6.3.3 Slab分配机制

采用伙伴算法分配内存时，每次至少分配一个页面。但当请求分配的内存大小为几十个字节或几百个字节时应该如何处理？如何在一个页面中分配小的内存区，小内存区的分配所产生的内碎片又如何解决？

Linux2.0采用的解决办法是建立了13个空闲区链表，它们的大小从32字节到132056字节。从Linux2.2开始，MM的开发者采用了一种叫做slab的分配模式，该模式早在1994年就被开发出来，用于Sun Microsystem Solaris 2.4操作系统中。Slab的提出主要是基于以下考虑：

- 内核对内存区的分配取决于所存放数据的类型。例如，当给用户态进程分配页面时，内核调用get_free_page()函数，并用0填充这个页面。 而给内核的数据结构分配页面时，事情没有这么简单，例如，要对数据结构所在的内存进行初始化、在不用时要收回它们所占用的内存。因此，Slab中引入了对象这个概念，所谓对象就是存放一组数据结构的内存区，其方法就是构造或析构函数，构造函数用于初始化数据结构所在的内存区，而析构函数收回相应的内存区。但为了便于理解，你也可以把对象直接看作内核的数据结构。为了避免重复初始化对象，Slab分配模式并不丢弃已分配的对象，而是释放但把它们依然保留在内存中。当以后又要请求分配同一对象时，就可以从内存获取而不用进行初始化，这是在Solaris 中引入Slab的基本思想。

  实际上，Linux中对Slab分配模式有所改进，它对内存区的处理并不需要进行初始化或回收。出于效率的考虑，Linux并不调用对象的构造或析构函数，而是把指向这两个函数的指针都置为空。Linux中引入Slab的主要目的是为了减少对伙伴算法的调用次数。
- 实际上，内核经常反复使用某一内存区。例如，只要内核创建一个新的进程，就要为该进程相关的数据结构（task_struct、打开文件对象等）分配内存区。当进程结束时，收回这些内存区。因为进程的创建和撤销非常频繁，因此，Linux的早期版本把大量的时间花费在反复分配或回收这些内存区上。从Linux2.2开始，把那些频繁使用的页面保存在高速缓存中并重新使用。
- 可以根据对内存区的使用频率来对它分类。对于预期频繁使用的内存区，可以创建一组特定大小的专用缓冲区进行处理，以避免内碎片的产生。对于较少使用的内存区，可以创建一组通用缓冲区（如Linux2.0中所使用的2的幂次方）来处理，即使这种处理模式产生碎片，也对整个系统的性能影响不大。
- 硬件高速缓存的使用，又为尽量减少对伙伴算法的调用提供了另一个理由，因为对伙伴算法的每次调用都会“弄脏”硬件高速缓存，因此，这就增加了对内存的平均访问次数。

Slab分配模式把对象分组放进缓冲区（尽管英文中使用了Cache这个词，但实际上指的是内存中的区域，而不是指硬件高速缓存）。因为缓冲区的组织和管理与硬件高速缓存的命中率密切相关，因此，Slab缓冲区并非由各个对象直接构成，而是由一连串的“大块（Slab）”构成，而每个大块中则包含了若干个同种类型的对象，这些对象或已被分配，或空闲，如图6.12所示。一般而言，对象分两种，一种是大对象，一种是小对象。所谓小对象，是指在一个页面中可以容纳下好几个对象的那种。例如，一个inode结构大约占300多个字节，因此，一个页面中可以容纳8个以上的inode结构，因此，inode结构就为小对象。Linux内核中把小于512字节的对象叫做小对象。

