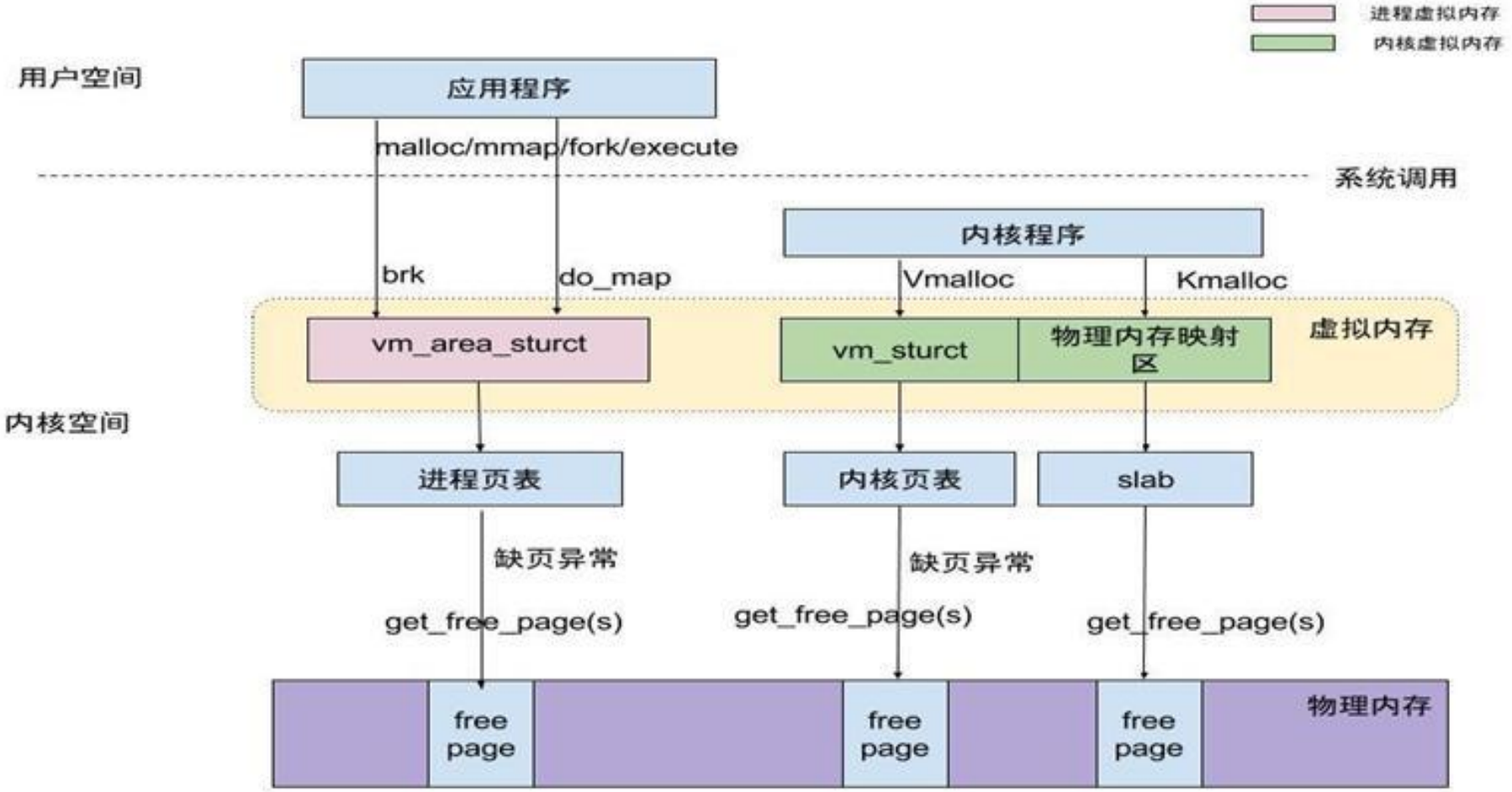
4.4 内存分配与回收机制（二）

西安邮电大学

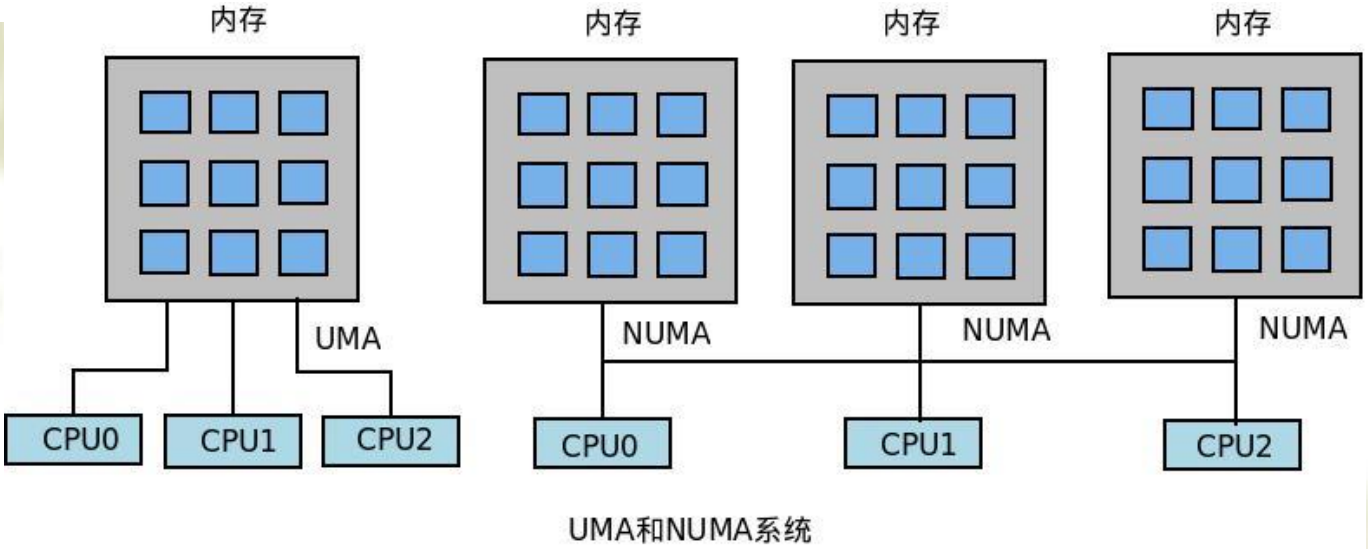
内存分配


进程虚拟内存
内核虚拟内存
用户空间
应用程序
malloc/mmap/fork/execute
系统调用
内核程序
brk
do_map
Vmalloc
Kmalloc
vm_area_sturct
vm_sturct
物理内存映射区
虚拟内存
内核空间
进程页表
内核页表
slab
缺页异常
缺页异常
get_free_page(s)
get_free_page(s)
get_free_page(s)
free page
free page
free page
物理内存

