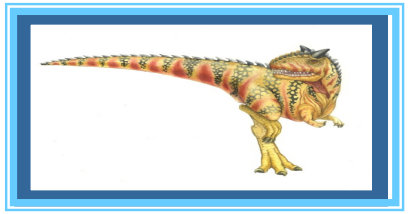
Linux内存管理



目录

- 虚拟内存空间的管理
- Linux的分页内存管理机制
- 物理内存的管理

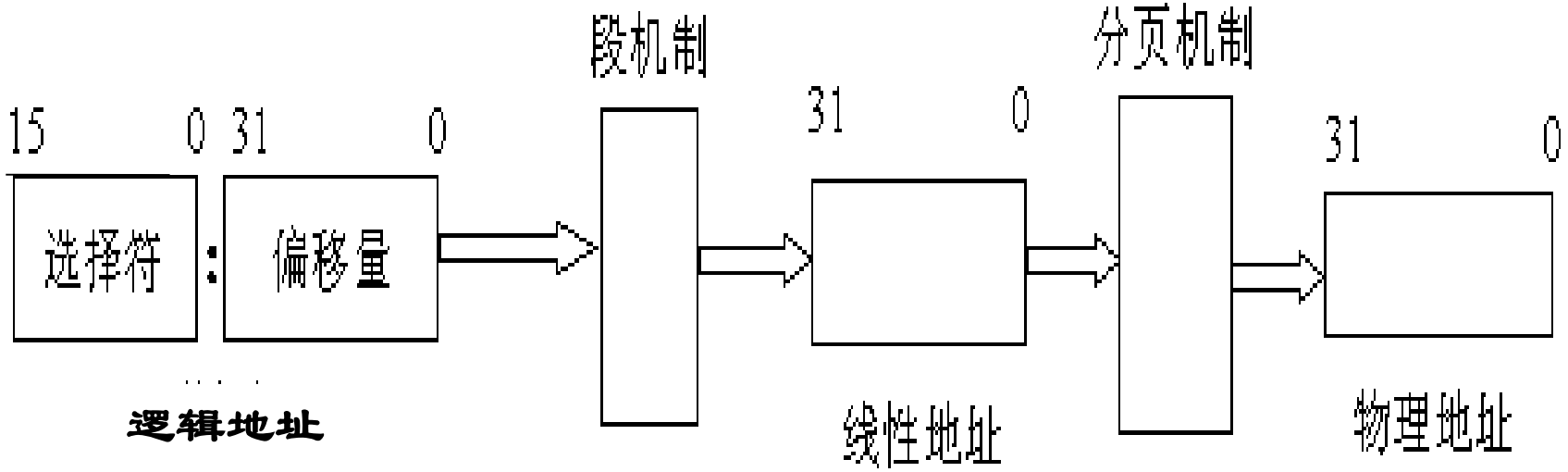
虚拟内存空间的管理

- Linux操作系统采用了请求式分页虚拟内存管理方法。
- 系统为每个进程提供了4GB(32位为例)的虚拟内存空间。各个进程的虚拟内存彼此独立。

IA32的内存寻址

- IA32（Intel Architecture 32 bit ,或i386、x86）体系结构具有两种内存管理模式：
  - 实地址模式--实模式（Real-address Mode)
  - 受保护的虚地址模式--保护模式(Protected Mode)
- 在保护模式下，IA32提供了实现虚拟内存的硬件机制
- IA32的地址转换机制
  - IA32中地址总线为32或36位（pentium pro开始，支持Physical Adress Extension,PAE,物理地址扩展），物理内存空间最大为4G或64G字节
  - IA32指令系统提供的的逻辑地址为48位，由它确定的虚拟地址空间可达64T字节。
- *有关IA32内存寻址内容，请阅读以下参考资料：*
  - *《边干边学》第14章*
  - *《understanding the linux kernel》第2章*

逻辑地址到物理地址转换



- **逻辑地址**：包含在机器语言指令中，用来指定一个操作数或一条指令的地址。
- **线性地址**（虚拟地址）：一个32位无符号整数，用来表示高达4GB的地址。
- **物理地址**：用于内存芯片级内存单元寻址。

进程的地址空间和特权级

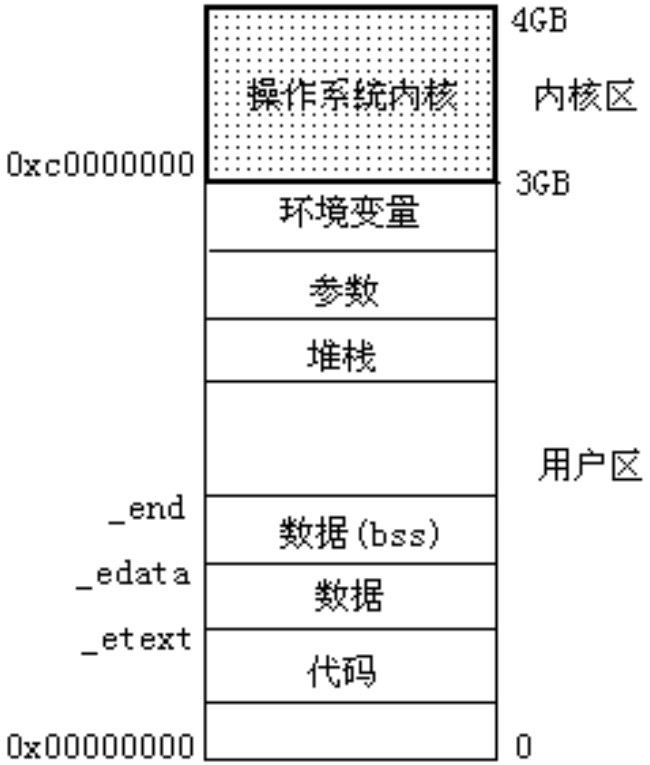
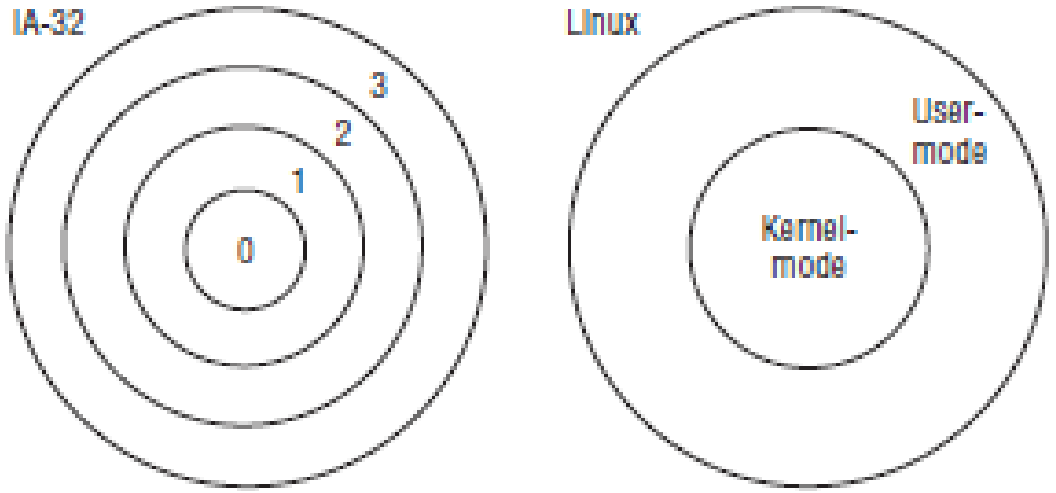
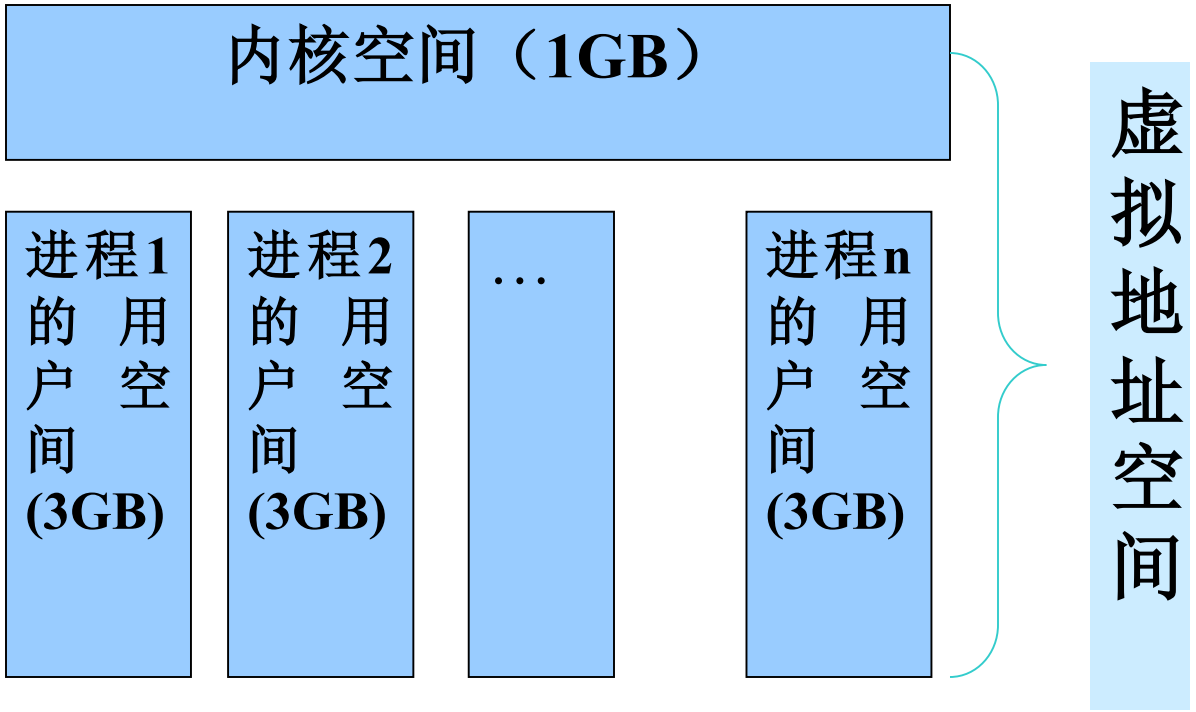


