的。

node_spanned_pages字段的值也等于0x7f5c0，node_present_pages则等于0x7f5c0减去两个段之间的hole所占的页数，node_start_pfn等于0。一个page对象对应一页物理内存，那么节点包含的物理内存对应的page对象存储在哪里？

从BIOS出来的信息显示，只有0x0-0x9dbff和0x100000-0x7f5bffff两段内存是可用的。内核提供了memblock机制（见8.3小节）来管理它们，每一段内存都有一个memblock_region对象与之对应。node的pglist_data结构体初始化的时候，calculate_node_totalpages函数会根据两段内存和各zone的边界计算得到node_spanned_pages和node_present_pages。

之后在alloc_node_mem_map函数中，根据node_start_pfn和node_spanned_pages的值计算出最终需要管理的内存的页数（主要是对齐调整）。页数就是该节点page对象的个数，据此申请内存，将内存地址赋值给node_mem_map字段，page对象就存在这里，代码如下

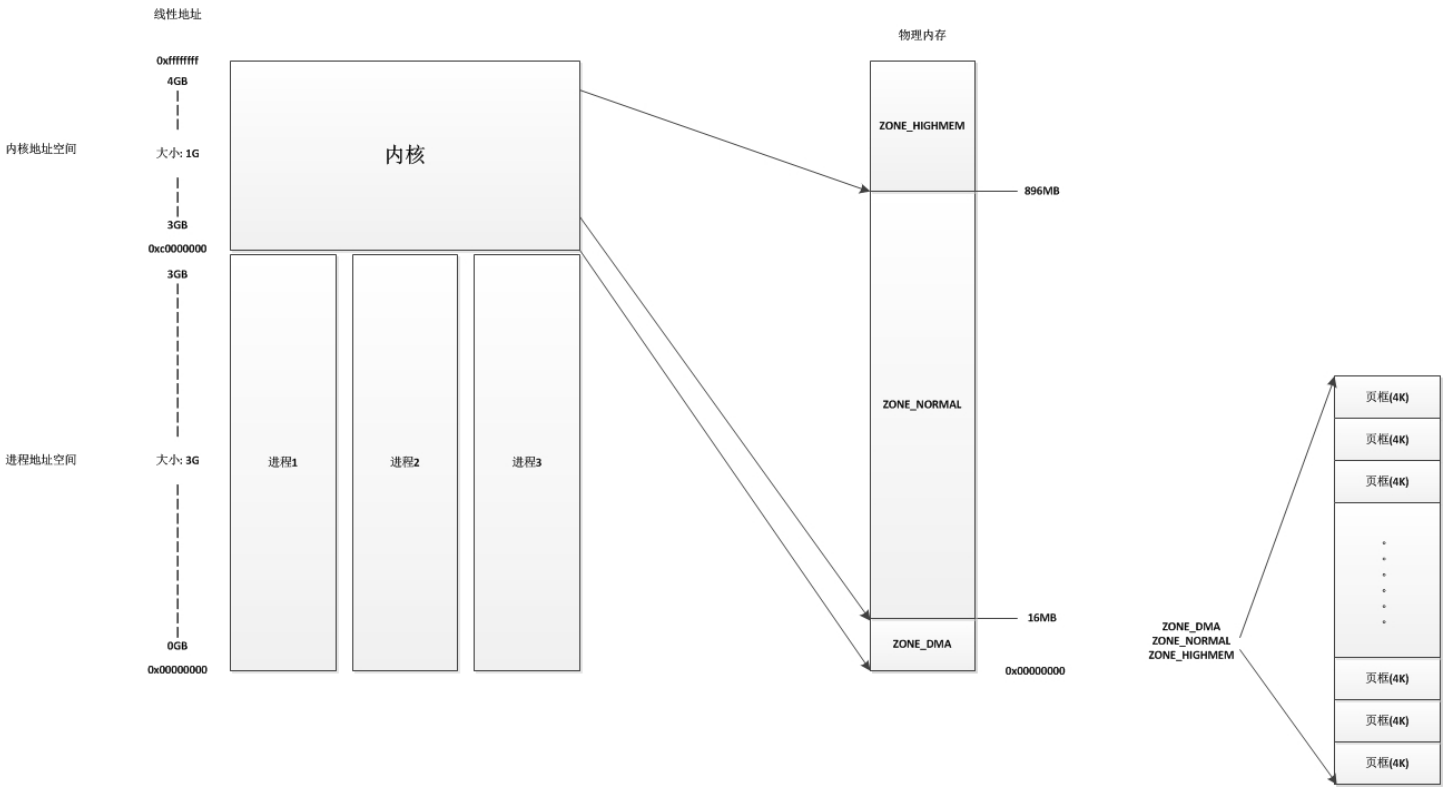
注意，使用的是node_spanned_pages，而不是node_present_pages。node的page对象本质是一个数组，这个数组对应着Flat Memory模式的整个平面物理内存，所以内存的hole也需要有page结构体与之对应。

node_zones字段是一个数组，表示节点的zone。

节点的内存根据使用情况划分为多个区域，每个区域都对应一个zone对象。zone最多分为ZONE_DMA、ZONE_DMA32、ZONE_NORMAL、ZONE_HIGHMEM、ZONE_MOVABLE和ZONE_DEVICE六种。ZONE_DMA32用在64位系统上，ZONE_MOVABLE是为了减少内存碎片化引入的虚拟zone，我们讨论DMA、NORMAL和HIGHMEM三个zone。

ZONE_DMA的内存区间为低16M，ZONE_NORMAL的内存区间为16M到max_low_pfn页框的地址，ZONE_HIGHMEM的内存区间为max_low_pfn到max_pfn。前面已经说过，max_pfn等于最大的页框号，不考虑预留High Memory的情况下，如果它大于MAXMEM_PFN（MAXMEM对应的页框，MAXMEM见下文），max_low_pfn就等于MAXMEM_PFN。如果它小于MAXMEM_PFN，max_low_pfn则等于max_pfn，这样就没有ZONE_HIGHMEM了。

page是zone的下一级单位，属于zone。但所有的page对象，都以数组的形式统一放在了pglist_data的node_mem_map字段指向的内存中。



对于ZONE_DMA和ZONE_NORMAL这两个管理区，内核地址都是进行直接映射，只有ZONE_HIGHMEM管理区系统在默认情况下是不进行直接映射的，只有在需要使用的时候进行映射(临时映射或者永久映射)。

1.2 启动程序

内核收到的内存信息是由BIOS给出的，主要包括各内存段的区间和使用情况，这些区间是Linux可见的所有内存，比如上文中打印的e820信息。

只有用途为usable的内存区间才属于内核直接管理（MMIO除外），但并不是所有的usable部分内存都直接归内核管理。除了memblock和slab系统外，还有一个更高的级别是启动程序，比如grub。grub可以通过参数（比如“mem=2048M”）来限制内核可以管理的内存上限，也就是说高于2048M的部分不归内核（memblock和buddy）管理。同样，memblock也可以扣留一部分内存，剩下的部分才轮到buddy系统管理。

你也许会疑问grub扣下的内存是给谁用的，当然是给修改grub的人用的，这部分内存可以在程序中映射、独享。

内核在不同阶段采用的内存管理方式也不同，主要分为memblock和buddy系统两种。

1.3 memblock分配器

block这个词意味着内存被分为一块块的，内核以memblock_region结构体表示一块，它的base和size字段分别表示块的起始地址和大小。块以数组的形式进行管理，该数组由memblock_type结构体的regions字段表示。内核共有两个memblock_type对象（也可以理解为有两个由内存块组成的数组），一个表示可用的内存，一个表示被预留的内存（预留就是公司中层占有的福利，不会进入下一层），分别由memblock结构体的memory和reserved字段表示。

扣除了grub预留的内存后，所有用途为usable的内存块都会默认进入可用内存块数组（下面简称memory数组），有一些模块也会选择预留一部分内存供其使用，这些块会进入被预留的内存块数组（下面简称reserve数组），内核为此提供了几个函数管理它们，如表8-3所示。

表8-3 memblock函数表

函　数	描　述
memblock_add	将内存块加入 memory 数组
memblock_remove	将内存块从 memory 数组删除，很少使用
memblock_reserve	预留内存块，加入 reserve 数组
memblock_free	释放内存块，从 reserve 数组删除

memblock是内存管理的第一个阶段，也是初级阶段，buddy系统会接替它的工作继续内存管理。它与buddy系统交接是在mem_init函数中完成的，标志为after_bootmem变量置为1。mem_init函数会调用set_highmem_pages_init和memblock_free_all分别释放highmem和lowmem到buddy系统。

块被加入reserve数组的时候并未从memory数组删除，只有在memory数组中，且不在reserve数组中的块才会进入buddy系统。与grub一样，被预留的内存块也是给预留内存块的模块使用的，模块自行负责内存的映射。所以，buddy系统管理的内存是经过grub和memblock“克扣”后剩下的部分。

1.4 伙伴系统

内核的buddy（伙伴）系统也是以块来管理内存的，不过块的大小并不是任意的，块以页为单位，仅有2 (0),2 (1)…2 (10)，共11（MAX_ORDER）种（1页，2页……1024页，4K，8K...4M）大小，在内核中以阶（order）来表示块的大小，所以阶有0, 1…10共11种。内存申请优先从小块开始是自然的，释放一个块的时候，如果它的伙伴也处于空闲状态，那么就将二者合并成一个更大的块。合并后的块，如果它的伙伴也空闲，就继续合并，依次类推。另外，块包含的内存必须是连续的，不能有洞（hole）。

那么两个块满足哪些条件才有成为伙伴的可能?

首先，两个块必须相邻，且处于同一个zone，这点容易理解，毕竟管理的是物理内存。

其次，由于每个块的大小都是2的整数次幂页，合并后也是如此，所以互为伙伴的两个块的阶必须相等。

最后，两个块必须来自同一个大块。

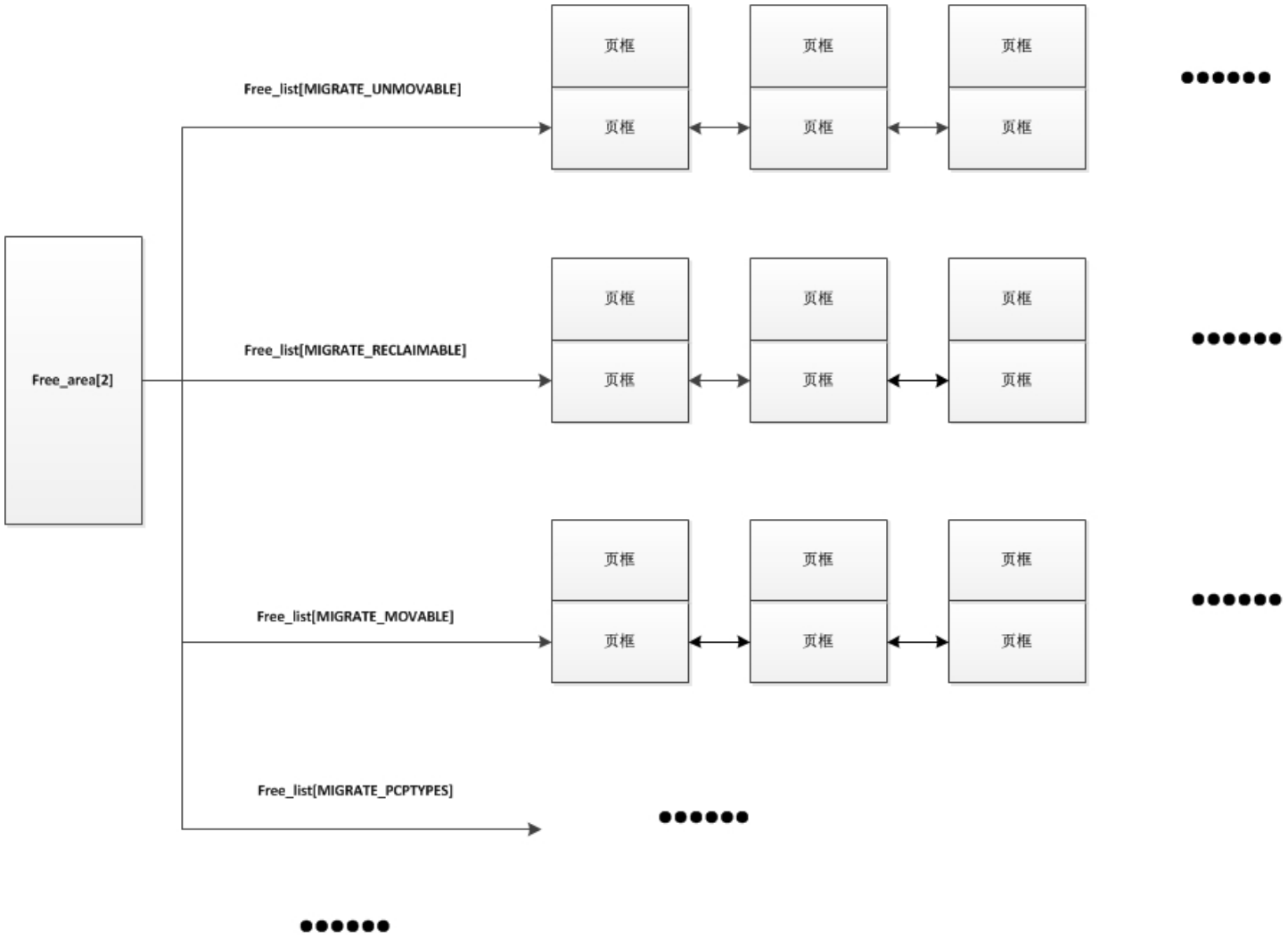
buddy的算法是基于zone的

页（page）的上一级是zone，所以块是由zone来记录的，zone的free_area字段是一个大小等于MAX_ORDER的free_area结构体类型的数组，数组的每一个元素对应伙伴系统当前所有大小为某个阶的块，阶的值等于数组下标。free_area的free_list字段也是一个数组，数组的每一个元素都是同种迁移（MIGRATE）类型的块组成的链表的头，它的nr_free字段表示空闲的块的数量。

```
/* 伙伴系统的一个块，描述1,2,4,8,16,32,64,128,256,512或1024个连续页框的块 */
struct free_area {
    /* 指向这个块中所有空闲小块的第一个页描述符，这些小块会按照MIGRATE_TYPES类型存放在不同
    struct list_head    free_list[MIGRATE_TYPES];
    /* 空闲小块的个数 */
    unsigned long        nr_free;
};
```

```
enum {
    MIGRATE_UNMOVABLE,          /* 不可移动页 */
    MIGRATE_RECLAIMABLE,        /* 可回收页 */
    MIGRATE_MOVABLE,            /* 可移动页 */
    MIGRATE_PCPTYPES,           /* 用来表示每CPU页框高速缓存的数据结构中的链表的可移动类
    MIGRATE_RESERVE = MIGRATE_PCPTYPES,
#ifdef CONFIG_CMA
    MIGRATE_CMA,
#endif
#ifdef CONFIG_MEMORY_ISOLATION
    MIGRATE_ISOLATE,            /* 不能从这个链表分配页框，因为这个链表专门用于NUMA结点移
#endif
    MIGRATE_TYPES
};
```

保存连续2个页框的free_area[2]的结构如下：



这样，zone内同阶同迁移类型的块都在同一个链表上，比如阶等于2、迁移类型为type（比如MIGRATE_MOVABLE）的块，都在zone->free_area[2]. free_list[type]链表上，另外，阶等于2的块的数量为zone->free_area[2]. nr_free个。

需要说明的是，伙伴系统只会维护空闲的块，已经分配出去的块不再属于伙伴系统，只有被释放后才会重新进入伙伴系统。

一个块可以用它的第一个页框和阶表示，即page和order。page的page_type字段的PG_buddy标志清零（不是置位）时表示它代表的块在伙伴系统中，private字段的值则表示块的阶。需要强调的是，只有块的第一个页框的page对象才有这个对应关系，中间页框的标志对伙伴系统而言没有意义。（目前怀疑块就是用page结构体表示，即free_list数组中的指针指的就是page）

内核提供了几个函数可以用来检查或者设置这两个字段，如表8-4所示。

表8-4 page的order函数表

函　数	描　述
PageBuddy	page 表示的块是否在伙伴系统中
page_order	page 表示的块的阶
set_page_order	设置 page 的 page_type 和 private 两个字段，块纳入伙伴系统
rmv_page_order	设置 page 的 page_type 和 private 两个字段，块从伙伴系统中删除。