内存分配

从图可以看出，从用户进程发出内存分配请求，到内核最终分配物理内存，这中间内核要做大量的工作，上一讲概要介绍了vmalloc()和kmalloc()，最终都要调用伙伴算法，通过get_free_page() 内核函数获得物理内存。

UMA和NUMA计算机的内存管理方式



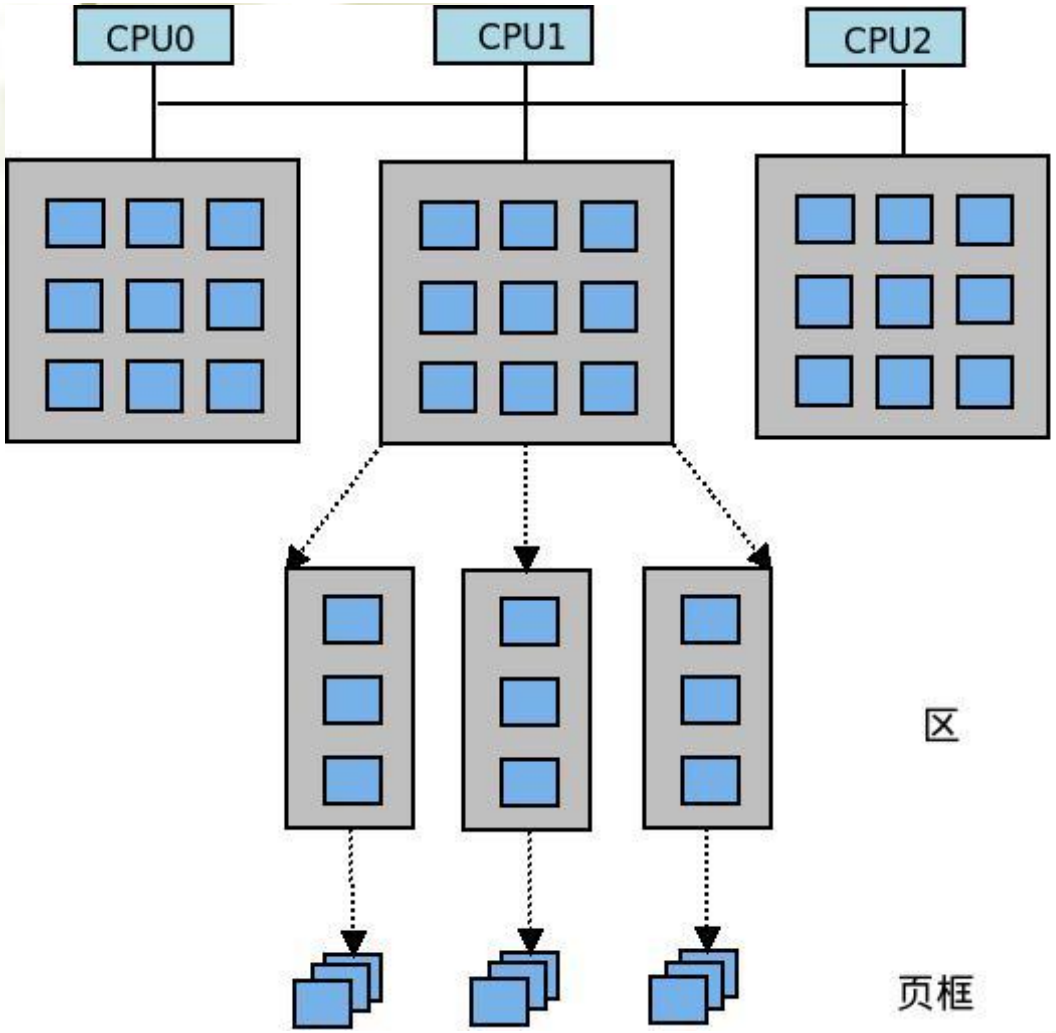
UMA和NUMA系统

目前有两种类型的计算机，分别以不同的方法管理物理内存：

NUMA计算机（non-uniform memory access）：是一种多处理器计算机，每个CPU拥有各自的本地内存。这样的划分使每个CPU都能以较快的速度访问本地内存，各个CPU之间通过总线连接起来，这样也可以访问其他CPU的本地内存，只不过速度比较慢而已；

