## 页框管理：

1. RAM的某些部分永久地分配给内核，用来存放内核代码以及静态内核数据结构。RAM的其余部分称为动态内存，实际上，整个系统的性能取决于如何有效的管理动态内存。
2. Linux采用4KB页框大小作为标准的内存分配单元。因为绝大多数情况下，主存和磁盘之间传输小块数据时更高效。
3. 页描述符：
4. 页框的状态信息保存在一个类型为page的页描述符中。所有的页描述符在mem\_map数组中。因为每个描述符长度为32字节，所以mem\_map需要的空间略小于整个RAM的1%。virt\_to\_page(addr)宏产生线性地址addr对应的页描述符地址。pfn\_to\_page(pfn)宏产生与页框号pfn对应的页描述符地址。
5. 字段：

类型 名字 说明

unsigned long flags 一组标志。也对页框所在的管理区进行编号

atomic\_t \_count 页框的引用计数器

atomic\_t \_mapcount 页框中的页表项数目（没有则为-1）

unsigned long private 可用于正在使用页的内核成分。如果页是空闲的，则该字段由伙伴系统使用

struct mapping address\_space\* mapping 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

unsigned long index 作为不同的含义被几种内核成分使用。

struct list\_head lru 包含页的最近最少使用双向链表的指针

（3）\_count:页的引用计数器。如果该字段为-1，则相应的页框空闲，并可被分配给任意进程或内核本身；如果该字段的值大于或等于0，则说明页框被分配给了一个或多个进程，或用于存放一些内核数据结构。page\_count()函数返回\_count加1后的值，也就是该页的使用者数目

（4）flags：包含多大32个用来描述页框状态的标志。对于每个PG\_xyz标志，内核都定义了操纵其值的一些宏、通常，PageXyz宏返回标志的值，而SetPageXyz和ClearPageXyz宏分别设置和清楚相应的位。

// 描述页框状态的标志 P312

## 非一致内存访问（NUMA）：

1. Linux2.6支持非一致内存访问（NUMA）模型，在这种模型中，给定CPU对不同内存单元（一般是8bit，一个字节）的访问时间可能不一样。系统的物理内存被划分为几个节点。在一个单独的节点内，任一给定的CPU访问页面所需要的时间都是相同的。然而，对不同的CPU，这个时间可能就不同。对每个CPU而言，内核都试图把耗时节点的访问次数减到最少，这就要小心地选择CPU最常引用的内核数据的存放位置。
2. 每个节点中的物理内存又可以分为几个管理区。每个节点都有一个类型为page\_data\_t的描述符。所有节点的描述符存放在一个但相恋表中，它的第一个元素有pgdat\_list变量指向。

类型 名字 说明

struct zone\_t[] node\_zones 节点中管理区描述符的数组

struct zonelist\_t[] node\_zonelists 页分配器使用的zonelist数据结构的数组

int nr\_zones 节点中管理区的个数

struct page\* node\_menm\_map 节点中页描述符的数组

struct bootmem\_data\* bdata 用在内核初始化阶段

unsigned long node\_start\_pfn 节点中第一个页框的下标

unsigned long node\_present\_pages 内存节点的大小（不包括洞，以页为单位）

unsigned long node\_spanned\_pages 内存节点的大小（不包括洞，以页为单位）

int node\_id 节点标识符

pg\_data\_t\* pgdat\_next 内存节点链表的下一项

wait\_queue\_head\_t kswapd\_wait kswapd页换出守护进程使用的等待队列

struct task\_struct\* kswapd 指针指向kswapd内核线程的进程描述符

int kswapd\_max\_order kswapd将要创建的空闲块大小取对数的值

1. 内核假定在所有的体系结构中物理内存都被划分为一个或多个节点。

## 内存管理区：

1. 在一个理想的计算机体系结构中，一个页框就是一个内存存储单元，任何种类的数据页都可以存放在任何页框中，没有什么限制。但是，实际的计算机体系结构有硬件的制约，这限制了页框可以使用的方式。尤其是，Linux内核必须处理80x86体系结构的两种硬件约束：
2. ISA总线（工业标准结构总线）的直接内存存取（DMA）处理器有一个严格的限制：它们只能对RAM的前16MB寻址
3. 在具有大容量的RAM的现代32位计算机中，CPU不能直接访问所有的物理内存，因为线性地址空间太小。
4. 为了应对上面的两种限制，Linux2.6把每个内存节点的物理地址内存划分为3个管理区(zone)。

为了应对这两种限制，Linux2.6把每个内存节点的物理地址划分为3个管理区。在80x86UMA体系结构中的管理区为：

ZONE\_DMA：包含低于16MB的内存页框

ZONE\_NORMAL：包含高于16MB且低于896MB的内存页框

ZONE\_HIGHMEN：包含从896MB开始高于896MB的内存页框

ZONE\_DMA区包含的页框可以由老式基于ISA的设备通过DMA使用。ZONE\_DMA和ZONE\_NORMAL区包含内存的“常规”页框，通过把它们线性地址映射到线性地址空间的第4个GB，内核就可以直接进行访问。相反，ZONE\_HIGHMEN区包含的内存页不能由内核直接访问，尽管它们也线性地映射到先行地址空间的第4个GB。在64位体系结构上的ZONE\_HIGHMEN总是空的。

