空天信息安全与可信计算
教育部重点实验室

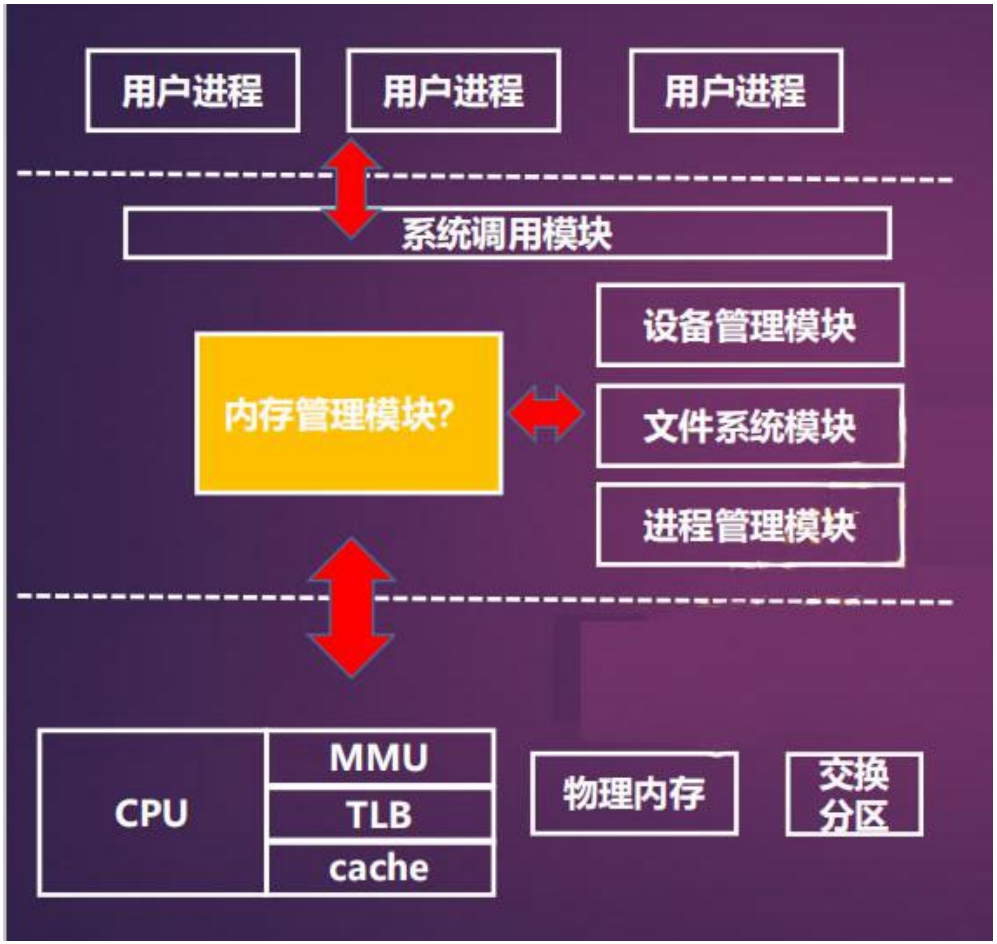
Linux分析与安全设计

第二章 Linux内存管理和内存攻防

第二章 Linux内存管理和内存攻防

- 内存管理究竟管什么
- 物理内存管理
- 虚拟内存管理
- 请页机制(缺页中断)
- 交换空间管理
- 缓冲机制

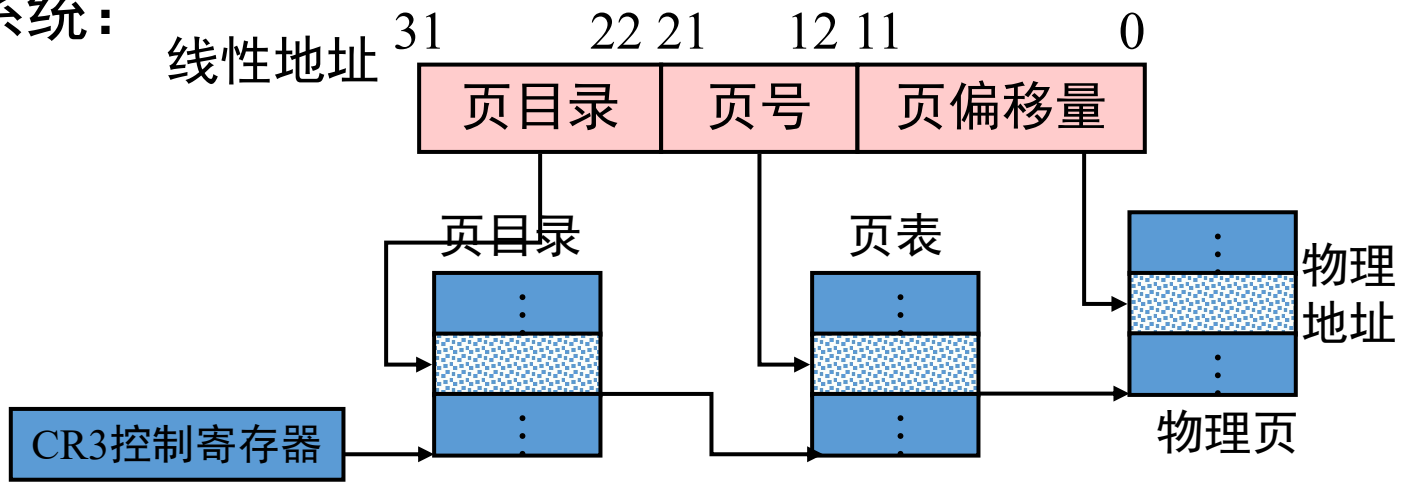
内存管理究竟管什么



- 内存初始化
  - 开启MMU
  - 线性映射
- 页表的管理
- 页面分配器
- 小块内存分配器：slab机制
- vmalloc机制
- 虚拟内存管理
- 缺页中断
- 页面回收

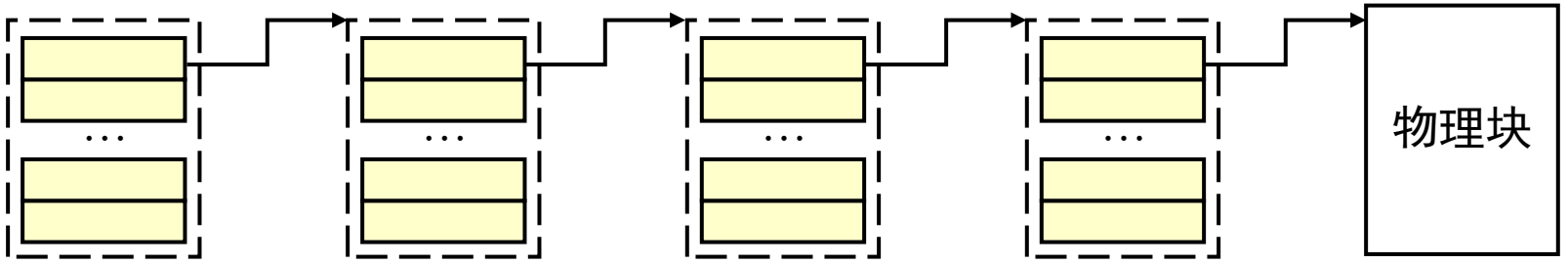
Linux内存管理

32位系统：



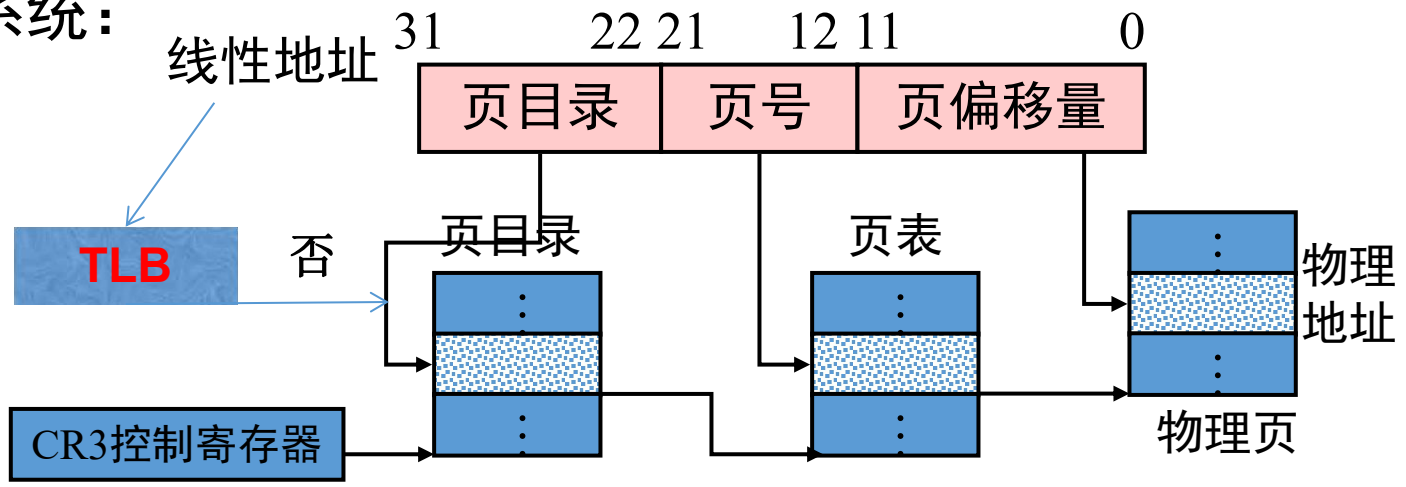
64位系统：

页全局目录(PGD) 页上级目录(PUD) 页中间目录(PMD) 页表(PT)



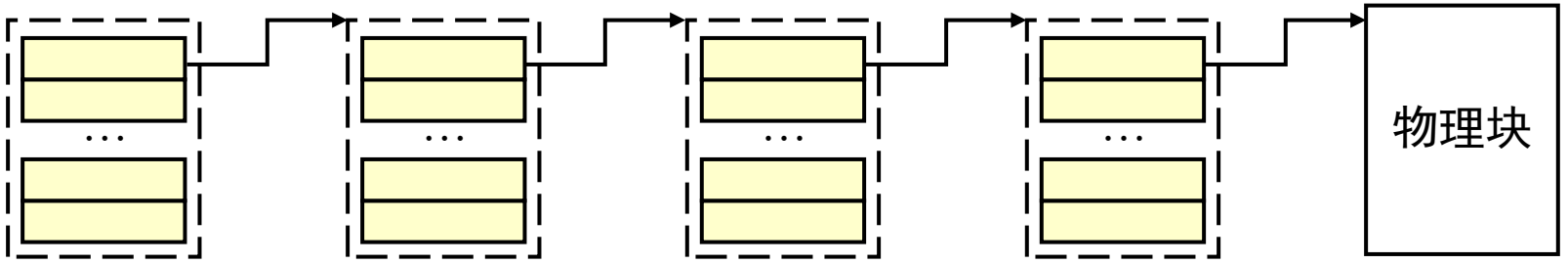
Linux内存管理

32位系统：



64位系统：

页全局目录(PGD) 页上级目录(PUD) 页中间目录(PMD) 页表(PT)



物理内存空间管理

- 数据结构
- 基于Buddy算法的内存页面管理
- 基于slab算法的内存分区管理

物理内存空间管理

- 数据结构
  - 分页管理结构
  - 设置了一个mem_map[]数组管理内存页面page，其在系统初始化时由 free_area_init()函数创建。数组元素是一个个page结构体，每个page结构体对应一个物理页面。
  - strcut page类型定义在/include/linux/mm_types.h中。
  - mem_map[]数组定义在mm/page_alloc.c

物理内存空间管理

```
struct page {
        unsigned flags;
        struct address_space *mapping;
        atomic_t _mapcount;
        atomic_t _refcount;
        unsigned int active;
        void *virtual;
        struct list_head lru
        struct page *next;

    }
```