图5.1 进程的虚存空间



- 用户态(user mode)禁止访问内核空间

虚拟内存、内核空间和用户空间



虚拟内存、内核空间和用户空间

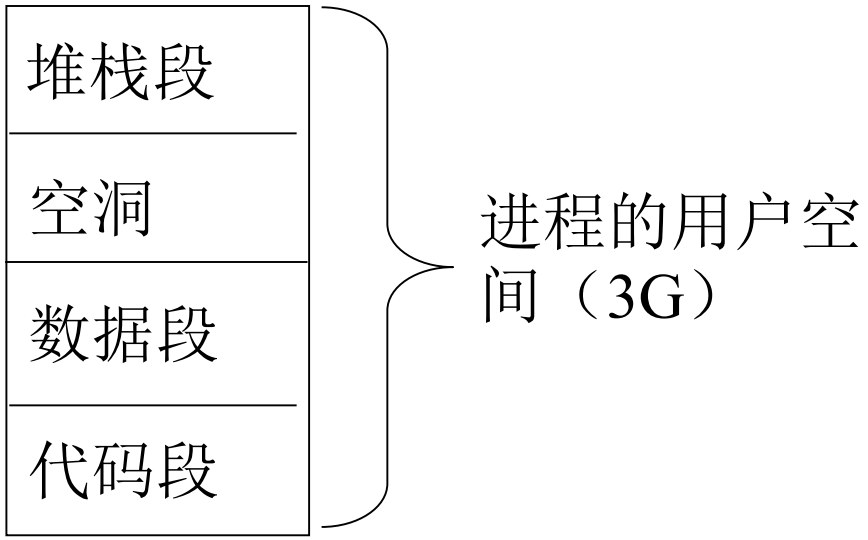
- 虚拟内存一共4G字节，分为内核空间（最高的1G字节）和用户空间（较低的3G字节）两部分，每个进程最大拥有3G字节私有虚存空间
  - 内核空间和用户空间大小的划分由宏定义PAGE_OFFSET决定，在文件src/include/asm-i386/page.h
- 地址转换—通过页表把虚存空间的一个地址转换为物理空间中的实际地址。

内核空间到物理内存的映射

- 内核空间由所有进程共享，其中存放的是内核代码和数据，即“内核映象”。“内核映象”不可回收和换出。
- 进程的用户空间中存放的是用户程序的代码和数据
- 内核空间映射到物理内存总是从最低地址（0x00000000）开始，使之在内核空间与物理内存之间建立简单的线性映射关系。
- 当系统启动时，Linux内核映像被装入在物理地址0x00100000开始的地方，即1MB开始的区间(第1M留作它用)。

进程用户空间的管理

- 每个程序经编译、链接后形成的二进制映像文件有一个代码段和数据段
- 进程运行时须有独占的堆栈空间



进程用户空间

- Linux把进程的用户空间划分为一个个区间，便于管理
- 一个进程的用户地址空间主要由mm_struct结构和vm_area_structs结构来描述。
- mm_struct结构它对进程整个用户空间进行描述（称为内存表述符memory descriptor）
- vm_area_structs结构对用户空间中各个区间(简称虚存区)进行描述
- mm_struct结构首地址在task_struct成员项mm中：

  struct mm_struct *mm;
- mm_struct结构定义在include/linux/mm_types.h中。P.367

mm_struct结构

```
struct mm_struct {
        struct vm_area_struct * mmap;           /* list of VMAs */
        rb_root_t mm_rb;//Pointer to the root of the red-black tree
        struct vm_area_struct * mmap_cache;/* last find_vma result */
        pgd_t * pgd;  //进程页目录指针
        atomic_t mm_users;                      /* How many users with user space? */
        atomic_t mm_count;      /* How many references to "struct mm_struct" (users count as 1) */
        int map_count;                          /* number of VMAs */
        struct rw_semaphore mmap_sem;
        spinlock_t page_table_lock;     /* Protects task page tables and mm->rss */
        struct list_head mmlist;        /* List of all active mm's.  These are globally strung      *together off init_mm.mmlist, and are protected
        by mmlist_lock      */
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;//驻留内存页框总数，VMA总数及被锁VMA总数
        unsigned long def_flags;
        unsigned long cpu_vm_mask;
        unsigned long swap_address;
        unsigned dumpable:1;
        /* Architecture-specific MM context */
        mm_context_t context;
};
```

域名	说　　明
count	对mm_struct结构的引用进行计数。为了在Linux中实现线程，内核调用clone派生一个线程，线程和调用进程共享用户空间，即mm_struct结构，派生后系统会累加mm_struct中的引用计数。
pgd	进程的页目录基地址，当调度程序调度一个进程运行时，就将这个地址转成物理地址，并写入控制寄存器（CR3）
map_count	在进程的整个用户空间中虚存区的个数
semaphore	对mm_struct结构进行互斥访问所使用的信号量
start_code,end_code,start_data, end_data	进程的代码段和数据段的起始地址和终止地址
start_brk, brk, start_stack;	每个进程都有一个特殊的地址区间，这个区间就是所谓的堆，也就是前图中的空洞。前两个域分别描述堆的起始地址和终止的地址，最后一个域描述堆栈段的起始地址。
arg_start, arg_end, env_start, env_end	命令行参数所在的堆栈部分的起始地址和终止地址； 环境串所在的堆栈部分的起始地址和终止地址
rss, total_vm, locked_vm	进程驻留在物理内存中的页面数，进程所需的总页数，被锁定在物理内存中的页数。
mmap	vm_area_struct虚存区结构形成一个单链表，其基址由小到大排列
mmap_avl	vm_area_struct虚存区结构形成一个颗AVL平衡树
mmap_cache	最近一次用到的虚存区很可能下一次还要用到，因此，把最近用到的虚存区结构放入高速缓存，这个虚存区就由mmap_cache指向。

VM_AREA_STRUCT 结构

- vm_area_struct结构是虚存空间中一个连续的区域，在这个区域中的信息具有相同的操作和访问特性。
- 各区间互不重叠,按线性地址的次序链接在一起。当区间的数目较多时,将建立AVL树以保证搜索速度。
- vm_area_struct结构定义在include/linux/mm_types.h中：　P.370

进程的虚存区举例

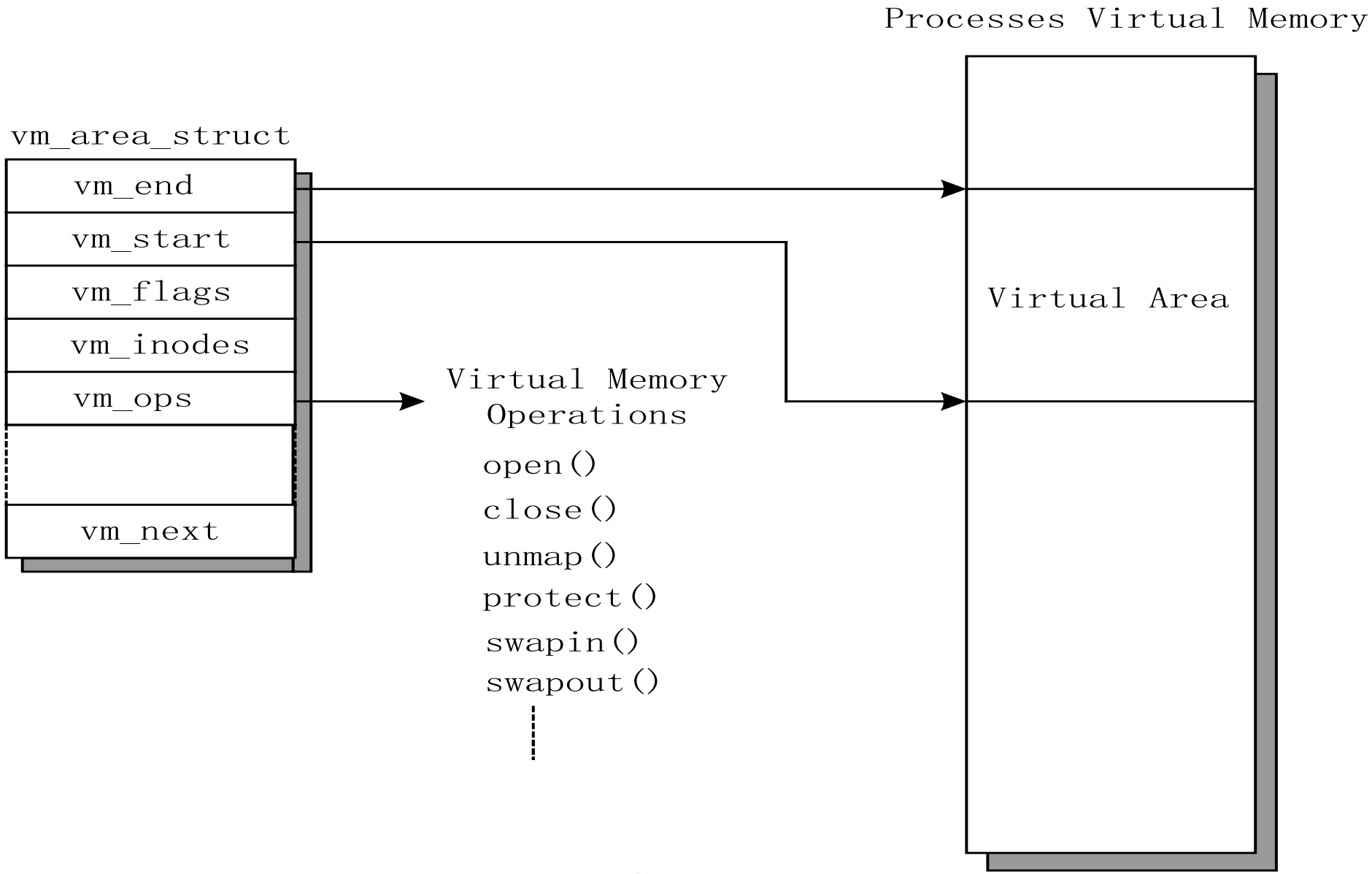
例：test.c

```
int main()
{
printf("virtual area test!");
}
```

地址范围	许可权	偏移量	所映射的文件
08048000-08048fff	r-xp	00000000	/home/test/test
08049000-08049fff	rw-p	00001000	/home/test/test
40000000-40014fff	r-xp	00000000	/lib/ld-2.3.2.so
40015000-40015fff	rw-p	00015000	/lib/ld-2.3.2.so
40016000-40016fff	rw-p	00000000	匿名
4002a000-40158fff	r-xp	00000000	/lib/libc-2.3.2.so
40159000-4015dfff	rw-p	0012f000	/lib/libc-2.3.2.so
4015e000-4015ffff	rw-p	00000000	匿名
fe000-bfffffff	rwxp	fffff000	匿名

test进程的虚存区

虚拟内存区域


Processes Virtual Memory
vm_area_struct
vm_end
vm_start
vm_flags
vm_inodes
vm_ops
vm_next
Virtual Memory Operations
open()
close()
unmap()
protect()
swapin()
swapout()
Virtual Area

vm_area_struct

```
struct vm_area_struct {
    struct mm_struct * vm_mm;  /* The address space we belong to. */
    unsigned long vm_start;        /* Our start address within vm_mm. */
    unsigned long vm_end;/* The first byte after our end address within vm_mm. */
        /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;
    pgprot_t vm_page_prot;        /* Access permissions of this VMA. */
    unsigned long vm_flags;       /* Flags, listed below. */
    struct rb_node vm_rb;
        /** For areas with an address space and backing store,one of the address_space->i_mmap{,shared} lists, for shm areas, the
        list of attaches, otherwise unused.     */
    struct list_head shared;
        /* Function pointers to deal with this struct. */
    struct vm_operations_struct * vm_ops;
        /* Information about our backing store: */
    unsigned long vm_pgoff;      /* Offset (within vm_file) in PAGE_SIZE
                                    units, *not* PAGE_CACHE_SIZE */
    struct file * vm_file;                  /* File we map to (can be NULL). */
    void * vm_private_data;                 /* was vm_pte (shared mem) */
};
```