1. 每个内存管理区都有自己的描述符字段：

类型 名称 说明

unsigned long free\_pages 管理区中空闲页的个数

unsigned long pages\_min 管理区保留的页的数目

unsigned long pages\_low 回收页框使用的下界，同时也被管理区分配器作为阀值使用

unsigned long pages\_high 回收页框使用的上界，同时也被管理区分配器当作阀值使用

unsigned long lowmen\_reserve 指明在处理内存不足的临界情况下每个管理区必须保留的页框数

struct per\_cpu\_pageset[] pageset 数据结构用于实现单一页框的特殊高速缓存

spinlock\_t lock 保护该描述符的自旋锁

struct free\_area[] free\_area 标识出管理区中的空闲页框块

spinlock\_t lru\_lock 活动以及非活动链表使用的自旋锁

struct list\_head active\_list 管理区中的活动页链表

struct list\_head inactive\_list 管理区中非活动页链表

unsigned long nr\_scan\_active 回收内存时需要扫描的活动页数目

unsigned long nr\_scan\_inactive 回收内存时需要扫描的非活动页数目

unsigned long nr\_active 管理区的活动链表上的页数目

unsigned long nr\_inactive 管理区的非活动链表的页数目

unsigned long pages\_scanned 管理区内回收页框时使用的计数器

int all\_unreclaimable 管理区中填满不可回收页时此标志被置位

int temp\_priority 临时管理区的优先级（回收页框时使用）

int prev\_priority 管理区优先级，范围在12和0之间

wait\_queue\_head\_t\* wait\_table 进程等待队列的散列表，这些进程正在等待管理区中的某页

unsigned long wait\_table\_size 等待队列散列表的大小

unsigned long wait\_table\_bits 等待队列散列表数组大小

struct pglist\_data\* zone\_pgdat 内存节点

struct page\* zone\_mem\_map 指向管理区的第一个页描述符的指针

unsigned long zone\_start\_pfn 管理区第一个页框的下标

unsigned long spanned\_pages 以页为单位的管理区的总大小，包括洞

unsigned long present\_pages 以页为单位的管理区的总大小，不包括洞

char\* name 指针指向管理区的传统名称:“DMA”，“NORMAL”或“HightMeM”

1. 每个页描述符都有到内存节点和节点内管理区的链接。为了节省空间，这些链接的方式与典型的指针不同，而是被编码成索引存放在flags字段的高位。page\_zone()函数接收一个页描述符的地址作为它的参数；它读取页描述符的flags字段的最高位，然后通过查看zone\_table数组来确定相应管理区描述符的地址。在启动时所有内存节点的所有管理区描述符的地址初始化这个数组
2. 当内核调用一个内存分配函数时，必须指明请求页框所在的管理区。内核通常指明它愿意使用哪个管理区。为了在内存分配请求中指定首选管理区，内核使用zonelist\_t数据结构，这就是管理区描述符指针数组。

## 保留的页框池：

1. 可以用两种不同的方法来满足内存分配的请求。如果有足够的空闲内存可用，请求就会被立刻满足。否则，必须回收一些内存，并且将发出请求的内核控制路径阻塞，直到有内存被释放。

不过，当请求内存时，一些内核控制路径不能被阻塞——例如，这种情况发生在处理中断或在执行临界区内的代码时。在这种情况下，一条内核控制路径应当产生原子内存分配请求。原子请求从不被阻塞：如果没有足够的空闲页，则仅仅是分配失败而已。

尽管无法保证一个原子内存分配请求绝不失败，但是内核会设法减少这种不幸事件发生的可能性。为做到这一点，内核为院子内存分配保留了一个页框池，只有在内存不足时才使用。

1. 保留内存的数量（以KB为单位）存放在min\_free\_kbytes变量中。它的初始值在内核初始化时设置，并取决于直接映射到内核线性地址空间的第4个GB的物理内存的数量。——也就是说，取决于包含在ZONE\_DMA和ZONE\_NORMAL内存管理区的页框数目:

公式： P317

ZONE\_DMA和ZONE\_NORMAL内存管理区将一定数量的页框贡献给保留内存，这个数量与两个管理区的相对大小成比例。例如，如果ZONE\_NORMAL管理区比ZONE\_DMA大8倍，那么页框的7/8从ZONE\_NORMAL获得，1/8从ZONE\_DMA获得。