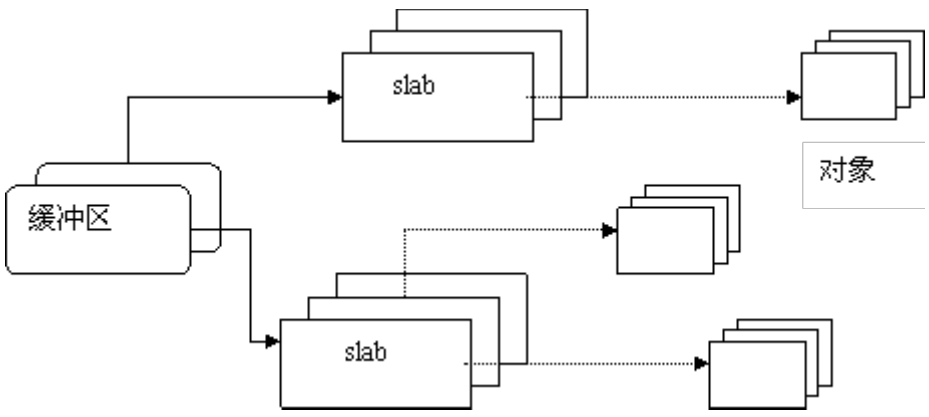


图6.10 Slab的组成

实际上，缓冲区就是主存中的一片区域，把这片区域划分为多个块，每块就是一个Slab，每个Slab由一个或多个页面组成，每个Slab中存放的就是对象。

因为Slab分配模式的实现比较复杂，我们不准备对其进行详细的分析，只对主要内容给予描述。

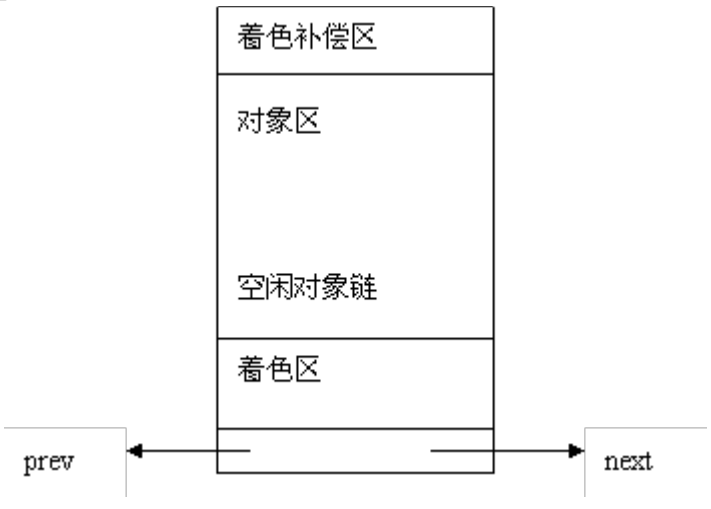
1．Slab的数据结构

Slab分配模式有两个主要的数据结构，一个是描述缓冲区的结构kmem_cache_t，一个是描述Slab的结构kmem_slab_t，下面对这两个结构给予简要讨论：

（1）Slab

Slab是Slab管理模式中最基本的结构。它由一组连续的物理页面组成，对象就被顺序放在这些页面中。其数据结构在mm/slab.c中定义如下：

```
/*
 * 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;
```



这里的链表用来将前一个Slab和后一个Slab链接起来形成一个双向链表，colouroff为该Slab上着色区的大小，指针s_mem指向对象区的起点，inuse是

Slab中所分配对象的个数。最后，free的值指明了空闲对象链中的第一个对象，kmem_bufctl_t其实是一个整数。Slab结构的示意图如图6.11所示：

图6.11 Slab结构示意图

对于小对象，就把Slab的描述结构slab_t放在该Slab中；对于大对象，则把Slab结构游离出来，集中存放。关于Slab中的着色区再给予具体描述：

- 每个Slab的首部都有一个小小的区域是不用的，称为"着色区（coloring area）"。着色区的大小使Slab中的每个对象的起始地址都按高速缓存中的"缓存行（cache line）"大小进行对齐（80386的一级高速缓存行大小为16字节，Pentium为32字节）。因为Slab是由1个页面或多个页面（最多为32）组成，因此，每个Slab都是从一个页面边界开始的，它自然按高速缓存的缓冲行对齐。但是，Slab中的对象大小不确定，设置着色区的目的就是将Slab中第一个对象的起始地址往后推到与缓冲行对齐的位置。因为一个缓冲区中有多个Slab，因此，应该把每个缓冲区中的各个Slab着色区的大小尽量安排成不同的大小，这样可以使得在不同的Slab中，处于同一相对位置的对象，让它们在高速缓存中的起始地址相互错开，这样就可以改善高速缓存的存取效率。
- 每个Slab上最后一个对象以后也有个小小的废料区是不用的，这是对着色区大小的补偿，其大小取决于着色区的大小，以及Slab与其每个对象的相对大小。但该区域与着色区的总和对于同一种对象的各个Slab是个常数。
- 每个对象的大小基本上是所需数据结构的大小。只有当数据结构的大小不与高速缓存中的缓冲行对齐时，才增加若干字节使其对齐。所以，一个Slab上的所有对象的起始地址都必然是按高速缓存中的缓冲行对齐的。