物理内存空间管理

- 说明：
  - flag

    用来存放页的状态，每一位代表一种状态，所以至少可以同时表示出32中不同的状态
  - mapping

    指向与该页相关的address_space对象
  - mapcount

    被页表映射的次数，也就是说该page同时被多少个进程共享
  - refcount

    引用计数，表示内核中引用该page的次数。当该值为0时，表示没有引用该page的位置，所以该page可以被解除映射，这往往在内存回收时是有用的

物理内存空间管理

- active

  经常被访问的处于活跃状态的页面

- virtual

  指向页对应虚拟地址的指针。

- list_head lru

  链表头，用于在各种链表上维护该页，主要有3个用途：伙伴算法，slab分配器，被用户态使用或被当做页缓存使用。

- page *next

  指向slab链表的指针

物理内存空间管理

- Buddy算法
  - Linux对空闲内存空间管理采用Buddy算法。
  - Buddy算法
    - 把内存中所有页面按照2^n划分，其中n=0~5，每个内存空间按1个页面、2个页面、4个页面、8个页面、16个页面、32个页面进行六次划分。
    - buddy算法的相关代码在mm/page_alloc.c中。

物理内存空间管理

- ✓ 划分后形成了大小不等的存储块，称为页面块，简称页块。包含1个页面的页块称为1页块，包含2个页面的称为2页块，依此类推。
- ✓ 每种页块按前后顺序两两结合成一对Buddy“伙伴”
- ✓ 系统按照Buddy关系把具有相同大小的空闲页面块组成页块组，即1页块组、2页块组……32页块组。
- ✓ 每个页块组用一个双向循环链表进行管理，共有 6 个链表，分别为1、2、4、8、16、32页块链表。

分别挂到free_area[] 数组上。

物理内存空间管理

- 位图数组
  - ✓ 标记内存页面使用情况，第0组每一位表示单个页面使用情况，1表示使用，0表示空闲，第2组每一位表示比邻的两个页面的使用情况，依次类推。默认为10个数组。
    - 当一对Buddy的两个页面块中有一个是空闲的，而另一个全部或部分被占用时，该位置1。
    - 两个页面块都是空闲，或都被全部或部分占用时， 对应位置0。

物理内存空间管理

- 内存分配和释放过程
  - 内存分配时，系统按照Buddy算法，根据请求的页面数在free_area[]对应的空闲页块组中搜索。
    - 若请求页面数不是2的整数次幂，则按照稍大于请求数的2的整数次幂的值搜索相应的页面块组。

物理内存空间管理

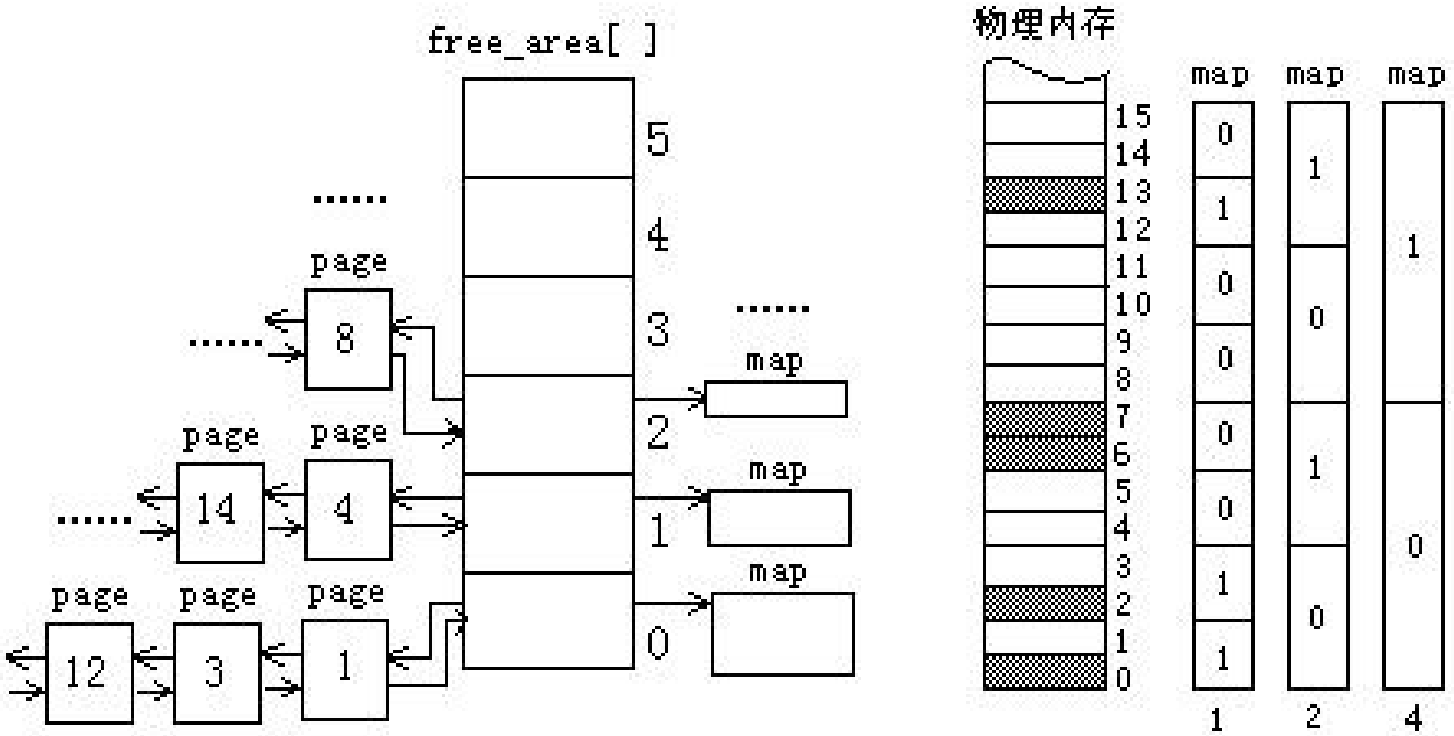
- 当相应页块组中没有可使用的空闲页面块时就查询更大一些的页块组，
- 在找到可用的空闲页面块后，分配所需页面。
- 当某一空闲页面块被分配后，若仍有剩余的空闲页面，则根据剩余页面的大小把它们加入到相应页块组中。

物理内存空间管理

- ✓ 内存页面释放时，系统将其做为空闲页面看待。

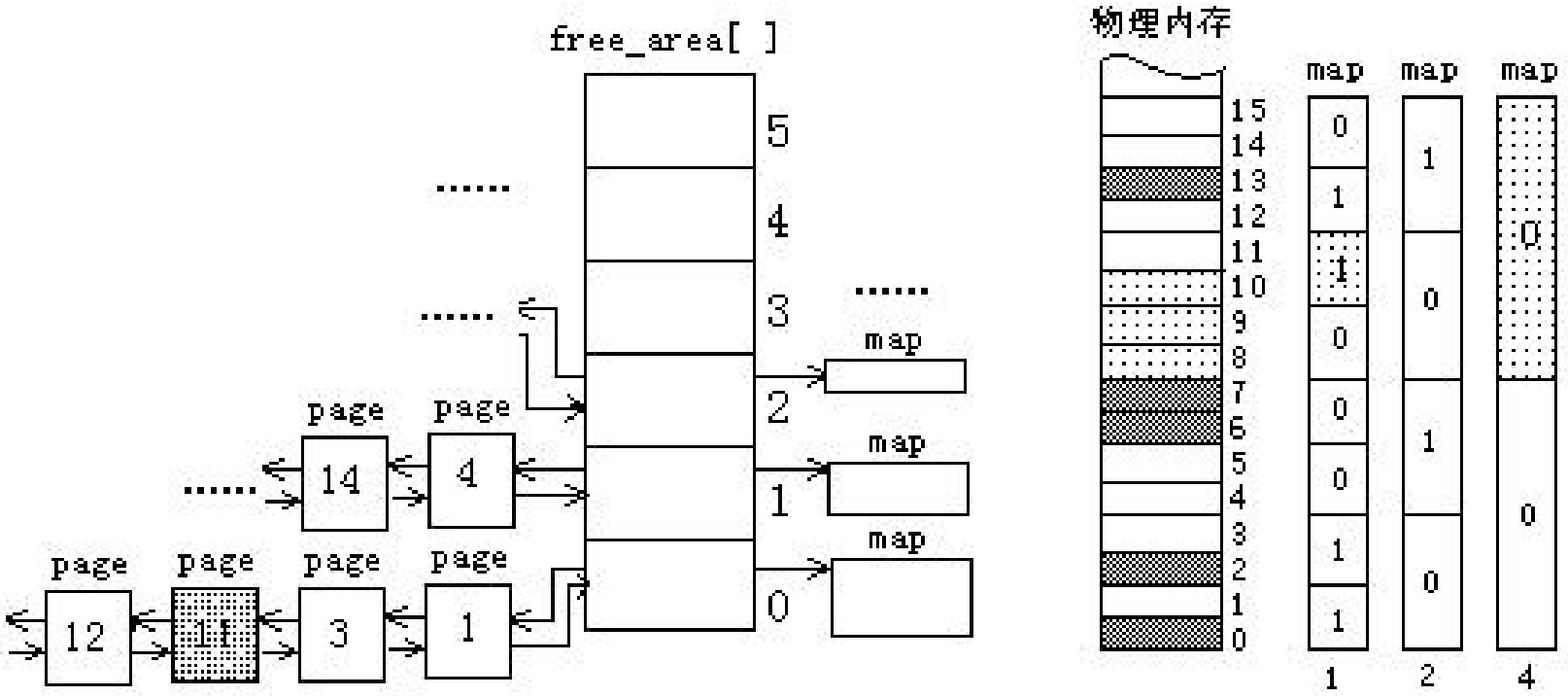
  检查是否存在与这些页面相邻的其它空闲页块，若存在，则合为一个连续的空闲区按Buddy算法重新分组。