在从伙伴系统中申请页框时，有可能会遇到一种情况，就是当前需求的连续页框链表上没有可用的空闲页框，这时后，伙伴系统会从下一级获取一个连续长度的页框块，将其拆分放入这级列表；当然在拥有者释放连续页框时伙伴系统也会适当地进行连续页框的合并，并放入下一级中。比如：我需要申请4个页框，但是长度为4个连续页框块链表没有空闲的页框块，伙伴系统会从连续8个页框块的链表获取一个，并将其拆分为两个连续4个页框块，放入连续4个页框块的链表中。释放时道理也一样，会检查释放的这几个页框的之前和之后的物理页框是否空闲，并且能否组成下一级长度的块。

每CPU页框高速缓存

每CPU页框高速缓存也是一个分配器，配合着伙伴系统进行使用，这个分配器是专门用于分配单个页框的，它维护一个单页框的双向链表，为什么需要这个分配器，因为每个CPU都有自己的硬件高速缓存，当对一个页进行读取写入时，首先会把这个页装入硬件高速缓存，而如果进程对这个处于硬件高速缓存的页进行操作后立即释放掉，这个页有可能还保存在硬件高速缓存中，这样我另一个进程需要请求一个页并立即写入数据的话，分配器将这个处于硬件高速缓存中的页分配给它，系统效率会大大增加。

在每CPU页框高速缓存中用一个链表来维护一个单页框的双向链表，每个CPU都有自己的链表(因为每个CPU有自己的硬件高速缓存)，那些比较可能处于硬件高速缓存中的页被称为“热页”，比较不可能处于硬件高速缓存中的页称为“冷页”。其实系统判断是否为热页还是冷页很简单，越最近释放的页就比较可能是热页，所以在双向链表中，从链表头插入可能是热页的单页框，在链表尾插入可能是冷页的单页框。分配时热页就从链表头获取，冷页就从链表尾获取。

在每CPU页框高速缓存中也可能会遇到没有空闲的页框(被分配完了)，这时候每CPU页框高速缓存会从伙伴系统中拿出页框放入每CPU页框高速缓存中，相反，如果每CPU页框高速缓存中页框过多，也会将一些页框放回伙伴系统。

在内核中使用struct per_cpu_pageset结构描述一个每CPU页框高速缓存，其中的struct per_cpu_pages是核心结构体，如下：

```
1 /* 描述一个CPU页框高速缓存 */
2 struct per_cpu_pageset {
3     /* 高速缓存页框结构 */
4     struct per_cpu_pages pcp;
5 #ifdef CONFIG_NUMA
6     s8 expire;
7 #endif
```

```
#ifdef CONFIG_SMP
    s8 stat_threshold;
    s8 vm_stat_diff[NR_VM_ZONE_STAT_ITEMS];
#endif
};

struct per_cpu_pages {
    /* 当前CPU高速缓存中页框个数 */
    int count;        /* number of pages in the list */
    /* 上界，当此CPU高速缓存中页框个数大于high，则会将batch个页框放回伙伴系统 */
    int high;        /* high watermark, emptying needed */
    /* 在高速缓存中将要添加或被删去的页框个数 */
    int batch;        /* chunk size for buddy add/remove */

    /* Lists of pages, one per migrate type stored on the pcp-lists */
    /* 页框的链表，如果需要冷高速缓存，从链表尾开始获取页框，如果需要热高速缓存，从链表头开
    struct list_head lists[MIGRATE_PCPTYPES];
};
```

关于页框回收

内存中并非所有物理页面都是可以进行回收的，内核占用的页不会被换出，只有与用户空间建立了映射关系的物理页面才会被换出。总的来说，以下这些种物理页面可以被 Linux 操作系统回收：

- 进程映射所占的页面，包括代码段，数据段，堆栈以及动态分配的“存储堆”（malloc分配的）。
- 用户空间中通过mmap()把文件内容映射到内存所占的页面。
- 匿名页面(没有映射到文件的都是匿名映射，用户空间的堆和栈)：进程用户模式下的堆栈以及是使用 mmap - 匿名映射的内存区(共享内存区)。注：堆栈所占页面一般不被换出。
- 特殊的用于 slab 分配器的缓存，比如用于缓存文件目录结构 dentry 的 cache，以及用于缓存索引节点 inode 的 cache
- tmpfs文件系统使用的页。

Linux 操作系统使用如下这两种机制检查系统内存的使用情况，从而确定可用的内存是否太少从而需要进行页面回收。

- 周期性的检查：这是由后台运行的守护进程 kswapd 完成的。该进程定期检查当前系统的内存使用情况，当发现系统内空闲的物理页面数目少于特定的阈值时，该进程就会发起页面回收的操作。
- “内存严重不足”事件的触发：在某些情况下，比如，操作系统忽然需要通过伙伴系统为用户进程分配一大块内存，或者需要创建一个很大的缓冲区，而当时系统中 的内存没有办法提供足够多的物理内存以满足这种内存请求，这时候，操作系统就必须尽快进行页面回收操作，以便释放出一些内存空间从而满足上述的内存请求。 这种页面回收方式也被称作“直接页面回收”。

如果操作系统在进行了内存回收操作之后仍然无法回收到足够多的页面以满足上述内存要求，那么操作系统只有最后一个选择，那就是使用 OOM(out of memory)killer，它从系统中挑选一个最合适的进程杀死它，并释放该进程所占用的所有页面

1.5 页的申请和释放

内核提供了几个宏协助完成页的申请和释放，如表8-5所示。

表8-5 申请和释放页函数表

宏	描　述
alloc_page/alloc_pages	申请一页/多页内存
free_page/free_pages	释放一页/多页内存
__free_page/__free_pages	释放一页/多页内存

所有的多页申请或释放内存的宏（XXX_pages）都有一个参数order表示申请的内存块的大小，也就是说申请的内存块只能是一整块，比如申请32页内存，得到的块order为5，申请33页内存，得到的块order为6，共64页，这确实是一种浪费。那申请33页内存，先申请32页，再申请1页是否可行呢？某些情况下并不可行，申请一个完整的块，得到的是连续的33页内存，分多次申请得到的内存不一定是连续的，要求连续内存的场景不能采用这种策略。

从另一个角度讲，如果程序的应用场景并不要求连续内存，应该优先多次使用alloc_page，而不是alloc_pages，多次申请几个分散的页，比一次申请多个连续的页，对整个系统要友善得多。只有每一个模块都“与人为善”，程序才能健康，程序员应该多从系统的角度考虑问题。

XXX_page都是通过XXX_pages实现的，只不过传递给后者的order参数为0。

free_XXX和__free_XXX是有区别的，与alloc_XXX对应的是__free_XXX，而不是前者（有点违背正常逻辑，可能是历史原因）。__free_XXX的参数为page和order，而free_XXX的参数为虚拟地址addr和order，addr经过计算得到page，然后调用__free_XXX继续释放内存。由addr计算得到page，只有映射到直接映射区的Low Memory才可以满足，所以free_XXX只能用于直接映射区的LowMemory。

alloc_pages最终调用__alloc_pages_nodemask实现，后者是伙伴系统页分配的核心，主要逻辑如下（代码有调整）。

```
1 struct page *
2 __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order, int preferred_nid,
3                         nodemask_t *nodemask)
4 {
5     struct page *page;
6     unsigned int alloc_flags = ALLOC_WMARK_LOW;
7     gfp_t alloc_mask; /* The gfp_t that was actually used for allocation */
8     struct alloc_context ac = { };
9
```

```
    /*
     * There are several places where we assume that the order value is sane
     * so bail out early if the request is out of bound.
     */
    if (unlikely(order >= MAX_ORDER)) {
        WARN_ON_ONCE(!(gfp_mask & __GFP_NOWARN));
        return NULL;
    }

    gfp_mask &= gfp_allowed_mask;
    alloc_mask = gfp_mask;
    if (!prepare_alloc_pages(gfp_mask, order, preferred_nid, nodemask, &ac, &allo
        return NULL;

    finalise_ac(gfp_mask, &ac);

    /*
     * Forbid the first pass from falling back to types that fragment
     * memory until all local zones are considered.
     */
    alloc_flags |= alloc_flags_nofragment(ac.preferred_zoneref->zone, gfp_mask);

    /* First allocation attempt */
    page = get_page_from_freelist(alloc_mask, order, alloc_flags, &ac);
    if (likely(page))
        goto out;

    /*
     * Apply scoped allocation constraints. This is mainly about GFP_NOFS
     * resp. GFP_NOIO which has to be inherited for all allocation requests
     * from a particular context which has been marked by
     * memalloc_no{fs,io}_{save,restore}.
     */
    alloc_mask = current_gfp_context(gfp_mask);
    ac.spread_dirty_pages = false;

    /*
     * Restore the original nodemask if it was potentially replaced with
     * &cpuset_current_mems_allowed to optimize the fast-path attempt.
     */
    if (unlikely(ac.nodemask != nodemask))
        ac.nodemask = nodemask;

    page = __alloc_pages_slowpath(alloc_mask, order, &ac);

out:
    if (memcg_kmem_enabled() && (gfp_mask & __GFP_ACCOUNT) && page &&
```

```
57          unlikely(memcg_kmem_charge(page, gfp_mask, order) != 0)) {
58          __free_pages(page, order);
59          page = NULL;
60      }
61
62      trace_mm_page_alloc(page, order, alloc_mask, ac.migratetype);
63
64      return page;
65  }
```

gfp_mask是alloc_pages传递的第一个参数，它是按位解释的，包含了优先选择的zone和内存分配的行为信息，可以由多种标志组合而成，如表8-6所示。

表8-6 申请内存标志表

标　志	描　述
__GFP_DMA	DMA zone
__GFP_HIGHMEM	最高为 HIGHMEM zone
__GFP_ATOMIC	不能睡眠，高优先级
__GFP_HIGH	可使用紧急内存池
__GFP_IO	可启动物理 IO 操作
__GFP_FS	可使用文件系统的操作

(续)

标　志	描　述
__GFP_NOMEMALLOC	若未置位，可次高优先级地分配内存
__GFP_MEMALLOC	最高优先级地分配内存
__GFP_DIRECT_RECLAIM	可以直接进入页面回收，需要时进入睡眠
__GFP_KSWAPD_RECLAIM	可以唤醒 kswapd 进行页面回收，kswapd 是负责页面回收的内核线程
__GFP_ZERO	清零
__GFP_RECLAIM	___GFP_DIRECT_RECLAIM|___GFP_KSWAPD_RECLAIM
GFP_KERNEL	__GFP_RECLAIM | __GFP_IO | __GFP_FS
GFP_ATOMIC	__GFP_ATOMIC | __GFP_KSWAPD_RECLAIM | __GFP_HIGH

_alloc_pages_nodemask先调用prepare_alloc_pages，根据参数计算得到ac、alloc_mask和alloc_flags，先以此调用get_page_from_freelist分配块，如果失败，调用__alloc_pages_slowpath，后者会根据gfp_mask计算新的alloc_flags，重新调用get_page_from_freelist。

需要注意的是，___GFP_DIRECT_RECLAIM被置位的情况下，会睡眠，不允许睡眠的情景中不可使用，GFP_KERNEL隐含了___GFP_DIRECT_RECLAIM，慎重使用。

get_page_from_freelist函数的主要逻辑如下。

```
static struct page *
get_page_from_freelist(gfp_t gfp_mask, unsigned int order, int alloc_flags,
                       const struct alloc_context *ac)
{
    struct zoneref *z;
    struct zone *zone;
    struct pglist_data *last_pgdat_dirty_limit = NULL;
    bool no_fallback;

retry:
    /*
     * Scan zonelist, looking for a zone with enough free.
     * See also __cpuset_node_allowed() comment in kernel/cpuset.c.
     */
    no_fallback = alloc_flags & ALLOC_NOFRAGMENT;
    z = ac->preferred_zoneref;
    for_next_zone_zonelist_nodemask(zone, z, ac->zonelist, ac->high_zoneidx,
                                    ac->nodemask) {
        struct page *page;
        unsigned long mark;

        if (cpusets_enabled() &&
            (alloc_flags & ALLOC_CPUSET) &&
            !__cpuset_zone_allowed(zone, gfp_mask))
                continue;
        /*
         * When allocating a page cache page for writing, we
         * want to get it from a node that is within its dirty
         * limit, such that no single node holds more than its
         * proportional share of globally allowed dirty pages.
         * The dirty limits take into account the node's
         * lowmem reserves and high watermark so that kswapd
         * should be able to balance it without having to
         * write pages from its LRU list.
         *
         * XXX: For now, allow allocations to potentially
         * exceed the per-node dirty limit in the slowpath
         * (spread_dirty_pages unset) before going into reclaim,
         * which is important when on a NUMA setup the allowed
         * nodes are together not big enough to reach the
         * global limit.  The proper fix for these situations
         * will require awareness of nodes in the
         * dirty-throttling and the flusher threads.
         */
        if (ac->spread_dirty_pages) {
            if (last_pgdat_dirty_limit == zone->zone_pgdat)
```

```
                continue;

            if (!node_dirty_ok(zone->zone_pgdat)) {
                last_pgdat_dirty_limit = zone->zone_pgdat;
                continue;
            }
        }

        if (no_fallback && nr_online_nodes > 1 &&
            zone != ac->preferred_zoneref->zone) {
            int local_nid;

            /*
             * If moving to a remote node, retry but allow
             * fragmenting fallbacks. Locality is more important
             * than fragmentation avoidance.
             */
            local_nid = zone_to_nid(ac->preferred_zoneref->zone);
            if (zone_to_nid(zone) != local_nid) {
                alloc_flags &= ~ALLOC_NOFRAGMENT;
                goto retry;
            }
        }

        mark = wmark_pages(zone, alloc_flags & ALLOC_WMARK_MASK);
        if (!zone_watermark_fast(zone, order, mark,
                       ac_classzone_idx(ac), alloc_flags)) {
            int ret;

#ifdef CONFIG_DEFERRED_STRUCT_PAGE_INIT
            /*
             * Watermark failed for this zone, but see if we can
             * grow this zone if it contains deferred pages.
             */
            if (static_branch_unlikely(&deferred_pages)) {
                if (_deferred_grow_zone(zone, order))
                    goto try_this_zone;
            }
#endif
            /* Checked here to keep the fast path fast */
            BUILD_BUG_ON(ALLOC_NO_WATERMARKS < NR_WMARK);
            if (alloc_flags & ALLOC_NO_WATERMARKS)
                goto try_this_zone;

            if (node_reclaim_mode == 0 ||
                !zone_allows_reclaim(ac->preferred_zoneref->zone, zone))
                continue;
```

```
94
95              ret = node_reclaim(zone->zone_pgdat, gfp_mask, order);
96              switch (ret) {
97              case NODE_RECLAIM_NOSCAN:
98                  /* did not scan */
99                  continue;
100             case NODE_RECLAIM_FULL:
101                 /* scanned but unreclaimable */
102                 continue;
103             default:
104                 /* did we reclaim enough */
105                 if (zone_watermark_ok(zone, order, mark,
106                         ac_classzone_idx(ac), alloc_flags))
107                     goto try_this_zone;
108
109                 continue;
110             }
111         }
112
113 try_this_zone:
114         page = rmqueue(ac->preferred_zoneref->zone, zone, order,
115                 gfp_mask, alloc_flags, ac->migratetype);
116         if (page) {
117             prep_new_page(page, order, gfp_mask, alloc_flags);
118
119             /*
120              * If this is a high-order atomic allocation then check
121              * if the pageblock should be reserved for the future
122              */
123             if (unlikely(order && (alloc_flags & ALLOC_HARDER)))
124                 reserve_highatomic_pageblock(page, zone, order);
125
126             return page;
127         } else {
128 #ifdef CONFIG_DEFERRED_STRUCT_PAGE_INIT
129             /* Try again if zone has deferred pages */
130             if (static_branch_unlikely(&deferred_pages)) {
131                 if (_deferred_grow_zone(zone, order))
132                     goto try_this_zone;
133             }
134 #endif
135         }
136     }
137
138     /*
139      * It's possible on a UMA machine to get through all zones that are
140      * fragmented. If avoiding fragmentation, reset and try again.
```

```
141          */
142         if (no_fallback) {
143             alloc_flags &= ~ALLOC_NOFRAGMENT;
144             goto retry;
145         }
146 
147         return NULL;
148 }
```

get_page_from_freelist会从用户期望的zone（ac->high_zoneidx）开始向下遍历，例如用户期望分配ZONE_HIGHMEM的内存，函数会按照ZONE_HIGHMEM、ZONE_NORMAL和ZONE_DMA的优先级分配内存。