(2)缓冲区

每个缓冲区管理着一个Slab链表，Slab按序分为三组。第一组是全满的Slab（没有空闲的对象），第二组Slab中只有部分对象被分配，部分对象还空闲，最后一组Slab中的对象全部空闲。只所以这样分组，是为了对Slab进行有效的管理。每个缓冲区还有一个轮转锁（spinlock），在对链表进行修改时用这个轮转锁进行同步。类型kmem_cache_s在mm/slab.c中定义如下：

```
 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

/* 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
.....
};
```

然后定义了kmem_cache_t，并给部分域赋给了初值：

```
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",
};
```

对该结构说明如下：

- 该结构中有三个队列slabs_full、slabs_partial以及slabs_free，分别指向满Slab、半满Slab和空闲Slab，另一个队列next则把所有的专用缓冲区链成一个链表。
- 除了这些队列和指针，该结构中还有一些重要的域：objsize是原始的数据结构的大小，这里初始化为kmem_cache_t的大小；num表示每个Slab上有几个缓冲区；gfporder则表示每个Slab大小的对数，即每个Slab由$2^{gfporder}$个页面构成。
- 如前所述，着色区的使用是为了使同一缓冲区中不同Slab上的对象区的起始地址相互错开，这样有了利于改善高速缓存的效率。colour_off表示颜

色的偏移量，colour表示颜色的数量；一个缓冲区中颜色的数量取决于Slab中对象的个数、剩余空间以及高速缓存行的大小。所以，对每个缓冲区都要计算它的颜色数量，这个数量就保存在colour中，而下一个Slab将要使用的颜色则保存在colour_next中。当colour_next达到最大值时，就又从0开始。着色区的大小可以根据（colour_off´colour）算得。例如，如果colour 为5，colour_off为8，则第一个Slab的颜色将为0，Slab中第一个对象区的起始地址（相对）为0，下一个Slab中第一个对象区的起始地址为8，再下一个为16，24，32，0…等。