## 分页框分配器：

1. 概念：分页框分配器是一个内核子系统，处理对连续页框组的内存分配请求。
2. 组成如图：P318

管理区分配器部分接受动态内存分配和释放的请求。在请求分配的情况下，该部分搜索一个能满足所请求的一组连续页框内存的管理区。在每个管理区内，页框被名为“伙伴系统”的部分来处理。为了达到更好的系统性能，一小部分页框保留在高速缓存中用于快速满足对单个页框的分配请求。

1. 请求和释放页框的函数：
2. alloc\_pages(gfp\_mask, order)：用这个函数请求2^order连续的页框。成功返回第一个所分配页框描述符的地址，失败返回NULL
3. alloc\_page(gfp\_mask)：用于获得一个单独页框的宏；它扩展为alloc\_pages(gfp\_mask, 0)
4. \_\_get\_free\_pages(gfp\_mask, order)：该函数类似于alloc\_pages()，但它返回第一个所分配页的线性地址。
5. \_\_get\_free\_page(gfp\_make)：用于获取一个单独页框的宏；它扩展为：\_\_get\_free\_pages(gfp\_mask, 0)
6. \_\_get\_zeroed\_page(gfp\_mask)：函数用来获取填满0的页框；它调用：alloc\_pages(gfp\_mask | \_\_GFP\_ZERO, 0)，然后返回所获取页框的线性地址。
7. \_\_get\_dma\_pages(gfp\_mask, order)：这个宏用这个宏获取适用于DMA的页框，它扩展为: \_\_get\_free\_pages(gfp\_mask | \_\_GFP\_DMA, order)

// 参数gfp\_mask是一组标志，它指明了如何寻找空闲的页框。能在gfp\_mask中使用的标志 P320，以\_\_GFP开头

（7）\_\_free\_pages(page, order)：该函数检查page指向的页描述符，如果该页框未被保留（PG\_reservered标志为0），就把描述符的count字段减1。如果count值变为0就假定从与page对应的页框开始的2^order个连续页框不再被使用。这种情况下，该函数释放页框。

（8）剩余释放函数P321

## 高端内存页框的内核映射：

1. 与直接映射的物理内存末端、高端内存的始端所对应的线性地址存放在high\_memory变量中，它被设置为896MB。896MB边界以上的页框并不映射在内核线性地址空间的第4个GB中，因此，内核不能直接访问它们。这就意味着，返回所分配页框线性地址的页分配器函数不适用于高端内存（内核不能直接访问），即不适用于ZONE\_HIGHMEN内存管理区内的页框。
2. 在64位硬件平台上不存在这个问题，因为可使用的线性地址空间远大于能安装的RAM大小。简言之，这些体系结构的ZONE\_HIGHMEM管理区总是空的。但是在32位平台上，如80x86体系结构，达到PAE所支持的64GB，采用方法如下：
3. 高端内存页框的分配只能通过alloc\_pages()函数和它的快捷函数alloc\_pags()。这些函数不返回第一个被分配页框的线性地址，因为如果该页框属于高端内存，那么这样的线性地址根本不存在，取而代之，这些函数返回第一个被分配页框的页描述符的线性地址。这些线性地址总是存在的，因为所有的页描述符一旦被分配在低端内存中，它们在内存初始化阶段就不会改变。
4. 没有线性地址的高端内存中的页框不能被内核访问。因此，内核线性地址空间的最后128MB中一部分专门用于映射高端内存的页框（一个线性地址对应一个页框）。当然，这种映射是暂时的，否则则只有128MB的高端内存可以被访问。取而代之，通过重复使用线性地址，使用整个高端内存能在不同的时间被访问。
5. 内核可以采用三种不同的机制将页框映射到高端内存：永久内核映射、临时内核映射、及非连续内存分配。
6. 建立永久内核映射可能阻塞当前进程，这发生在空闲页表项不存在时。永久内核映射不能用于中断处理程序和可延迟函数。相反，建立临时内核映射绝不会要求阻塞当前进程，不过，它的缺点是只有很少的临时内核映射可以同时建立起来。
7. 使用临时内核映射的内核控制路径必须保证当前没有其他的内核控制路径在使用同样的映射。这意味着内核控制路径永远不能被阻塞，否则其他内核路径有可能使用同一个窗口（页表项，对应一个页）来映射其他的高端内存页框。
8. 永久内核映射：
9. 永久内核映射允许建立高端页框地址空间的长期映射。它们使用主内核页表中一个专门的页表，其地址存放在pk\_map\_page\_table变量中。页表中表项数有LAST\_PKMAP宏产生。页表包含512或1024项，这取决于PAE是否被激活。因此，内核一次最多访问2MB或4MB的高端内存。
10. 该页表映射的线性地址从PKMAP\_BASE开始。pkmap\_count数组包含LAST\_PKMAP个计数器，pkmap\_page\_table页表中的每一项都有一个。每一个计数器说明页表项的映射情况。我们区分以下三种情况：
11. 计数器为0：对应的页表项没有映射任何高端内存页框，并且是可用的。
12. 计数器为1：对应的页表项没有映射任何高端内存页框，但是它不能使用，因为自从它最后一次使用以来，其相应的TLB表现还未被刷新。
13. 计数器为n（远大于1）：相应的页表项映射一个高端内存页框，这意味着正好有n-1个内核成分在使用这个页框。
14. 为了记录高端内存页框与永久内核映射包含的线性地址之间的联系，内核试用了page\_address\_htable散列表。该表包含了一个page\_address\_map数据结构，用于高端内存中的每一个页框进行当前映射。而该数据结构还包含一个指向页描述符的指针和分配给该页框的线性地址。
15. page\_address()函数返回页框对应的线性地址，如果页框在高端内存中并且没有被映射，则返回NULL。这个函数接收一个页描述符page作为其参数，并区分以下两种情况：
16. 如果页框不在高端内存中（PG\_highmem标志为0），则线性地址总是存在并且是通过计算页框的下标，然后将其转化为物理地址，最后根据相应的物理地址得到线性地址。
17. 如果页框在高端内存中（PG\_highmem标志为1）中，该函数就到page\_address\_htable散列表中查找。如果散列表中找到页框，page\_address()就返回它的线性地址，否则返回NULL。
18. kmap()函数用于建立永久内核映射。P324
19. kumap()函数用于撤销先前有kmap()建立的永久内核映射。
20. 临时内核映射：
21. 留给临时内核映射的窗口数是非常少的。
22. 每个CPU都有它自己的包含13个窗口的集合，它们用enum km\_type数据结构表示。该数据结构中定义的每个符号，如KM\_BOUNCE\_READ等标识了窗口的线性地址。
23. 内核必须保证同一个窗口永不会被两个不同的控制路径同时使用。因此，km\_type结构中的每个符号只能由一种内核成分使用，并以该成分命名。最后一个KM\_TYPE\_NR本身并不代表一个线性地址，由每个CPU用来产生不同的可用窗口数。
24. 在km\_type中的每个符号（除最后一个）都是固定映射的线性地址的一个下标。enum fixed\_addresses数据结构包含符号FIX\_KMAP\_BEGIN和FIX\_KMAP\_END；后者赋值为下标FIX\_KMAP\_BEGIN + （KM\_TYPE\_NR\*NR\_CPUS）-1。在这种方式下，系统中每个CPU都有KM\_TYPE\_NR个固定映射的线性地址。此外，内核用fix\_to\_virt(FIX\_KMAP\_BEGIN)线性地址对应的页表项的地址初始化kmap\_pte变量。
25. 建立临时映射函数kmap\_atomic()：P326