第1步，zone_watermark_fast判断当前zone是否能够分配该内存块，由内存块的order和zone的水位线（watermark）决定。zone的水位线由它的_watermark字段表示，该字段是个数组，分别对应WMARK_MIN、WMARK_LOW和WMARK_HIGH三种情况下的水位线，它们的值默认在模块初始化的时候设定，也可以运行时改变。

水位线表示zone需要预留的内存页数，内核针对不同紧急程度的内存需求有不同的策略，它会预留一部分内存应对紧急需求，根据zone分配内存后是否满足水位线的最低要求来决定是否从该zone分配内存。

紧急程度共分为四个等级，分别是默认等级、ALLOC_HIGH、ALLOC_HARDER和ALLOC_OOM，各等级对应的最低剩余内存页数分别为watermark、watermark/2、watermark*3/8和watermark/4。例如，zone的watermark值为32，当前zone剩余空闲内存为40页，申请的内存块的order为4，分配了内存块之后剩余24页（40-16）；如果是默认等级，需要剩余32页，分配失败；如果是ALLOC_HIGH，只需剩余16页（32/2），分配成功。

除了以上四个等级，还有一个额外的ALLOC_NO_WATERMARKS，它表示内存申请不受watermark的限制，这就是第2步的逻辑。

紧急程度和进程优先级与gfp_mask的对应关系，如表8-7所示。

表8-7 申请内存紧急程度表

紧急程度	gfp_mask
ALLOC_HIGH	__GFP_HIGH
ALLOC_HARDER	__GFP_ATOMIC 且!__GFP_NOMEMALLOC 或者实时进程且不在中断上下文中
ALLOC_OOM	Out Of Memory
ALLOC_NO_WATERMARKS	__GFP_MEMALLOC
默认等级	其他

第3步，找到了合适的zone或者在ALLOC_NO_WATERMARKS置位的情况下，调用rmqueue分配块，rmqueue分两种情况处理。

第一种情况，order等于0（只申请一页），内核并不立即使用zone->free_area[0].free_list[migratetype]链表上的块，而是调用rmqueue_pcplist函数先查询per_cpu_pages结构体（简称pcp）维护的链表是否可以满足内存申请。pcp对象由zone->pageset的pcp字段表示，其中zone->pageset是每cpu变量。

pcp的lists字段维护了多种迁移类型的块组成的链表，比如MIGRATE_UNMOVABLE、MIGRATE_RECLAIMABLE和MIGRATE_MOVABLE。它们相当于几个order为0的块的缓存池，内核直接从池中分配块，如果池中的资源不足，则向伙伴系统申请，一次申请pcp->batch块。很显然，它的设计是基于其他模块申请一页内存的次数很多的假设，这样一次从伙伴系统申请多个order为0的块缓存下来，就不需要每次都经过伙伴系统，有利于提高内存申请的效率。

第二种情况，order大于0，调用__rmqueue从伙伴系统申请块（实际上，order等于0时，如果pcp池中的资源不足也会调用__rmqueue），它从当前的order开始向上查询zone->free_area[order].free_list[migratetype]链表，如果某个order上的链表不为空，则分配成功。如果最终使用的order（记为order_bigger）大于申请的块的order，则拆分使用的块，生成大小等于order+1, order+2,…, order_bigger-1的块各一个，大小等于order的块两个，其中一个返回给用户。

1.6 一些重要问题

struct page是如何与物理内存的一页相对应的?

所有的struct page 为一个连续的数组 ，起始地址存放在 struct page *mem_map中

因此如果知道了struct page的地址pd，pd - mem_map就得到了pd是哪个page frame的描述符

那在已知一个struct page的情况下，怎样取到该页的内容

以X86 32位通常情况为例：

已知一个page descriptor，可由它得到它所描述的物理页是整个内存的第几页，假设是第M个物理页。

那么这个物理页的物理地址是physAddr = M << PAGE_SHIFT

在得知该物理页的物理地址是physAddr后，就可以视physAddr的大小得到它的虚拟地址：

1.physAddr < 896M 对应虚拟地址是 physAddr + PAGE_OFFSET （PAGE_OFFSET=3G)

2.physAddr >= 896M 对应虚拟地址不是静态映射的，通过内核的高端虚拟地址映射得到一个虚拟地址。

在得到该页的虚拟地址之后，内核就可以正常访问这个物理页了。

高端内存是什么?

要理解high memory是要解决什么问题，首先要了解下内核地址转换的方式。在内核中我们往往要频繁地进行虚拟/物理地址操作，在这种情况下，快速高效的virtual to physical转换就很重要。可如果按照多级页表path walk去查找，内存访问开销就比较大，因此一种简单的"fix-mapping"思路是：将

0xC0000000-0xFFFFFFFF的虚拟地址直接映射到0x00000000-0x3FFFFFFF，也就是将最高的1G地址全部映射到最低的1G，这样虚拟地址与物理地址之间就有固定的3G offset，每当遇到一个内核中的符号，我们需要得到其物理地址时，直接减去3G即可。

有人可能会问，那0-3G的比较低的那些虚拟地址怎么转换呢？答案是不用，也就是内核自己不使用0-3G的虚拟地址（除非是处理syscall）。

上述这种简单粗暴的处理方式很方便理解，效率也比较高（只需要简单的减法操作），但也有自己的局限性。在32位处理器下，按照经典用户态与内核3:1的划分比例，内核能够使用的虚拟地址只有1G大，按照固定offset的映射方式，这意味着内核能够使用的物理地址大小也只有1G。但...随着内核越来越复杂，各种数据结构对内存的需求也越来越高，比如用来物理页的page结构体，仅仅在其上增加一个12字节的reverse mapping管理结构，就会使得page总体占用的内存增高400KB，将近96个物理页大小；即便内存技术的发展使得高于4G的内存变得十分常见，受限于32位系统与这种fix-mapping，内核可用的物理内存大小仍然被死死地限制在1G。

以上，算是对high memory要解决问题的背景介绍。通俗地讲，"high memory"要解决的是32位下虚拟地址空间不足带来的问题（而显然，对64位系统这个问题就不存在了）。实际上在很早以前这个问题就在lwn上讨论过了 ，在当时已经有一些临时的方法去规避这个问题，比如重新划分用户/内核的地址空间比例，变为2.5:1.5等等，但在特定场景下（比如用户态使用的内存非常非常多）会使得用户态运行效率降低，同时带来一些非对其问题，因此也不是一个很好的办法。

怎么解决呢？

如 @Swee Neil所提到的，我们可以把这1G，划分成两部分，一部分用来fix-mapping，一部分用来dynamic-mapping。以x86为例，实际中的做法是，0xC0000000-0xF7FFFFFF的896MB用作fix-mapping，0xF8000000-0xFFFFFFFF的128MB用作dynamic-mapping，前者仍然对应于物理地址的0x00000000-0x37FFFFFF（只不过部分要优先分配给DMA）；后者就是所谓的high memory。当内核想访问高于896MB物理地址内存时，从0xF8000000 ~ 0xFFFFFFFF地址空间范围内找一段相应大小空闲的虚拟地址空间，借用一会。

借用这段虚拟地址空间，建立映射到想访问的那段物理内存（即填充内核PTE页面表），临时用一会，用完后归还。这样别人也可以借用这段地址空间访问其他物理内存，实现了使用有限的地址空间，访问所有所有物理内存。

当然，high memory也有自己的缺点，就是效率比较低（既然是动态的，就绕不开重映射、pte操作等等）。

实际上high memory还被划分为了3个区域([3])，一部分用于vmalloc分配虚拟地址上连续的内存，一部分用于较长期的动态映射（persistent kernel mappings），还有一部分用于编译时可以直接分配物理地址的高端固定映射（fixmaps）：

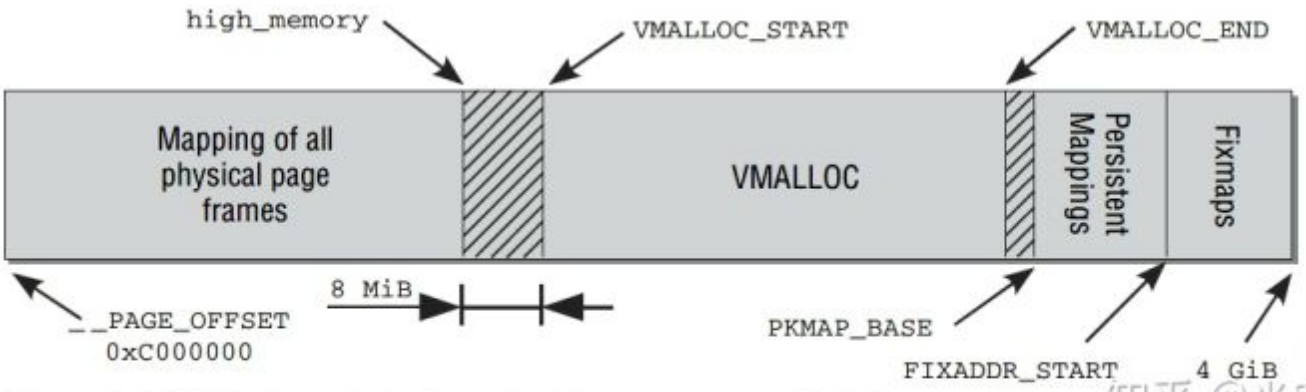


Figure 3-15: Division of the kernel address space on IA-32 systems.

来到64位系统，这个问题天然就不存在，因此在64位系统的memlayout([3])中就没有high memory，但vmalloc仍然是内核的一个重要部分，因此memlayout中仍然有这一部分：

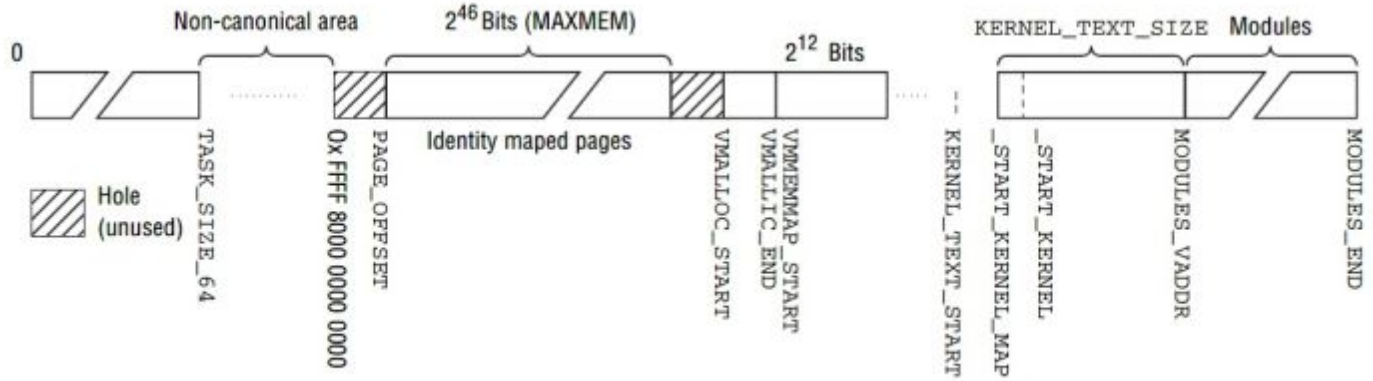


Figure 3-19: Organization of the virtual address space on AMD64 systems. The image is not drawn to scale, naturally.

二、内存的线性空间

内核线性空间各个区的本质区别就在于映射的不同

32位系统中，原则上一个普通进程可以访问4G的线性空间。但这4G空间并不都属于内核，按照经典的3:1划分，进程在用户态可以访问线性地址的前3G，在内核态的时候可以访问第4G。

在这种划分方式下，内核访问的线性地址起始于0xC0000000，结束于0xFFFFFFFF，共1G。这意味着内核中有效的指针的值都在该区间，用户空间的指针的值在0xC0000000之前。

3：1的划分并不是一成不变的，可以通过“Memory split”编译选项来更改CONFIG_VMSPLIT_3G对3：1的划分。划分的方式决定了内核使用线性地址的起始点，PAGE_OFFSET，该宏在x86上等同于CONFIG_PAGE_OFFSET，后者就由我们选择的划分方式确定。如果是3：1的划分，PAGE_OFFSET就等于0xC0000000。CONFIG_VMSPLIT_3G是默认的选择，本书基于此讨论，其他情况类似。

X86_64上PAGE_OFFSET在五级页表使能的情况下，值等于0xff11000000000000，否则等于0xffff888000000000。

我们接下来主要讨论X86_32，请注意，PAGE_OFFSET是内核线性地址空间的起始，并不是物理内存的开始，物理内存映射到该空间即可访问。

表9-1 线性空间变量表

名　称	描　述
high_memory	Low Memory 和 High Memory 的分界，线性地址 一般，high_memory = max_low_pfn<<PAGE_SIZE + PAGE_OFFSET
PAGE_SIZE	表示一页内存使用的二进制位数，一般为 12
PAGE_OFFSET	前文已经介绍过，内核线性空间的起始位置，一般为 0xC0000000
FIXADDR_TOP	固定映射的结尾位置
FIXADDR_START	固定映射的开始位置
VMALLOC_END	动态映射区的结尾
VMALLOC_START	动态映射区的开始

有了以上的铺垫，下面正式讨论内核线性空间的布局，在HIGHMEM使能的情况下（未使能的情况下大同小异），布局如图9-1所示。

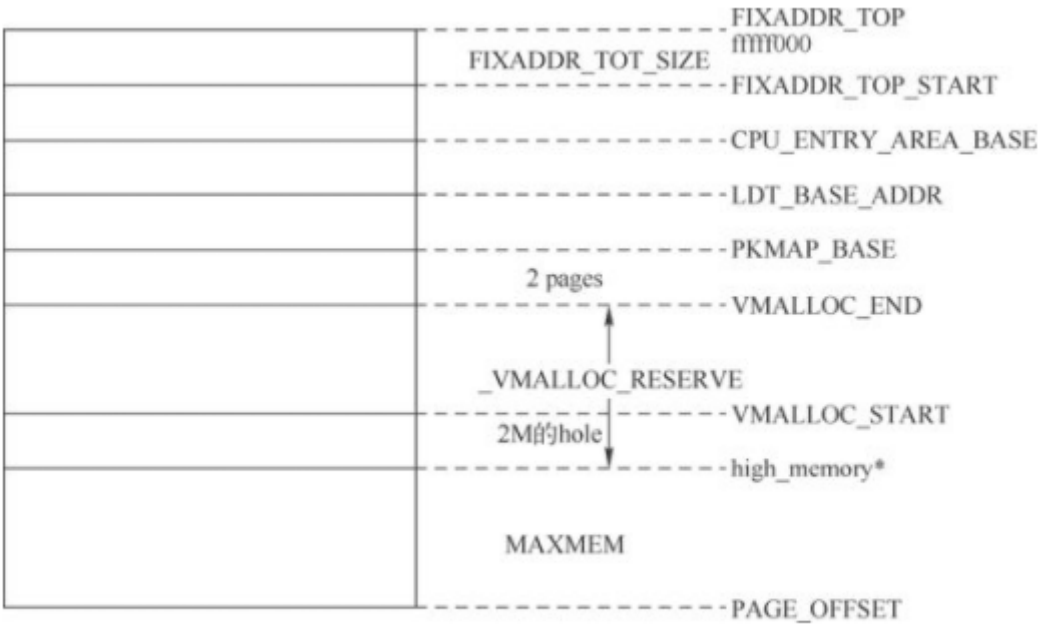


图9-1 线性空间布局

1G的线性地址空间 [0xC0000000, 0xFFFFFFFF]，最高的4K不用，由FIXADDR_TOP开始向下算起，FIXADDR的TOP和START之间是固定映射区。PKMAP_BASE和FIXADDR之间是永久映射区和CPU Entry区（存放cpu_entry_area对象）。VMALLOC_END和VMALLOC_START之间是动态映射区，之前介绍的ioremap获取的虚拟内存区间就属于这个区。接下来8M的hole没有使用，目的是越界检测，_VMALLOC_RESERVE默认为128M减去8M，实际上动态映射区为120M。

需要说明的是，high_memory是计算得来的，它和PAGE_OFFSET之间的间隔可能更小，但它和VMALLOC_START之间的8M是固定的，也就是说，动态映射区可能比120M大。

以上几个区域是必需的，1G的空间减去这些剩下约896M，这就是896这个数字的来源，以MAXMEM表示，该区间称为直接映射区。

我们要有一个意识，内存并不是随意映射的，物理内存和合适的线性地址区间缺一不可。那么内核为什么要有一套如此复杂，而且对每个进程都适用的规则呢？因为内核的空间是进程直接或者间接共享的！