VMA

- 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_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
  - ……
- vm_page_prot 虚存区域的页面的保护特性。

VMA

- 所有vm_area_struct结构体按地址递增链接成一个单向链表，vm_next指向下一个vm_area_struct结构体。链表的首地址由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的函数的指针。
  - vm_operations_struct结构体在include/linux/mm.h 文件中定义。

- vm_next_share和vm_prev_share，把有关的vm_area_struct 结合成一个共享内存时使用的双向链表。

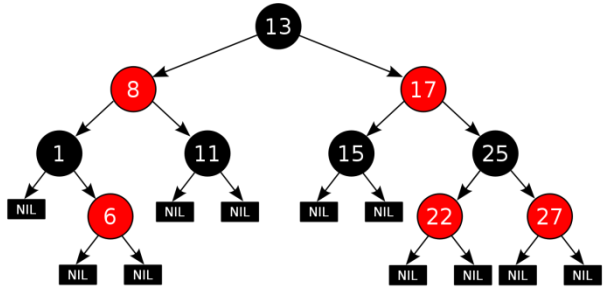
红黑树

- 为了提高对vma段查询、插入、删除操作的速度，Linux内核为每个进程维护了一棵红黑树(Red Black Tree)，所有vm_area_struct结构体组成一个红黑树。红黑树的节点和根节点的结构定义在include/linux/rbtree.h文件中：

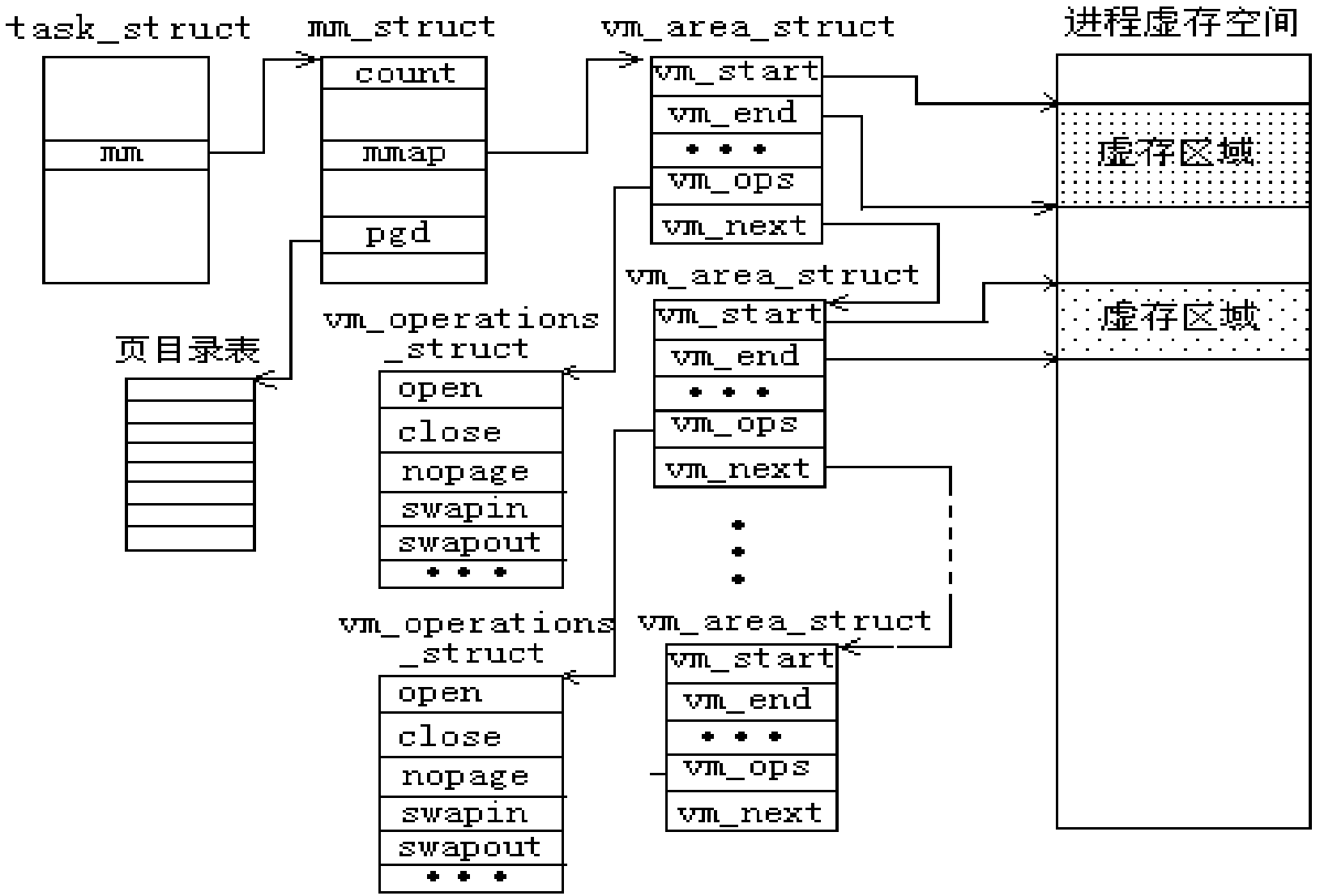
```
100 typedef struct rb_node_s
101 {
102         struct rb_node_s * rb_parent;
103         int rb_color;
104 #define RB_RED          0
105 #define RB_BLACK        1
106         struct rb_node_s * rb_right;
107         struct rb_node_s * rb_left;
108 }
109
110 struct rb_root
111 {
112         struct rb_node * rb_node;
113 }
```

红黑树

- 在树中，所有的vm_area_struct虚存段都作为树的一个节点。节点中vm_rb的左指针rb_left指向相邻的低地址虚存段，右指针rb_right指向相邻的高地址虚存段。
- Red Black Tree is a kind of self-balancing binary search tree
  - Every node must be either red or black.
  - The root of the tree must be black.
  - The children of a red node must be black.
  - 从一个节点到后代叶子节点的每个路径都包含相同数量的黑节点。当统计黑节点个数时，空指针也算作黑节点。



- 红黑树的一些操作定义在lib/rbtree.c中


task_struct
mm_struct
vm_area_struct
进程虚存空间
mm
count
mmap
pgd
vm_start
vm_end
• • •
vm_ops
vm_next
虚存区域
vm_area_struct
vm_start
vm_end
• • •
vm_ops
vm_next
虚存区域
页目录表
vm_operations
_struct
open
close
nopage
swapin
swapout
• • •
vm_operations
_struct
open
close
nopage
swapin
swapout
• • •
vm_area_struct
vm_start
vm_end
• • •
vm_ops
vm_next

进程虚存空间管理

虚存段的建立

- Linux使用do_mmap()函数完成可执行映像向虚存段的映射，由它建立有关的虚存段。
- do_mmap()函数定义在include/linux/mm.h文件中

  unsigned long do_mmap(struct file * file, unsigned long addr, unsigned long len,unsigned long prot, unsigned long flags, unsigned long off)
- do_mmap()函数参数含义：
  - file是指向该文件结构体的指针，若file为NULL，称为匿名映射(anonymous mapping)。
  - off是相对于文件起始位置的偏移量。
  - addr虚存段在虚拟内存空间的开始地址。
  - len是这个虚存段的长度。

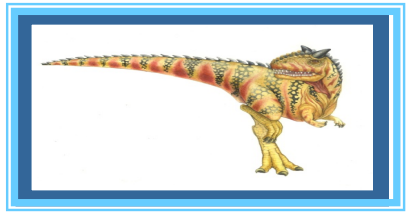
虚存段的建立

- prot指定了虚存段的访问特性：
  - PROT_READ　0x1　对虚存段允许读取
  - PROT_WEITE　0x2　对虚存段允许写入
  - PROT_EXEC　0x4　虚存段（代码）允许执行
  - PROT_NONE　0x0　不允许访问该虚存段
- flag指定了虚存段的属性：
  - MAP_FIXED　指定虚存段固定在addr的位置上。
  - MAP_SHARED　指定对虚存段的操作是作用在共享页面上
  - MAP_PRIVATE指定了对虚存段的写入操作将引起页面拷贝。

创建进程用户空间

- fork()系统调用在创建新进程时也为该进程创建完整的用户空间
- fork()是通过拷贝或共享父进程的用户空间来实现的，即内核调用copy_mm()函数，为新进程建立所有页表和mm_struct结构
- Linux利用“写时复制 copy on write”技术来快速创建进程

Linux的分页内存管理机制



Linux的分页内存管理机制

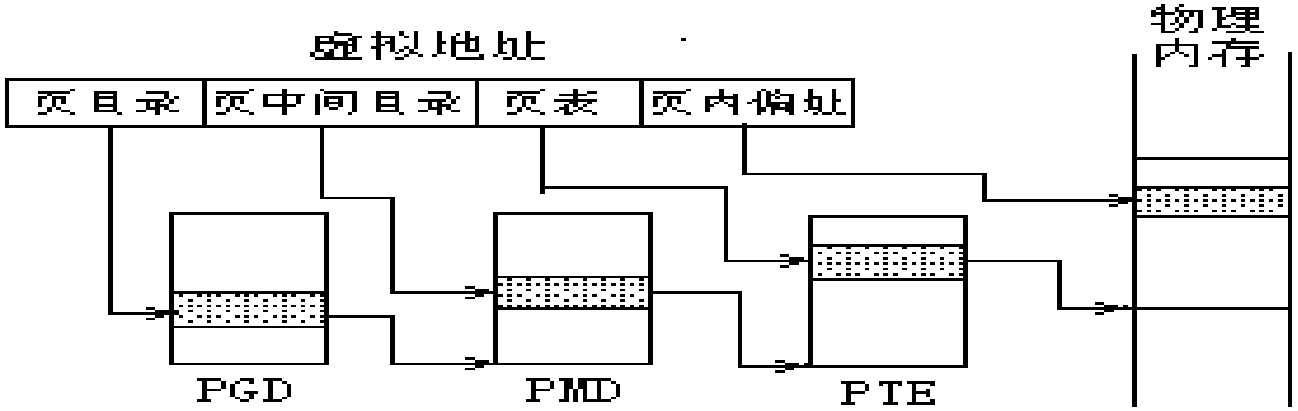
一、Linux的页表机制：

- Linux总是假定处理器支持三级页表结构。Linux在64位体系结构上采用三级页表机制。对于以IA32体系结构的32位机器,则采用二级也表机制，页中间目录层实际不起作用。
- 三级分页管理把虚拟地址分成四个位段：