UMA计算机（uniform memory access）：将可用内存以连续方式组织起来。如图所示。

物理内存的组织



为了兼容NUMA模型，内核引入了内存节点，每个节点关联一个CPU。各个节点又被划分几个内存区，每个内存区中又包含若干个页框。

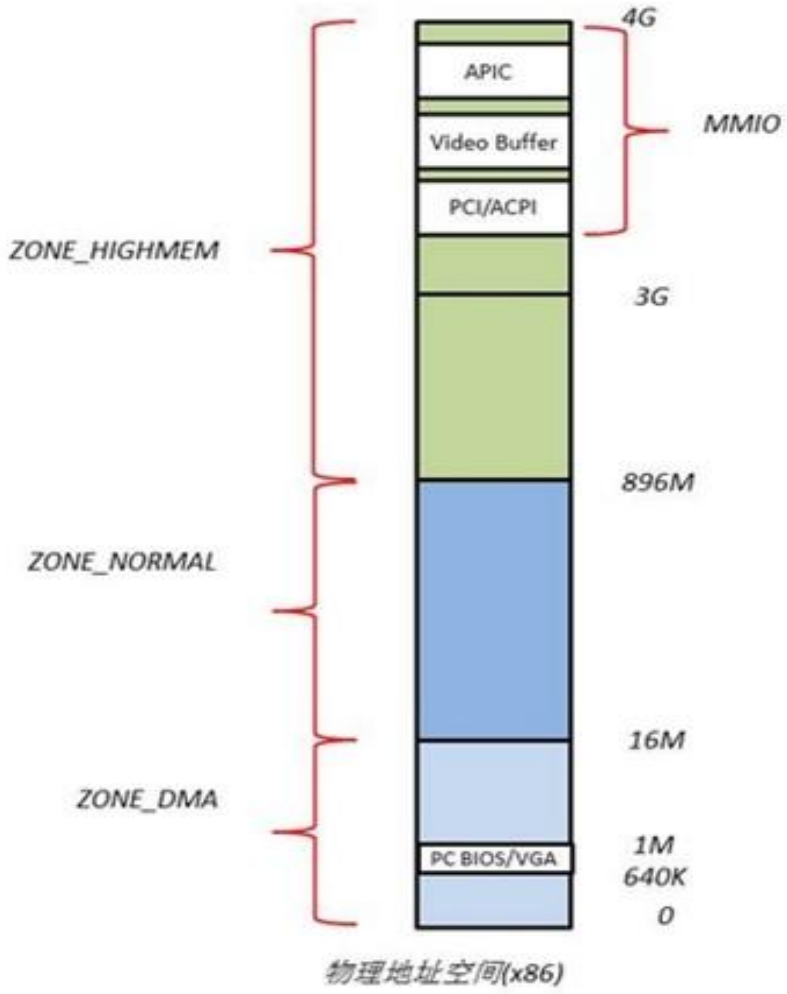
物理内存在逻辑上被划分为三级结构，分别使用pg_data_t，zone和page这三种数据结构依次描述节点，区和页框。

内存节点

NUMA计算机中每个CPU的物理内存称为一个内存节点，内核通过pg_data_t数据结构来描述一个内存节点，系统内的所有结点形成一个双链表。

UMA模型下的物理内存只对应一个节点，也就是说整个物理内存形成一个节点，因此上述的节点链表中也就只有一个元素。

内存管理区



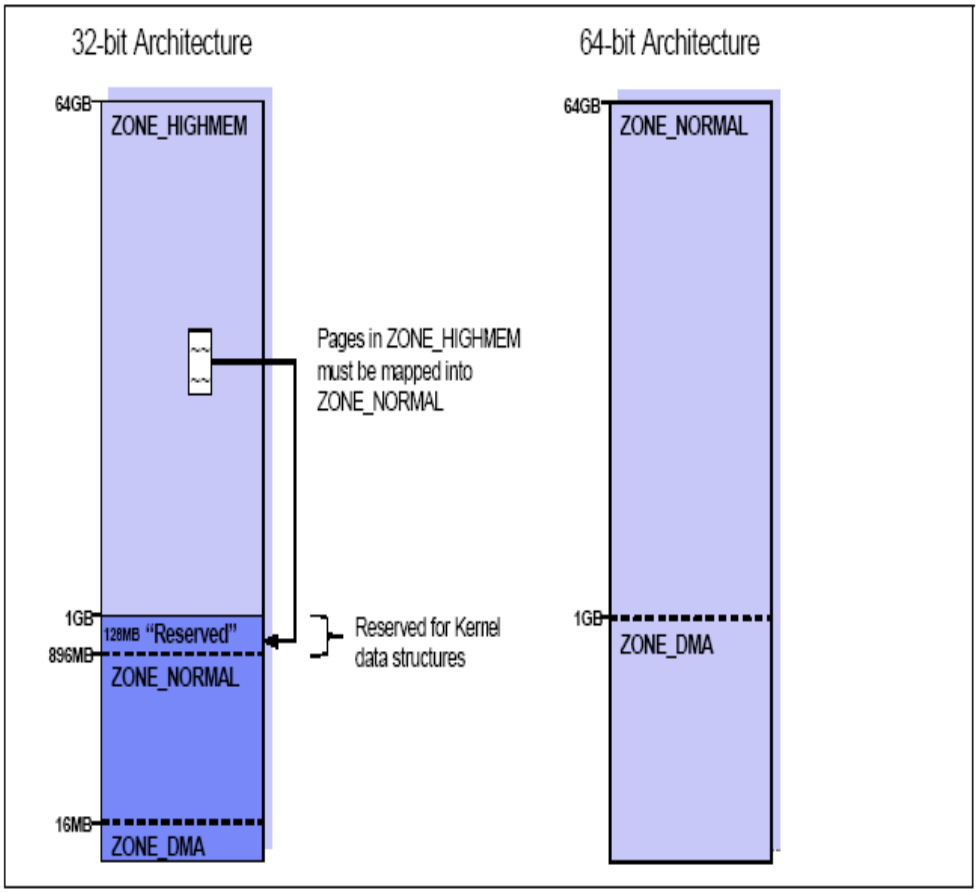
各个节点划分为若干个区，也是对物理内存的进一步细分。通过下面几个宏来标记物理内存不同的区：