线性地址用户空间和内核空间比例为3：1，3G的用户空间每个进程独立拥有，互不影响，也就是说进程负责维护自己的这部分页表，称为用户页表（从这里开始，书中提到的页表不再是虚拟页的概念，而是辅助完成寻址的页框）。当然，它只需要映射自己使用的那部分内存，并不需要维护整套用户页表。内核空间的1G则不同，基本所有进程共有，也就是说它们的页表的内核部分（内核页表）很多情况下是相同的，属于公共部分。

内核的线性空间并不属于某一个进程，而是大家共同拥有。一个进程对公共部分做的改动，对其他进程很多情况下是可见的，有影响的。通俗一点就是，坑就那么多，栽进去一个萝卜，就少了一个萝卜的坑。

直接映射区

直接映射区理论上最大为MAXMEM，所谓的直接映射非常简单，映射后的虚拟地址va等于物理地址pa加PAGE_OFFSET（0xC0000000）。物理地址从最小页pfn=0开始，依次按照pa+PAGE_OFFSET → va的方式映射到该区间

从pfn=0开始，一直映射到没有多余空间或者没有物理内存为止。所以，如果系统本身内存不足MAXMEM（896M），且没有特意预留High Memory的情况下，全部的物理内存（不包括MMIO）都会映射到直接映射区。如果系统内存大于896M，或者预留了High Memory，内存不能（不会）全部映射到该区。

直接映射区是唯一的Low Memory映射的区域，页框0到页框max_low_pfn映射到该区间。映射一旦完成，系统运行期间不会改变，这是相对于其他区的优势；从下面几个区的分析中会看到，High Memory映射区域在运行期间是可以改变的，所以系统中需要一直稳定存在的结构体和数据只能使用直接映射区的内存，比如page对象

在水果店中，有些水果一年四季都有，比较容易保存，且占空间较多（销量大），比如苹果，老板可以一直都不换它们的位置。直接映射区与水果店的苹果区类似。

动态映射区

VMALLOC_START和VMALLOC_END之间的120M空间为动态映射区，它是内核线性空间中最灵活的一个区。其他几个区都有固定的角色，它们不能满足的需求，都可以由动态映射区来满足，常见的ioremap、mmap一般都需要使用它。

使用动态映射区需要申请一段属于该区域的线性区间，内核提供了get_vm_area函数族来满足该需求，它们的区别在于参数不同，但最终都通过调用__get_vm_area_node函数实现。

内核提供了vm_struct结构体来表示获取的线性空间及其映射情况，主要字段如表9-2所示。

表9-2 vm_struct字段表

字　段	类　型	描　述
addr	void *	线性空间的起始地址
size	unsigned long	线性空间的大小
nr_pages	unsigned int	包含的页数
pages	page **	page 对象组成的动态数组
phys_addr	phys_addr_t	对应的物理地址
next	vm_struct *	下一个 vm_struct，组成链表

在__get_vm_area_node的参数中，start和end表示用户希望的目标线性区间所在的范围，get_vm_area传递的参数为VMALLOC_START和VMALLOC_END。内核会将已使用的动态映射区的线性区间记录下来

每一个区间以vmap_area结构体表示。vmap_area结构体除了va_start和va_end字段表示区间的起始外，还有两个组织各个vmap_area对象的字段：rb_node和list。

一个vmap_area对象既在以vmap_area_root为根的红黑树中（rb_node），又以list字段链接，rb_node字段的作用是快速查找，list字段则是根据查找的结果继续遍历（各区间是按照顺序链接的）。

__get_vm_area_node会查找一个没有被占用的合适区间，如果找到则将该区间记录到红黑树和链表中，然后利用得到的vmap_area对象给vm_struct对象赋值并返回。vm_struct结构体是其他模块可见的，vmap_area结构体是动态映射区内部使用的。

区间的管理有点像管理员管理街道菜市场，菜市场的面积是固定的，管理员需要记录每个摊位的位置和尺寸，当有新的摊位申请时，他就找一个没有被占用的能满足申请尺寸的地方作为新的摊位，将摊位记录在案，然后返回给申请者。

申请到vm_struct线性区间后，就可以将物理内存映射到该区域，ioremap使用的是ioremap_page_range，有的模块使用map_vm_area，remap_vmalloc_range等。

最后，free_vm_area和remove_vm_area可以用来取消映射并释放申请的区间，free_vm_area还会释放vm_struct对象所占用的内存。

对应水果店的例子，有些水果是时令的，比如杨梅、水蜜桃等，它们占用空间的时间较短，过了季节就会退出市场，动态映射区与杨梅区类似。

永久映射区

永久映射区的线性地址起于PKMAP_BASE，大小为LAST_PKMAP个页，开启了PAE的情况下为512，否则为1024，也就是一页页中级目录表示的大小（2M或者4M）。

虽然名字为“永久”（Permanent），但实际并非永久，如果你只记得“永久”二字，是理解不了它的，记下kmap就可以了。内核提供了kmap函数将一页内存映射到该区，kunmap函数用来取消映射。kmap的参数为page结构体指针，如果对应的页不属于High Memory，则直接返回页对应的虚拟地址vaddr（等于paddr+PAGE_OFFSET），否则内核会在该线性区内寻址一个未被占用的页，作为虚拟地址，并根据page和虚拟地址建立映射。

传递的参数决定了kmap一次只映射一页，也就相对于永久映射区被分成了以页为单位的“坑”，内核利用数组来管理一个个“坑”，pkmap_count数组就是用来记录“坑”的使用次数的，使用次数为0的表示未被占用。

kmap可能会引起睡眠，不可以在中断等环境中使用。

例如，水果店里总会有些营销策略，比如某些水果打特价。特价区不需要占用较大空间，布置它是为了吸引顾客光顾，隔一段时间换一个种类，永久映射区与之类似。

固定映射区

上面说的“永久”非永久，本节的“固定”确实是固定。固定映射区起始于FIXADDR_TOT_START，终止于FIXADDR_TOP，分为多个小区间，它们的边界由fixed_addresses定义，每一个小区间都有特定的用途，如ACPI等。这些小区间的映射不会脱离内存映射的本质，区别仅在于区间的作用不同，我们不做一一介绍。

所有的小区间中，有一个比较特殊，它有一个特殊的名字叫临时映射区。它在固定映射区的偏移量为FIX_KMAP_BEGIN和FIX_KMAP_END，区间大小等于KM_TYPE_NR*NR_CPUS页。

从它的大小可以看出，每个CPU都有属于自己的临时映射区，大小为KM_TYPE_NR页。内核提供了kmap_atomic、kmap_atomic_pfn和kmap_atomic_prot函数来映射物理内存到该区域，将获得的物理内存的

page对象的指针或者pfn作为参数传递至这些函数即可，kunmap_atomic函数用来取消映射。

与永久映射区一样，如果对应的物理页不属于High Memory，则直接返回页对应的虚拟地址vaddr（等于paddr + PAGE_OFFSET），否则内核会找到一页未使用的线性地址，并建立映射。

临时映射区的区间很小，每个CPU只占几十页，它的管理也很简单。内核使用一个每CPU变量__kmap_atomic_idx来记录当前已使用的页的号码，映射成功变量加1，取消映射变量减1。这意味着小于该变量的号码对应的页都被认为是已经映射的。另外，取消映射的时候，内核直接取消变量当前值对应的映射，并不是传递进来的虚拟地址对应的映射，所以保证二者一致是程序员的责任。

临时映射区的管理设计得如此简单，优点是快速，kmap_atomic比kmap快很多。它的另一个优点是不会睡眠，kmap会睡眠，所以它的适应性更广。

另一方面，临时映射区的使用是有限制的。“临时”二字究竟意味着多短呢？如果永久映射区与快捷酒店住宿类似，那临时映射区最多只能算是“钟点房”，休息完就走。kmap_atomic会调用preempt_disable禁内核抢占，kunmap_atomic来重新使能它，所以基本上只能用来临时存放一些数据，数据处理完毕立即释放。

固定映射区与水果店的特色水果区类似，水果店的招牌固定放在显眼的位置。至于临时映射区，它是固定映射区的一个子区域，不妨叫它“钟点房”区吧。

mmap机制

前面讲述了内核的线性空间，此处插入一节，介绍下mmap机制。之所以在这里引入这个话题，是因为mmap也使用线性空间。但必须强调一下，mmap使用的不是内核线性空间，是用户线性空间，与

前面的讨论一定不要混在一起；另外，它并没有脱离内核，因为它本身就是通过系统调用实现的。

函数原型

mmap用来将文件或设备映射进内存，映射之后不需要再使用read/write等操作文件，可以像访问普通内存一样访问它们，可以明显地提高访问效率。mmap用来建立映射，munmap用来取消映射，mmap的函数原型（用户空间）如下。

```
        void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
        int munmap(void *addr, size_t length);
```

参数fd是已打开文件的文件描述符，addr是期望获得的虚拟地址，也就是希望内核将文件或设备映射到该虚拟地址，一般为0（由内核根据自身方便确定虚拟地址），length是映射的区间大小，offset表示选择的文件或设备的偏移量，以该偏移量作为起始映射点，offset必须是物理页大小（sysconf(_SC_PAGE_SIZE)）的整数倍，prot是期望的内存保护标志，flags表示映射的标志。

简言之，希望内核将文件或设备（fd）从offset开始，到offset+length结束的区间映射到addr开始的虚拟内存中，并将addr返回，映射后的内容访问权限由prot决定。

prot不能与文件的打开标志冲突，一般是PROT_NONE或者读写执行三种标志的组合，如表9-3所示。

表9-3 prot标志表

标　志	描　述
PROT_NONE	映射的内容不可访问
PROT_EXEC	映射的内容可执行
PROT_READ	映射的内容可读
PROT_WRITE	映射的内容可写

flags用来传递映射标示，常见的是MAP_PRIVATE和MAP_SHARED的其中一种，和以下多种标志的组合，如表9-4所示。

表9-4 flags标志表

标　志	描　述
MAP_PRIVATE	私有映射
MAP_SHARED	共享映射
MAP_FIXED	严格按照 mmap 的 addr 参数表示的虚拟地址进行映射
MAP_ANONYMOUS	匿名映射，映射不与任何文件关联
MAP_LOCKED	锁定映射区的页面，防止被交换出内存

私有和共享是相对于其他进程来说的，MAP_SHARED对映射区的更新对其他映射同一区域的进程可见，所做的更新也会体现在文件中，可以使用msync来控制何时写回文件。MAP_PRIVATE采用的是写时复制策略，映射区的更新对其他映射同一区域的进程不可见，所做的更新也不会写回文件。

需要强调的是，以上对prot和flags的描述，并不适用于所有文件，比如有些文件只接受MAP_PRIVATE，不接受MAP_SHARED，或者相反。应用程序并不能随意选择参数，普通文件一般在不违反自身权限的情况下可以自由选择，设备文件等特殊文件可以接受的参数由驱动决定。

数据结构

mmap使用的线性区间管理与内核的线性区间管理是不同的，内核的线性区间是进程共享的，mmap使用的线性区间则是进程自己的用户线性空间，在不考虑进程间关系的情况下，进程的用户线性空间是独立的。

但二者的管理又没有本质的区别，前面说过动态映射区的管理就像管理员管理菜市场，这里的线性区间管理则演变成了每家分配三亩地（用户空间3G，内核空间1G），各家规划自家的庄稼。既然是规划，一样需要地段分配、分配记录、地段回收等功能。村里有一亩良田可以产黄金，全村人共有，剩下的普通田每户三亩。村里管理这一亩地和每家管理三亩地，实际上没有本质区别，只是角度不同罢了。

内核以vm_area_struct结构体表示一个用户线性空间的区域，主要字段如表9-5所示。

表9-5 vm_area_struct字段表

字　段	类　型	描　述
vm_start	unsigned long	线性区间的开始
vm_end	unsigned long	线性区间的结束，实际的区间为[vm_start, vm_end)
vm_page_prot	pgprot_t	映射区域的保护方式
vm_flags	unsigned long	映射的标志
vm_ops	vm_operations_struct *	映射的相关操作
vm_file	file *	映射文件对应的 file 对象
vm_pgoff	unsigned long	在文件内的偏移量
vm_rb	rb_node	将它插入红黑树中
vm_next vm_prev	vm_area_struct *	将它链接到链表中
vm_mm	mm_struct *	所属的 mm_struct

既然进程要管理自己的线性区间分配，就需要管理它所有的vm_area_struct对象。进程的mm_struct结构体有两个字段完成该任务，mmap字段是进程的vm_area_struct对象组成的链表的头，mm_rb字段是所有vm_area_struct对象组成的红黑树的根。内核使用链表来遍历对象，使用红黑树来查找符合条件的对象。vm_area_struct结构体的vm_next和vm_prev字段用来实现链表，vm_rb字段用来实现红黑树。

从应用的角度看，mmap成功返回就意味着映射完成，但实际上从驱动的角度并不一定如此。很多驱动在mmap中并未实现内存映射，应用程序访问该地址时触发异常后，才会做最终的映射工作，这就用到了vm_ops字段

mmap的实现

mmap使用的系统调用与平台有关，但最终都调用ksys_mmap_pgoff函数实现。需要说明的是，mmap传递的offset参数会在调用的过程中转换成页数（offset/MMAP_PAGE_UNIT），一般由glibc完成，所以内核获得的参数已经是以页为单位的（pgoff可以理解为page offset）。

ksys_mmap_pgoff调用vm_mmap_pgoff，经过参数检查后，最终调用do_mmap函数完成mmap，do_mmap的主要逻辑如下。

```
unsigned long do_mmap(struct file *file, unsigned long addr,
            unsigned long len, unsigned long prot, unsigned long flags,
            vm_flags_t vm_flags, unsigned long pgoff,


            unsigned long *populate, struct list_head *uf)
{
    //省略变量的定义和一些对 flags 和 prot 等参数的检查
    len = PAGE_ALIGN(len);
    addr = get_unmapped_area(file, addr, len, pgoff, flags);
    vm_flags = calc_vm_prot_bits(prot) | calc_vm_flag_bits(flags) |
        mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
    if (file) {
        switch (flags & MAP_TYPE) {
    case MAP_SHARED:
        flags &= LEGACY_MAP_MASK;
        /* fall through */
        case MAP_SHARED_VALIDATE:
            if ((prot&PROT_WRITE) && !(file->f_mode&FMODE_WRITE))
                return -EACCES;

            vm_flags |= VM_SHARED | VM_MAYSHARE;
            if (!(file->f_mode & FMODE_WRITE))
                vm_flags &= ~(VM_MAYWRITE | VM_SHARED);

            /* fall through */
    case MAP_PRIVATE:
        if (!(file->f_mode & FMODE_READ))
            return -EACCES;
        if (path_noexec(&file->f_path)) {
            if (vm_flags & VM_EXEC)
                return -EPERM;
            vm_flags &= ~VM_MAYEXEC;
        }

        if (!file->f_op->mmap)
            return -ENODEV;
        break;
    }
    }
    addr = mmap_region(file, addr, len, vm_flags, pgoff);
    return addr;
}
```

第一步当然是做合法性检查和调整，映射的长度len（区间大小）被调整为页对齐，所以mmap返回后，实际映射的长度可能比要求的要大。比如某个mmap调用中，offset和length等于0和0x50，实际映射为0和0x1000。

为什么offset本身必须是页对齐的呢？回忆一下，offset在glibc中会被变成pgoff，假设offset和length等于0x10和0x50，实际映射为0和0x1000，(0, 0x1000)是可以覆盖(0x10, 0x10+0x50)的；但如果offset变成了0xff0，(0, 0x1000)就覆盖不了(0xff0, 0xff0+0x50)了。强制要求offset页对齐可以将问题简化，让调用者自行处理特殊需求。

检查完毕就开始找一个合适的线性区间，也就是找“坑”，通过get_unmapped_area实现。get_unmapped_area优先调用文件的get_unmapped_area操作，如果文件没有定义，则使用当前进

程的get_unmapped_area函数（current->mm->get_unmapped_area），该函数与平台有关，一般的逻辑如下。

如果参数addr等于0，内核会根据当前进程的mm_struct对象的mm_rb字段指向的进程所有的vm_area_struct对象组成的红黑树查找一段合适的区域，如果addr不等于0，且(addr, addr + len)没有被占用，则返回addr，否则与addr等于0等同。如果参数flags的MAP_FIXED标示被置位，则直接返回addr。