  页目录、页中间目录、页表、页内偏移。
- 系统设置三级页表：
  - 页目录PGD（Page Directory）
  - 页中间目录PMD（Page Middle Directory）
  - 页表PTE（Page Table）
- 2.6.11以后Linux内核采用四级页表模型来使用硬件分页机制 (支持64位CPU架构)，分别是：
  - Page Global Directory(pgd_t), Page Upper Directory(pud_t),Page Middle Directory(pmd_t) 和 Page Table(pte_t)。
  - 当硬件分页机制实际是两级页表时，Linux内核把Page Upper Directory 和Page Middle Direcotry跳过了。

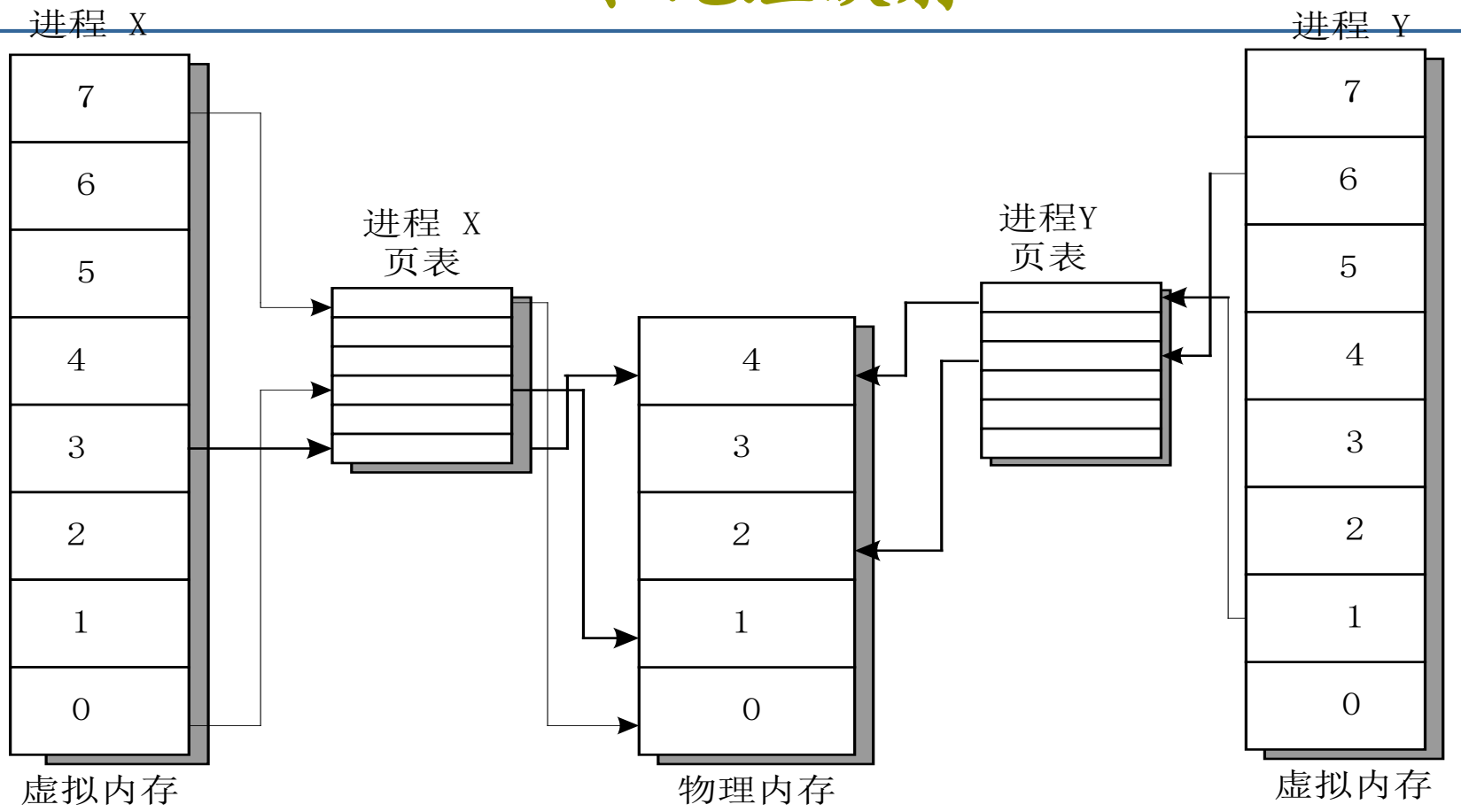
Linux三级页表结构

- 三级分页结构是Linux提供的与硬件无关的分页管理方式。
- 当Linux运行在某种机器上时，需要利用该种机器硬件的内存管理机制来实现分页内存。
- Linux内核中对不同的机器配备了不同的分页结构的转换方法。
- 对IA32，提供了把三级分页管理转换成两级分页机制的方法。其中一个重要的方面就是把PGD与PMD合二为一，使所有关于PMD的操作变为对PGD的操作。



Linux的三级分页管理

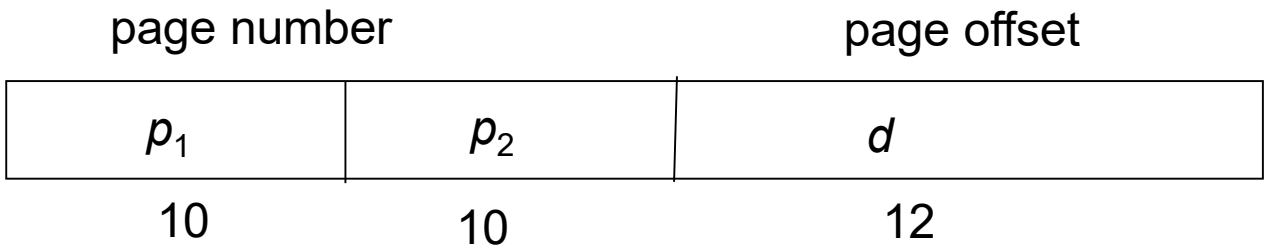
二、地址映射



- 虚拟内存系统中的所有地址都是虚拟地址而不是物理地址。通过操作系统所维护的一系列表，由MMU实现从虚拟地址到物理地址的转换。

地址映射

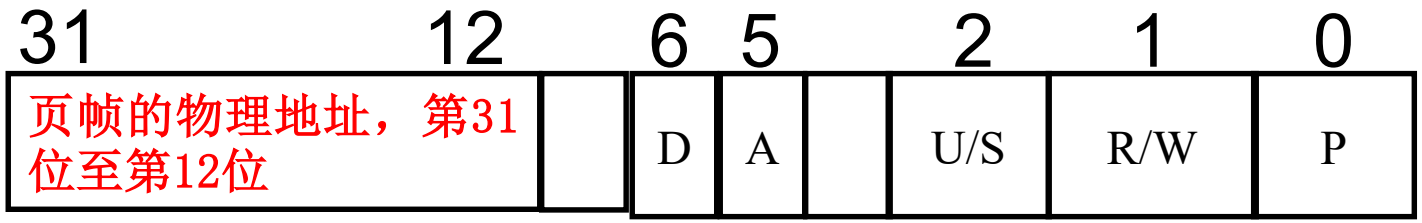
- 以IA32体系结构为例。IA32系列既支持分段机制,也支持分页机制,Linux主要采用了分页机制, 页面可以映射到任一物理页帧。IA32下进程的线性地址为32位,分为以下三个部分:



  - 页目录p1：置于高10位,记录在页目录中的索引。
  - 页表p2：占据中间的10位,记录在页表中的索引。
  - 偏移d：占据低12位,用来表示在4KB的页帧中的偏移。

页目录项和页表项

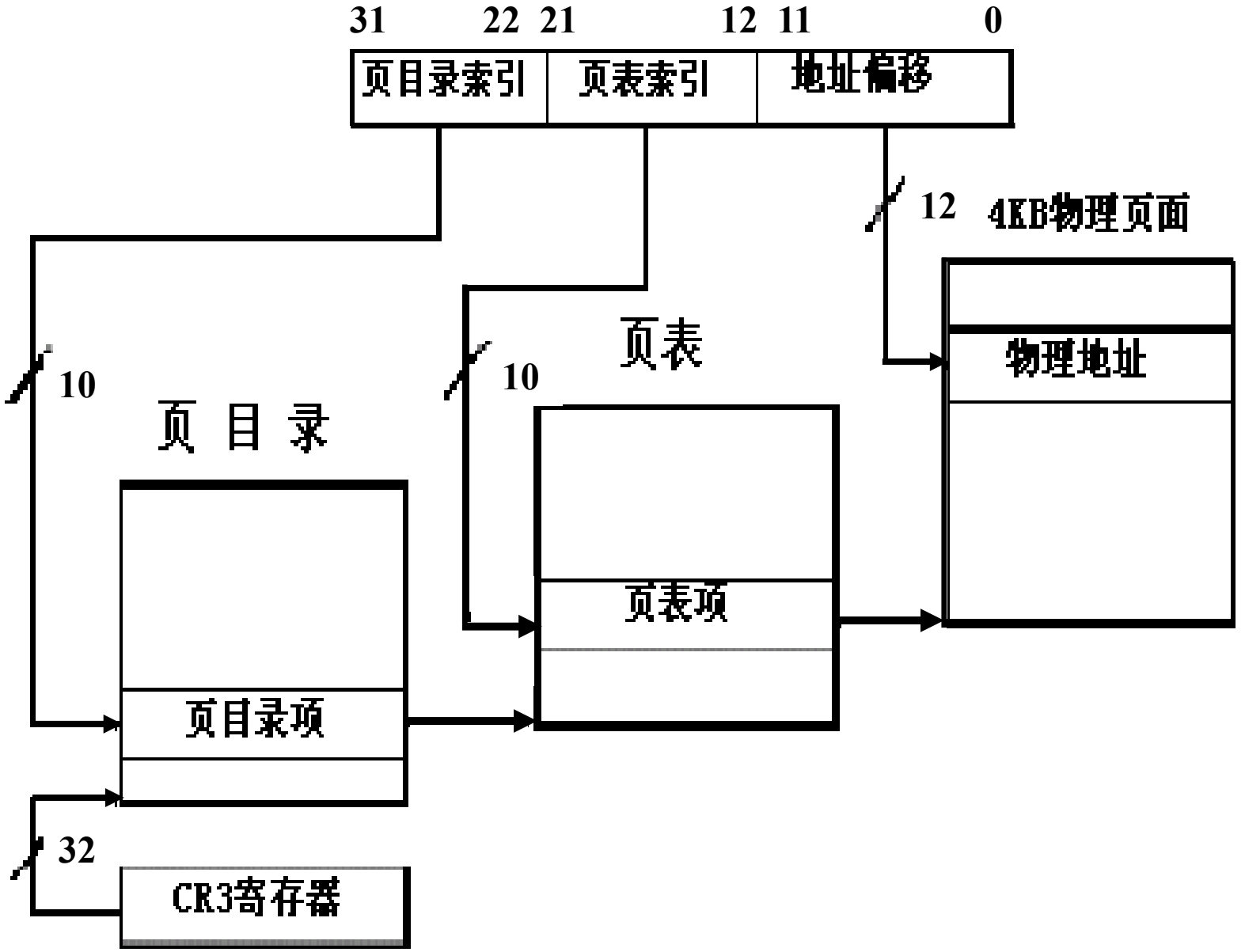
- P ：1 当前物理页存在；P=0则当前物理页不再内存
- R/W：1则该页可写，可读，且可执行；R/W=0则该页可读，可执行，但不可写
- U/S：1则该页可在任何特权级下访问；0则该页只能在特权级0、1和2下访问
- A ：访问位
- D ：已写标志位