ZONE_DMA：标记适合DMA的内存区。

ZONE_NORMAL：可以直接映射到内核空间的物理内存。

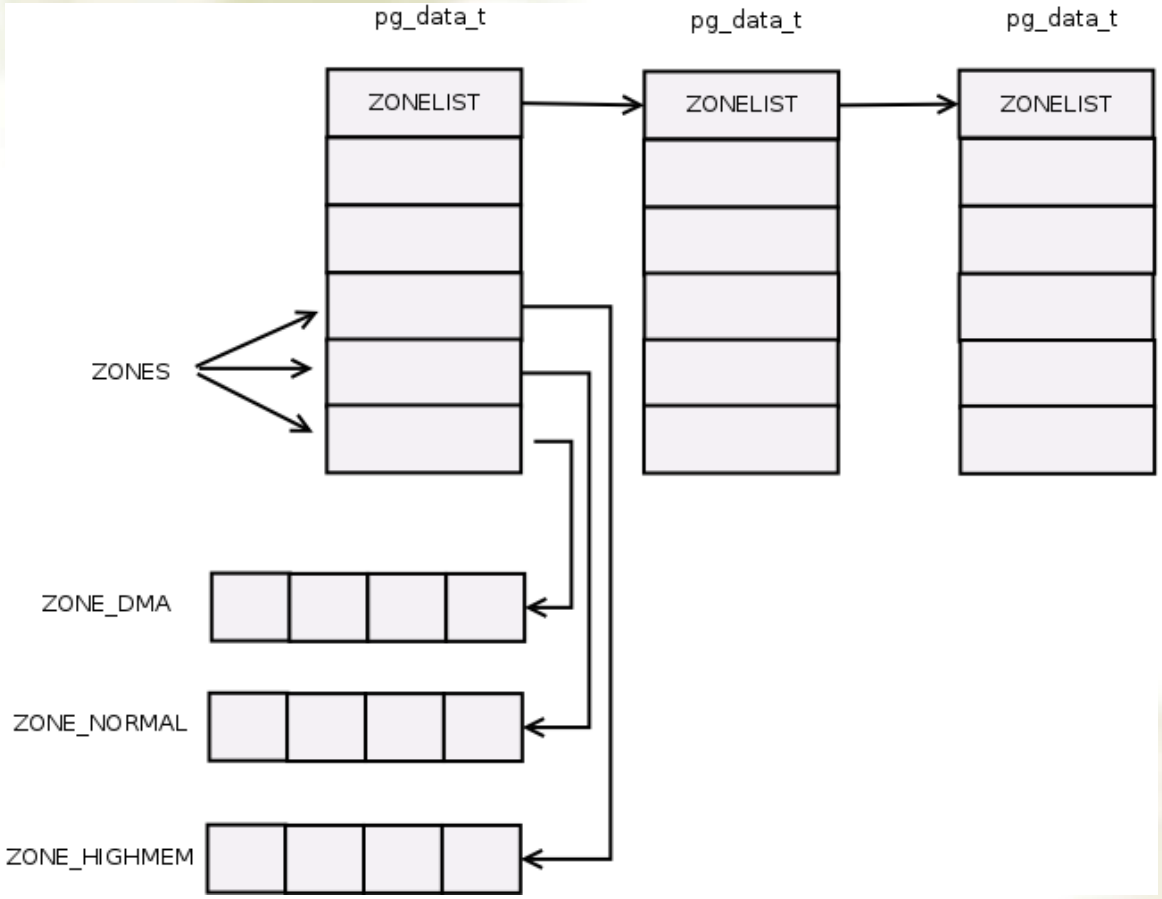
ZONE_HIGHMEM：高端物理内存。

内存管理区



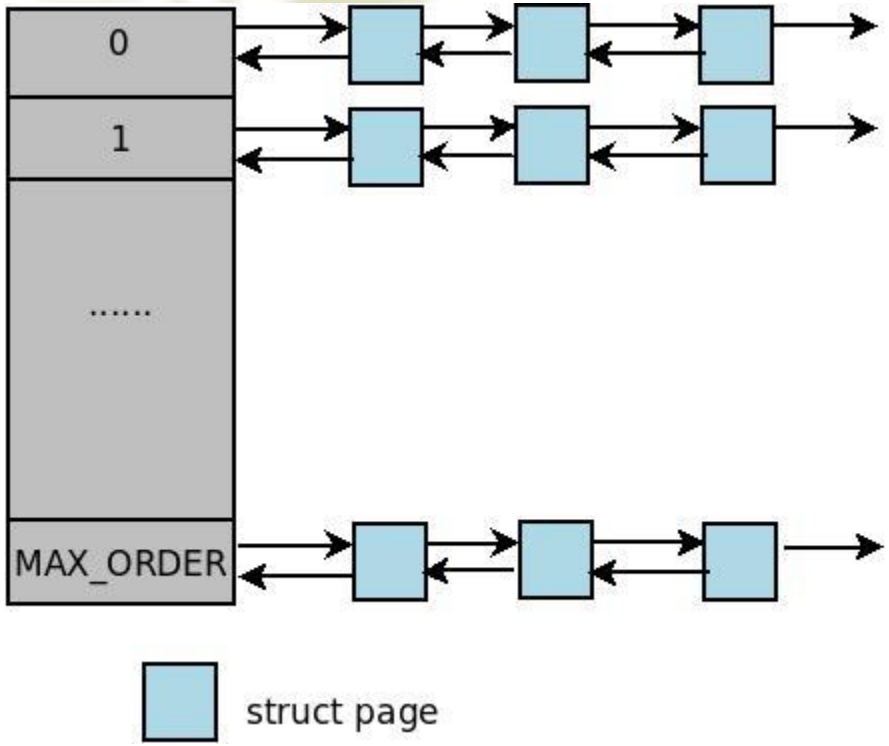
那么32位和64位操作系统对内存区管理上有什么差异？最大的区别是64位操作系统不再有高端内存的概念，可以支持大于4GB的内存寻址。ZONE_NORMAL空间将扩展到64GB或者128GB，（64位系统上的映射更简单了）

基本数据结构之间的关系



物理内存先被划分为内存节点，内存区用pg_data_t表示，每个节点关联一个CPU，对于NUMA结构来说，因为有多个节点，因此各节点之间形成一个链表。每个节点又被划分几个内存管理区（ZONES），在一个内存管理区中则是一个个的页框。页框是内存管理的基本单位，它可以存放任何种类的数据。

伙伴算法概述



那么物理内存如何分配，Linux内核中主要采用伙伴算法。

为什么叫伙伴算法呢？大小相同、物理地址连续的两个页块被称为伙伴。

Linux的伙伴算法把所有的空闲页面分为多个块链表（该默认大小为11）个，每个链表中的一个块含有2的幂次个页面，即页块或简称块。

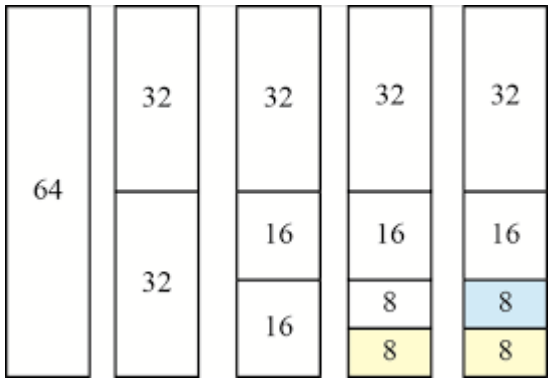
与伙伴算法有关的数据结构

```
01 struct zone {
02 ……
03 struct free_area         free_area[MAX_ORDER];
04 ……
05 }
06
07 #ifndef CONFIG_FORCE_MAX_ZONEORDER
08 #define MAX_ORDER 11
09 #else
10 #define MAX_ORDER CONFIG_FORCE_MAX_ZONEORDER
11 #endif
```

```
1 struct free_area {
2         struct list_head        free_list[MIGRATE_TYPES];
3         unsigned long           nr_free;
4 };
```

每个物理页框对应一个struct page实例。每个内存区关联了一个struct zone实例，该结构中使用free_area数组对空闲页框进行管理。

伙伴算法的分配原理



伙伴算法的分配原理是，如果分配阶为n的页框块，那么先从第n条页框块链表中查找是否存在这么大小的空闲页块。如果有则分配，否则在第n+1条链表中继续查找，直到找到为止。