物理内存空间管理



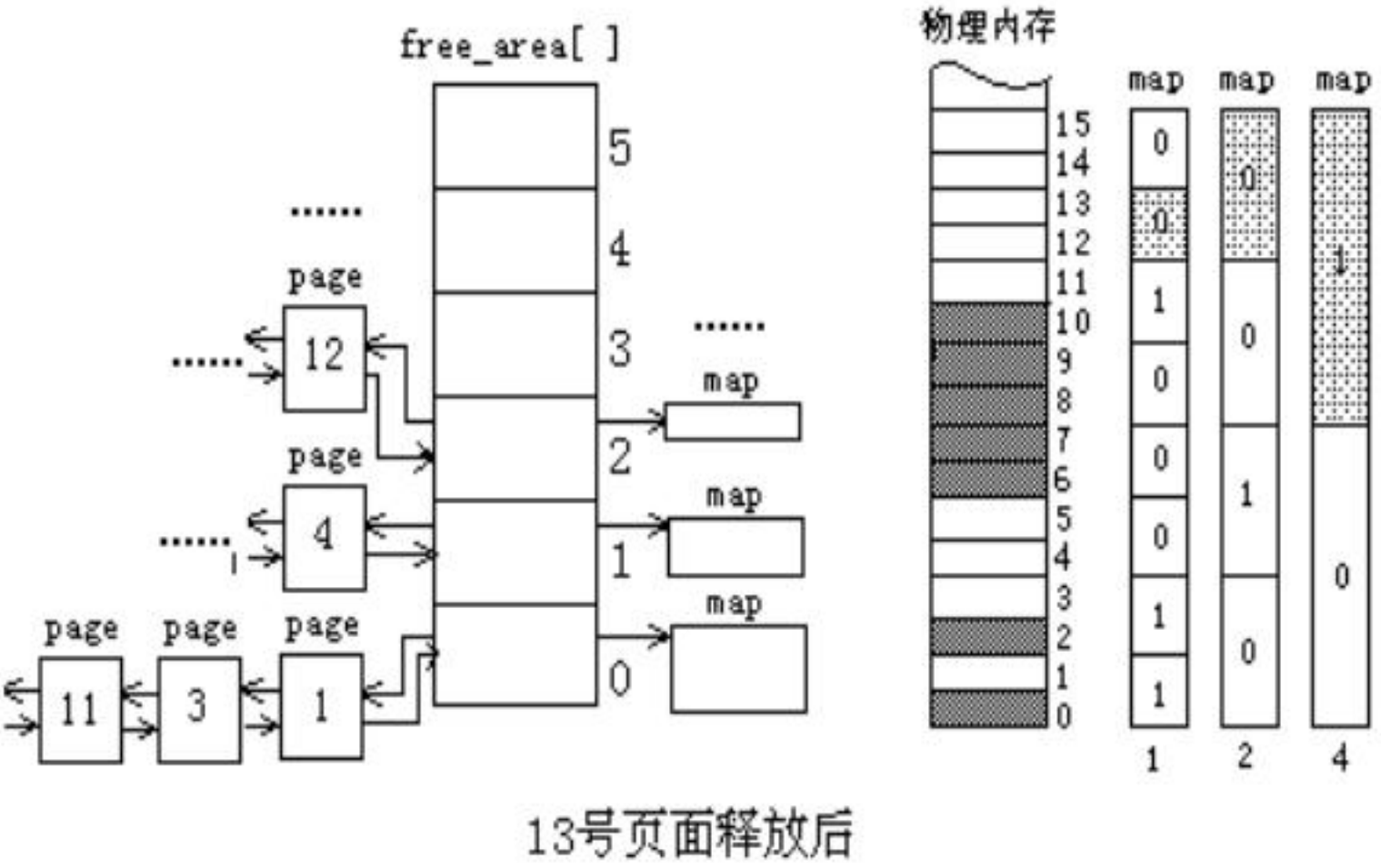
内存分配的Buddy算法

物理内存空间管理



8、9、10页面分配后的示意

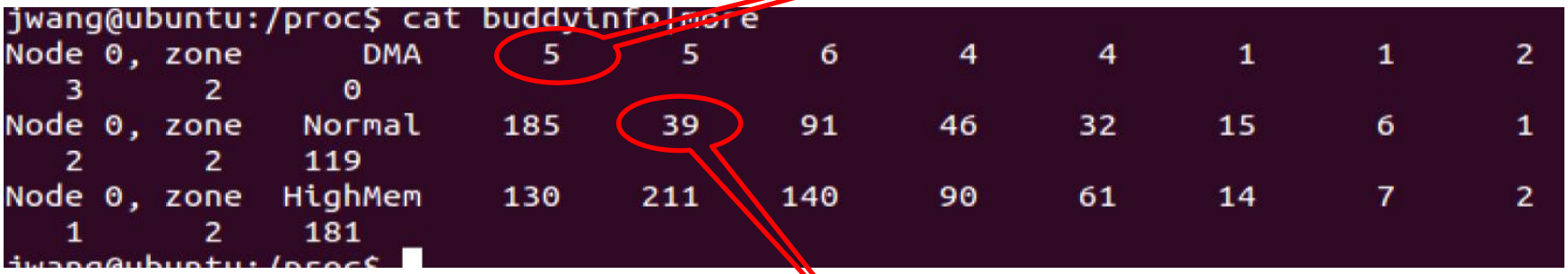
物理内存空间管理



13号页面释放后

物理内存空间管理

查看系统中buddy内存空情况 cat proc/buddyinfo



buddy算法的相关代码在mm/page_alloc.c中

物理内存空间管理

- Slab算法
  - 采用buddy算法，解决了外碎片问题，这种方法适合大块内存请求，不适合小内存区请求。例如：mm_struct就只需要差不多半个页面。
  - Linux2.0采用传统内存分区算法，按几何分布提供内存区大小。虽然减少了内碎片，但没有显著提高系统效率。

物理内存空间管理

- Linux2.4以后采用了slab分配器算法

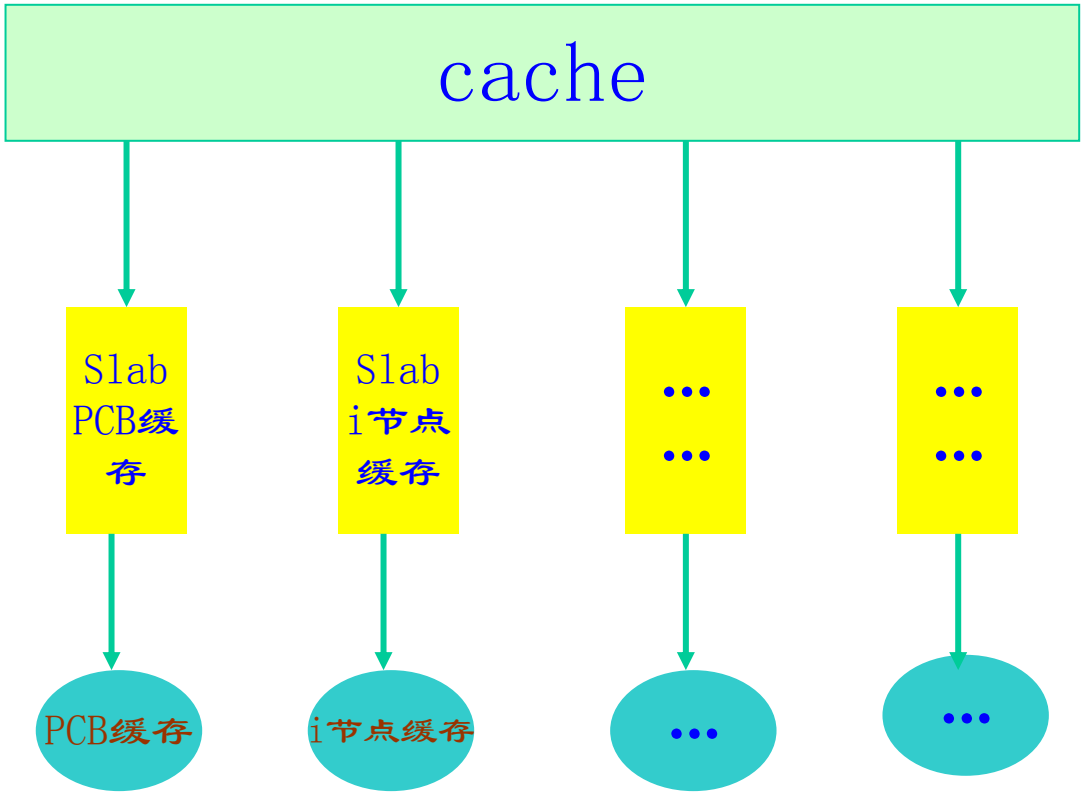
  该算法比传统的分配器算法有更好性能和内存利用率，最早在solaris2.4上使用。

- slab分配器是基于对象进行管理的，所谓的对象就是内核中的数据结构（例如：task_struct,file_struct 等）

物理内存空间管理

- Slab分配器思想
  - 小对象的申请和释放通过slab分配器来管理。
  - slab分配器有一组高速缓存，每个高速缓存保存同一种对象类型，如i节点缓存、PCB缓存等。
  - 内核从它们各自的缓存中分配和释放对象。
  - 每种对象的缓存区由一连串slab构成，每个slab由一个或者多个连续的物理页面组成。这些页面种包含了已分配的缓存对象，也包含了空闲对象。

物理内存空间管理



Slab分配器的组成

物理内存空间管理

- 数据结构
  - 高速缓存

    Struct kmem_cache_s
  - 高速缓存中的每个slab数据结构

    Struct kmem_slab_s

物理内存空间管理

如：创建一个32个字节的新 slab 缓存

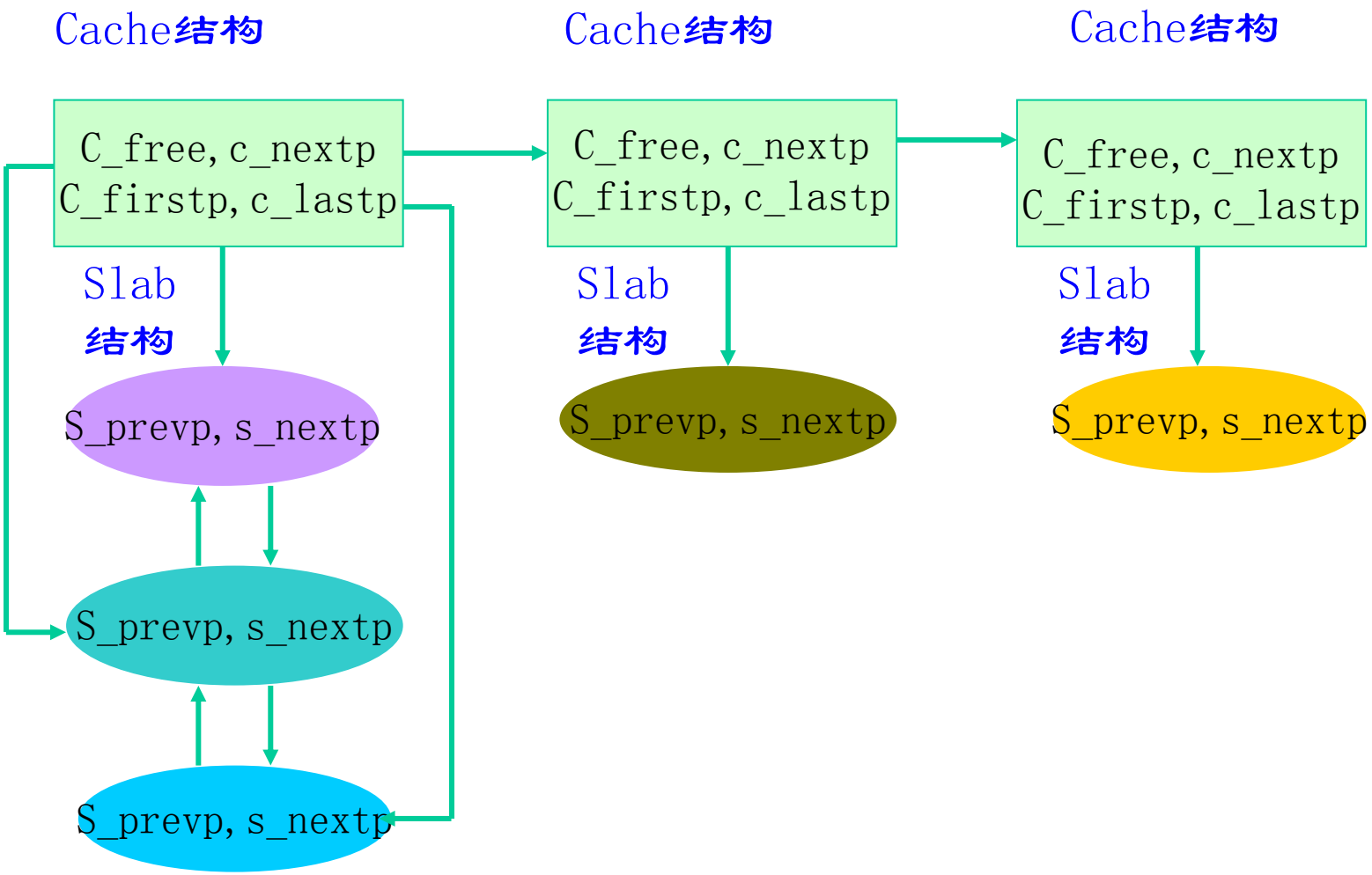
```
static struct kmem_cache *my_cachep;

static void init_my_cache( void )
{

   my_cachep = kmem_cache_create(
                  "my_cache",              /* Name */
                  32,                      /* Object Size */
                  0,                       /* Alignment */
                  SLAB_HWCACHE_ALIGN,      /* Flags */
                  NULL, NULL );            /* Constructor/Deconstructor */

   return;
}
```