## 伙伴系统算法：

1. Linux采用著名的伙伴系统算法来解决外碎片的问题。把所有的空闲页框分组为11块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512，1024个连续的页框。
2. 常规页的大小：2^12B = 4096B = 4KB
3. 详细算法：328
4. Linux2.6为每个管理区使用不同的伙伴系统。第一种处理适合ISA DMA的页框，第二种处理“常规”页框，第三种处理高端内存页框。每个伙伴系统使用的主要数据结构如下：
5. mem\_map数组（存放所有页描述符）：实际上，每个管理区都关系到mem\_map元素的子集。子集中第一个元素和元素的个数分别由管理区描述符的zone\_mem\_map和size字段指定。
6. 包含有11个元素，元素类型为free\_area的一个数组，每个元素对应一种块大小。该数组存放在管理区描述符的free\_area字段中。

5、

（1）我们考虑管理区描述符中free\_area数组的第k个元素，它标识所有大小为2^k的空闲块。这个free\_list字段是双向循环链表的头，这个双向循环链表集中了大小为2^k页的空闲块对应的页描述符。更精确的讲，该链表包含每个空闲页框**块**的起始地址页框的页描述符，指向链表中相邻元素的指针存放在页描述符的lru字段中。

（2）除了链表头外，free\_area数组的第k个元素同样包含字段nr\_free，它指定了大小为2^k页的空闲块的个数。当然，如果没有大小为2^k的空闲页框块，则nr\_free等于0且free\_list为空（free\_list的两个指针都指向它自己的free\_list字段）。

（3）最后，一个2^k的空闲页块的第一个页的描述符的private字段存放了块的order，也就是数组k。正是由于这个字段，当页块被释放时，内核可以确定这个块的伙伴是否也空闲，如果是的话，它就可以把两个块结合成大小为2^(k+1)页的单一块。