MMU地址映射过程

- 每个进程都有一个页目录,当进程运行时,寄存器CR3指向该页目录的基址。地址映射过程:
  1. 从CR3取得页目录的基地址。页目录用一个物理页帧存储,用来保存页表的基址。
  2. 以逻辑地址的页目录段为索引,在页目录中找到页表的基址。页表也是用一个物理页帧存储,用来保存物理页帧号。
  3. 以逻辑地址的页表段为索引,在页表中找相应的物理页帧号。
  4. 物理页帧号加上逻辑地址的偏移段即得到了对应的物理地址。

线 性 地 址

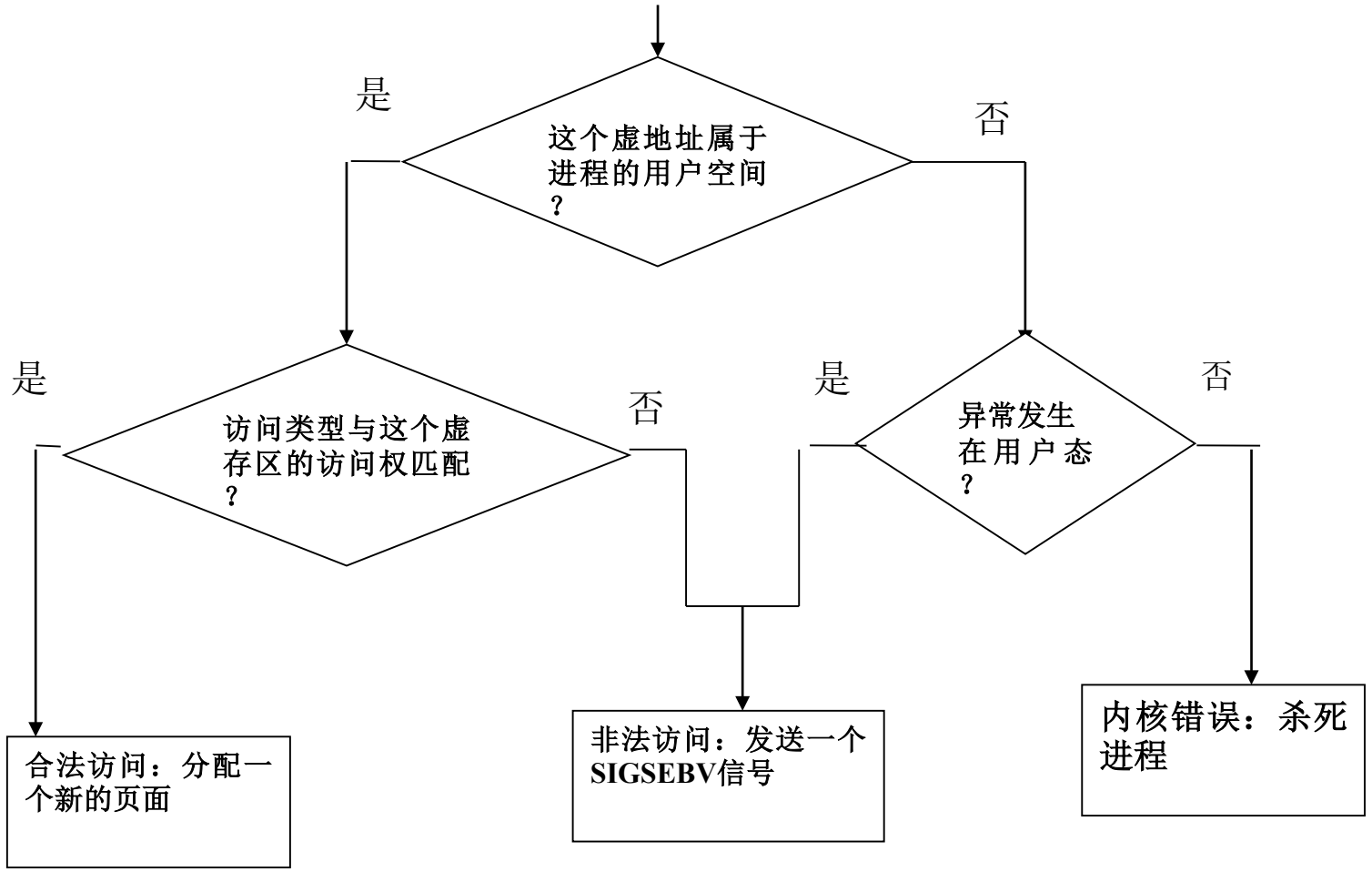

31
22 21
12 11
0
页目录索引
页表索引
地址偏移
12
4KB物理页面
物理地址
10
页 目 录
10
页表
页表项
页目录项
32
CR3寄存器

缺页异常的处理

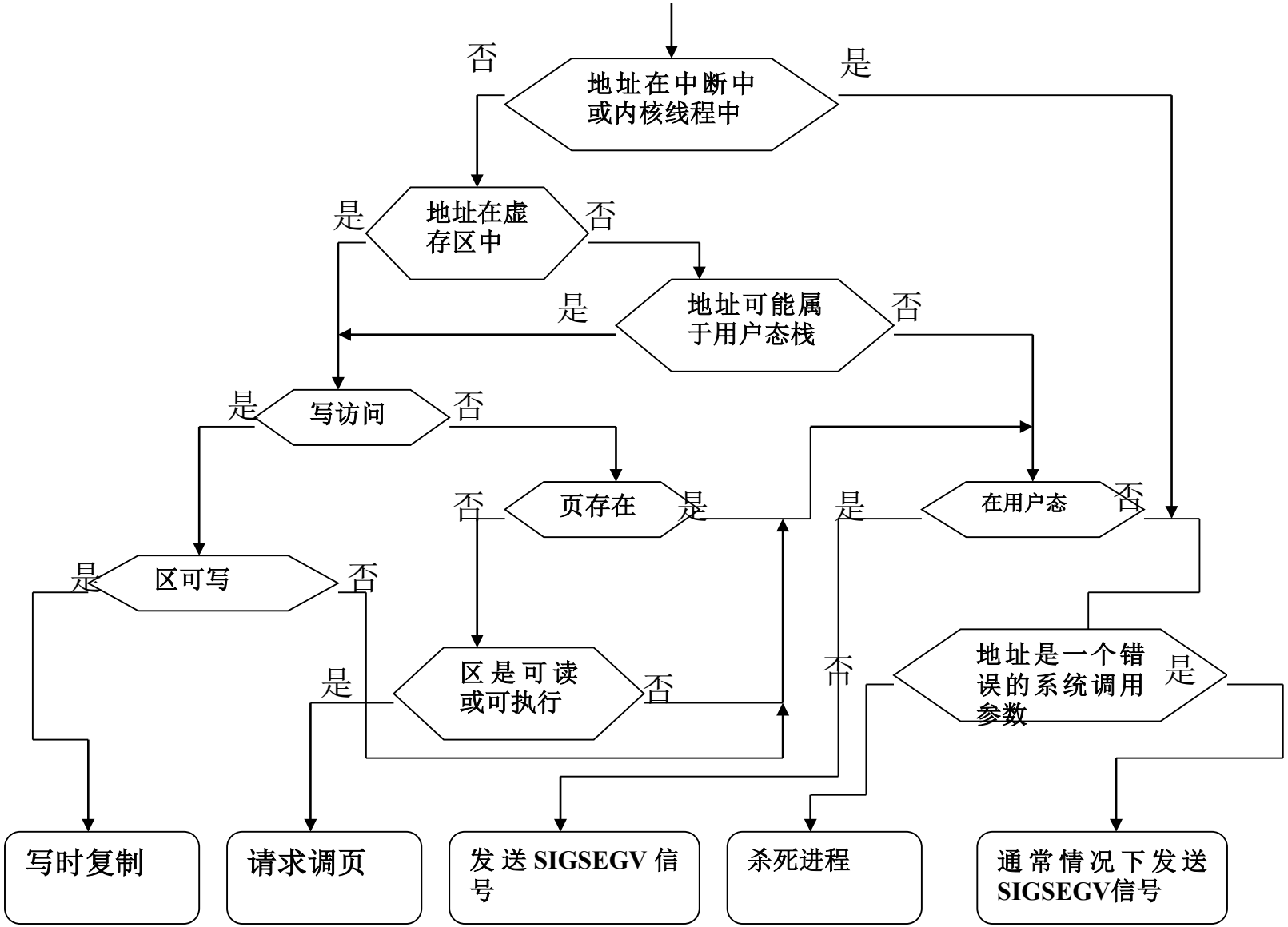
- 导致缺页异常（缺页中断）的原因有:
  - 编程错误。可分为内核程序错误和用户程序错误。
  - 操作系统故意引发的异常。操作系统合理利用硬件机制，在适当时间触发异常，使得该异常的处理程序被调用，以达到预期目的。
- 80x86缺页中断服务程序是do_page_fault函数