物理内存空间管理



Cache结构与slab结构之间的关系

物理内存空间管理

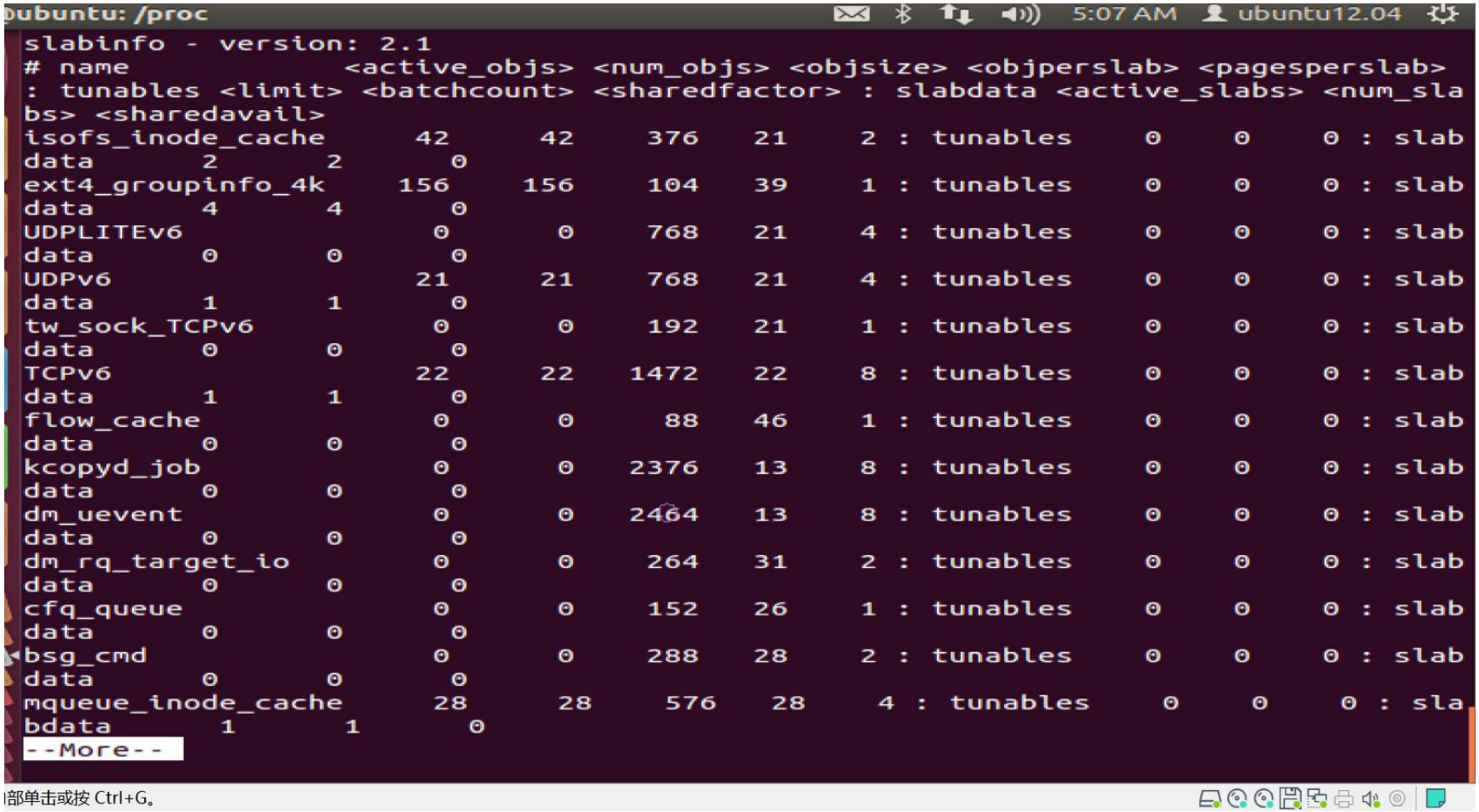
- 创建slab对象过程
  - Slab分配器调用Kmem_cache_creat和Kmem_cache_grow() 函数为缓存分配一个新的slab对象。
  - 调用kmem_getpages()获取一组连续内存页框；
  - 调用kmem_cache_slabmgamt()分配一个新的slab数据结构；
  - 调用kmem_cache_init_objs（）为新slab中包含的所有对象定义构造方法；
  - 调用kmem_slab_link_end()将新的slab插入到这个高速缓存的双向链表的末尾。

物理内存空间管理

- 优点：
  - 充分利用了空间，减少了内部碎片
  - 管理局部化，尽可能减少了与buddy分配器打交道，提高了效率。
  - 从版本 2.6.24 开始，SLUB 分配器取代 SLAB，成为 Linux 内核的默认分配器。
    - SLUB 通过减少 SLAB 分配器所需的大量开销，来解决 slab 分配的性能问题
    - 对于 SLAB 分配器，每个 CPU 都有队列以维护每个 cache 内的对象，SLUB 会删除这些队列。