6、分配块：

（1）\_\_rmqueue()函数用来在管理区中找到一个空闲块。P329

7、释放块：

（1）\_\_free\_pages\_bulk()函数用来释放块。 P330

## 每CPU页框高速缓存：

1. 概念：每CPU页框高速缓存包含了一些预先分配的页框，它们备用于满足本地CPU发出的单一内存请求。实际上，这里为每个内存管理区和每个CPU提供了两个高速缓存：一个热高速缓存，它存放的页框中所包含的内容很可能就在CPU硬件高速缓存中；还有一个冷高速缓存。
2. 实现每CPU页框高速缓存的主要数据结构是存放在内存管理区描述符的pageset字段中的一个per\_cpu\_pageset数组数据结构。该数组包含为每个CPU提供的一个元素，这个元素依次由两个per\_cpu\_pages描述符组成，一个留给热高速缓存而另一个留给冷高速缓存。
3. per\_cpu\_pages字段：

类型 名称 说明

int count 高速缓存中的页框个数

int low 下界，表示高速缓存需要补充

int high 上界，表示高速缓存用尽

int batch 在高速缓存中将要添加或被删除的页框的个数

strcut list\_head list 高速缓存中包含的页框描述符链表

// 内核使用两个位标来监视热高速和冷高速缓存的大小：如果页框的个数低于下界low，内核通过伙伴系统中分配batch个页框来补充对应的高速缓存；否则，如果页框个数高于上界high，内核从高速缓存中释放batch个页框到伙伴系统中。值batch、low和high本质上取决于内存管理区中包含的页框个数。

## 通过每CPU页框高速缓存分配页框：

1. buffered\_rmqueue()函数在指定的内存管理区中分配页框。它使用每CPU页框高速缓存来处理单一页框请求。P333
2. free\_hot\_page()和free\_cold\_page()函数用于释放单个页框到每CPU页框高速缓存。P334；在当前的Linux2.6内核版本中，从没有页框被释放到冷高速缓存中；至于硬件高速缓存，内核总是假设被释放的页框是热的。当然，这并不意味着冷高速缓存是空的。当达到下界时，通过buffered\_rmqueue()补充冷高速缓存。

## 管理区分配器：

1. 概念：管理分配器是内核页框分配器的前端。该成分必须分配一个包含足够多的空闲页框的内存管理区，使它能满足内存请求。P334
2. \_\_alloc\_pages()函数是管理区分配器的核心（用于申请包含足够页框的内存管理区）。P335
3. \_\_free\_pages()函数用于释放页框。P338

## 内存区管理：

1. slab分配器：
2. slab分配器把内存去看作对象，这些对象由一组数据结构和几个叫做构造、析构函数组成。为了避免重复初始化对象，slab分配器并不丢弃已分配的对象，而是释放但把它们保存在内存中。当以后又要请求新的对象时，就可以从内存中获取而不用重新初始化。
3. 内核函数倾向于反复请求同一类型的内存区。在没有slab分配器时，内核把时间浪费在反复分配和回收那些包含同一内存区的页框上，slab分配器把那些页框保存在高速缓存中并很快的重新使用它们。
4. 对内存区的请求可以根据它们发生的频率来分类。对于预期频繁请求一个特定大小的内存区而言，可以通过创建一组具有适当大小的专用对象来高效的处理，由此以避免内存碎片的产生。

// P339

1. slab分配器把对象分组放进高速缓存。每个高速缓存都是同种类型对象的一种“储备”。
2. 包含高速缓存的主存区被划分为多个slab，每个slab由一个或者多个连续的页框组成，这些页框中既包含已分配的对象，也包含空闲的对象。
3. slab分配器的组成：P340
4. 高速缓存描述符 kmem\_cachae\_t类型的数据结构（等价于struct kmem\_cache\_s类型）： P340

类型 名称 说明

struct array\_cache\*[] array 每CPU指针数组包含指向空闲对象的本地高速缓存

unsigned int batchcount 要转移进本地高速缓存或从本地高速缓存中转移出的大批对象的数量

unsigned int limit 本地高速缓存中空闲对象的最大数组

struct kmem\_list3 lists 参见下一个表

unsigned int objsize 高速缓存中包含的对象的大小

unsigned int flags 描述高速缓存永久属性的一组标志

unsigned int num 封装在一个单独slab中的对象个数（高速缓存中的所有slab具有相同的大小）

unsigned int free\_limit 整个slab高速缓存中空闲对象的上限

spinlock\_t spinlock 高速缓存自旋锁

unsigned int gfporder 一个单独slab中包含的连续页框数目的对数

unsigned int gfpflags 分配页框时传递给伙伴系统函数的一组标志

size\_t colour slab使用的颜色个数

unsigned int colour\_off slab中的基本对齐偏移

unsigned int colour\_next 下一个被分配的slab使用的颜色

kmem\_cache\_t\* slabp\_cache 指针指向包含slab描述符的普通slab高速缓存

unsigned int slab\_size 单个slab的大小

unsigned int dflags 描述高速缓存动态属性的一组标志

void\* ctor 指向与高速缓存相关的构造方法的指针

void\* dtor 指向与高速缓存相关的析构方法的指针

const char\* name 存放高速缓存名字的字符数组

stuct list\_head next 高速缓存描述符双向链表使用的指针

1. lists字段是kmem\_list3结构体：

P341

1. slab描述符：P342
2. slab描述符可以存放在两个可能的地方：
3. 外部描述符：存放在slab外部，位于cache\_sizes指向的一个不适合ISA DMA的普通高速缓存中
4. 存放在slab内部，位于分配给slab的第一个页框的起始位置
5. 如果slab描述符放在slab外部，那么高速缓存描述符的flags字段中的CFLGS\_OFF\_SLAB标志被置1，否则它被置0