注意，得到的addr如果不是以页为单位的，直接返回EINVAL退出，如果应用希望自己决定addr，必须保证这点。

在MAP_FIXED被置位的情况下，为什么不检查区间是否被占用就直接返回呢？这是因为MAP_FIXED有个隐含属性，就是“抢地盘”。如果得到的区间已被占用，内核会在mmap_region函数中取消该区间的映射，将区间让出来。根据前面的分析，只有MAP_FIXED被置位的情况下，获得的区间才有可能已经被占用，所以也只有MAP_FIXED会“抢地盘”。如此霸道的属性，所以MAP_FIXED并不推荐使用，当然有些情况下，它却是必需的，比如下文介绍的brk。

MAP_FIXED的另外一个版本MAP_FIXED_NOREPLACE会友好很多。如果指定的区间已经被占用，MAP_FIXED_NOREPLACE会直接返回错误（EEXIST）。

有了“坑”之后，就可以栽萝卜了。首先计算vm_flags，可以看到，如果文件没有打开写权限，VM_MAYWRITE和VM_SHARED标示会被清除；如果文件没有打开读权限，则直接返回EACCES。实际上，在一般情

况下共享隐含着“写”，私有隐含着“读”。当然，从代码逻辑上看，即使文件没有打开写权限，MAP_SHARED也可以成功，但是映射的页并没有写权限[vm_flags = calc_vm_prot_bits(prot) | (…)，因为文件没有打开写权限，prot也就没有PROT_WRITE标志，所以vm_flags也不会有VM_WRITE]。

VM_MAYXXX和VM_XXX有什么区别呢？VM_XXX表示映射的页的权限，VM_MAYXXX表示VM_XXX可以被置位，如表9-6所示。为了不引起歧义，还是要强调下。

表9-6 VM_XXX和VM_MAYXXX关系表

VM_XXX	VM_MAYXXX
VM_READ：映射的页可读	VM_MAYREAD：VM_READ 标示可以被设置
VM_WRITE：映射的页可写	VM_MAYWRITE：VM_WRITE 标示可以被设置
VM_EXEC：映射的页可执行	VM_MAYEXEC：VM_EXEC 标示可以被设置
VM_SHARED：映射的页可共享	VM_MAYSHARED：VM_SHARED 标示可以被设置

完成了以上准备工作，就可以调用mmap_region进行映射了，主要逻辑如下。

```
unsigned long mmap_region(struct file *file, unsigned long addr,
        unsigned long len, vm_flags_t vm_flags, unsigned long pgoff)
{
    //省略变量定义、合法性检查和出错处理等
    struct mm_struct *mm = current->mm;

    find_vma_links(mm, addr, addr + len, &prev, &rb_link, &rb_parent)
    vma = vm_area_alloc(mm);
    vma->vm_start = addr;
    vma->vm_end = addr + len;
    vma->vm_flags = vm_flags;
    vma->vm_page_prot = vm_get_page_prot(vm_flags);
    vma->vm_pgoff = pgoff;
    if (file) {
        vma->vm_file = get_file(file);
        error = file->f_op->mmap(file, vma);    //call_mmap
        addr = vma->vm_start;
    }else if (vm_flags & VM_SHARED) {
        error = shmem_zero_setup(vma);
    }
    vma_link(mm, vma, prev, rb_link, rb_parent);
    vma_set_page_prot(vma);
    return addr;
}
```

可以看到，mmap_region主要做三件事，初始化vm_area_struct对象、完成映射、将对象插入链表和红黑树。不考虑匿名映射，mmap最终是靠驱动提供的文件的mmap操作实现的（file->f_op->mmap），也就是说mmap究竟产生了什么效果最终是由驱动决定的。

到目前为止，我们仍然只看到了“坑”（线性地址空间，vm_area_struct），却没有看到“萝卜”的身影，你肯定猜到了，“萝卜”是由驱动决定的。没错，内核的mmap模块主要负责找到“坑”，至于是否栽“萝卜”，栽哪个“萝卜”，怎么栽，是由驱动决定的。

内存映射的总结

内存映射基本分析完毕，总结来说，主要包括三部分，物理内存的组织（“萝卜”）、线性空间的管理（“坑”）和物理内存与虚拟内存的映射（五级页表）。无论内核还是用户空间，内存的映射基本离不开这三个部分。

系统启动后，程序不能直接访问物理内存，只能访问映射后的虚拟内存，或者是触发异常完成映射。至于物理内存和线性空间的管理，不同用途的实现其实并没有本质的区别，无非是选择不同的数据结构（数组、链表和红黑树等），采用不同的组织方式而已。

三、缺页异常

我们前面关于内存的讨论基本都是已经分配了物理内存的情况，虚拟内存的访问按部就班即可，但很多情况下，得到虚拟内存的时候并没有物理内存与之对应，比如我们在c语言中使用malloc申请堆内存，得到的虚拟地址可能在访问之前都只是“空有其表”，访问这些虚拟地址会导致缺页异常（Page Fault）。

当然了，缺页异常不止没有对应的物理内存一种，访问权限不足等也会导致异常。为了帮助我们区分、处理缺页异常，CPU会额外提供两项信息：错误码和异常地址。

错误码error_code存储在栈中，包含以下信息。

1. 异常的原因是物理页不存在，还是访问权限不足，由X86_PF_PROT标志区分，当error_code&X86_PF_PROT等于0时，表示前者。
2. 导致异常时，处于用户态还是内核态，由X86_PF_USER标志区分。
3. 导致异常的操作是读还是写，由X86_PF_WRITE标志区分。
4. 物理页存在，但页目录项或者页表项使用了保留的位，X86_PF_RSVD标志会被置位。
5. 读取指令的时候导致异常，X86_PF_INSTR标志会被置位。
6. 访问违反了protection key导致异常，X86_PF_PK标志会被置位。所谓protection key，简单理解就是有些寄存器可以控制部分内存的读写权限，越权访问会导致异常。

至于异常地址，就是导致缺页异常的虚拟地址，存储在CPU的cr2寄存器中。

讨论代码之前，我们先从逻辑上分析常见的导致缺页异常的场景和合理的处理方式，也就是常见的“症状”和“药方”。

第一种场景是程序逻辑错误，分为以下三类。

第一类是访问不存在的地址，最简单的是访问空指针

第二类是访问越界，比如用户空间的程序访问了内核的地址。

第三类是违反权限，比如以只读形式映射内存的情况下写内存。

缺页异常对此无能为力，程序的执行没有按照程序员的预期执行，应该是程序员来解决。缺页异常并不是用来解决程序错误的，对此只能是oops、kernel panic等。

第二种场景是访问的地址未映射物理内存。这是正常的，也是对系统有益的，程序中申请的内存并不都会使用，物理内存毕竟有限，使用的时候再去映射物理内存可以避免浪费。

第三种场景是TLB过时，页表更新后，TLB未更新，这种情况下，绕过TLB访问内存中的页表即可。

第四种场景是COW（Copy On Write，写时复制）等，内存没有写权限，写操作导致缺页异常，但它与第一种场景的第三类错误是不同的产生异常的内存是可以有写权限的，是“可以有”和“真没有”的区别。

处理缺页异常

缺页异常的处理函数是page_fault，它是由汇编语言完成的，除了保存现场外，它还会从栈中获得error_code，然后调用do_page_fault函数。后者读取cr2寄存器得到导致异常的虚拟地址，然后调用__do_page_fault，__do_page_fault根据不同的场景处理异常，展开如下。

```
void __do_page_fault(struct pt_regs *regs, unsigned long error_code,
        unsigned long address)
{
```

```
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct mm_struct *mm;
    int fault, major = 0;
    unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE;

    //省略出错处理等
    tsk = current;
    mm = tsk->mm;
    if (unlikely(fault_in_kernel_space(address))) {
        //do_kern_addr_fault
        if (!(error_code & (X86_PF_RSVD | X86_PF_USER | X86_PF_PROT))) {
            if (vmalloc_fault(address) >= 0)    //1
                return;
        }

        if (spurious_kernel_fault(error_code, address))    //2
            return;
        bad_area_nosemaphore(regs, error_code, address, NULL);

        return;
    }
    //do_user_addr_fault
    if (unlikely(error_code & X86_PF_RSVD))    //3
        pgtable_bad(regs, error_code, address);
    vma = find_vma(mm, address);    //4
    if (unlikely(!vma)) {
        bad_area(regs, error_code, address);
        return;
    }
    if (likely(vma->vm_start <= address))
        goto good_area;
    if (unlikely(!(vma->vm_flags & VM_GROWSDOWN))) {
        bad_area(regs, error_code, address);
        return;
    }
    if (unlikely(expand_stack(vma, address))) {
        bad_area(regs, error_code, address);
        return;
    }
good_area:
    if (unlikely(access_error(error_code, vma))) {    //5
        bad_area_access_error(regs, error_code, address, vma);


      return;
    }
    fault = handle_mm_fault(vma, address, flags);    //6
    //…
}
```

X86_PF_RSVD类型的异常被当作一种错误：使用保留位有很大的风险，它们都是X86预留未来使用的，使用它们有可能会造成冲突。所以无论产生异常的地址属于内核空间（第1步）还是用户空间（第3步），都不会尝试处理这种情况，而是产生错误。

如果导致异常的虚拟地址address属于内核空间，X86_PF_USER意味着程序在用户态访问了内核地址，同样是错误，能够得到处理的只有vmalloc和spurious等。使用vmalloc申请内存的时候更新的是主内核页表，并没有更新进程的页表（进程拥有独立页表的情况下），进程在访问这部分地址时就会

产生异常，此时只需要复制主内核页表的内容给进程的页表即可，这也是第1步的vmalloc_fault函数的逻辑。

内核中，页表的访问权限更新了，出于效率的考虑，可能不会立即刷新TLB。比如某段内存由只读变成读写，TLB没有刷新，写这段内存可能导致X86_PF_PROT异常，spurious_fault就是处理这类异常的，也就是导致缺页异常的第三种场景。

如果异常得不到有效处理，就属于bad_area，会调用bad_area、bad_area_nosemaphore、bad_area_access_error等函数，它们的逻辑类似，如果产生异常时进程处于用户态（X86_PF_USER），发送SIGSEGV（SEGV的全称为Segmentation Violation）信号给进程；如果进程处于内核态，尝试修复错误，失败的情况下使进程退出。

从第3步开始，导致异常的虚拟地址都属于用户空间，所以问题就变成找到address所属的vma，映射内存。第4步，find_vma找到第一个结尾地址（vma->vm_end）大于address的vma（进程的vma是有顺序的），找不到则出错。如果该vma包含了address，那么它就是address所属的vma，否则尝试扩大vma来包含address，失败则出错。

我们前面将虚拟内存和物理内存比喻成坑和萝卜，按照这个比喻，第4步是要找到合适的坑。坑里面没有萝卜是可以解决的，如果连坑都没有，多半是程序逻辑错误。

第5步，找到了vma之后，过滤几种因为权限不足导致异常的场景：内存映射为只读，尝试写内存；读取不可读的内存；内存映射为不可读、不可写、不可执行。这几种场景都是程序逻辑错误，不予处理。

到了第6步，才真正进入处理缺页异常的核心部分，前5步讨论了address属于内核空间的情况和哪些情况算作错误两个话题，算是“前菜”。至此，我们可以把导致缺页异常的场景总结为错误和异常两类，除了vmalloc_fault和spurious_fault外，前面讨论的场景均为错误，都不会得到处理，理解缺页异常的第一个关键就是清楚处理异常并不是为了纠正错误。

handle_mm_fault调用__handle_mm_fault继续处理异常，深入分析之前，我们需要先明了现状、目标、面临的问题和对策。现状是我们已经找到了address所属的vma，目标是完成address所需的映射。面临的问题可以分为以下三种类型。

第一种，没有完整的内存映射，也就是没有映射物理内存，申请物理内存完成映射即可。

第二种，映射完整，但物理页已经被交换出去，需要将原来的内容读入内存，完成映射。

第三种，映射完整，内存映射为可写，页表为只读，写内存导致异常，常见的情况就是COW。

__do_page_fault的第5步和类型3都提到了内存映射的权限和页表的权限，在此总结。

用户空间虚拟内存访问权限分为两部分，一部分存储在vma->vm_flags中（VM_READ、VM_EXEC和VM_WRITE等），另一部分存储在页表中。前者表示内存映射时设置的访问权限（内存映射的权限），表示被允许的访问权限，是一个全集，允许范围外的访问是错误，比如尝试写以PROT_READ方式映射的内存。后者表示实际的访问权限（页表的权限），内存映射后，物理页的访问权限可能会发生变化，比如以可读可写方式映射的内存，在某些情况下页表被改变，变成了只读，写内存就会导致异常，这种情况不是错误，因为访问是权限允许范围内的，COW就是如此。这是理解缺页异常的第二个关键，区分内存访问权限的两部分。

明确了问题之后，__handle_mm_fault的逻辑就清晰了，它访问address对应的页目录，如果某一级的项为空，表示是第一种问题，申请一页内存填充该项即可。它访问到pmd（页中级目录），接下来这三种类型的问题的“分水岭”出现了：pte内容的区别导致截然不同的处理逻辑，由handle_pte_fault函数完成，具体如下。

```
int handle_pte_fault(struct vm_fault *vmf)
{
    pte_t entry;
    //省略出错处理等
    if (unlikely(pmd_none(*vmf->pmd))) {   //1
        vmf->pte = NULL;
    } else {
        vmf->pte = pte_offset_map(vmf->pmd, vmf->address);
        vmf->orig_pte = *vmf->pte;
        if (pte_none(vmf->orig_pte)) {
            pte_unmap(vmf->pte);
            vmf->pte = NULL;
        }
    }

    if (!vmf->pte) {    //2
        if (vma_is_anonymous(vmf->vma))
            return do_anonymous_page(vmf);
        else
            return do_fault(vmf);
    }


   if (!pte_present(vmf->orig_pte))    //3
       return do_swap_page(vmf);

   entry = vmf->orig_pte;
   if (unlikely(!pte_same(*vmf->pte, entry)))
       return 0;
   if (vmf->flags & FAULT_FLAG_WRITE) {    //4
       if (!pte_write(entry))
           return do_wp_page(vmf);
   }
   //…
}
```

vm_fault结构体（以下简称vmf）是用来辅助处理异常的，保存了处理异常需要的信息，主要字段如表10-3所示。

表10-3 vm_fault字段表

字　段	类　型	描　述
vma	vm_area_struct *	address 对应的 vma
flags	unsigned int	FAULT_FLAG_xxx 标志
pgoff	pgoff_t	address 相对于映射文件的偏移量，以页为单位
address	unsigned long	导致异常的虚拟地址
pmd	pmd_t *	页中级目录项的指针
pud	pud_t *	页上级目录项的指针
orig_pte	pte_t	导致异常时页表项的内容
pte	pte_t *	页表项的指针
cow_page	page *	COW 使用的内存页
page	page *	处理异常函数返回的内存页

显然，进入handle_pte_fault函数前，除了与pte和page相关的字段外，其他多数字段都已经被__handle_mm_fault函数赋值了，它处理到pmd为止。pgoff由计算得来，等于(address-vma->vm_start)>>PAGE_SHIFT加上vma->vm_pgoff。

第1步，判断pmd项是否有效（指向一个页表），无效则属于第一种问题；有效则判断pte是否有效，无效也属于第一种问题，有效则属于后两种。

请注意区分pte_none和!vmf->pte，前者判断页表项的内容是否有效，后者判断pte是否存在。pmd项没有指向页表的情况下后者为真，页表项没有期望的内容时前者为真。

第2步针对第一种问题，非匿名映射由do_fault函数处理。do_fault根据不同的情况调用相应的函数。

如果是读操作导致异常，调用do_read_fault；如果是写操作导致异常，以MAP_PRIVATE映射的内存，调用do_cow_fault；如果是写操作导致异常，以MAP_SHARED映射的内存，调用do_shared_fault。

以上三个do_xxx_fault都会调用__do_fault，后者回调vma->vm_ops->fault函数得到一页内存（vmf->page）；得到内存后，再调用finish_fault函数更新页表完成映射。do_read_fault和do_shared_fault的区别在于内存的读写权限，do_cow_fault与它们的区别在于最终使用的物理页并不是得到的vmf->page，它会重新申请一页内存（vmf->cow_page），将vmf->page复制到vmf->cow_page，然后使用vmf->cow_page更新页表。

此处需要强调两点，首先，vma->vm_ops->fault是由映射时的文件定义的（vma->vm_file），文件需要根据vmf的信息返回一页内存，赋值给vmf->page。其次，do_cow_fault最终使用的物理页是新申请的vmf->cow_page，与文件返回的物理页只是此刻内容相同，此后便没有任何关系，之后写内存并不会更新文件。