P.375

缺页异常处理程序的总体方案



缺页处理程序流程图


地址在中断中或内核线程中
否
是
地址在虚存区中
是
否
地址可能属于用户态栈
是
否
写访问
是
否
页存在
否
是
在用户态
是
否
区可写
是
否
区是可读或可执行
是
否
地址是一个错误的系统调用参数
否
是
写时复制
请求调页
发送SIGSEGV信号
杀死进程
通常情况下发送SIGSEGV信号

do_page_fault( )

- 页面异常的处理程序是do_page_fault( )函数，该函数有两个参数：
  - 一个是指针,指向异常发生时寄存器值存放的地址。
  - 另一个错误码,由三位二进制信息组成：
    - ✓ 第0位——访问的物理页帧是否存在；
    - ✓ 第1位——写错误还是读错误或执行错误；
    - ✓ 第2位——程序运行在核心态还是用户态。
- do_page_fault( )函数定义在arch/x86/mm/fault.c文件中

do_page_fault( )

- do_page_fault()函数的执行过程如下：
  - 首先得到导致异常发生的线性地址（虚拟地址）,对于IA32该地址放在CR2寄存器中。
  - 检查异常是否发生在中断或内核线程中,如是,则进行出错处理。
  - 检查该线性地址属于进程的某个vm_area_struct区间。如果不属于任何一个区间,则需要进一步检查该地址是否属于栈的合理可扩展区间。一但是用户态产生异常的线性地址正好位于栈区间的vm_start前面的合理位置,则调用expand_stack( )函数扩展该区间,通常是扩充一个页面,但此时还未分配物理页帧。

至此,线性地址必属于某个区间。

do_page_fault( )

- 根据错误码的值确定下一个步骤:
  - 如果错误码的值表示为写错误,则检查该区间是否允许写,不允许则进行出错处理。
  - 如果允许写就是属于写时拷贝(COW)。如果错误码的值表示为页面不存在,这就是所谓的按需调页(demand paging)。

do_page_fault( )

写时拷贝的处理过程：

- 首先改写对应页表项的访问标志位，表明其刚被访问过,这样在页面调度时该页面就不会被优先考虑。
- 如果该页帧目前只为一个进程单独使用,则只需把页表项置为可写。
- 如果该页帧为多个进程共享,则申请一个新的物理页帧并标记为可写,复制原来物理页帧的内容,更改当前进程相应的页表项,同时原来的物理页帧的共享计数减一。

do_page_fault( )

从文件区读入

按需调页的处理过程:

- 第一种情况页面从未被进程访问,这种情况页表项的值全为0。

  (1)如果所属区间的vm_ops->nopage不为空,表示该区间映射到一个文件,并且vm_ops->nopage指向装入页面的函数,此时调用该函数装入该页面。

  (2)如果vm_ops或vm_ops->nopage为空,则该调用do_anonymous_page( )申请一个页面;

- 另一种情况是该页面被进程访问过,但是目前已被写到交换分区,页表项的存在标志位为0,但其他位被用来记录该页面在交换分区中的信息。调用do_swap_page( )函数从交换分区调入该页面。

从交换区读入

换页

- Linux使用近似最近最少使用（LRU）页面置换算法来公平地选择将要从系统中换出的页面。
- 这种策略为系统中的每个页面设置一个年龄，它随页面访问次数而变化。页面被访问的次数越多则页面年龄越年轻；相反则越衰老。年龄较老的页面是待交换页面的最佳侯选者。

交换机制

- 当物理内存不足时，Linux通过某种机制选出内存中的某些页面换到磁盘上，以便留出空闲区来调入需要使用的页面
- 交换的基本原理：当空闲内存数量小于一个固定的极限值时，就执行换出操作（包括把进程的整个地址空间拷贝到磁盘上）。反之，当调度算法选择一个进程运行时，整个进程又被从磁盘中交换进来

页面交换

- 在Linux中，进行交换的单位是页面（页帧）而不是进程
- 在页面交换中，页面置换算法是影响交换性能的关键性指标，其复杂性主要与换出有关：
  - 哪种页面要换出？
  - 如何在交换区中存放页面？
  - 如何选择被交换出的页面？

选择被换出的页面策略

- 只有与用户空间建立了映射关系的物理页帧才会被换出，内核空间中内核所占的页帧则常驻内存
- 进程映像所占的页帧，其代码段、数据段可被换入换出，但堆栈段一般不换出
- 通过系统调用mmap()把文件内容映射到用户空间时，页帧所使用的交换区就是被映射的文件本身
- 进程间共享内存区其页帧的换入换出比较复杂
- 映射到内核空间中的页帧都不会被换出
- 内核在执行过程中使用的页帧要经过动态分配，但永驻内存

在交换区中存放页面

- 交换区也被划分为块，每个块的大小恰好等于一页，一块叫做一个页插槽
- 换出时，内核尽可能把换出的页放在相邻的插槽中，从而减少访问交换区时磁盘的寻道时间
- 若系统使用了多个交换区，快速交换区可以获得比较高的优先级
- 当查找一个空闲插槽时，要从优先级最高的交换区中开始搜索
- 如果优先级最高的交换区不止一个，应该循环选择相同优先级的交换区

页面交换策略

- 策略1：需要时才交换
- 策略2：系统空闲时交换
- 策略3：换出但并不立即释放
- 策略4：把页面换出推迟到不能再推迟为止

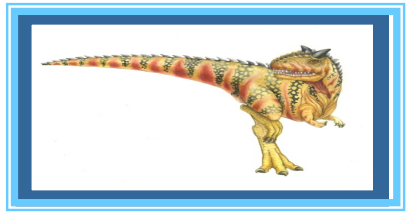
页面换入/换出及回收的基本思想

- 释放页帧：如果一个页帧变为空闲可用，就把该页帧的page结构链入某个空闲队列free_area，同时页帧的使用计数count减1。
- 分配页帧：调用___get_free_page()从某个空闲队列分配内存页帧，并将其页帧的使用计数count置为1。
- 活跃状态：已分配的页帧处于活跃状态，该页帧的数据结构page通过其队列头结构lru链入活跃页帧队列active_list，并且在进程地址空间中至少有一个页与该页帧之间建立了映射关系。
- 不活跃“脏”状态：处于该状态的页帧其page结构通过其队列头结构lru链入不活跃“脏”页帧队列inactive_dirty_list，并且原则是任何进程的页面表项不再指向该页帧，也就是说，断开页面的映射，同时把页帧的使用计数count减1。将不活跃“脏”页帧的内容写入交换区，并将该页帧的page结构从不活跃“脏”页帧队列inactive_dirty_list转移到不活跃“干净”页帧队列，准备被回收。
- 不活跃“干净”状态：页帧page结构通过其队列头结构lru链入某个不活跃“干净”页帧队列。如果在转入不活跃状态以后的一段时间内，页帧又受到访问，则又转入活跃状态并恢复映射。当需要时，就从“干净”页帧队列中回收页帧，也就是说或者把页帧链入到空闲队列，或者直接进行分配。

页帧交换守护进程kswapd

- Linux内核利用守护进程（Daemon） kswapd定期地检查系统内的空闲页帧数是否小于预定义的极限，一旦发现空闲页帧数太少，就预先将若干页帧换出
- kswapd相当于一个进程，它有自己的进程控制块task_struct结构，与其它进程一样受内核调度，但没有独立的地址空间

物理内存的管理



物理内存的管理

- 内存用途
  - 存储内核映像；
  - 其他内核子系统的内存需求；
  - 进程的需求；
  - 缓冲需求。
- 功能要求
  - 快速响应请求；
  - 尽可能地利用内存同时减少内存碎片。
- 解决方案
  - 基于区的伙伴系统及slab分配器。

页帧与区域

- 页帧：物理内存是以页帧(page frame)为基本单位，页帧的大小固定，IA32默认为4KB，PAGE_SIZE宏定义。页帧大小：
  - 命令：getconf PAGESIZE

```
ji@ubuntu:~$ getconf PAGESIZE
4096
```

  - 系统调用：getpagesize()
- 结点：访问速度相同的一个内存区域称为一个结点（Node），每个结点关联到一个处理器。
- 区：每个结点的物理内存因为用途不同又分成不同的区(zone)。例如x86，分成如下三个区：
  - DMA ZONE 低于16MB的内存，是DMA方式能够访问的物理内存。在内存分配时，尽可能保留这部分内存以供DMA方式使用。
  - NORMAL ZONE 介于16MB与896MB之间，直接被内核映射。
  - HIGHMEM ZONE 高端内存，超过896MB以上的部分，不能被内核直接映射。
- 区的划分没有任何物理意义，是内核内核为了管理页帧而采取逻辑上的分组。include/linux/mmzone.h定义了区结构
- Linux对不同zone的内存使用单独的伙伴系统(buddy system)管理,而且独立地监控空闲页帧。

参考资料：深入Linux内核架构

页帧与区

- Linux设置了一个mem_map[]数组管理内存页帧。
- mem_map[]在系统初始化时由free_area_init()函数创建，它存放在物理内存的低地址部分
- mem_map[]数组的元素是一个个的page结构体，每一个page结构体它对应一个物理页帧。
- page结构进一步被定义为mem_map_t类型，32字节，其定义在include/linux/mm_types.h中：

物理页帧

```
//    include/linux/mm_types.h
struct page {
        page_flags_t          flags;
        atomic_t      _count;       //Page frame's reference counter
        atomic_t       _mapcount;
        unsigned long          private;
        struct address_space  *mapping;
        pgoff_t               index;
        struct list_head      lru;
            //Contains pointers to the least recently used doubly linked list of pages.
        void                  *virtual;
       …
};
```