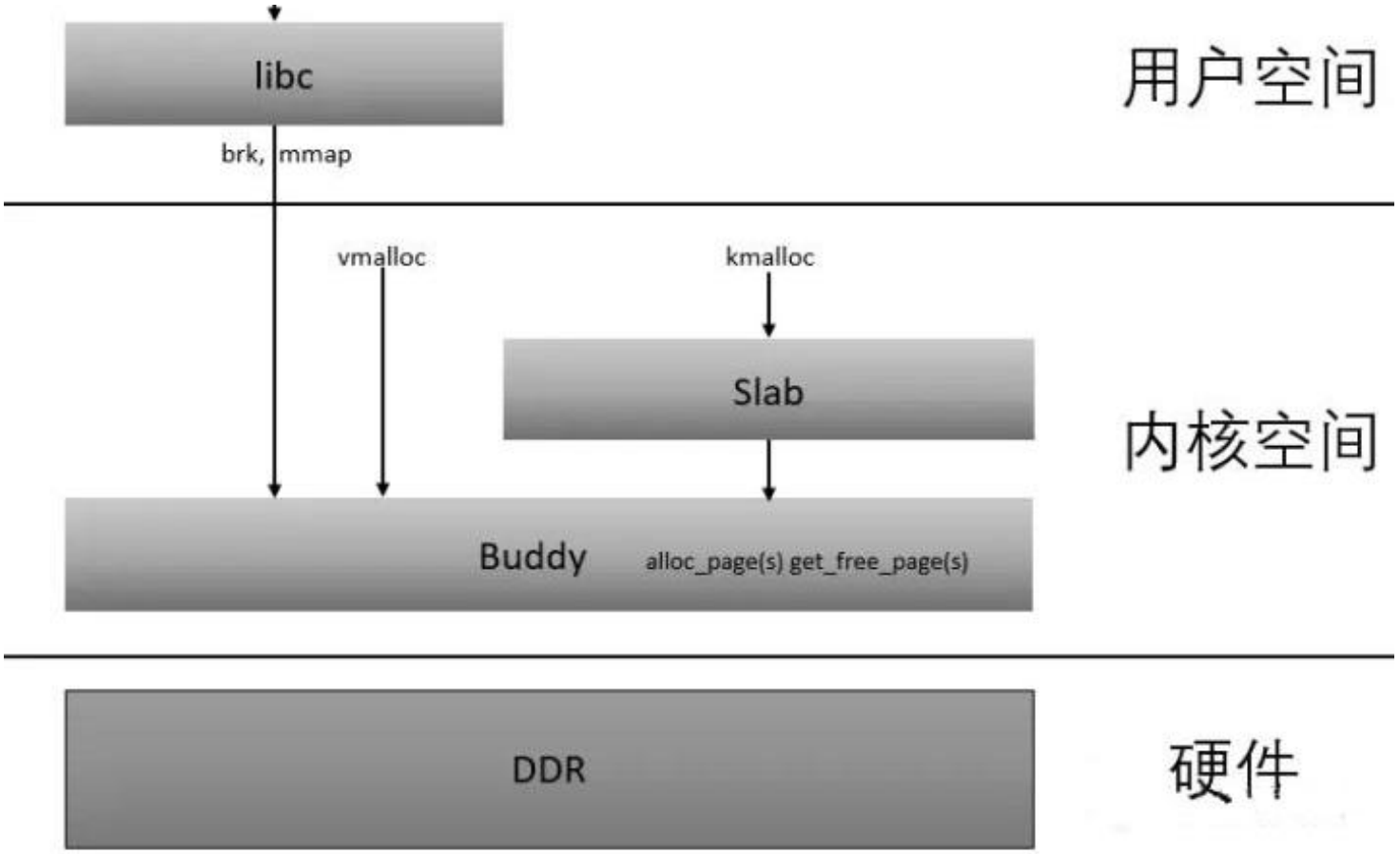
物理内存空间管理

查看系统中slab情况 cat proc/slabinfo



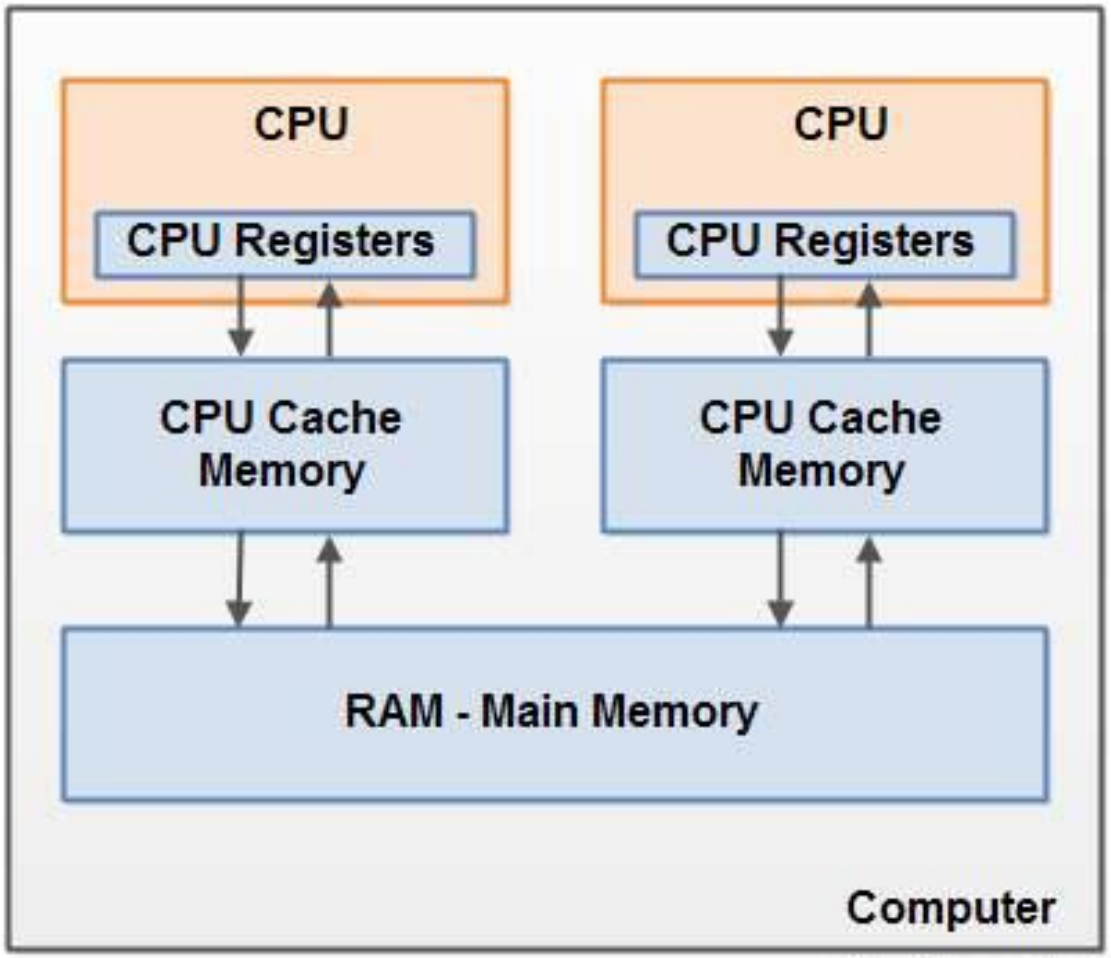
物理内存空间管理

Slab与Buddy的关系



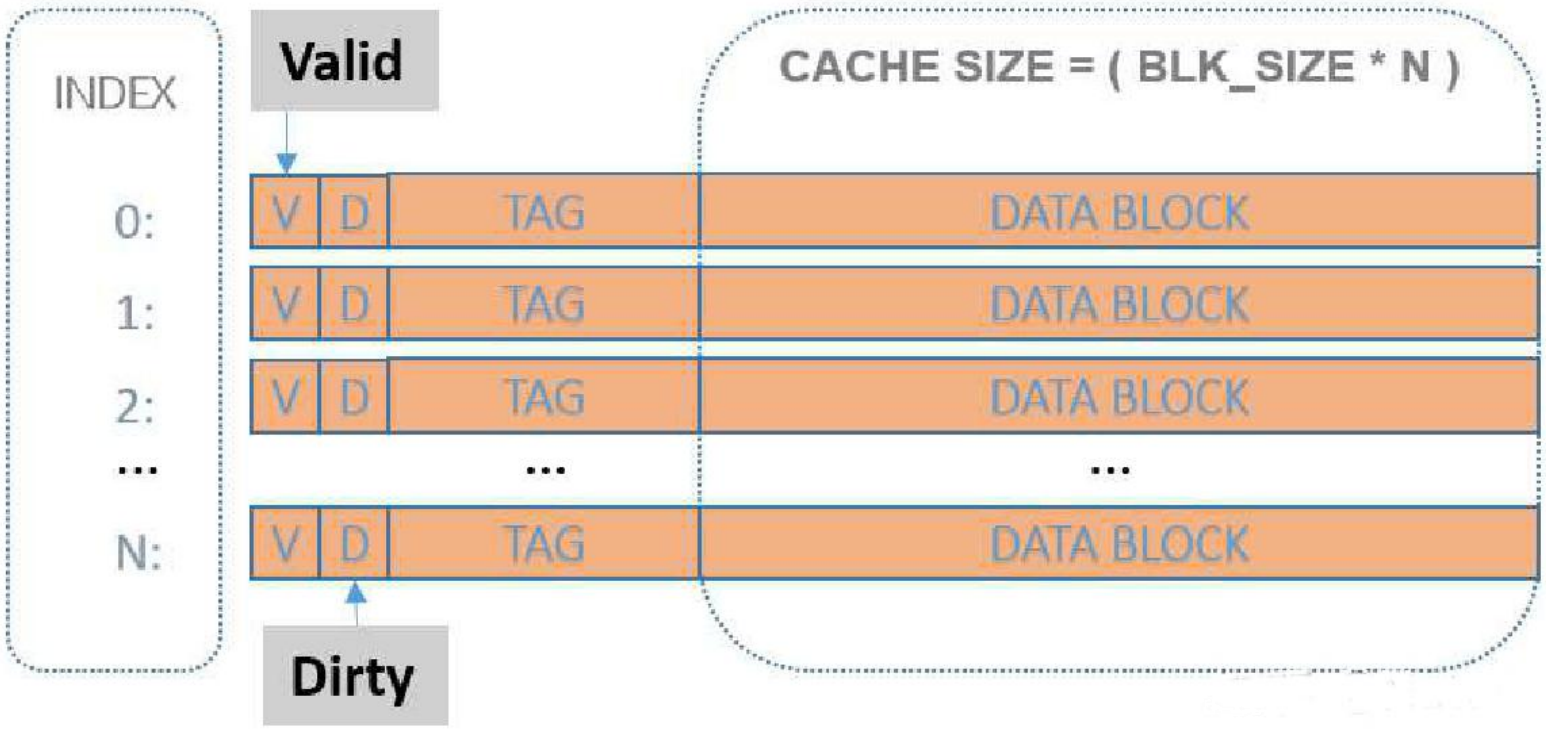
物理内存空间管理

Cache



物理内存空间管理

Cache存储



虚拟地址空间管理

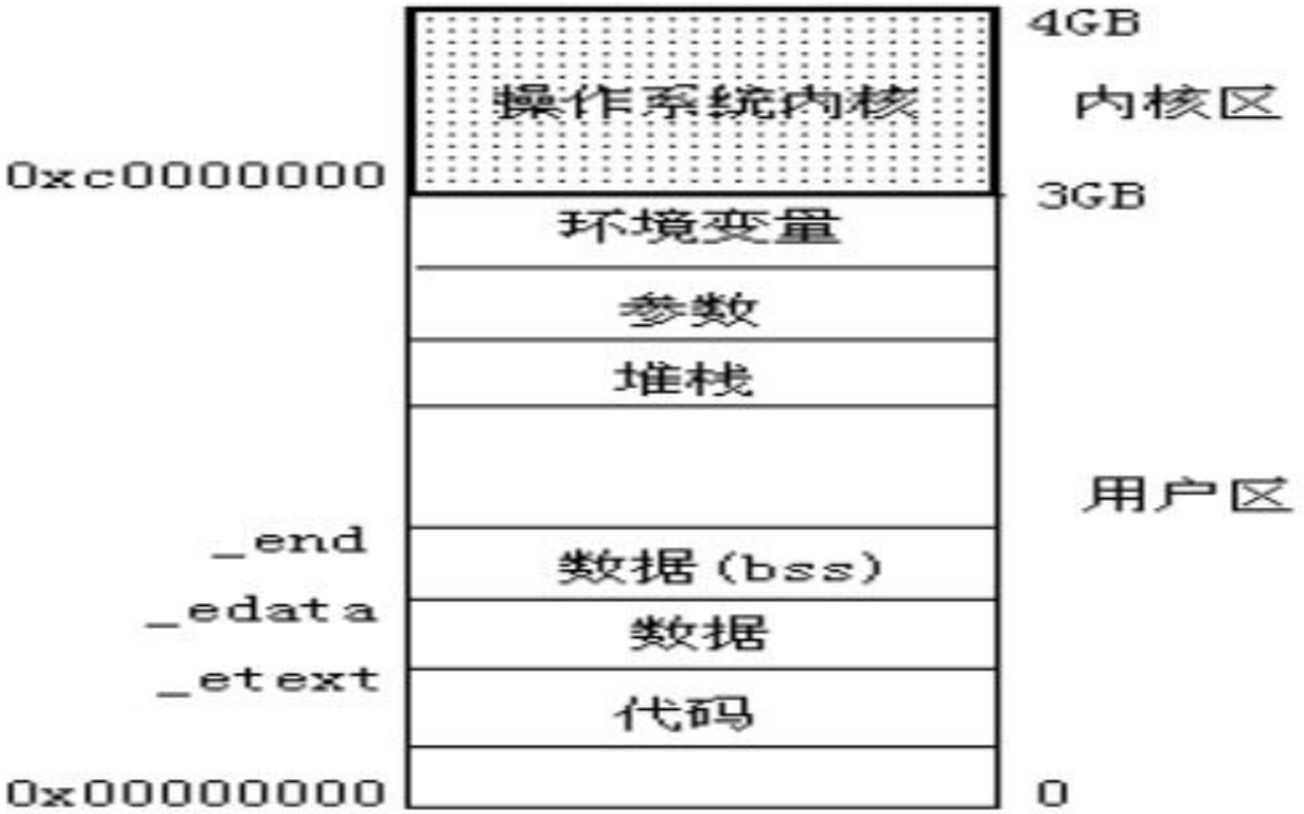
- 进程地址空间
- 虚拟内存空间管理（内存映射）

进程地址空间

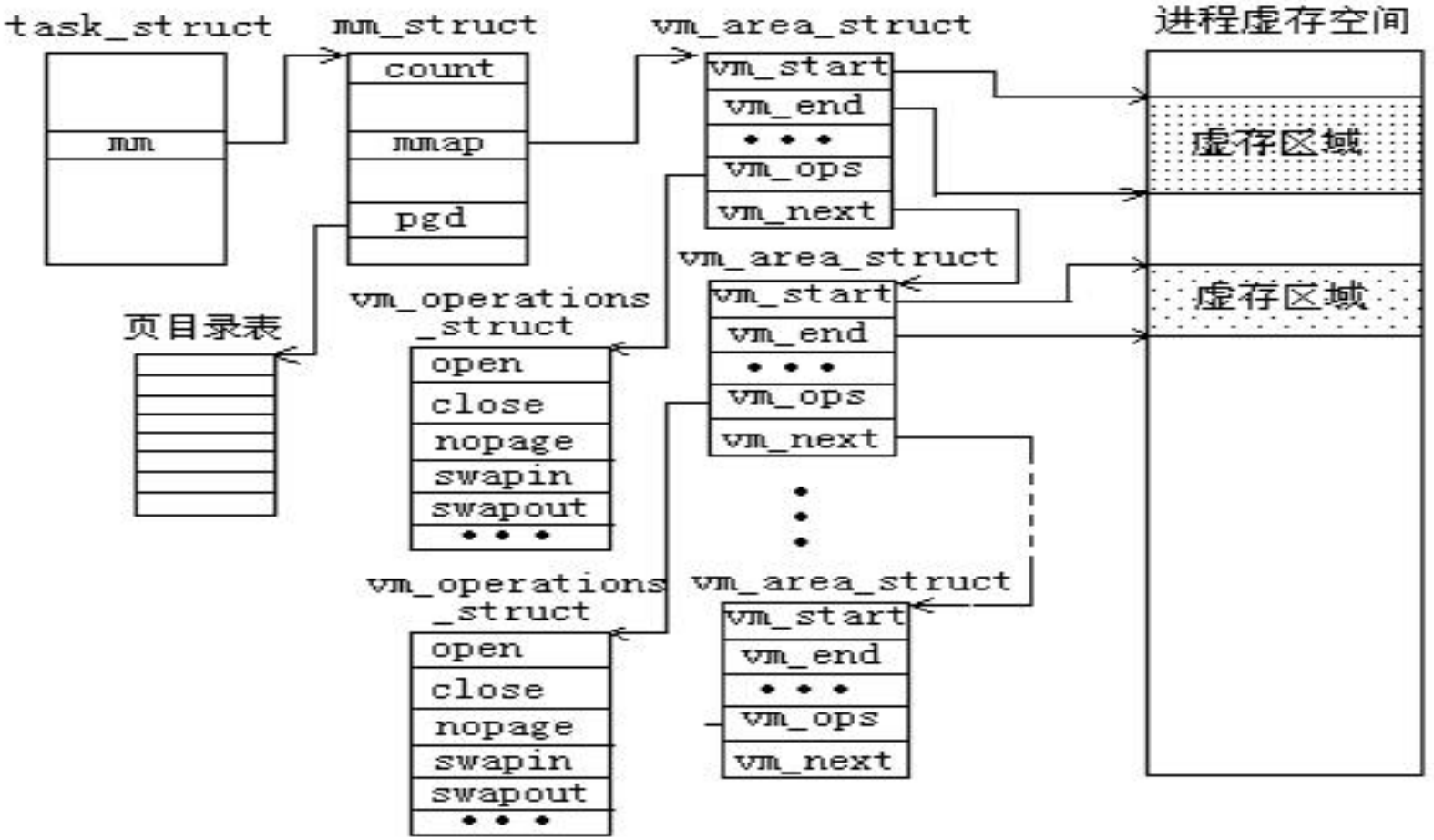
- Linux把进程虚拟空间分成两部分：内核区和用户区
- 操作系统内核的代码和数据等被映射到内核区。进程可执行映像（代码和数据）映射到虚拟内存的用户区。