如果申请大小为8的页块（分配阶为3），但却在页块大小为32的链表中找到空闲块，则先将这32个页面对半等分，前一半作为分配使用，另一半作为新元素插入下级大小为16的链表中；继续将前一半大小为16的页块等分，一半分配，另一半插入大小为8的链表中。

页框分配的实现

这里介绍最主要的两个函数：

__rmqueue_smallest()：在指定的内存区上，从所请求分配阶order对应的链表开始查找所需大小的空闲块，如果不成功则从高一阶的链表上继续查找。

expand()：如果所得到的内存块大于所请求的内存块，则按照伙伴算法的分配原理将大的页框块分裂成小的页框块。

__rmqueue_smallest

```
static inline
struct page *__rmqueue_smallest(struct zone *zone, unsigned int order,
                                                int migratetype)
{
        unsigned int current_order;
        struct free_area * area;
        struct page *page;

        /* Find a page of the appropriate size in the preferred list */
        for (current_order = order; current_order < MAX_ORDER; ++current_order) {
                area = &(zone->free_area[current_order]);
                if (list_empty(&area->free_list[migratetype]))
                        continue;

                page = list_entry(area->free_list[migratetype].next,
                                                        struct page, lru);
                list_del(&page->lru);
                rmv_page_order(page);
                area->nr_free--;
                expand(zone, page, order, current_order, area, migratetype);
                return page;
        }

        return NULL;
}
```