## 普通和专用高速缓存：

1. 普通高速缓存只由slab分配器用于自己的目的，而专用高速缓存由内核的其余部分使用
2. 普通高速缓存具体介绍：
3. 第一个高速缓存叫做kemem\_cache，包含由内核使用的其余高速缓存的高速缓存描述符。cache\_cache变量包含第一个高速缓存的描述符。
4. 另外一些高速缓存包含用作普通用途的内存区。内存区大小的范围一般包括13个几何分布的内存区。一个叫malloc\_sizes的表（其元素类型为cache\_sizes）分别指向26个高速缓存描述符，与其相关的内存区大小为32，65，128，256，512，1024，2048，4096，8192，16384，32768，65536和131072字节。每种大小，都有两个高速缓存：一个用于ISA DMA分配，另一个用于常规分配。
5. 在系统初始化期间调用kmem\_cache\_init()和kmem\_cache\_sizes\_init()来建立普通高速缓存。
6. 专用高速缓存是由kmem\_cache\_create()函数创建的。这个函数首先根据参数确定处理新高速缓存的最佳方法（例如，是在slab的内部还是外部包含slab描述符）。然后它从cache\_cache普通高速缓存中为新的高速缓存分配一个高速缓存描述符，并把这个描述符插入插入到高速缓存描述符的cache\_chain链表中（当获得了用于保护链表避免被同时访问的cache\_chain\_sem信号量后，插入操作完成）。
7. 还可以调用kemem\_cache\_destory()撤销一个高速缓存并将它从cache\_chain链表中删除。这个函数主要用于模块中，即模块装入时创建自己的高速缓存，卸载时撤销高速缓存。为了避免浪费内存空间，内核必须在撤销高速缓存本身之前就撤销其所有的slab。kmem\_cache\_shrink()函数通过反复调用slab\_destory()撤销高速缓存中的所有的高速缓存中的所有slab。

## slab分配器与分区页框分配器的接口：

1. 当slab分配器创建新的slab时，它依靠分区页框分配器来获得一组连续的空闲页框。kmem\_getpages()函数P344
2. 通过调用kmem\_freepages()函数可以释放分配给slab的页框 P345

## 给高速缓存分配slab：

1. 一个新创建的高速缓存中没有包含任何slab，因此也没有空闲的对象。只有当以下两个条件都为真时，才给高速缓存分配slab：
2. 已发出一个分配新对象的请求。
3. 高速缓存不包含任何空闲对象
4. 当这些情况发生时，slab分配器通过调用cache\_grow()函数给高速缓存分配一个新的slab。而这个函数调用kmem\_getpages()从分区页框中获得一组页框来存放一个单独的slab，然后又调用alloc\_slabmgmt()获得一个新的slab描述符。如果高速缓存描述符的CFLGS\_OFF\_SLAB标志置位，则从高速缓存描述符的slabp\_cache字段指向的普通高速缓存中分配给这个新的slab 描述符；否则，从slab的第一个页框中分配这个slab描述符。
5. 注意不可能从ZONE\_HIGHMEM内存管理区分配页框，因为kmem\_getpages()函数返回由page\_address()函数产生的线性地址，该函数为未映射的高端内存页框返回NULL
6. 给定一个页框，内核必须确定它是否被slab分配器使用，如果是，就迅速得到相应高速缓存和slab描述符的地址。因此，cache\_grow()扫描分配给新slab的页框的所有页描述符的地址，并将高速缓存描述符和slab描述符的地址分别赋给页描述符中lru字段的next和prev字段。这项工作不会出错，因为只有当页框空闲时伙伴系统的函数才会使用lru字段，而只要涉及伙伴系统，slab分配器函数所处理的页框就不空闲并将PG\_slab标志置位。
7. 接着，cache\_grow()调用cache\_init\_objs()，它将构造方法应用到了新slab包含的所有对象上。
8. 最后，cache\_grow()调用list\_add\_tail()来将新得到的slab描述符\*slabp，添加到高速缓存描述符\*cachep的全空slab链表的末端，并更新高速缓存中的空闲对象计数器。

## 从高速缓存中释放slab：

1. 在两种条件下才能撤销slab：
2. slab高速缓存中有太多的空闲对象
3. 被周期性调用的定时器函数确定是否有完全未使用的slab能被释放
4. 在两种情况下，调用slab\_destory()函数撤销一个slab，并释放相应页框到分区页框分配器。P346