进程地址空间

进程的虚拟内存空间



虚拟地址空间管理



进程虚存空间管理

虚拟地址空间管理

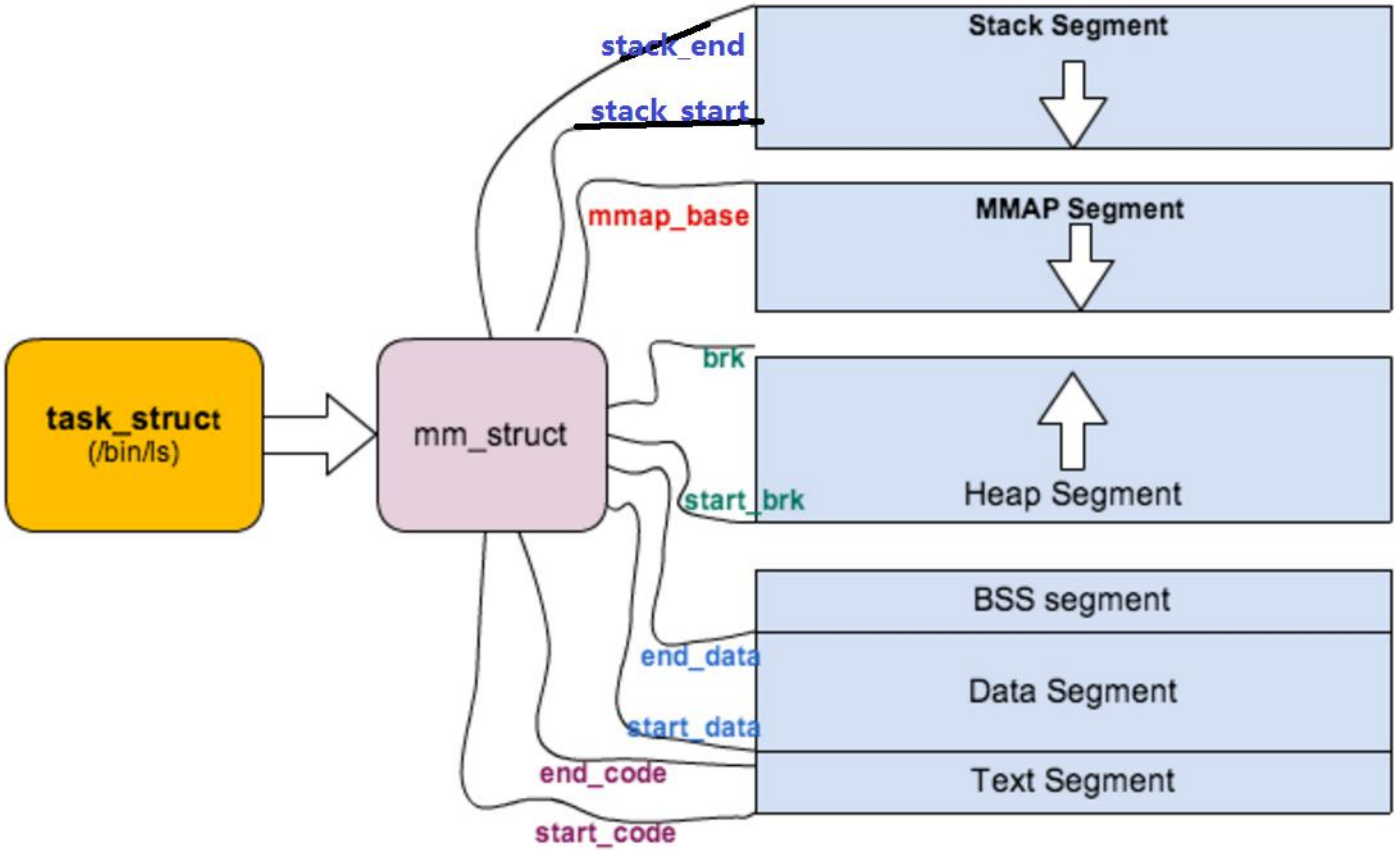
- 有关数据结构
  - ✓mm_struct结构体

    定义了每个进程的虚存用户区，首地址在任务结构体中，定义在/include/linux/mm_types.h中。

虚拟地址空间管理

```
struct mm_struct {
        int count;      //*引用计数/
        pgd_t * pgd;    /*指向全局页目录表/
        unsigned long start_code, end_code, start_data,
    end_data;
        unsigned long start_brk, brk, start_stack,
    start_mmap;
        unsigned long arg_start, arg_end, env_start,
    env_end;
        unsigned long rss, total_vm, locked_vm
        struct vm_area_struct * mmap;
        struct vm_area_struct * mmap_avl;
};
```

虚拟地址空间管理



虚拟地址空间管理

- ✓进程虚存区域

  一个虚存区域是虚存空间中一个连续区域，每个虚拟区域用一个vm_area_struct结构体描述，定义在/include/linux/mm.h中。

虚拟地址空间管理

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    pgprot_t vm_page_prot;
    unsigned short vm_flags;
    short vm_avl_height;
        struct vm_area_struct * vm_avl_left;
        struct vm_area_struct * vm_avl_right;
        struct vm_area_struct * vm_next;
        struct vm_area_struct * vm_next_share;
        struct vm_area_struct * vm_prev_share;
        struct vm_operations_struct * vm_ops;
        unsigned long vm_offset;
        struct inode * vm_inode;
        unsigned long vm_pte;
};
```

虚拟地址空间管理

✓说明：

- vm_mm指针指向进程的mm_struct结构体
- vm_start和vm_end 虚拟区域的开始和终止地址。
- vm_page_prot　虚存区域的页面的保护特性
- vm_inode

  若虚存区域映射的是磁盘文件或设备文件内容，则vm_inode指向这个文件的inode结构体，否则vm_inode为NULL。

虚拟地址空间管理

- vm_flags指出了虚存区域的操作特性：

VM_READ	虚存区域允许读取
VM_WRITE	虚存区域允许写入
VM_EXEC	虚存区域允许执行
VM_SHARED	虚存区域允许多个进程共享
VM_GROWSDOWN	虚存区域可以向下延伸
VM_GROWSUP	虚存区域可以向上延伸
VM_SHM	虚存区域是共享存储器的一部分
VM_LOCKED	虚存区域可以加锁
VM_STACK_FLAGS	虚存区域做为堆栈使用

虚拟地址空间管理

- vm_offset

  该区域的内容相对于文件起始位置的偏移量，或相对于共享内存首址的偏移量。

- vm_next

  所有vm_area_struct结构体链接成一个单向链表

  vm_next指向下一个vm_area_struct结构体。链表首地址由mm_struct中成员项mmap指出。

- vm_ops是指向vm_operations_struct结构体的指针，该结构体中包含着指向各种操作函数的指针。

虚拟地址空间管理

- vm_avl_left
  左指针指向相邻的低地址虚存区域
- vm_avl_right
  右指针指向相邻的高地址虚存区域
- mmap_avl
  表示进程AVL树的根
- vm_avl_hight
  表示AVL树的高度。
- vm_next_share和vm_prev_share
  把有关vm_area_struct结合成一个共享内存时使用的双向链表。

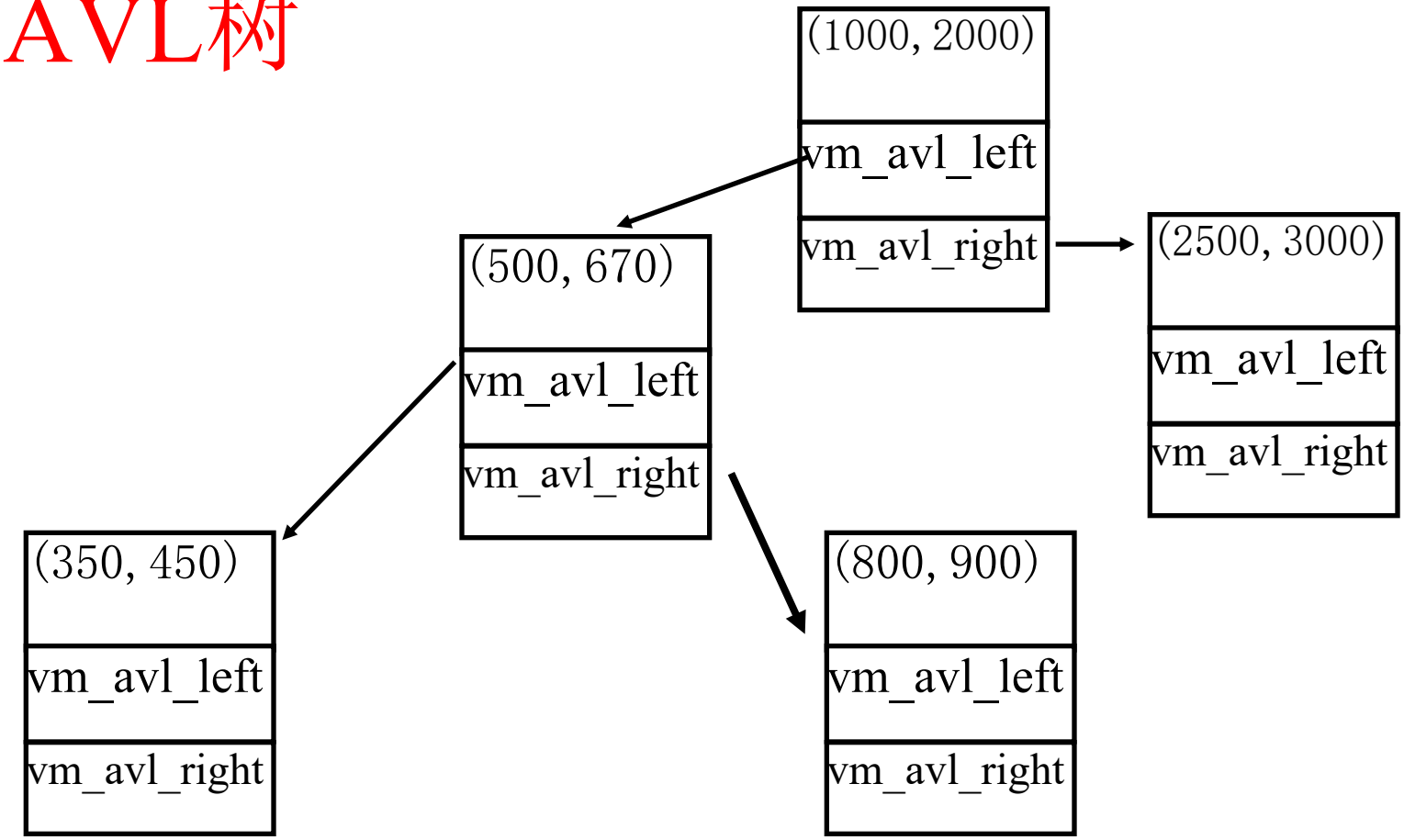
虚拟地址空间管理

AVL树

- ✓AVL树是最先发明的自平衡二叉查找树。
- ✓在AVL树中任何节点的两个子树的高度最大差别为1，所以它也被称为高度平衡树。
- ✓增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

虚拟地址空间管理

AVL树



虚拟地址空间管理

红黑树

- ✓红黑树是满足一定特征的二叉树，其本质还是二叉树。
- ✓红黑树，满足以下性质，即只有满足以下全部性质的树，才称之为红黑树：

1）每个结点要么是红的，要么是黑的。

2）根结点是黑的。

3）每个叶结点，即空结点（NIL）是黑的。

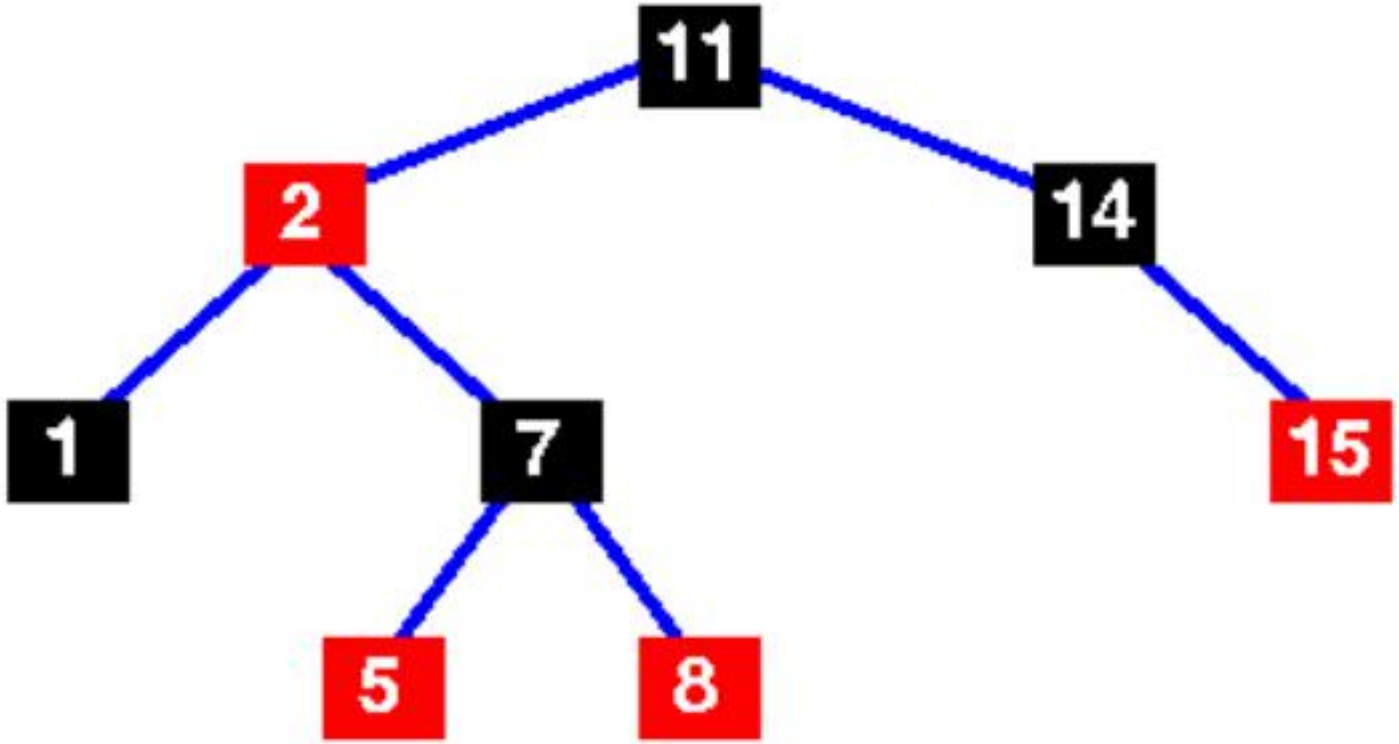
4）如果一个结点是红的，那么它的俩个儿子都是黑的。

5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

虚拟地址空间管理

红黑树

struct rb_root mm_rb;



其每一个红黑树的节点包括三个参数node_color,node_left,node_right

虚拟地址空间管理

- 虚存区域建立
  - Linux使用do_mmap()函数完成可执行映像向虚存区域的映射，建立有关的虚存区域。
  - do_mmap()函数定义在/mm/mmap.c文件中

虚拟地址空间管理