- cache_cache变量实际上就是缓冲区结构的头指针。

由此可以看出，缓冲区结构kmem_cache_t相当于Slab的总控结构，缓冲区结构与Slab结构之间的关系如图6.12所示：

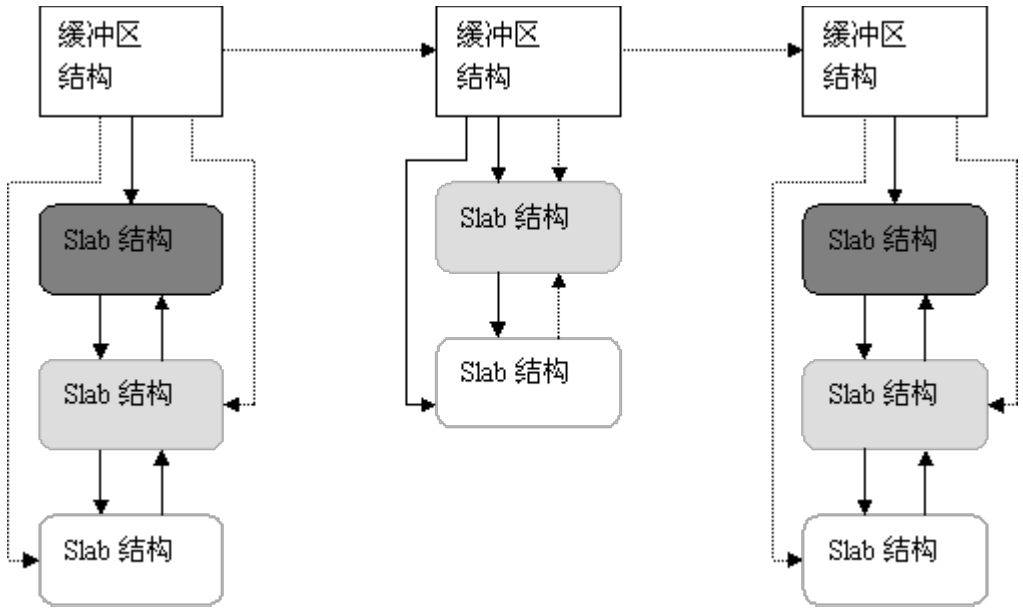


图6.12缓冲区结构kmem_cache_t与Slab结构slab_t之间的关系

在图6.12中，深灰色表示全满的Slab，浅灰色表示含有空闲对象的Slab，而无色表示空的Slab。缓冲区结构之间形成一个单向链表，Slab结构之间形成一个双向链表。另外，缓冲区结构还有分别指向满、半满、空闲Slab结构的指针。

2．专用缓冲区的建立和撤销

专用缓冲区是通过kmem_cache_create（）函数建立的，函数原型为：

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

对其参数说明如下：

name： 缓冲区名（19 个字符)

size： 对象大小

offset ：所请求的着色偏移量。

c_flags ：对缓冲区的设置标志：

SLAB_HWCACHE_ALIGN：表示与第一个高速缓存中的缓冲行边界（16或32字节）对齐。 SLAB_NO_REAP：不允许系统回收内存

SLAB_CACHE_DMA：表示Slab使用的是DMA内存

ctor ：构造函数（一般都为NULL)

dtor ：析构函数（一般都为NULL)

objp ：指向对象的指针

cachep ：指向缓冲区

对专用缓冲区的创建过程简述如下：

kmem_cache_create（）函数要进行一系列的计算，以确定最佳的Slab构成。包括：每个Slab由几个页面组成，划分为多少个对象；Slab的描述结构slab_t应该放在Slab的外面还是放在Slab的尾部；还有“颜色”的数量等等。并根据调用参数和计算结果设置kmem_cache_t结构中的各个域，包括两个函数指针ctor和dtor。最后，将kmem_cache_t结构插入到cache_cache的next队列中。

但请注意，函数kmem_cache_create（）所创建的缓冲区中还没有包含任何Slab，因此，也没有空闲的对象。只有以下两个条件都为真时，才给缓冲区分配Slab：

（1） 已发出一个分配新对象的请求；

（2） 缓冲区不包含任何空闲对象；

当这两个条件都成立时，Slab分配模式就调用kmem_cache_grow（）函数给缓冲区分配一个新的Slab。其中，该函数调用kmem_gatepages()从伙伴系统获得一组页面；然后又调用kmem_cache_slabgmt()获得一个新的Slab结构；还要调用kmem_cache_init_objs()为新Slab中的所有对象申请构造方法（如果定义的话）；最后，调用kmem_slab_link_end()把这个Slab结构插入到缓冲区中Slab链表的末尾。

Slab分配模式的最大好处就是给频繁使用的数据结构建立专用缓冲区。但到目前的版本为止，Linux内核中多数专用缓冲区的建立都用NULL作为构造函数的指针，例如，为虚存区间结构vm_area_struct建立的专用缓冲区vm_area_cachep：

```
vm_area_cachep = kmem_cache_create("vm_area_struct",
                    sizeof(struct vm_area_struct), 0,
                    SLAB_HWCACHE_ALIGN, NULL, NULL);
```

就把构造和析构函数的指针置为NULL，也就是说，内核并没有充分利用Slab管理机制所提供的好处。为了说明如何利用专用缓冲区，我们从内核代码中选取一个构造函数不为空的简单例子，这个例子与网络子系统有关，在net/core/buff.c中定义：

```
void __init  skb_init(void)
    {
      int  i;
  skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
                        sizeof(struct sk_buff),
                        0,
                        SLAB_HWCACHE_ALIGN,
                        skb_headerinit, NULL);
if (!skbuff_head_cache)
      panic("cannot create skbuff cache");

for (i=0; i<NR_CPUS; i++)
      skb_queue_head_init(&skb_head_pool[i].list);
}
```