## 对象描述符：

1. 每个对象都有类型为kmem\_bufctl\_t的一个描述符。对象描述符存放在一个数组中，位于相应的slab描述符之后。与slab描述符类似，slab的对象描述符也可以用两种可能的方式来存放：
2. 外部对象描述符：存放在slab的外面，位于高速缓存描述符的slabp\_cache字段指向的一个普通高速缓存中。内存区的大小取决于在slab中所存放的对象个数。
3. 内存对象描述符：存放在slab内部，正好位于描述符所描述的对象之前。
4. 数组中的第一个对象描述符描述的是slab中的第一个对象，依此类推。对象描述符只不过是一个无符号整数，只有在对象空闲时才有意义。它包含的是下一个空闲对象在slab中的下标，因此实现了slab内部空闲对象的一个简单链表。空闲对象链表中的最后一个元素的对象描述符用常规BUFCTL，END（0xffff）标记。
5. slab分配器所管理的对象可以在内存中进行对齐，也就是说，存放它们的内存单元的起始物理地址是一个给定常量的倍数，通常是2的倍数。这个常量就叫对齐因子。
6. slab分配器所允许的最大对齐因子是4096，即页框大小。这就意味着通过访问对象的物理地址或线性地址就可以对齐对象。
7. 通常情况下，如果内存单元的物理地址是字大小（即计算机内部内存总线的宽度）对齐的，那么，微机对内存单元的存取会非常快。因此，缺省情况下，kmem\_cache\_create()函数根据BYTES\_PER\_WORD宏所指定的字大小来对齐对象。对于80x86处理器，这个宏产生的值为4B，因为字长是32位。
8. kmem\_cache\_create() P349

## slab着色：

1. 同一个硬件高速缓存行可以映射RAM中很多不同块。相同大小的对象倾向与存放在高速缓存内相同的偏移处，因此，在不同的slab内具有相同偏移量的对象最终很可能映射在同一高速缓存行中。高速缓存的硬件很可能因此而花费内存周期在同一个高速缓存行与RAM单元之间来回传送几个内存单元。而其他的高速缓存行并未充分使用。
2. free变量表示在slab内未用的字节（没有分配给任一对象的字节）的个数。slab分配器利用空闲未用的字节free来对slab着色。“着色”只是用来再细分slab，并允许内存分配器把对象展开在不同的线性地址中。
3. 具有不同颜色（随机数）的slab把第一个对象存放在不同的内存单元，同时满足对齐约束。可用的颜色个数是free/aln（这个值存放在高速缓存描述符的colour字段）。
4. aln：对齐
5. 概念：利用slab中空闲的空间去做不同的偏移，这样就可以让相同大小的不同对象映射到不同的高速缓存行上，以提高访存效率。
6. 参考资料：<http://blog.csdn.net/njuitjf/article/details/18314807>
7. 续：<http://blog.chinaunix.net/uid-20196318-id-28854.html>

## 空闲Slab对象的本地高速缓存：

1. 为了减少处理器对自旋锁的竞争并更好地利用硬件高速缓存，slab分配器的每个高速缓存包含一个被称作slab本地高速缓存的每CPU数据结构，该数据结构由一个指向被释放对象的小指针数组组成。slab对象的大多数分配和释放只影响本地数组，只有在本地数组下溢或上溢时才涉及slab数据结构。
2. 高速缓存描述符的array字段是一组指向array\_cache数据结构的指针，系统的每个CPU对应于一个元素。每个array\_cache数据结构是空闲对象的本地高速缓存的一个描述符。
3. array\_cache结构的字段：

类型 名称 说明

unsigned int avail 指向本地高速缓存中可使用对象的指针的个数。它同时也作为高速缓存中第一个空槽的下标

unsigned int limit 本地高速缓存的大小，也就是本地高速缓存中指针的最大个数

unsigned int batchcount 本地高速缓存重新填充或腾空时使用的块大小

unsigned int touched 如果本地高速缓存最近已经被使用过，则该标志设为1

1. 本地高速缓存描述符并不包含本地高速缓存本身的地址，因为本地高速换正好位于本地缓存描述符之后。本地缓存存放的是指向已释放对象的指针，而不是对象本身，对象本身总是位于高速缓存的slab中。
2. 当创建一个新的slab高速缓存时，kmem\_cache\_create()函数决定本地高速缓存的大小（将这个值存放在高速缓存描述符的limit字段中）、分配本地高速缓存，并将本地高速缓存的指针存放在高速缓存描述符的array字段。这个大小取决于存放在slab高速缓存中对象的大小，范围从1（相对非常大的对象）到120（相对小的对象）。此外，batchcount字段的初始值，也就是从一个本地高速缓存的块里添加或删除的对象的个数，被初始化为本地高速缓存大小的一半。
3. 在多处理器系统中，小对象使用slab高速缓存同样包含一个附加的本地高速缓存，它的地址被存放在高速缓存描述符的lists.shared字段中（该字段存放被所有CPU共享的本地高速缓存的地址）。共享的本地高速缓存正如它的名字暗示的那样，被所有CPU共享，它使得将空闲对象从一个本地高速缓存移动到另一个高速缓存的任务更容易。