```
unsigned long do_mmap(struct file *
  file, unsigned long addr,
        unsigned long len,
        unsigned long prot,
        unsigned long flags,
        unsigned long off)
```

addr　虚存区域在虚拟内存空间的开始地址

len　是这个虚存区域的长度。

file　是指向该文件结构体的指针

off　是相对于文件起始位置的偏移量。

虚拟地址空间管理

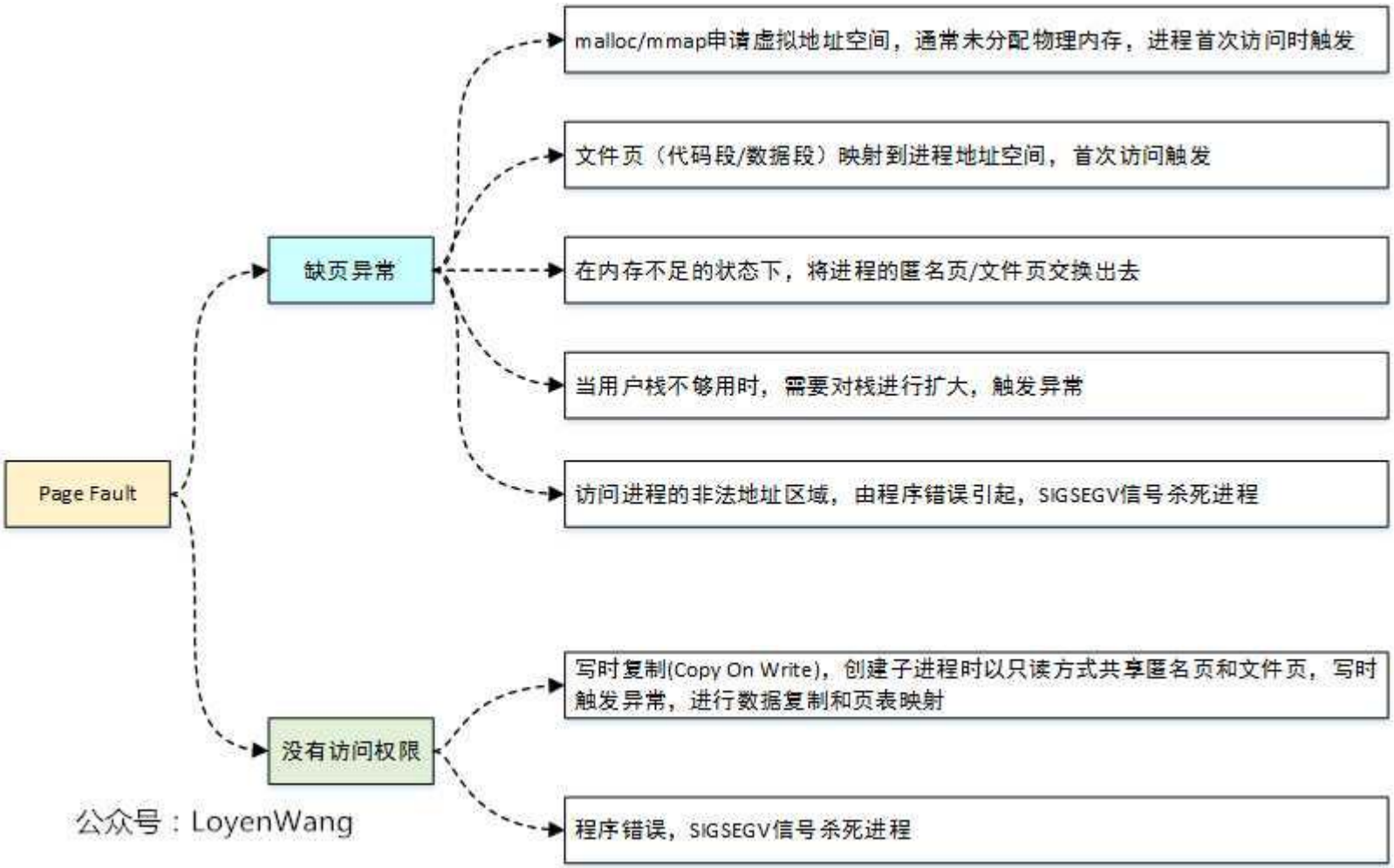
prot　指定了虚存区域的访问特性：

PROT_READ	0x1	对虚存区域允许读取
PROT_WEITE	0x2	对虚存区域允许写入
PROT_EXEC	0x4	虚存区域（代码）允许执行
PROT_NONE	0x0	不允许访问该虚存区域

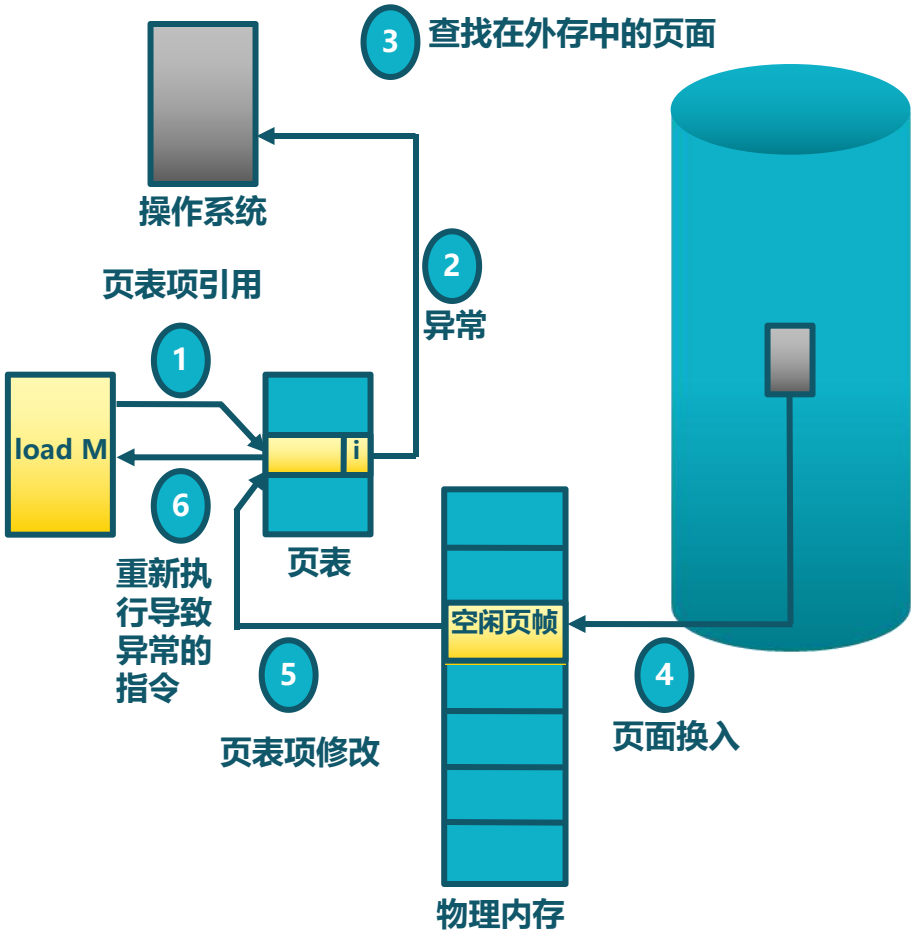
flag　指定了虚存区域的属性：

MAP_FIXED	指定虚存区域固定在addr的位置上
MAP_SHARED	指定对虚存区域的操作是作用在共享页面上
MAP_PRIVATE	指定了对虚存区域的写入操作将引起页面拷贝

请页机制（缺页中断）



请页机制（缺页中断）



- 在内存中有空闲物理页面时，分配一物理页帧f，转第E步；

B.依据页面置换算法选择将被替换的物理页帧f，对应逻辑页q

C.如q被修改过，则把它写回外存；

D.修改q的页表项中驻留位置为0；

E.将需要访问的页p装入到物理页面f

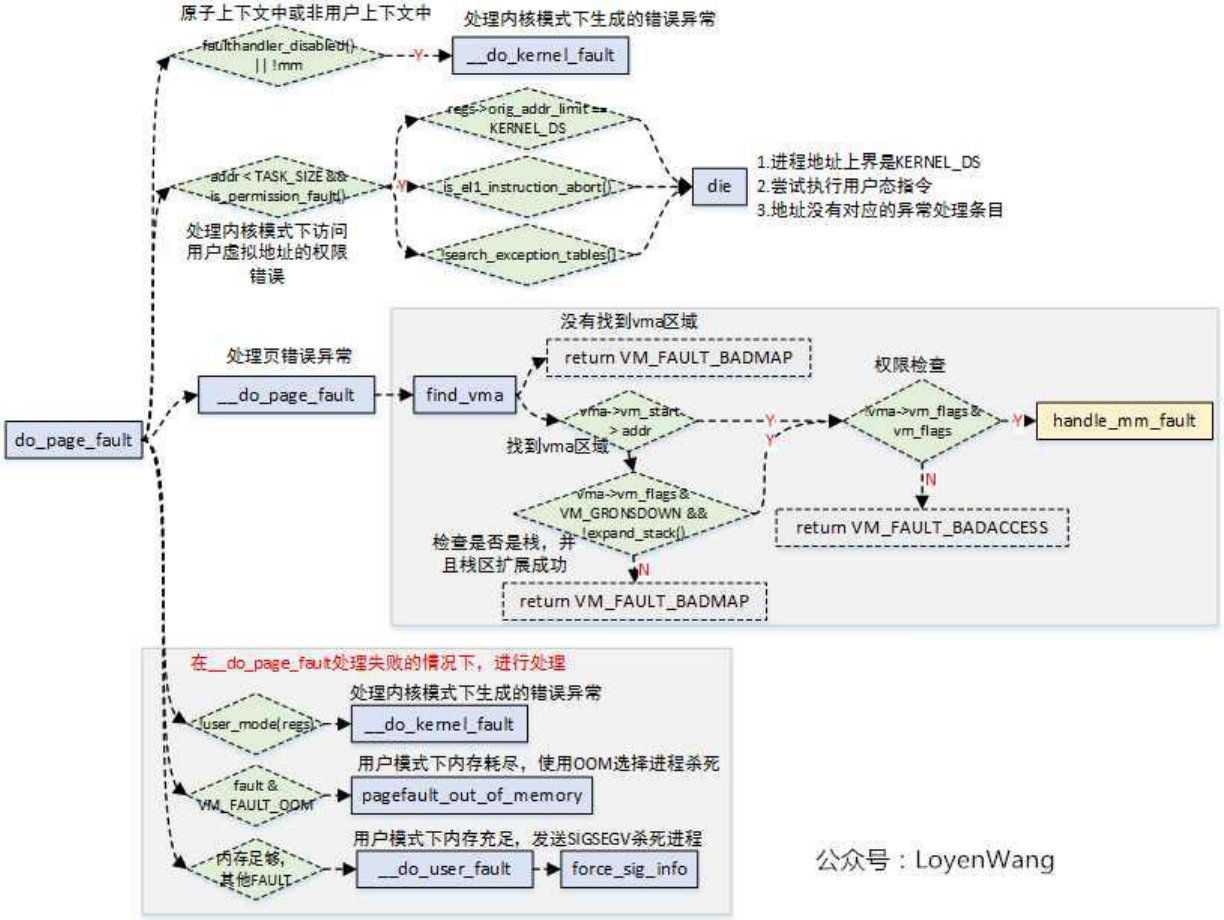
F.修改p的页表项驻留位为1，物理页帧号为f；

G.重新执行产生缺页的指令

请页机制（缺页中断）

- 缺页中断处理函数

  Do_page_fault()



公众号：LoyenWang

交换机制

- 引入目的

  利用外存解决物理内存存储量不足问题

- 方法

  将磁盘上一部分空间作为交换空间，当物理内存不足时，将一部分暂不用的数据换入到交换空间中，从而保证内存有足够空间空间

- 优缺点

  解决了主存不足问题，增加了处理机开销

交换机制

- 交换空间
- 交换空间格式
- 守护进程kswapd（）

交换机制

- 交换空间
  - ✓ 交换设备和交换文件统称为交换空间
  - ✓ 交换设备中同一个页面中数据块连续存放
  - ✓ 交换文件中是零散存放的，需要通过交换文件索引点来检索。

交换机制

- 交换空间格式
  - ✓ 交换空间被划分为块（页插槽），每个块等于一个物理页。第一个页插槽中，存放了一个以”swap_space“结尾的位图，位图每一位对应一个页插槽，即第一个插槽后是真正用于交换的页插槽，这样，每个交换空间可容纳：

    （4096－10）×8－1＝32687个页面
  - ✓ Linux允许建立8个交换空间

交换机制

- 页面交换守护进程kswapd（）
  - ✓ 是一个无限循环的线程，完成页面交换工作，保证系统内有足够内存。
  - ✓ Linux将页面分为活跃状态、非活跃”脏“状态、非活跃”干净“状态。需要时，将非活跃”脏“状态页面内容写入到外存交换空间，并将该页面从”脏“队列移动到”干净“队列。回收页面总是从非活跃”干净“链表队列中进行

缓冲机制

- 目的

  改善处理机和外围设备之间速度不匹配的矛盾，提高系统性能。

- Linux采用了多种与主存相关的高速缓存。
  - 页面高速缓存
  - 缓冲区高速缓存
  - 交换区缓存

缓冲机制

- 页面高速缓存
  - 是为了加快对磁盘文件访问而设立的。
  - 当页面从磁盘被读入主存时，被存入页面高速缓存。
  - 如文件系统的一些系统调用，如read()、write()等，都是通过页面高速缓存完成的。

缓冲机制

- 缓冲区高速缓存
  - 大小固定，是针对块设备的I/O开辟的。
  - 缓存区高速缓存以块设备标识符和块号作为索引，快速查找数据块，若数据块已在缓存中，则不必从物理设备中读取，从而加快了读取数据速度。
  - 缓冲区高速缓存大小可变化，当没有空闲的缓冲区而又必须分配新缓冲区时，内核就按需分配缓冲区，当主存空间不足时，可以释放缓冲区。

缓冲机制

- 交换缓存
  - Swap分区在系统的物理内存不够用的时候，把硬盘内存中的一部分空间释放出来，以供当前运行的程序使用。
  - 那些被释放的空间可能来自一些很长时间没有操作的程序，这些被释放的空间被临时保存到Swap分区中，等到那些程序要运行时，再从Swap分区中恢复保存的数据到内存中。

Linux常用的内存分配函数

函数名	分配原理	最大内存	其他
_get_free_pages	直接对页框进行操作	4MB	适用于分配较大量的连续物理内存
kmem_cache_alloc	基于slab机制实现	128KB	适合需要频繁申请释放相同大小内存块时使用
kmalloc	基于kmem_cache_alloc实现	128KB	最常见的分配方式，需要小于页框大小的内存时使用
vmalloc	建立非连续物理内存到虚拟地址的映射		物理不连续，适合需要大内存，但对地址连续性没有要求的场合
dma_alloc_coherent	基于_alloc_pages实现	4MB	适用于DMA操作
Ioremap	实现已知物理地址到虚拟地址的映射		适用于物理地址已知的场合，如设备驱动
alloc_bootmem	在启动kernel时，预留一段内存，内核看不见		小于物理内存大小，内存管理要求较高

kmalloc函数

- 用于内核分配在物理上是连续的内存