第3步，页表项内容有效，但物理页不存在，也就是第二种问题，由do_swap_page函数处理。

第4步，写操作导致异常，但物理页没有写权限，也就是第三种问题，由do_wp_page函数处理。需要注意的是，写映射为只读的内存导致异常的情况已经被__do_page_fault函数的第5步过滤掉了，所以此处的情况是，内存之前被映射为可写，但实际不可写，具体场景读者不必纠结于此，下节详解。

do_wp_page主要处理三种情况，前两种情况见下面的分析，第三种在下节分析。

第一种，以PROT_WRITE（可写）和MAP_SHARED（共享）标志映射的内存，调用wp_pfn_shared函数尝试将其变为可写即可。

第二种，以PROT_WRITE（可写）和MAP_PRIVATE（不共享）标志映射的内存，就是所谓的COW，调用wp_page_copy函数处理。wp_page_copy新申请一页内存，复制原来页中的内容，更新页表，后续写操作只会更新新的内存，与原内存无关。

缺页异常的处理至此结束了，代码本身的逻辑并不是很复杂，难点在于理解代码对应的场景，我们在此列举几个常见的例子方便读者理解。

例1，mmap映射内存，得到虚拟地址，如果实际上并没有物理地址与之对应，访问内存会导致缺页异常，由handle_pte_fault的第2步处理。接下来不同的情况由不同的函数处理，结果也不一样，如表10-4所示。

表10-4 非完整映射情况表

情　况	处 理 函 数	结　果
读访问	do_read_fault	完成映射，不尝试将内存变为可写
写访问，MAP_SHARED 映射内存	do_shared_fault	完成映射，尝试将内存变为可写
写访问，MAP_PRIVATE 映射内存	do_cow_fault	申请一页新的内存，复制内容，完成映射，新内存可写

例2，接例1，读访问导致异常处理后，内存依然不可写，但此时内存映射是完整的，写内存会导致异常，由handle_pte_fault的第4步处理，至于是do_wp_page函数处理的哪种情况由映射的方式决定。

例3，mmap映射内存，得到虚拟地址，并没有物理地址与之对应的情况下，内核调用get_user_pages尝试访问该内存。由于此时内存映射不完整，内核会调用handle_mm_fault完成映射，这种情况下物理页的访问权限由内核的需要决定。如果访问权限为只读，用户空间下次写该内存就会导致异常，处理过程与例2相同。

例4，以MAP_PRIVATE方式映射内存，得到的内存不可写，写内存导致异常，处理过程与例1的情况3类似。

COW的精髓

COW的全称是Copy On Write，也就是写时复制，handle_pte_fault的第2步和第4步都有它的身影。从缺页异常的处理过程来看，缺页异常认定为COW的条件是以MAP_PRIVATE方式映射的内存，映射不完整，或者物理内存权限为只读；写内存（FAULT_FLAG_WRITE）。

由此，我们可以总结出COW的认定条件。

首先，必须是以MAP_PRIVATE方式映射的内存，它的意图是对内存的修改，对其他进程不可见，而以MAP_SHARED方式映射的内存，本意就是与其他进程共享内存，并不存在复制一说，也就不存在COW中的Copy了。

其次，写内存导致异常。有两层含义，首先必须是写，也就是COW中的Write；其次是导致异常，也就是映射不完整或者访问权限不足。映射不完整容易理解，就是得到虚拟内存后，没有实际的物理内存与之对应。至于访问权限不足，上节中的例2到例4都属于这种情况，访问权限不足的场景，与子进程的创建有关。

子进程被创建时，会负责父进程的很多信息，包括部分内存信息，其中复制内存映射信息的任务由dup_mmap函数完成。

dup_mmap函数复制父进程的没有置位VM_DONTCOPY标志的vma，调用copy_page_range函数尝试复制vma的页表项信息。后者访问子进程的四级页目录，如果不存在则申请一页内存并使用它更新目录项，然后复制父进程的页表项，完成映射。这就是复制vma信息的一般过程，不过实际上存在以下几种情况需要特殊处理。

copy_page_range不会复制父进程以MAP_SHARED方式映射普通内存对应的vma。所谓普通内存是相对于MMIO等内存来说的，MMIO的映射信息是需要复制的。这种情况下，子进程得到执行后，访问这段内存时会导致缺页异常，由缺页异常程序来处理。

不复制是基于一个事实：子进程被创建后，多数情况下会执行新的程序，拥有自己的内存空间，父进程的内存它多半不会全部使用，所以创建进程时尽量较少复制。

针对MAP_PRIVATE和PROT_WRITE方式映射的内存，首先，仅复制页表项信息是不够的，因为复制了页表项，子进程和父进程随后可以访问相同的物理内存，这违反了MAP_PRIVATE的要求。其次，如果不复制，由缺页异常来处理，缺页异常会申请新的一页，使用新页完成映射。这看似没有问题，但实际上退化成了映射不完整的场景中，丢失了内存中的数据。

copy_page_range函数针对这种情况的策略就是将页表项的权限降级为只读，复制页表项完成映射。子进程和父进程写这段内存时，就会导致缺页异常，由handle_pte_fault的第4步处理。

COW基本分析完毕，但仍有两个问题值得深思。

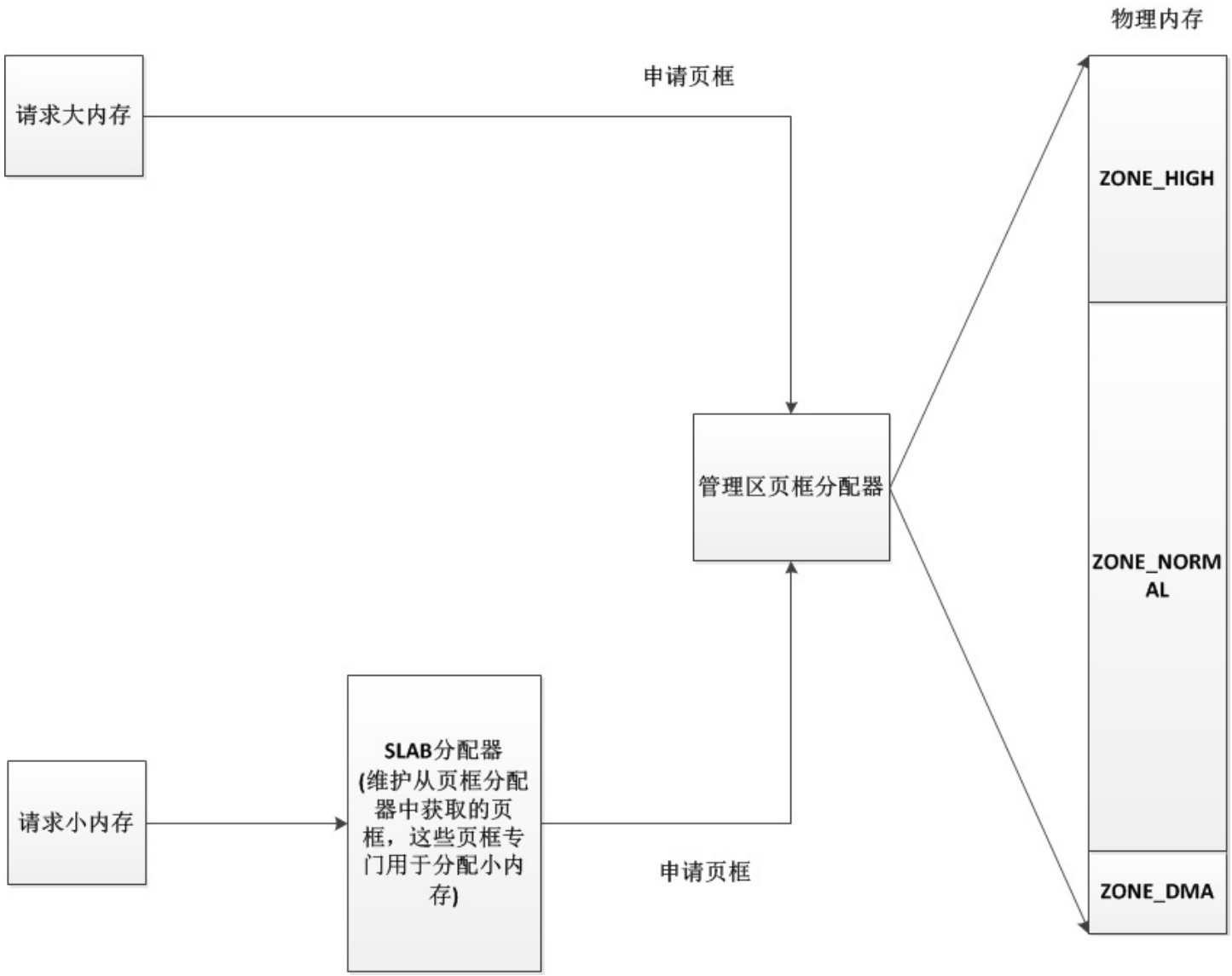
子进程创建后，子进程和父进程写COW的内存都会导致异常，如果父进程先写，就会触发复制，即使子进程不需要写。COW在这种情况下失效，所以子进程先执行对COW更加有利，如果子进程直接执行新的程序，就不需要复制了。内核定义了一个变量sysctl_sched_child_runs_first，它可以控制子进程是否抢占父进程，我们可以通过写/proc/sys/kernel/sched_child_runs_first文件设置它的值。

另外，如果子进程执行新的程序，或者取消了它与COW有关的映射，父进程再次写COW的内存导致缺页异常后还需要复制吗？这取决于拥有COW内存的进程的数量，如果仅剩一个进程映射这段内存，只需要修改页表项将内存标记为可写即可；如果还有其他进程需要访问这段内存，父进程也需要复制内存，这就是do_wp_page处理的第3种情况。当然了，如果父进程放弃了映射，子进程写内存可能同样不需要复制内存，二者在程序上并没有地位上的差别。

存在的问题：实际使用下mmap

四、slab分配器

之前说了管理区页框分配器，这里我们简称为页框分配器，在页框分配器中主要是管理物理内存，将物理内存的页框分配给申请者，而且我们知道也可页框大小为4K(也可设置为4M)，这时候就会有个问题，如果我只需要1KB大小的内存，页框分配器也不得不分配一个4KB的页框给申请者，这样就会有3KB被白白浪费掉了。为了应对这种情况，在页框分配器上一层又做了一层SLAB层，SLAB分配器的作用就是从页框分配器中拿出一些页框，专门把这些页框拆分成一小块一小块的小内存，当申请者申请的是小内存时，系统就会从SLAB中获取一小块分配给申请者。它们的整个关系如下图：



可以看出，SLAB分配器和页框分配器并没有什么直接的联系，对于页框分配器来说，SLAB分配器也只是一个从它那里申请页框的申请者而已。

在SLAB分配器中将SLAB分为两大类：专用SLAB和普通SLAB。专用SLAB用于特定的场合(比如TCP有自己专用的SLAB，当TCP模块需要小内存时，会从自己的SLAB中分配)，而普通SLAB就是用于常规分配的时候。我们可以使用命令查看SLAB的状态

```
cat /proc/slabinfo
```

命令结果如下：

```
radix_tree_node     3536   3536    312   26    2 : tunables    0    0    0 : slabdata    136    136      0
idr_layer_cache      240    240   1072   30    8 : tunables    0    0    0 : slabdata      8      8      0
dma-kmalloc-8192       0      0   8192    4    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-4096       0      0   4096    8    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-2048       0      0   2048   16    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-1024       0      0   1024   32    8 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-512       32     32    512   32    4 : tunables    0    0    0 : slabdata      1      1      0
dma-kmalloc-256        0      0    256   32    2 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-128        0      0    128   32    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-64         0      0     64   64    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-32         0      0     32  128    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-16         0      0     16  256    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-8          0      0      8  512    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-192        0      0    192   21    1 : tunables    0    0    0 : slabdata      0      0      0
dma-kmalloc-96         0      0     96   42    1 : tunables    0    0    0 : slabdata      0      0      0
kmalloc-8192          71     84   8192    4    8 : tunables    0    0    0 : slabdata     21     21      0
kmalloc-4096          68     80   4096    8    8 : tunables    0    0    0 : slabdata     10     10      0
kmalloc-2048         192    192   2048   16    8 : tunables    0    0    0 : slabdata     12     12      0
kmalloc-1024         968   1056   1024   32    8 : tunables    0    0    0 : slabdata     33     33      0
kmalloc-512         1666   1696    512   32    4 : tunables    0    0    0 : slabdata     53     53      0
kmalloc-256         1292   1376    256   32    2 : tunables    0    0    0 : slabdata     43     43      0
kmalloc-192         6977   7161    192   21    1 : tunables    0    0    0 : slabdata    341    341      0
kmalloc-128         1209   1472    128   32    1 : tunables    0    0    0 : slabdata     46     46      0
kmalloc-96         19824  19824     96   42    1 : tunables    0    0    0 : slabdata    472    472      0
kmalloc-64          4672   4672     64   64    1 : tunables    0    0    0 : slabdata     73     73      0
kmalloc-32         19090  19456     32  128    1 : tunables    0    0    0 : slabdata    152    152      0
kmalloc-16         13568  13568     16  256    1 : tunables    0    0    0 : slabdata     53     53      0
kmalloc-8           7680   7680      8  512    1 : tunables    0    0    0 : slabdata     15     15      0
kmem_cache_node      128    128     32  128    1 : tunables    0    0    0 : slabdata      1      1      0
kmem_cache            96     96    128   32    1 : tunables    0    0    0 : slabdata      3      3      0
```

如刚才所有，我们看到有些SLAB的名字比较特别，如 `TCP` , `UDP` , `dquot` 这些，它们都是专用SLAB，专属于它们自己的模块。而后面这张图，如 `kmalloc-8` ， `kmalloc-16` ...还有dma-kmalloc-96，dma-kmalloc-192...这些都是普通SLAB，当需要为一些小数据分配内存时(比如一个结构体)，就会从这些普通SLAB中获取内存。值得注意的是，对于kmalloc-8这些普通SLAB，都有一个对应的dma-kmalloc-8这种类型的普通SLAB，这种类型是专门使用了ZONE-DMA区域的内存，方便用于DMA模式申请内存。

kmem_cache结构

虽然叫SLAB分配器，但是在SLAB分配器中，最顶层的数据结构却不是SLAB，而是 `kmem_cache` ，我们暂且叫它SLAB缓存吧，每个SLAB缓存都有它自己的名字，就是上图中的 `kmalloc-8` ， `kmalloc-16` 等。总的来说， `kmem_cache` 结构用于描述一种SLAB，并且管理着这种SLAB中所有的对象。所有的 `kmem_cache` 结构会保存在以 `slab_caches` 作为头的链表中。在内核模块中可以通过 `kmem_cache_create` 自行创建一个 `kmem_cache` 用于管理属于自己模块的SLAB。