从代码中可以看出，skb_init（）调用kmem_cache_create（）为网络子系统建立一个sk_buff数据结构的专用缓冲区，其名称为“skbuff_head_cache”（你可以通过读取／proc/slabinfo/文件得到所有缓冲区的名字）。调用参数offset为0，表示第一个对象在Slab中的位移并无特殊要求。但是参数flags为SLAB_HWCACHE_ALIGN，表示Slab中的对象要与高速缓存中的缓冲行边界对齐。对象的构造函数为skb_headerinit（），而析构函数为空，也就是说，在释放一个Slab时无需对各个缓冲区进行特殊的处理。

当从内核卸载一个模块时，同时应当撤销为这个模块中的数据结构所建立的缓冲区，这是通过调用kmem_cache_destroy()函数来完成的。从Linux2.4.16内核代码中进行查找，对这个函数的调用非常少。

3．通用缓冲区

在内核中初始化开销不大的数据结构可以合用一个通用的缓冲区。通用缓冲区非常类似于物理页面分配中的大小分区，最小的为32，然后依次为64、128、…直至128K（即32个页面），但是，对通用缓冲区的管理又采用的是Slab方式。从通用缓冲区中分配和释放缓冲区的函数为：

```
void *kmalloc(size_t size, int flags);
Void  kree(const void *objp);
```

因此，当一个数据结构的使用根本不频繁时，或其大小不足一个页面时，就没有必要给其分配专用缓冲区，而应该调用kmallo()进行分配。如果数据结构的大小接近一个页面，则干脆通过alloc_page()为之分配一个页面。

事实上，在内核中，尤其是驱动程序中，有大量的数据结构仅仅是一次性使用，而且所占内存只有几十个字节，因此，一般情况下调用kmallo()给内核数据结构分配内存就足够了。另外，因为，在Linux2.0以前的版本一般都调用kmallo()给内核数据结构分配内存，因此，调用该函数的一个优点是（让你开发的驱动程序）能保持向后兼容。

6.3.4 内核空间非连续内存区的管理

我们说，任何时候，CPU访问的都是虚拟内存，那么，在你编写驱动程序，或者编写模块时，Linux给你分配什么样的内存？它处于4G空间的什么位置？这就是我们要讨论的非连续内存。

首先，非连续内存处于3G到4G之间，也就是处于内核空间，如图6.13所示：

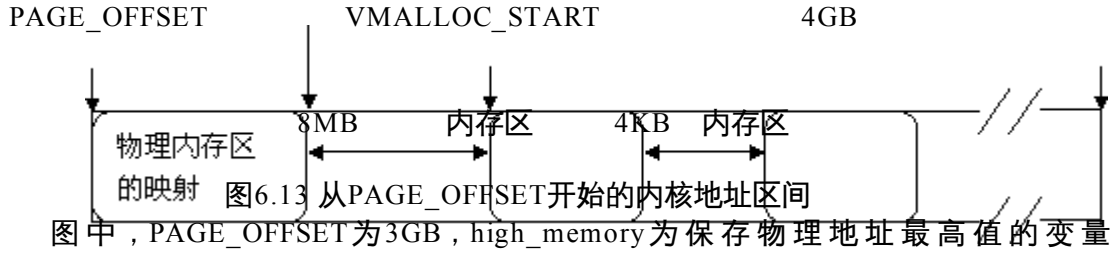


图6.13 从PAGE_OFFSET开始的内核地址区间

图中，PAGE_OFFSET为3GB，high_memory为保存物理地址最高值的变量，VMALLOC_START为非连续区的的起始地址，定义于include／i386/pgtable.h中：

```
#define VMALLOC_OFFSET  (8*1024*1024)
#define VMALLOC_START   (((unsigned long) high_memory + 2*VMALLOC_OFFSET-1) & \~(VMALLOC_OFFSET-1))
```

在物理地址的末尾与第一个内存区之间插入了一个8MB（VMALLOC_OFFSET）的区间，这是一个安全区，目的是为了“捕获”对非连续区的非法访问。出于同样的理由，在其他非连续的内存区之间也插入了4K大小的安全区。每个非连续内存区的大小都是4096的倍数。