对__rmqueue_smallest()解释

该函数的实现比较简单，从当前指定的分配阶到最高分配阶依次进行遍历。在每次遍历的分配阶链表中，根据参数迁移类型（migratetype）选择正确的迁移队列。根据以上的限定条件，当选定一个页块链表后，只要该链表不为空，就说明可以分配该阶对应的页块。

一旦选定在当前遍历的分配阶链表上分配页框，那么就通过list_entry（）将该页块从链表上移除。以上这个过程通过rmv_page_order()完成。此外，还要更新页块链表nr_free的值。

expand()

```
static inline void expand(struct zone *zone, struct page *page,
        int low, int high, struct free_area *area,
        int migratetype)
{
        unsigned long size = 1 << high;

        while (high > low) {
                area--;
                high--;
                size >>= 1;
                VM_BUG_ON(bad_range(zone, &page[size]));
                list_add(&page[size].lru, &area->free_list[migratetype]);
                area->nr_free++;
                set_page_order(&page[size], high);
        }
}
```

对分裂函数expand()的解释

分裂函数的实现也是显而易见的，它完全遵照伙伴算法的分裂原理。这里有两个分配阶，一个是申请页框时指定的低阶low，一个是在上级函数中遍历时所选定的高阶high。该函数从high分配阶开始递减向low遍历，也就是从较大的页块开始依次分裂。