flags

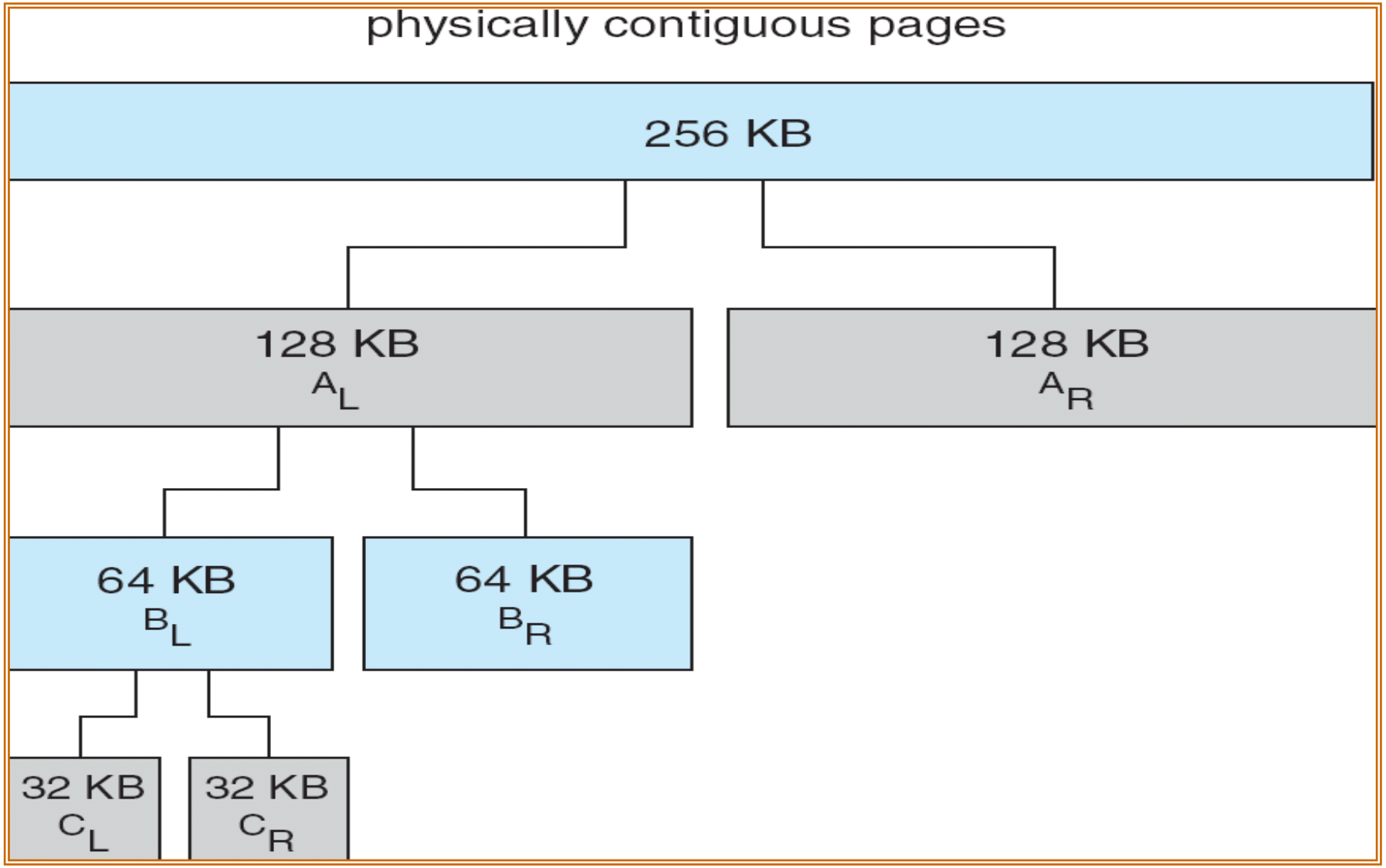
符号常量	意 义
PG_locked	页帧处于闭锁状态，正在装入该页帧
PG_error	页帧装入时发生错误
PG_referenced	页帧已装入，可以访问
PG_uptodate	页帧内容更新过
PG_free_after	关于页帧的I/O过程结束，页帧被释放
PG_decr_after	关于页帧的I/O过程结束，页帧计数减少
PG_swap_unlock_after	读出交换页帧后，页帧解除闭锁
PG_DMA	页帧可以用于DMA传送
PG_reserved	页帧被保留以后使用，当前禁止使用

物理内存的分配和回收

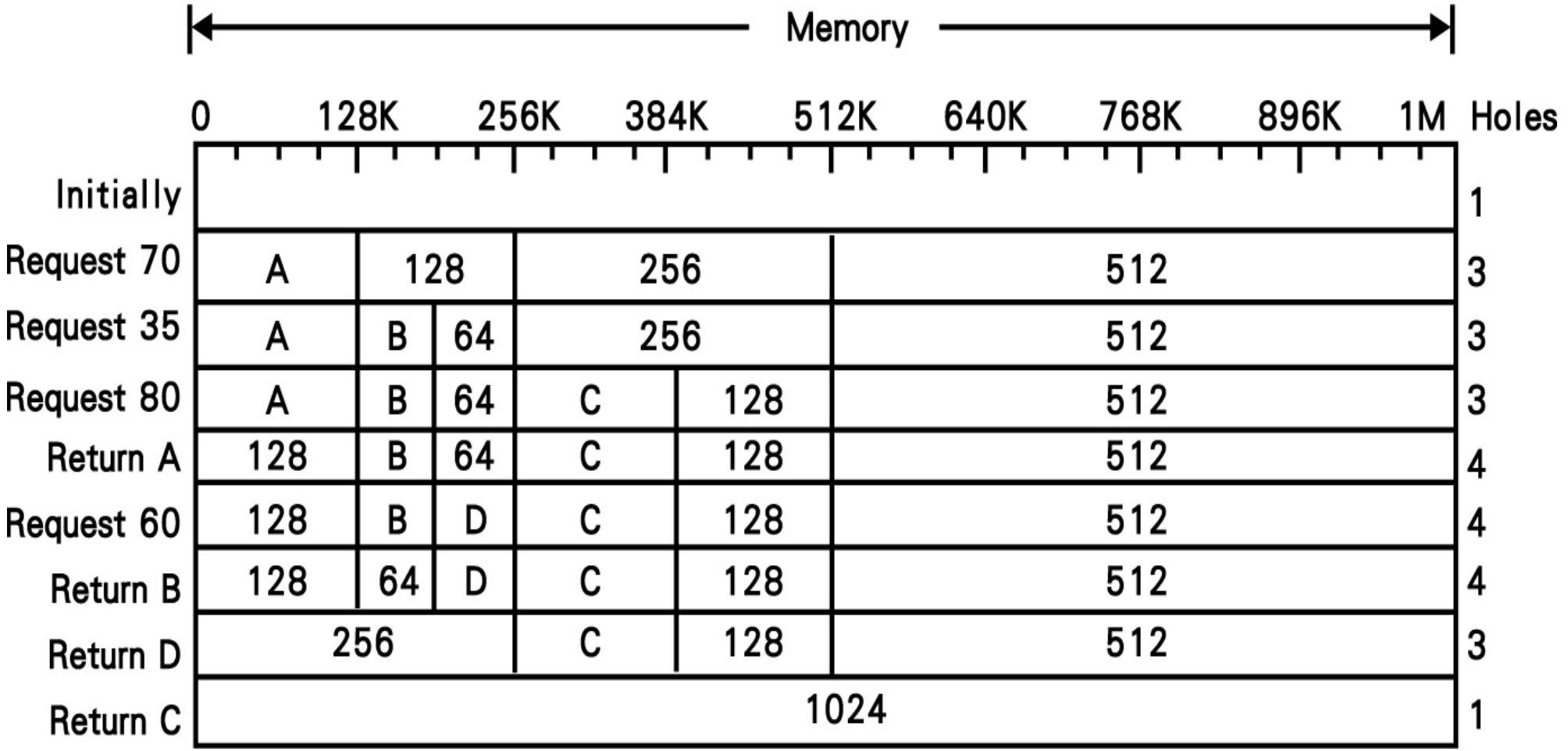
- 内存管理子系统采用基于内存区域的伙伴（Buddy）分配算法来管理可用物理页帧的分配和回收。
  - 最小的分配单位是页帧
  - 目的是分配一组连续的物理页帧

- 两组连续页帧被认为是一对“伙伴”必须满足如下条件:
  - 大小相同,比如说都有n个页帧;
  - 物理地址连续;
  - 第2个页帧块（组）的后面页帧号必须是$2\times n$的倍数。

Buddy System Allocator



buddy system例



Linux伙伴算法

- Buddy算法是把内存中的所有页帧按照2^n划分，其中n=0～10。对内存空间按1个页帧、2个页帧、4个页帧、8个页帧、16个页帧、32个页帧、…、1024个页帧进行划分。
- 划分后形成了大小不等的存储块，称为页帧块，简称页块。包含1个页帧的页块称为1页块，包含2个页帧的称为2页块，依此类推。
- Linux把物理内存划分成了1、2、4、8、…、1024共十一种页块。

Linux伙伴算法

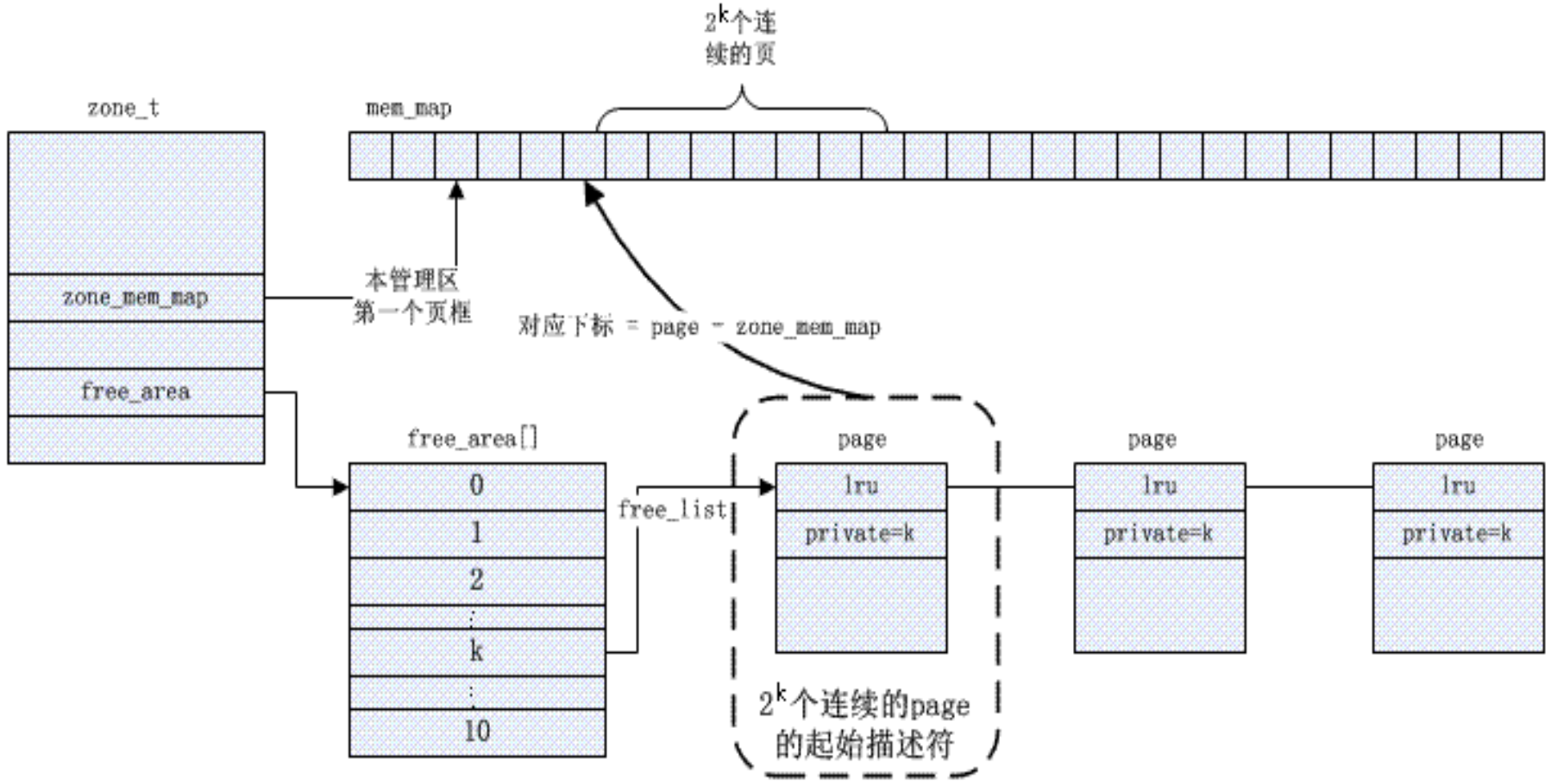
- Linux把空闲的页帧按照页块大小分组进行管理，数组free_area[]来管理各个空闲页块组。

```
struct free_area_struct {
        struct list_head       free_list[MIGRATE_TYPES];
        unsigned long          nr_free;
};
```