我们先看看kmem_cache结构：

```
1 /* slab分配器中的SLAB高速缓存 */
2 struct kmem_cache {
3     /* 指向包含空闲对象的本地高速缓存，每个CPU有一个该结构，当有对象释放时，优先放入本地CPU
4     struct array_cache __percpu *cpu_cache;
5
6 /* 1) Cache tunables. Protected by slab_mutex */
7     /* 要转移进本地高速缓存或从本地高速缓存中转移出去的对象的数量 */
8     unsigned int batchcount;
9     /* 本地高速缓存中空闲对象的最大数目 */
```

```
10      unsigned int limit;
11      /* 是否存在CPU共享高速缓存，CPU共享高速缓存指针保存在kmem_cache_node结构中 */
12      unsigned int shared;
13
14      /* 对象长度 + 填充字节 */
15      unsigned int size;
16      /* size的倒数，加快计算 */
17      struct reciprocal_value reciprocal_buffer_size;
18
19
20  /* 2) touched by every alloc & free from the backend */
21      /* 高速缓存永久属性的标识，如果SLAB描述符放在外部(不放在SLAB中)，则CFLAGS_OFF_SLAB置
22      unsigned int flags;         /* constant flags */
23      /* 每个SLAB中对象的个数(在同一个高速缓存中slab中对象个数相同) */
24      unsigned int num;         /* # of objs per slab */
25
26
27  /* 3) cache_grow/shrink */
28      /* 一个单独SLAB中包含的连续页框数目的对数 */
29      unsigned int gfporder;
30
31      /* 分配页框时传递给伙伴系统的一组标识 */
32      gfp_t allocflags;
33
34      /* SLAB使用的颜色个数 */
35      size_t colour;
36      /* SLAB中基本对齐偏移，当新SLAB着色时，偏移量的值需要乘上这个基本对齐偏移量，理解就是1
37      unsigned int colour_off;
38      /* 空闲对象链表放在外部时使用，其指向的SLAB高速缓存来存储空闲对象链表 */
39      struct kmem_cache *freelist_cache;
40      /* 空闲对象链表的大小 */
41      unsigned int freelist_size;
42
43      /* 构造函数，一般用于初始化这个SLAB高速缓存中的对象 */
44      void (*ctor)(void *obj);
45
46
47  /* 4) cache creation/removal */
48      /* 存放高速缓存名字 */
49      const char *name;
50      /* 高速缓存描述符双向链表指针 */
51      struct list_head list;
52      int refcount;
53      /* 高速缓存中对象的大小 */
54      int object_size;
55      int align;
56
```

```
57
58  /* 5) statistics */
59      /* 统计 */
60  #ifdef CONFIG_DEBUG_SLAB
61      unsigned long num_active;
62      unsigned long num_allocations;
63      unsigned long high_mark;
64      unsigned long grown;
65      unsigned long reaped;
66      unsigned long errors;
67      unsigned long max_freeable;
68      unsigned long node_allocs;
69      unsigned long node_frees;
70      unsigned long node_overflow;
71      atomic_t allochit;
72      atomic_t allocmiss;
73      atomic_t freehit;
74      atomic_t freemiss;
75
76      /* 对象间的偏移 */
77      int obj_offset;
78  #endif /* CONFIG_DEBUG_SLAB */
79  #ifdef CONFIG_MEMCG_KMEM
80      /* 用于分组资源限制 */
81      struct memcg_cache_params *memcg_params;
82  #endif
83      /* 结点链表，此高速缓存可能在不同NUMA的结点都有SLAB链表 */
84      struct kmem_cache_node *node[MAX_NUMNODES];
85  };
```

从结构中可以看出，在这个kmem_cache中所有对象的大小是相同的(object_size)，并且此kmem_cache中所有SLAB的大小也是相同的(gfporder、num)。

在这个结构中，最重要的可能就属 `struct kmem_cache_node * node[Max_NUMNODES]` 这个指针数组了，指向的 `struct kmem_cache_node` 中保存着slab链表，在NUMA架构中每个node对应数组中的一个元素，因为每个SLAB高速缓存都有可能在不同结点维护有自己的SLAB用于这个结点的分配。我们看看 `struct kmem_cache_node`

```
1  /* SLAB链表结构 */
2  struct kmem_cache_node {
3      /* 锁 */
4      spinlock_t list_lock;
5
6  /* SLAB用 */
7  #ifdef CONFIG_SLAB
```

```
    /* 只使用了部分对象的SLAB描述符的双向循环链表 */
    struct list_head slabs_partial;    /* partial list first, better asm code */
    /* 不包含空闲对象的SLAB描述符的双向循环链表 */
    struct list_head slabs_full;
    /* 只包含空闲对象的SLAB描述符的双向循环链表 */
    struct list_head slabs_free;
    /* 高速缓存中空闲对象个数(包括slabs_partial链表中和slabs_free链表中所有的空闲对象) */
    unsigned long free_objects;
    /* 高速缓存中空闲对象的上限 */
    unsigned int free_limit;
    /* 下一个被分配的SLAB使用的颜色 */
    unsigned int colour_next;    /* Per-node cache coloring */
    /* 指向这个结点上所有CPU共享的一个本地高速缓存 */
    struct array_cache *shared;    /* shared per node */
    struct alien_cache **alien;    /* on other nodes */
    /* 两次缓存收缩时的间隔，降低次数，提高性能 */
    unsigned long next_reap;
    /* 0:收缩  1:获取一个对象 */
    int free_touched;        /* updated without locking */
#endif

/* SLUB用 */
#ifdef CONFIG_SLUB
    unsigned long nr_partial;
    struct list_head partial;
#ifdef CONFIG_SLUB_DEBUG
    atomic_long_t nr_slabs;
    atomic_long_t total_objects;
    struct list_head full;
#endif
#endif

};
```

在这个结构中，最重要的就是`slabs_partial`、`slabs_full`、`slabs_free`这三个链表头。

- slabs_partial：维护部分对象被使用了的SLAB链表，保存的是SLAB描述符。
- slabs_full：维护所有对象都被使用了的SLAB链表，保存的是SLAB描述符。
- slabs_free：维护所有对象都没被使用的SLAB链表，保存的是SLAB描述符。

可能到这里大家会比较郁闷，怎么又有SLAB链表，SLAB到底是什么东西？SLAB就是一组连续的页框，它的描述符结合在页描述符（struct page）中，也就是页描述符描述SLAB的时候，就是SLAB描述符。这三个链表保存的是这组页框的首页框的SLAB描述符。链表的组织形式与伙伴系统的组织页框的形式一样。

刚开始创建kmem_cache完成后，这三个链表都为空，只有在申请对象时发现没有可用的slab时才会创建一个新的SLAB，并加入到这三个链表中的一个中。也就是说kmem_cache中的SLAB数量是动态变化的，当SLAB数量太多时，kmem_cache会将一些SLAB释放回页框分配器中。

我们看看SLAB描述符中相关字段：

```
struct page {
    /* First double word block */
    /* 用于页描述符，一组标志(如PG_locked、PG_error)，也对页框所在的管理区和node进行编号
    unsigned long flags; /
    union {
        /* 用于页描述符，当页被插入页高速缓存中时使用，或者当页属于匿名区时使用 */
        struct address_space *mapping;
        /* 用于SLAB描述符，指向第一个对象的地址 */
        void *s_mem;            /* slab first object */
    };


    /* Second double word */
    struct {
        union {
            /* 作为不同的含义被几种内核成分使用。例如，它在页磁盘映像或匿名区中标识存放在
            pgoff_t index;        /* Our offset within mapping. */
            /* 用于SLAB描述符，指向空闲对象链表 */
            void *freelist;
            /* 当管理区页框分配器压力过大时，设置这个标志就确保这个页框专门用于释放其他页
            bool pfmemalloc;
        };

        union {
#if defined(CONFIG_HAVE_CMPXCHG_DOUBLE) && \
    defined(CONFIG_HAVE_ALIGNED_STRUCT_PAGE)
            /* Used for cmpxchg_double in slub */
            /* SLUB使用 */
            unsigned long counters;
#else
            /* SLUB使用 */
            unsigned counters;
#endif

            struct {

                union {
                    /* 页框中的页表项计数，如果没有为-1，如果为PAGE_BUDDY_MAPCOUNT_VA
                    atomic_t _mapcount;

```

```
41                      struct { /* SLUB使用 */
42                          unsigned inuse:16;
43                          unsigned objects:15;
44                          unsigned frozen:1;
45                      };
46                      int units;    /* SLOB */
47                  };
48                  /* 页框的引用计数，如果为-1，则此页框空闲，并可分配给任一进程或内核；如果
49                  atomic_t _count;        /* Usage count, see below. */
50              };
51              /* 用于SLAB时描述当前SLAB已经使用的对象 */
52              unsigned int active;    /* SLAB */
53          };
54      };
55
56
57      /* Third double word block */
58      union {
59          /* 包含到页的最近最少使用(LRU)双向链表的指针，用于插入伙伴系统的空闲链表中，只有块
60          struct list_head lru;
61
62
63          /* SLAB使用 */
64          struct {        /* slub per cpu partial pages */
65              struct page *next;    /* Next partial slab */
66  #ifdef CONFIG_64BIT
67              int pages;    /* Nr of partial slabs left */
68              int pobjects;    /* Approximate # of objects */
69  #else
70              short int pages;
71              short int pobjects;
72  #endif
73          };
74
75          /* SLAB使用 */
76          struct slab *slab_page; /* slab fields */
77          struct rcu_head rcu_head;    /* Used by SLAB
78                                        * when destroying via RCU
79                                        */
80  #if defined(CONFIG_TRANSPARENT_HUGEPAGE) && USE_SPLIT_PMD_PTLOCKS
81          pgtable_t pmd_huge_pte; /* protected by page->ptl */
82  #endif
83      };
84
85
86      /* Remainder is not double word aligned */
87      union {
```

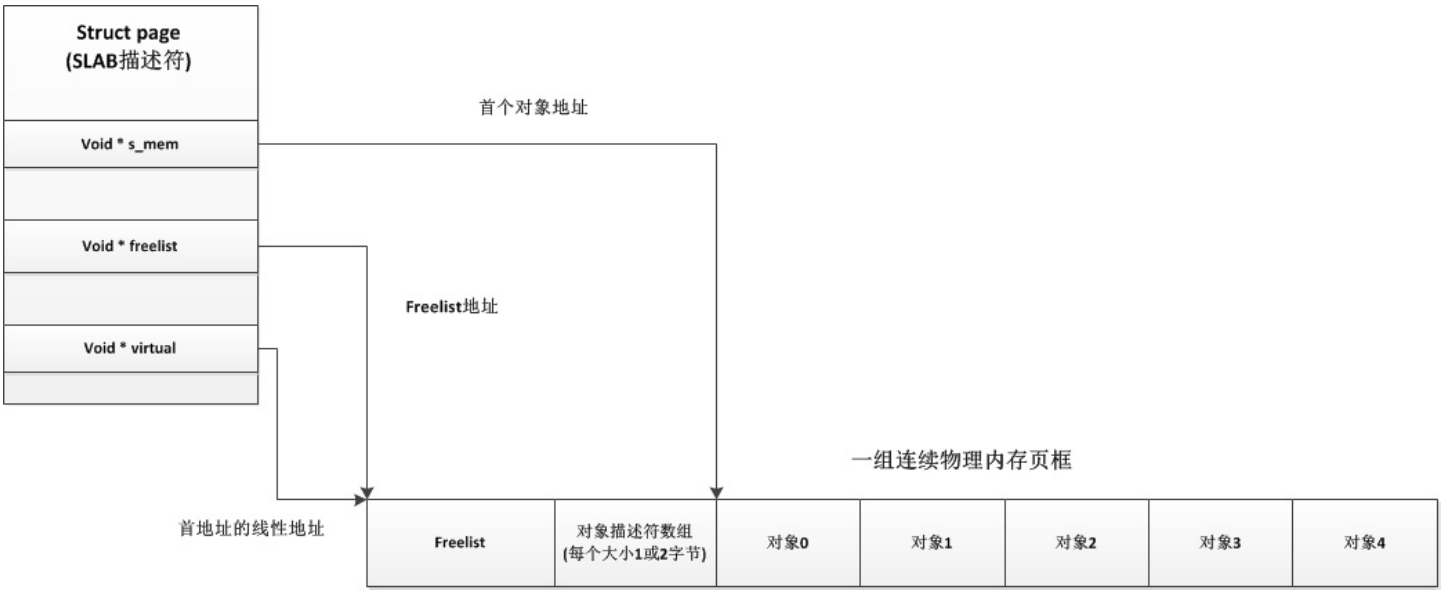
```
88          /* 可用于正在使用页的内核成分(例如：在缓冲页的情况下它是一个缓冲器头指针，如果页是
89          unsigned long private;
90  #if USE_SPLIT_PTE_PTLOCKS
91  #if ALLOC_SPLIT_PTLOCKS
92          spinlock_t *ptl;
93  #else
94          spinlock_t ptl;
95  #endif
96  #endif
97          /* SLAB描述符使用，指向SLAB的高速缓存 */
98          struct kmem_cache *slab_cache;    /* SL[AU]B: Pointer to slab */
99          struct page *first_page;    /* Compound tail pages */
100     };
101
102 #if defined(WANT_PAGE_VIRTUAL)
103     /* 线性地址，如果是没有映射的高端内存的页框，则为空 */
104     void *virtual;            /* Kernel virtual address (NULL if
105                          not kmapped, ie. highmem) */
106 #endif /* WANT_PAGE_VIRTUAL */
107 #ifdef CONFIG_WANT_PAGE_DEBUG_FLAGS
108     unsigned long debug_flags;    /* Use atomic bitops on this */
109 #endif
110
111 #ifdef CONFIG_KMEMCHECK
112     void *shadow;
113 #endif
114
115 #ifdef LAST_CPUPID_NOT_IN_PAGE_FLAGS
116     int _last_cpupid;
117 #endif
118 }
```

在SLAB描述符中，最重要的可能就是s_mem和freelist这两个指针。s_mem用于指向这段连续页框中第一个对象，freelist指向空闲对象链表。

空闲对象链表是一个由数组制成的简单链表，它保存的地方有两种情况：

- 空闲对象链表是一个由数组制成的简单链表，它保存的地方有两种情况：
- 保存在内部，保存在这个SLAB所代表的连续页框的头部。

不过一般没有什么其他情况空闲对象链表都是保存在内部居多，这里我们只讨论将空闲对象链表保存在内部的情况，这种情况下，这个SLAB所代表的连续页框的头部首先放的就是空闲对象链表，后面接着放的是对象描述符数组(1,2个字节大小)，之后紧接着就是对象所代表的内存了，如下图：



我们看看freelist数组是怎么形成一个链表的，之前我们也说了分配时会优先分配最近释放的对象，整个freelist跟struct page中的active有很大联系，可以说active决定了下个分配的对象是谁，在freelist数组制作成的链表中，active作为下标，保存目标空闲对象的对象号，在活动过程中，动态修改这个数组中的值。我们用一幅图可以很清楚看出freelist是如何实现：

初始化阶段:各个数组中的值设置为对象号，并初始**active = 0**

分配阶段:分配时会根据**active**值在数组中获取对象号，然后**active++**，这里模拟已经分配了**4**个对象，下次分配的对象将是**freelist[4]**中的对象**4**

释放阶段:假定此时释放了对象**0**，会先将**active--**，然后在对应数组元素中写入释放对象的对象号，在下次分配时就会分配到对象**0**

SLAB中的连续页框个数与kmem_cache结构中的gfporder有关，而这个gfporder在初始化时通过对象数量、大小、freelist大小、对象描述符数组大小和着色区计算出来的。而对于对象的大小，也并不是你创建时打算使用的大小，比如，我打算创建一个kmem_cache的对象大小是10字节，而在创建过程中，系统会帮你优化和初始化这些对象，包括将你的对象保存地址放在内存对其标志，在对象的两边放入一些填充区域(RED_ZONE)进行防止越界等工作。

关于SLAB着色:

看名字很难理解，其实又很好理解，我们知道内存需要处理时要先放入CPU硬件高速缓存中，而CPU硬件高速缓存与内存的映射方式有多种。在同一个kmem_cache中所有SLAB都是相同大小，都是相同

连续长度的页框组成，这样的话在不同SLAB中相同对象号对于页框的首地址的偏移量也相同，这样有很可能导致不同SLAB中相同对象号的对象放入CPU硬件高速缓存时会处于同一行，当我们交替操作这两个对象时，CPU的cache就会交替换入换出，效率就非常差。SLAB着色就是在同一个kmem_cache中对不同的SLAB添加一个偏移量，就让相同对象号的对象不会对齐，也就不会放入硬件高速缓存的同一行中，提高了效率，如下图：

kmalloc-8(未着色)

SLAB0

Freelist	对象描述符数组 (每个大小1或2字节)	对象0	对象1	对象2	对象3	对象4	空闲区域

SLAB1

Freelist	对象描述符数组 (每个大小1或2字节)	对象0	对象1	对象2	对象3	对象4	空闲区域

kmalloc-8(着色)

SLAB0

Freelist	对象描述符数组 (每个大小1或2字节)	对象0	对象1	对象2	对象3	对象4	空闲区域

SLAB1

空闲区域	Freelist	对象描述符数组 (每个大小1或2字节)	对象0	对象1	对象2	对象3	对象4	空闲区域

着色空间就是前端的空闲区域，这个区有大小都是在分配新的SLAB时计算好的，计算方法很简单，node结点对应的kmem_cache_node中的colour_next乘上kmem_cache中的colour_off就得到了偏移量，然后colour_next++，当colour_next等于kmem_cache中的colour时，colour_next回归到0。

```
1 偏移量 = kmem_cache.colour_off * kmem_cache.node[NODE_ID].colour_next;
2
3     kmem_cache.node[NODE_ID].colour_next++;
4     if (kmem_cache.node[NODE_ID].colour_next == kmem_cache.colour)
5         kmem_cache.node[NODE_ID].colour_next = 0;
```