物理内存分配器

基于伙伴算法、每CPU高速缓存和slab高速缓存形成两种内存分配器。

第一种是分区页框分配器（zoned page frame allocator），处理对连续页框的内存分配请求。

第二种是slab分配器，它将各种分配对象分组放进高速缓存，即每个高速缓存都对同类型分配对象的一种“储备”。

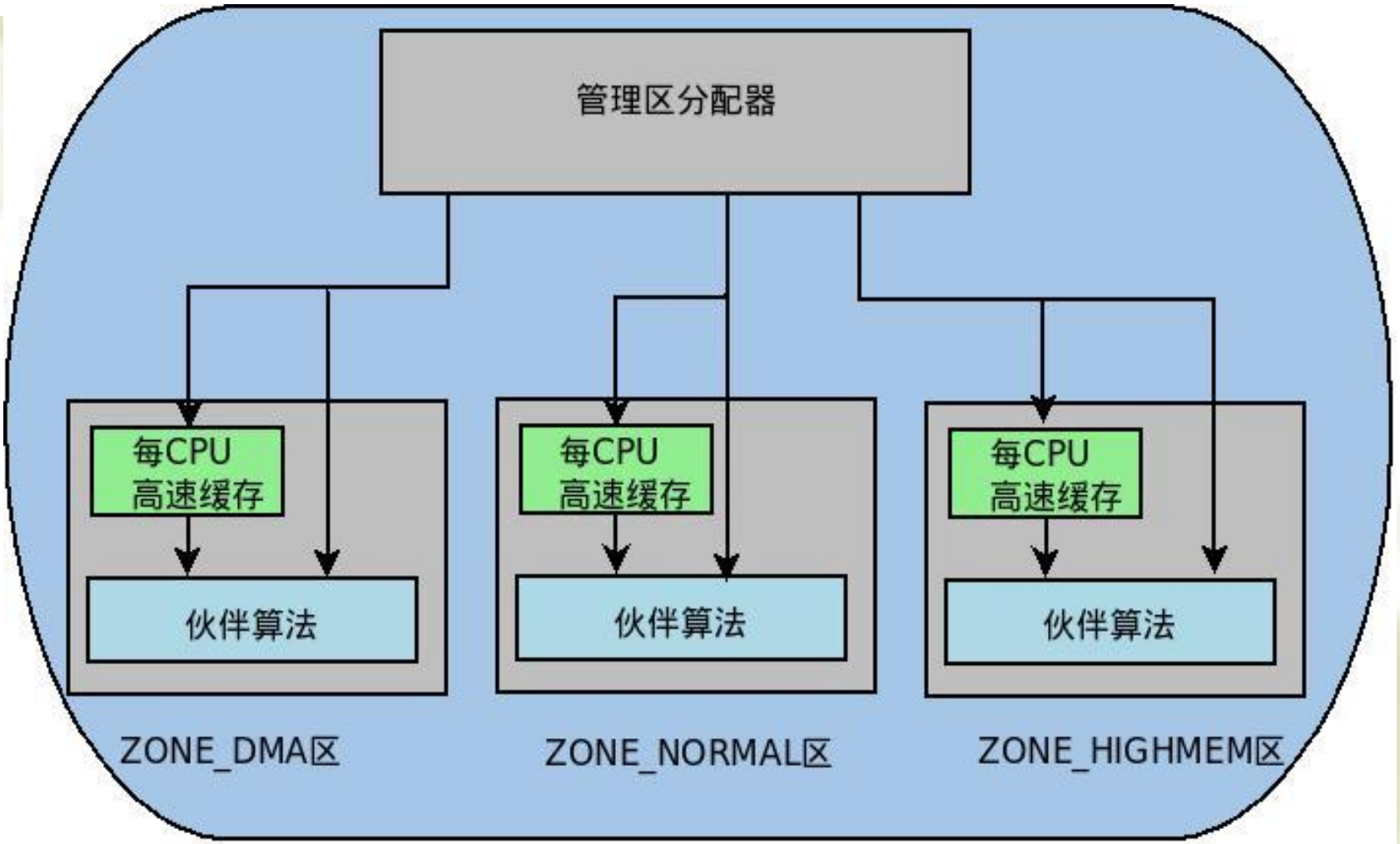
分区页框分配器图示

分区页框管理器分为两大部分：前端的管理区分配器和伙伴系统。

管理区分配器负责搜索一个能满足请求页框块大小的管理区。

在每个管理区中，具体的页框分配工作由伙伴系统负责。为了达到更好的系统性能，单个页框的申请工作直接通过每CPU页框高速缓存完成。

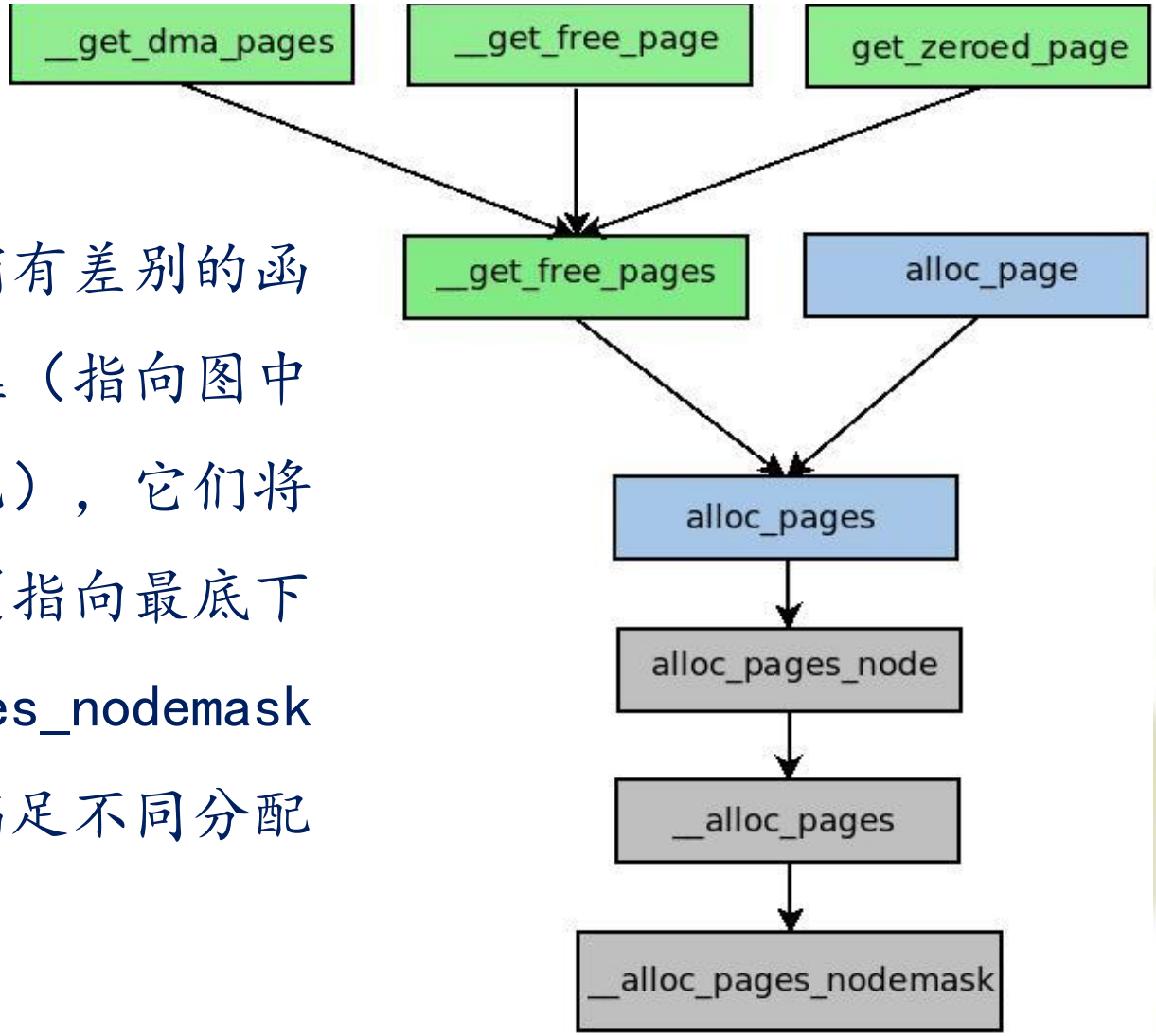
分区页框分配器图示



分区页框分配器示例图

页框分配函数的关系图

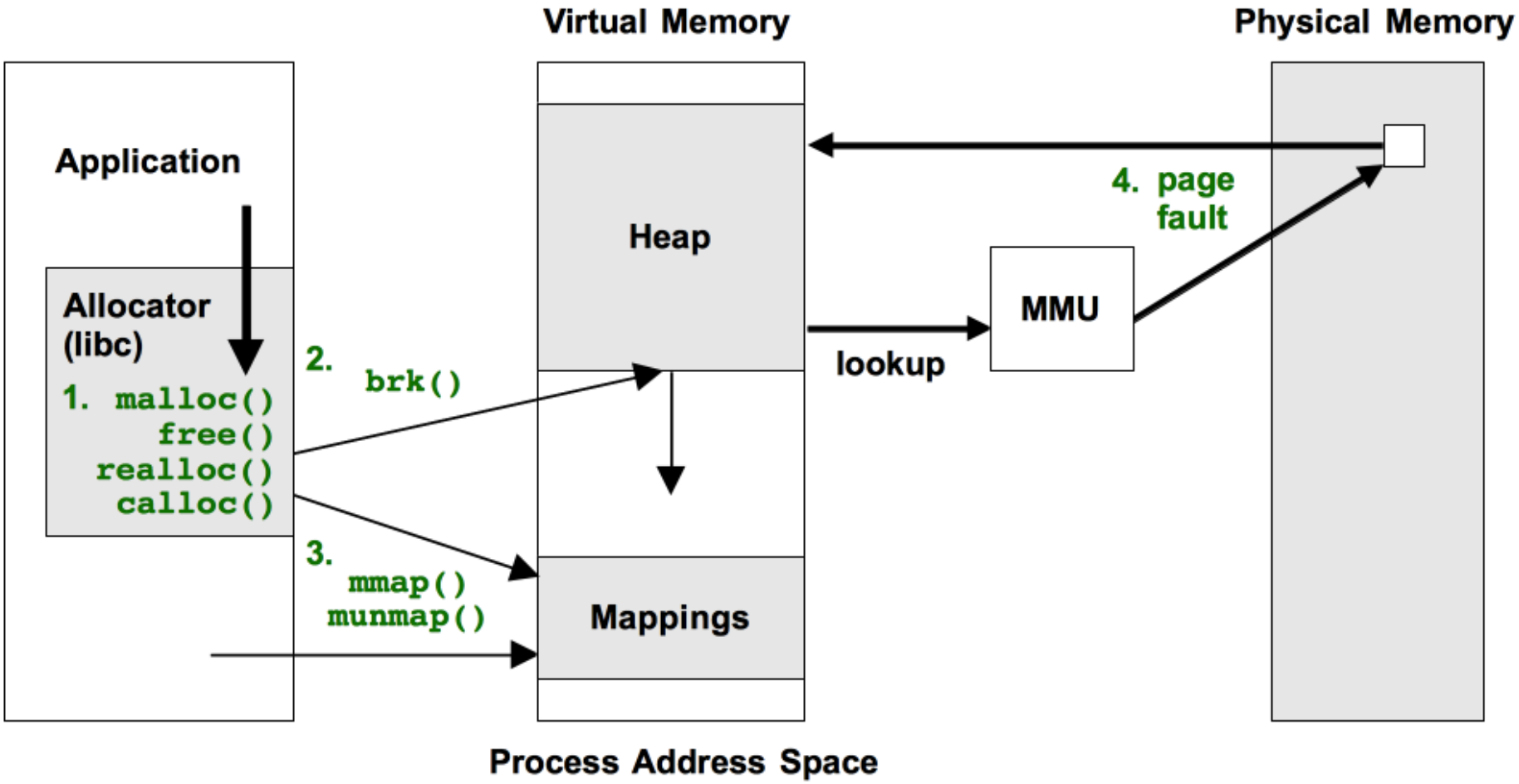