## 分配slab对象：

1. 通过kmem\_cache\_alloc()函数可以获得新对象。P352

## 释放Slab对象：

1. 通过kmem\_cache\_free()函数释放一个曾经有slab分配器分配给某个内核函数的对象。P353

## 通用对象：

1. 如果对存储区的请求不频繁，就用一组普通的高速缓存来处理，普通高速缓存中的对象具有几何分布的大小，范围为32~131072字节。
2. 调用kmalloc()函数就可以得到这种类型的对象 P355
3. 书上讲述较少，日后自己再学。

## 内存池：

1. 基本上讲，一个内存池允许一个内核成分，仅在内存不足的紧急情况下分配一些动态内存来使用。
2. 不应该将内存池与前面“保留的页框池”一节中描述的保留页框混淆。实际上这些页框只能用于满足中断处理程序或内部临界区发出的原子内存分配请求。而内存池是动态内存的储备，只能被特定的内核成分（即池的拥有者）使用。拥有者通常不使用储备，但是，如果动态内存变得极其稀有以至于所有普通内存分配请求都将失败的话，那么作为最后的解决手段，内核成分就能调用特定的内存池函数提取储备得到所需的内存。
3. 一个内存池常常叠加在slab分配器之上——也就是说，它被用来保存slab对象的储备。但是一般而言，内存池能被用来分配任何一种类型的动态内存。因此，我们一般将内存池处理的内存单元看作“内存元素”。
4. 内存池由mempool\_t对象描述：

类型 名称 说明

spinlock\_t lock 用来保护对象字段的自旋锁

int min\_nr 内存池中元素的最大个数

int curr\_nr 当前内存池中元素的个数

void\*\* elements 指向一个数组的指针，该数组由指向保留元素的指针组成

void\* pool\_data 池的拥有者可获得的私有数据

mempool\_alloc\_t\* alloc 分配一个元素的方法

mempool\_free\_t\* free 释放一个元素的方法

wait\_queue\_head\_t wait 当内存池为空时使用的等待队列

1. 当内存元素是slab对象时，alloc和free方法一般由mempool\_alloc\_slab()和mempool\_free\_slab()函数实现。它们只是分别调用kmem\_cache\_alloc()和kmem\_cache\_free()函数。在这种情况下，mempool\_t对象的pool\_data字段存放了slab高速缓存描述符的地址。
2. mempool\_create()函数创建一个新的内存储。P357
3. mempool\_destory()函数释放池中所有的内存元素，然后释放元素数组和mempool\_t对象自己。P357
4. mempool\_alloc()函数：从内存池分配一个元素 P357
5. mempool\_free()函数：释放一个元素到内存池

# 非连续内存区管理：

非连续内存区：连续的线性地址映射到非连续的页框

## 非连续内存区的线性地址：358

1. 非连续内存区的描述符：每个非连续内存区都对应一个vm\_struct类型的描述符 P359

类型 名称 说明

void\* addr 内存区内第一个内存单元的线性地址

unsigned long size 内存区的大小加4096（4096是内存区之间的安全区间的大小）

unsigned long flags 非连续内存区映射的内存的类型

struct page\*\* pages 指向nr\_pages数组的指针，该数组由指向页描述符的指针组成。

unsigned int nr\_pages 内存区填充的页的个数

unsigned long phys\_addr 该字段设为0，除非内存已被创建来映射一个硬件设备的I/O共享内存

struct vm\_struct\* next 指向下一个vm\_struct结构的指针

1. flags字段标识了非连续区映射的内存的类型：VM\_ALLOC表示使用vmalloc()得到的页，VM\_MAP表示使用vmap()映射的已经被分配的页，而VM\_IORENAP表示使用ioremap()映射的硬件设备的板上内存。
2. get\_vm\_area()函数在线性地址VMALLOC\_START和VMALLOC\_END之间查找一个空闲区域。 P359
3. vmalloc()函数给内核分配一个非连续内存区。P360 （除了vmalloc()函数外，非连续内存区还能由vmalloc\_32()函数分配，该函数与vmalloc很相似，但是它只从ZONE\_NORMAL和ZONE\_DMA内存管理区中分配页框）
4. pud\_alloc()来为新内存区创建一个页上级目录，并把它的物理地址写入内核页全局目录的适合表项。P362
5. alloc\_area\_pud()为新的页上级目录分配相关的页表。 P362
6. vmalloc()修改内核页表项，使得内核可以通过页表项来把线性地址转化为物理地址。
7. vfree()函数释放vmalloc()或vmalloc\_32()创建的非连续内存区，vumap()函数释放vmap()创建的内存区。P363