1．非连续区的数据结构

描述非连续区的数据结构为struct vm_struct，定义于include/linux/vmalloc.h中：

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

非连续区组成一个单链表，链表第一个元素的地址存放在变量vmlist中。Addr域是内存区的起始地址；size是内存区的大小加4096(安全区的大小)。

2．创建一个非连续区的结构

函数get_vm_area（）创建一个新的非连续区结构，其代码在mm/vmalloc.c中：

```
struct vm_struct * get_vm_area(unsigned long size, unsigned long flags)
{
        unsigned long addr;
        struct vm_struct **p, *tmp, *area;

        area = (struct vm_struct *) kmalloc(sizeof(*area), GFP_KERNEL);
        if (!area)
                return NULL;
        size += PAGE_SIZE;
        addr = VMALLOC_START;
        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;
        }
        area->flags = flags;
        area->addr = (void *)addr;
        area->size = size;
        area->next = *p;
        *p = area;
        write_unlock(&vmlist_lock);
        return area;

out:
        write_unlock(&vmlist_lock);
        kfree(area);
```

```
        return NULL;
}
```

这个函数比较简单，就是在单链表中插入一个元素。其中调用了kmalloc()和kfree()函数,分别用来为vm_struct结构分配内存和释放所分配的内存。

3．分配非连续内存区

vmalloc（）函数给内核分配一个非连续的内存区，在/include/linux/vmalloc.h中定义如下：

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

vmalloc（）最终调用的是__vmalloc（）函数，该函数的代码在mm/vmalloc.c中：

```
 void * __vmalloc (unsigned long size, int gfp_mask, pgprot_t prot)
{
       void * addr;
       struct vm_struct *area;

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

函数首先把size参数取整为页面大小（4096）的一个倍数，也就是按页的大小进行对齐，然后进行有效性检查，如果有大小合适的可用内存，就调用get_vm_area（）获得一个内存区的结构。但真正的内存区还没有获得，函数vmalloc_area_pages（）真正进行非连续内存区的分配：

```
    inline int vmalloc_area_pages (unsigned long address, unsigned long size,
                        int gfp_mask, pgprot_t prot)
 {
        pgd_t * dir;
        unsigned long end = address + size;
        int ret;

        dir = pgd_offset_k(address);
        spin_lock(&init_mm.page_table_lock);
             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));
        spin_unlock(&init_mm.page_table_lock);
        return ret;
    }
```

该函数有两个主要的参数，address表示内存区的起始地址，size表示内存区的大小。内存区的末尾地址赋给了局部变量end。其中还调用了几个主要的函数或宏：

- pgd_offset_k（）宏导出这个内存区起始地址在页目录中的目录项；
- pmd_alloc（）为新的内存区创建一个中间页目录；
- alloc_area_pmd（）为新的中间页目录分配所有相关的页表，并更新页的总目录；该函数调用pte_alloc_kernel()函数来分配一个新的页表，之后再调用alloc_area_pte（）为页表项分配具体的物理页面。

从vmalloc_area_pages（）函数可以看出，该函数实际建立起了非连续内存区到物理页面的映射。

4．kmalloc()与vmalloc()的区别

kmalloc()与vmalloc() 都是在内核代码中提供给其他子系统用来分配内存的函数，但二者有何区别？

从前面的介绍已经看出，这两个函数所分配的内存都处于内核空间，即从3GB～4GB；但位置不同，kmalloc()分配的内存处于3GB～high_memory之间，而vmalloc()分配的内存在VMALLOC_START～4GB之间，也就是非连续内存区。一般情况下在驱动程序中都是调用kmalloc()来给数据结构分配内存，而vmalloc()用在为活动的交换区分配数据结构，为某些I/O驱动程序分配缓冲区，或为模块分配空间，例如在include/asm-i386/module.h中定义了如下语句：

```
 #define module_map(x)          vmalloc(x)
```

其含义就是把模块映射到非连续的内存区。

与kmalloc()和vmalloc()相对应，两个释放内存的函数为kfree()和vfree()。