```
#include<linux/slab.h>
void *kmalloc( size_t size, int flags );
```

分配的内存大小（字节数）
通常最好不要分配大于128KB的内存

GFP_KERNEL:
为内核空间分配内存，可能睡眠
GFP_ATOMIC:
在中断处理例程或其他运行子程上下文之外的代码中分配内存，不会休眠。

kmalloc函数

- 通常用于设备驱动程序。
- Slab分配器
  - Kmalloc的机制是基于slab分配器的
  - 后备高速缓存的操作
    - create
    - destroy
    - allocate
    - free

get_free_page

- Kmalloc要求分配的内存大小应该小于128KB,大于128Kb的怎么办?
- 分配页面函数或宏

```
unsigned long get_zeroed_page(unsigned int flags);
unsigned long __get_free_page(unsigned int flags);
unsigned long __get_free_pages(unsigned int flags，unsigned int order)
    ;
```

- 释放页面函数

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr，unsigned long order);
```

vmalloc

- Vmalloc分配虚拟地址空间的连续区域，但这段区域在物理上可能是不连续的。

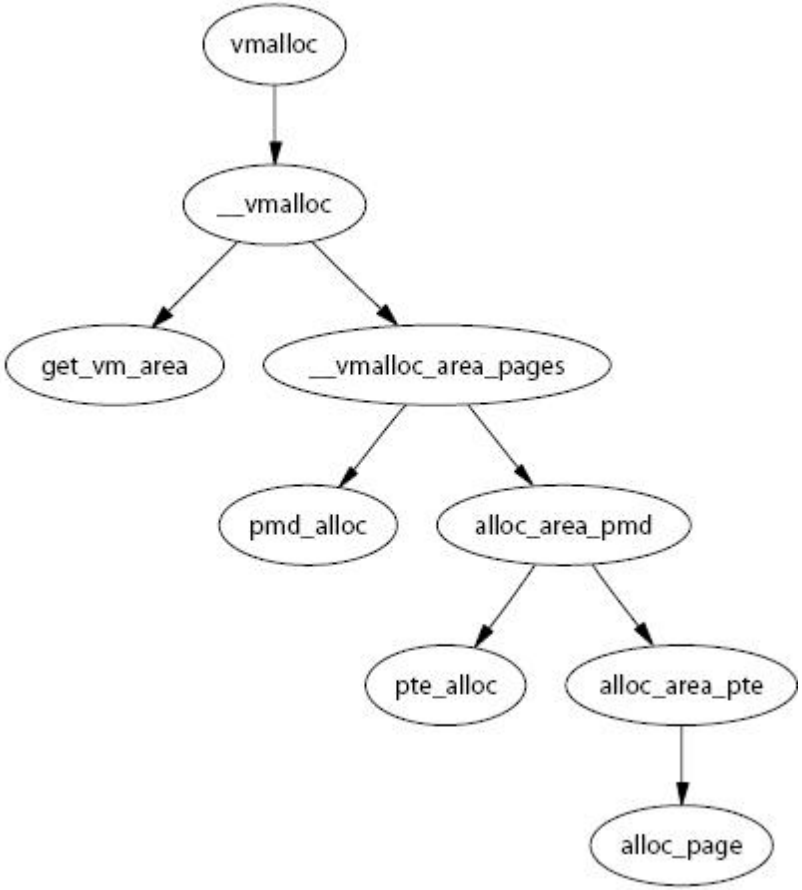
```
#include<linux/vmalloc.h>
void *vmalloc(unsigned long size);
void vfree(void *addr);
```

vmalloc

- vmalloc分配得到的地址是不能在微处理器之外使用的，当驱动程序需要真正的物理地址时(像外设用以驱动系统总线的DMA地址)，就不能使用vmalloc;
- 使用vmalloc函数的正确场合是在分配一大块连续的、只在软件中存在的、用于缓冲的内存区域的时候。
- 因为vmalloc不但获取内存，还要建立页表，它的开销比get_free_pages 大，因此，用vmalloc函数分配仅仅一页的内存空间是不值得的。

vmalloc

- Vmalloc:调用图



虚拟地址与物理地址关系

- 内核虚拟地址转化为物理地址

```
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
extern inline unsigned long virt_to_phys(volatile void *address)
{
        return __pa(address);
}
```

- 物理地址转化为内核虚拟地址
-

```
#define __va(x) ((void *)((unsigned long)(x)+PAGE_OFFSET))
extern inline void * phys_to_virt(unsigned long address)
{
        return __va(address);
}
```

实验

1、安装linux源代码，基于本讲内容，查看和分析Linux内核主要数据结构及内存分配原理

C203