本地CPU空闲对象链表

现在说说本地CPU空闲对象链表。这个在kmem_cache结构中用cpu_cache表示，整个数据结构是struct array_cache，它的目的是将释放的对象加入到这个链表中，我们可以先看看数据结构：

```
struct array_cache {
    /* 可用对象数目 */
    unsigned int avail;
    /* 可拥有的最大对象数目，和kmem_cache中一样 */
    unsigned int limit;
    /* 同kmem_cache，要转移进本地高速缓存或从本地高速缓存中转移出去的对象的数量 */
    unsigned int batchcount;
    /* 是否在收缩后被访问过 */
    unsigned int touched;
    /* 伪数组，初始没有任何数据项，之后会增加并保存释放的对象指针 */
    void *entry[];    /*
};
```

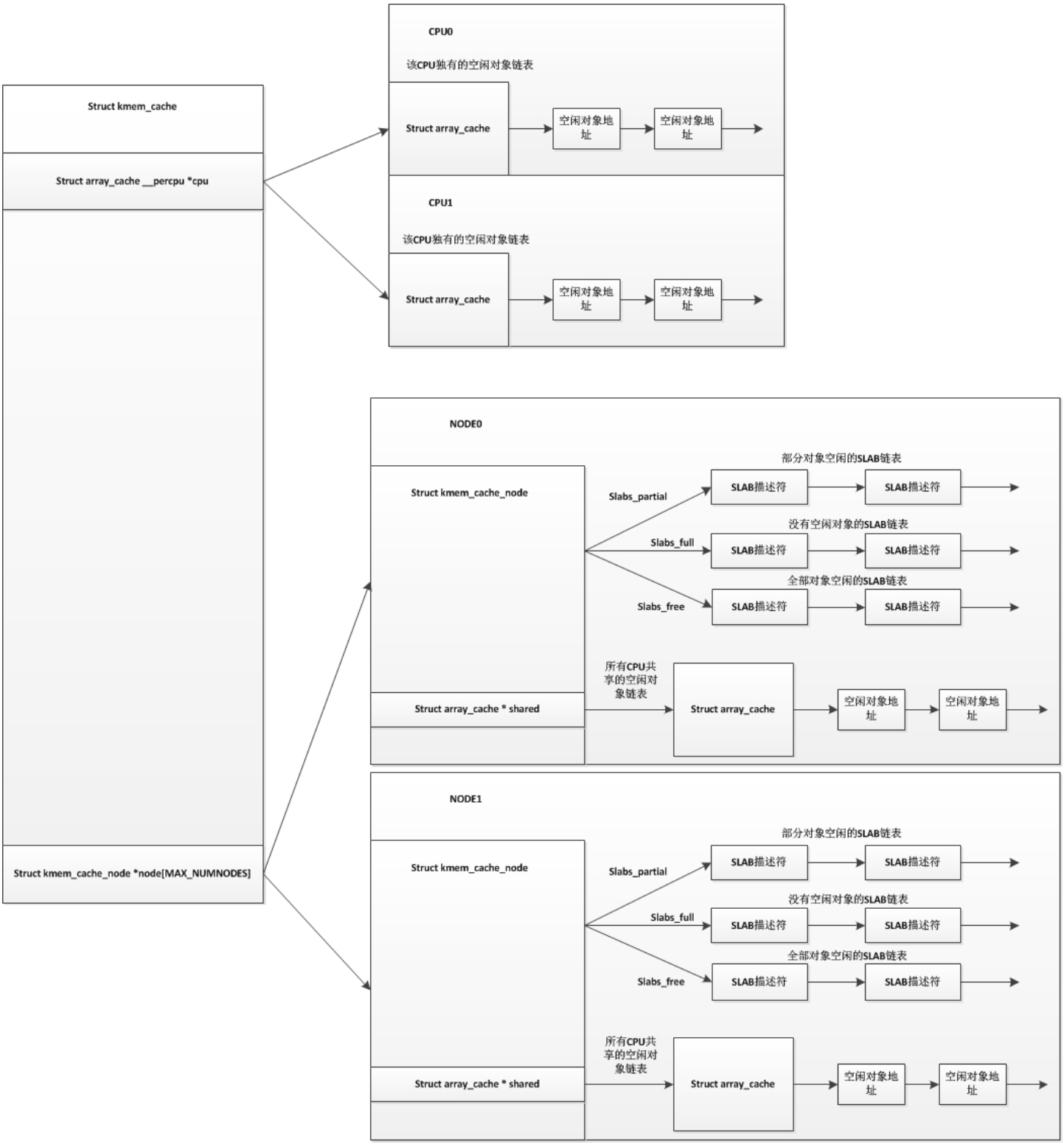
因为每个CPU都有它们自己的硬件高速缓存，当此CPU上释放对象时，可能这个对象很可能还在这个CPU的硬件高速缓存中，所以内核为每个CPU维护一个这样的链表，当需要新的对象时，会优先尝试从当前CPU的本地CPU空闲对象链表获取相应大小的对象。这个本地CPU空闲对象链表在系统初始化完成后是一个空的链表，只有释放对象时才会将对象加入这个链表。当然，链表对象个数也是有所限制，其最大值就是limit，链表数超过这个值时，会将batchcount个数的对象返回到所有CPU共享的空闲对象链表(也是这样一个结构)中。

注意在array_cache中有一个entry数组，里面保存的是指向空闲对象的首地址的指针，注意这个链表是在kmem_cache结构中的，也就是kmalloc-8有它自己的本地CPU高速缓存链表，dquot也有它自己的本地CPU高速缓存链表，每种类型kmem_cache都有它自己的本地CPU空闲对象链表。

有CPU共享的空闲对象链表

原理和本地CPU空闲对象链表一样，唯一的区别就是所有CPU都可以从这个链表中获取对象，一个常规的对象申请流程是这样的：**系统首先会从本地CPU空闲对象链表中尝试获取一个对象用于分配；如果失败，则尝试来到所有CPU共享的空闲对象链表链表中尝试获取；如果还是失败，就会从SLAB中分配一个；这时如果还失败，kmem_cache会尝试从页框分配器中获取一组连续的页框建立一个新的SLAB，然后从新的SLAB中获取一个对象。对象释放过程也类似，首先会先将对象释放到本地CPU空闲对象链表中，如果本地CPU空闲对象链表中对象过多，kmem_cache会将本地CPU空闲对象链表中的batchcount个对象移动到所有CPU共享的空闲对象链表链表中，如果所有CPU共享的空闲对象链表链表的对象也太多了，kmem_cache也会把所有CPU共享的空闲对象链表链表中batchcount个数的对象移回它们自己所属的SLAB中，这时如果SLAB中空闲对象太多，kmem_cache会整理出一些空闲的SLAB，将这些SLAB所占用的页框释放回页框分配器中。**

这个所有CPU共享的空闲对象链表也不是肯定会有的，kmem_cache中有个shared字段如果为1，则这个kmem_cache有这个高速缓存，如果为0则没有。



五、分页

多级页表

硬件上可以有多种分页方式，32位的地址可以分成多种级别来解释。Linux为了兼容32位和64位的平台和各种分页方式，3.10版内核统一将线性地址分成4个级别来解释，分别叫作页全局目录（PageGlobalDirectory，PGD）、页上级目录（PageUpperDirectory，PUD）、页中级目录（PageMiddleDirectory，PMD）和页表（PageTable，PT），5.05版内核在PGD和PUD之间添加了P4D，总共5级。

对应常规分页（2级），5个级别可以解释的偏移位数分别为10、0、0、0和10，PUD、P4D和PMD的项数都为1；对应PAE的4K页框分页（2+9+9），5个级别可以解释的偏移分别为2、0、0、9和9，P4D和PUD的项数为1；对应PAE的2M页框分页（2+9），5个级别可以解释的偏移位数分别为2、0、0、0和9。5个级别的优先级分别为页全局目录、页表、页中级目录、页上局目录、页四级目录。

值得一提的是，PAE和常规分页并不能共存，开启PAE需要设置CPU的寄存器。软件上也是如此，内核以CONFIG_X86_PAE来确定编译时是否引入PAE功能。

内核提供了丰富的函数协助完成分页（映射）工作，为了更好地理解各函数的含义，首先介绍几个常见的内存模块缩写，如表7-1所示。

表7-1 内存模块缩写表

缩　写	描　述
pgd、p4d、pud、pmd	PGD、P4D、PUD、PMD PGD、P4D、PUD、PMD 的项
pte	page table entry，页表项
pfn	pageframenumber，页框号 按照 4K 每个页框，从 0 到物理内存最大值依次编号
pg、ofs	page、offset
PTRS	项的数目

配合上面的缩写，内核提供了很多get/set函数和宏，如表7-2所示。

表7-2 页表属性函数表

函 数 和 宏	描　述
xxx_index	xxx 可以是 pgd、p4d、pud、pmd、pte 表示线性地址在对应级别内的偏移量
xxx_offset	xxx 可以是 pgd、p4d、pud 和 pmd 用来获取该级别目标偏移量对应的项的指针
pte_offset_kernel	意义同上
set_xxx	xxx 可以是 pgd、p4d、pud、pmd、pte 设置项
xxx_pfn	xxx 可以是 pgd、p4d、pud、pmd、pte 获取项指向的页框号
pfn_xxx	xxx 可以是 pud、pmd、pte 根据 pfn 和属性生成项的值，利用 __xxx 完成
__xxx	xxx 可以是 pgd、p4d、pud、pmd、pte 根据一个整数值生成项的值
xxx_val	xxx 可以是 pgd、p4d、pud、pmd、pte 根据项的值生成一个整数
xxx_clear	xxx 可以是 pgd、p4d、pud、pmd、pte 清除项的值

注意，应用程序中使用的指针为虚拟地址，以上函数中使用的参数和返回值中的指针也不例外，但项中存放的是物理地址。不同的条件下，各宏的实际意义也不相同，比如在PAE条件下，pmd_offset返回的是pmd中的项的指针，但如果是常规分页，它会直接返回上一级指针。

利用以上函数和宏，基本可以完成软件上的分页任务。比如某一个4K的页框，页框号为0x12，它的物理地址就为0x12000，采用常规分页的情况下设置它对应的映射的代码如下。

```
//pgd pud pmd and pteare all pointers, pfn is 0x12
//pgd
pgd_idx = pgd_index((pfn<<PAGE_SHIFT) + PAGE_OFFSET);
pgd = pgd_base + pgd_idx;

// p4d pud and pmd
p4d = p4d_offset(pgd, 0);
pud = pud_offset(p4d, 0);
pmd = pmd_offset(pud, 0);

//page table
pte_ofs = pte_index((pfn<<PAGE_SHIFT) + PAGE_OFFSET);
if (!(pmd_val(*pmd) & _PAGE_PRESENT)) {
    pte_t * page_table = (pte_t *)alloc_low_page();
    set_pmd(pmd, __pmd(__pa(page_table) | _PAGE_TABLE));
}
pte = pte_offset_kernel(pmd, pte_ofs);
set_pte(pte, pfn_pte(pfn, prot));
```

该代码段以常规分页为例，演示了一般流程。xxx_index的参数为线性地址，公式中的PAGE_SHIFT和PAGE_OFFSET的作用详见下文。由于是常规分页，p4d_offset、pud_offset和pmd_offset直接返回它们的第一个项，所以p4d、pud、pmd与pgd的值相等。如果pte对应的页框还没有分配，通过alloc_low_page分配一个页框，并利用其物理地址设置pmd的项。最后，调用set_pte设置页表的项。

完成了各级项的设置之后，虚拟地址0xc0012000与0x12页框之间的映射就完成了。假设程序中需要访问0xc0012001c地址，MMU会定位到0x12页框的偏移量为0x1c的字节。

这段代码中唯一将pfn（0x12）直接作为物理地址使用的地方是在set_pte，所以无论pgd、pud和pmd的位置和内容如何，只要最终pte的参数没有变，那么MMU最终定位到的就是同一个页框。“殊途同归”，又一次阐述了内存共享。这是内存共享的原理，重要的事情还是要多说几遍。

TLB

在80x86Linux虚拟机中MMU还包含一个称为转换后援缓冲器或TLB的高速缓存用于加速虚拟地址的线性转换，当一个虚拟地址第一次使用时，通过慢访问内存中的页表计算出相应的物理地址。同时，物理地址被存放在一个TLB表项中，以便以后对同一虚拟地址的引用可以快速得到转换。

在多处理系统中（每个CPU都对应一个MMU内存管理单元），每个CPU都有自己的TLB，这叫做该CPU的本地TLB。多个CPU之间的TLB中对应项不必同步，这是因为运行在各个CPU上的不同进程可能虚拟地址相同但映射的物理地址不同，因为每个进程都有单独的页表。

当CPU的cr3控制器被修改时（进程切换时会修改），内核使当前CPU中的TLB失效，这是因为新的一组页表被启用而TLB指向的是旧数据。CPU不能自动刷新它们自己TLB，因为决定虚拟地址和物理地址直接何时不再有效的是内核而不是硬件。

TLB的刷新机制

TLB刷新的时机

- 改变内核维持的页表时，由于内核页表作为所有进程内核虚拟地址的映射页表模板，所有影响所有进程的页表，所以此时全部CPU的TBL失效刷新。
- 更换一个范围内内核页表时，由于内核页表作为所有进程内核虚拟地址的映射页表模板，所有影响所有进程的页表，所以此时全部CPU的指定虚拟地址范围内的TLB失效刷新。
- 执行进程切换时（cr3被修改），运行当前进程的CPU中非全局页（全局页是指内核地址空间相关页，非全局指进程私有地址空间相关页）相关的TLB失效刷新。
- 触发缺页异常时，运行当前进程的CPU中单个页表相关的TLB失效刷新。

cr3寄存器被改变但不触发TLB失效的特殊情况

- 在进程切换时一般情况下会将CPU的cr3寄存器的值修改为新进程的页全局目录的地址，且当前CPU的TLB会被刷新。不过内核在下列情况下进行进程切换时会避免修改cr3刷新TLB：
- 相同进程里的线程切换时不会修改cr3寄存器
- 当两个使用相同页表集的普通进程直接进行系统切换时，由于页表集相同，所以不会修改cr3
- 当一个普通进程和一个内核线程间执行进程切换时，内核线程并不拥有自己的页表集，更确切的说内核线程使用刚在CPU上执行过的普通进程的页表集，所以不会修改cr3

六、不同接口的使用

常用类型标志

GFP_KERNEL	常用标志之一、可能会被阻塞，即分配过程中可能会睡眠
GFP_ATOMIC	不能睡眠，且保证会分配成功，可以访问系统预留内存
GFP_NOWAIT	分配中不允许睡眠等待
GFP_NOFS	不会访问任何的文件系统的接口和操作
GFP_NOIO	不需要启动任何的IO操作。如使用直接回收机制丢弃干净的页面或者为slab分配的页面
GFP_USER	通常用户空间的进程用来分配内存，这些内存可以被内核或者硬件使用。常用的一个场景是硬件使用的DMA缓冲器要映射到用户空间，如显卡的缓冲器
GFP_DMA GFP_DMA32	使用ZONE_DMA或者ZONE_DMA32来分配内存
GFP_HIGHUSER	用户空间进程用来分配内存，优先使用ZONE_HIGHMEM，这些内存可以映射到用户空间，内核空间不会直接访问这些内存。另外，这些内存不能迁移。
GFP_HIGHUSER_MOVABLE	类似于GFP_HIGHUSER，但页面可以迁移

GFP_TRANSHUGE GFP_TRANSHUGE_LIGHT	通常用于透明页面的分配

kmalloc

kmalloc与kfree相对应；

```
1 static __always_inline void *kmalloc(size_t size, gfp_t flags);
```

```
1 void kfree(const void *objp);
```

vmalloc

vmalloc与vfree相对应；

```
1 void *vmalloc(unsigned long size);
```

```
1 void vfree(const void *addr);
```

__get_free_pages

不能用于高端内存，返回虚拟地址

```
1 unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);
```

__vmalloc

```
1 void *__vmalloc(unsigned long size, gfp_t gfp_mask, pgprot_t prot)
```

prot:分配内存的内存属性

alloc_pages

free_pages和__free_pages是有区别的，与alloc_pages对应的是__free_pages，而不是前者

```
1 static inline struct page *alloc_pages(gfp_t gfp_mask, unsigned int order);
```

```
1 void __free_pages(struct page *page, unsigned int order){
```

```
1 void free_pages(unsigned long addr, unsigned int order){
```

free_pages的参数为虚拟地址addr和order，addr经过计算得到page，然后调用__free_pages继续释放内存。由addr计算得到page，只有映射到直接映射区的Low Memory才可以满足，所以free_pages只能用于直接映射区的LowMemory