内核中有6个稍有差别的函数和宏来请求页框（指向图中4个绿色和2个蓝色），它们将核心的分配函数（指向最底下的）__alloc_pages_nodemask进行封装，形成满足不同分配需求的分配函数。



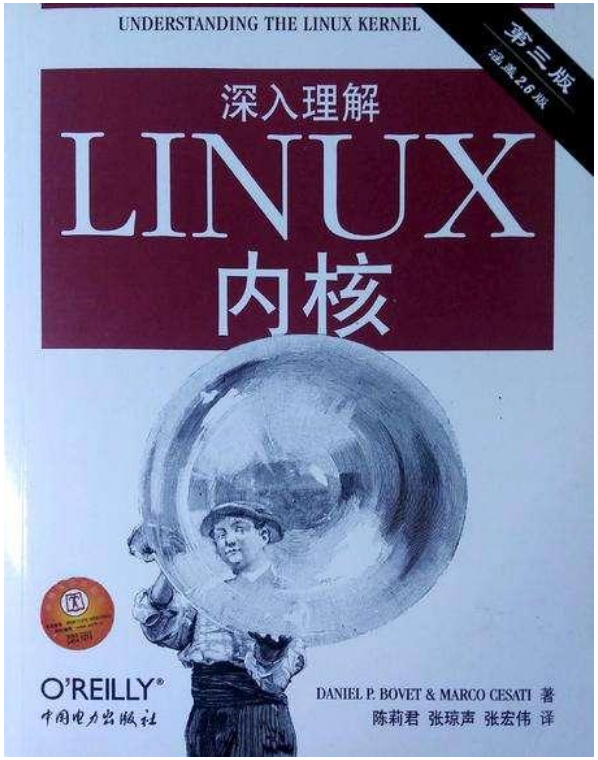
总结：从用户态到内核态的内存分配

当用户程序通过调用系统调用申请内存时，首先陷入内核，建立虚拟地址空间的映射，获得一块虚拟内存区VMA。当进程对这块虚存区进行访问时，如果物理内存尚未分配，那么此时发生一个缺页异常，通过get_free_pages 申请一个或多个物理页面，并将此物理内存和虚拟内存的映射关系写入页表。

总结：从用户态到内核态的内存分配



参考资料



深入理解Linux内核 第三版第八章

博客：http://edsionte.com/techblog/

“内存管理哪些事儿”栏目，有比较详细的系列文章。

带着思考离开

在物理内存为1G的计算中，能否malloc（1.6G）？为什么？

谢谢大家！

THANK YOU