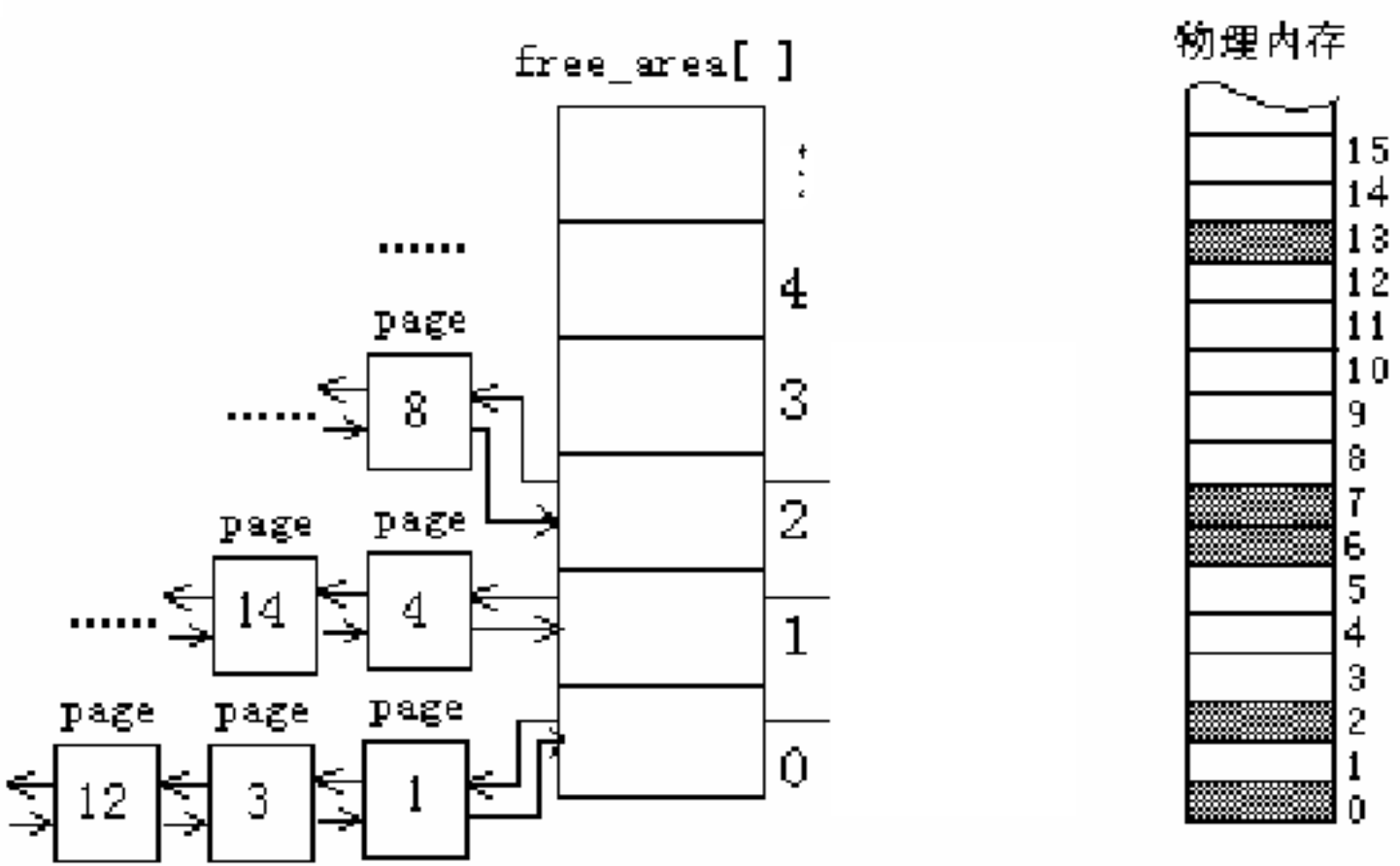
  - free_list字段是双向循环链表的头，该链表包含每个空闲页块(大小为2^n)的起始页帧的page结构；指向链表中相邻元素的指针存放在page结构的lru字段中。nr_free，它指定了大小为2^n页的空闲块的个数。

```
static struct free_area_struct  free_area[NR_MEM_LISTS];
```

数据结构


2^k个连续的页
zone_t
mem_map
zone_mem_map
free_area
本管理区第一个页框
对应下标 = page - zone_mem_map
free_area[]
0
1
2
:
k
:
10
free_list
page
lru
private=k
page
lru
private=k
page
lru
private=k
2^k个连续的page的起始描述符

Buddy算法



内存分配的Buddy算法

Buddy算法

- 系统按照Buddy关系把具有相同大小的空闲页帧块组成页块组，即1页块组、2页块组……1024页块组。
- 每个页块组用一个双向循环链表进行管理，共有11个链表，分别为1、2、4、8、…、1024页块的链表。
- 这些链表是由空闲页帧的page结构体双向连接而成，分别挂到free_area[] 数组上。

  free_area[0]的指针指向1页帧块的链表，

  free_area[1]的指针指向2页帧块的链表，

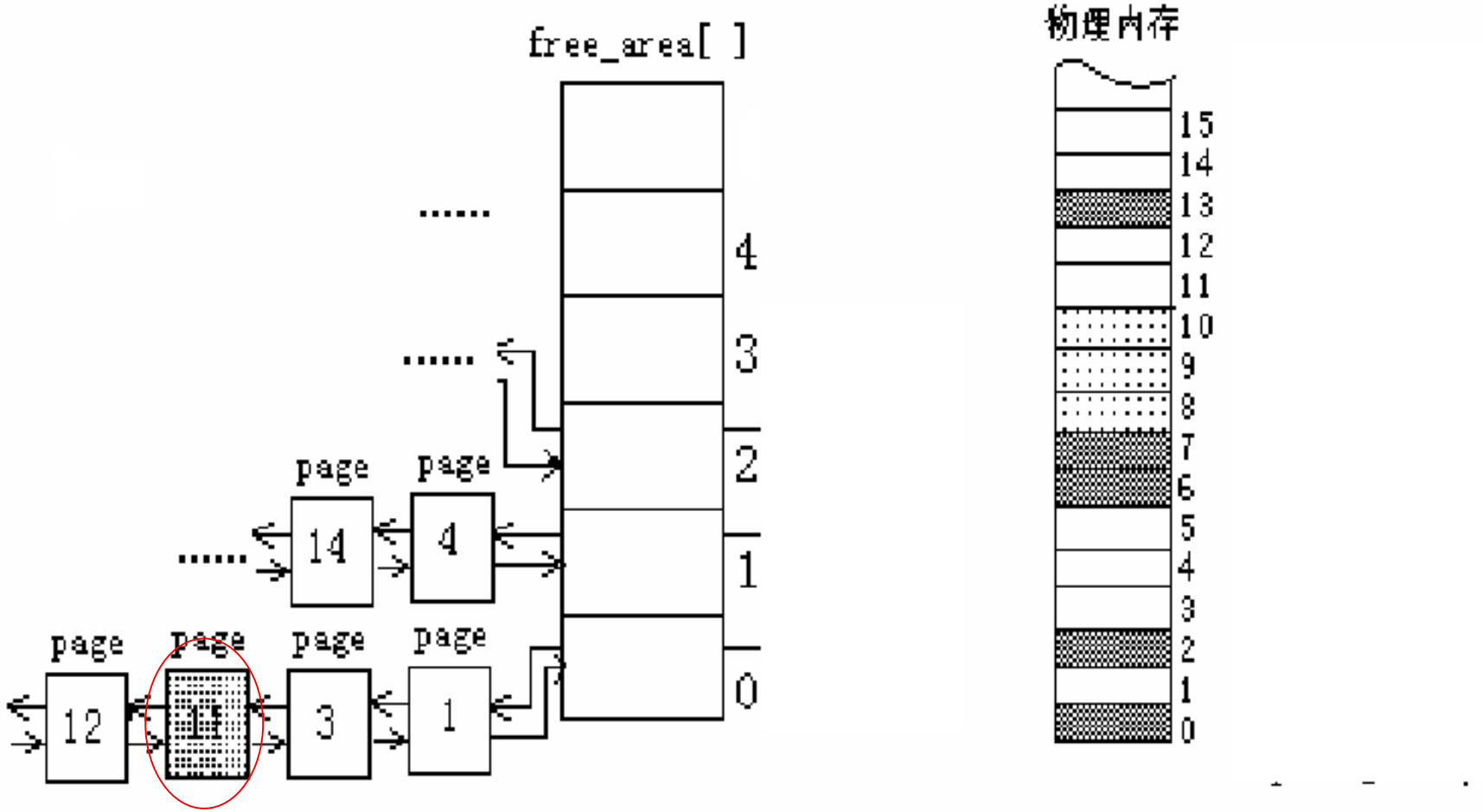
  ……，

  free_area[10]的指针指向1024页帧块的链表。

Buddy系统的内存分配

- 在请求内存分配时，系统按照Buddy算法，根据请求的页帧数在free_area[]对应的空闲页块组中搜索。
- 若请求的页帧数不是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帧块组。
- 当相应的页块组中没有可使用的空闲页帧块时就查询更大一些的页块组，在找到可利用的空闲页帧块后，分配所需的页帧。
- 当某一空闲页帧块被分配后，若仍有剩余的空闲页帧，则根据剩余页帧的大小把它们加入到相应的页块组中。
- 例：系统分配3帧后图（8、9、10帧）

8、9、10页帧分配后图

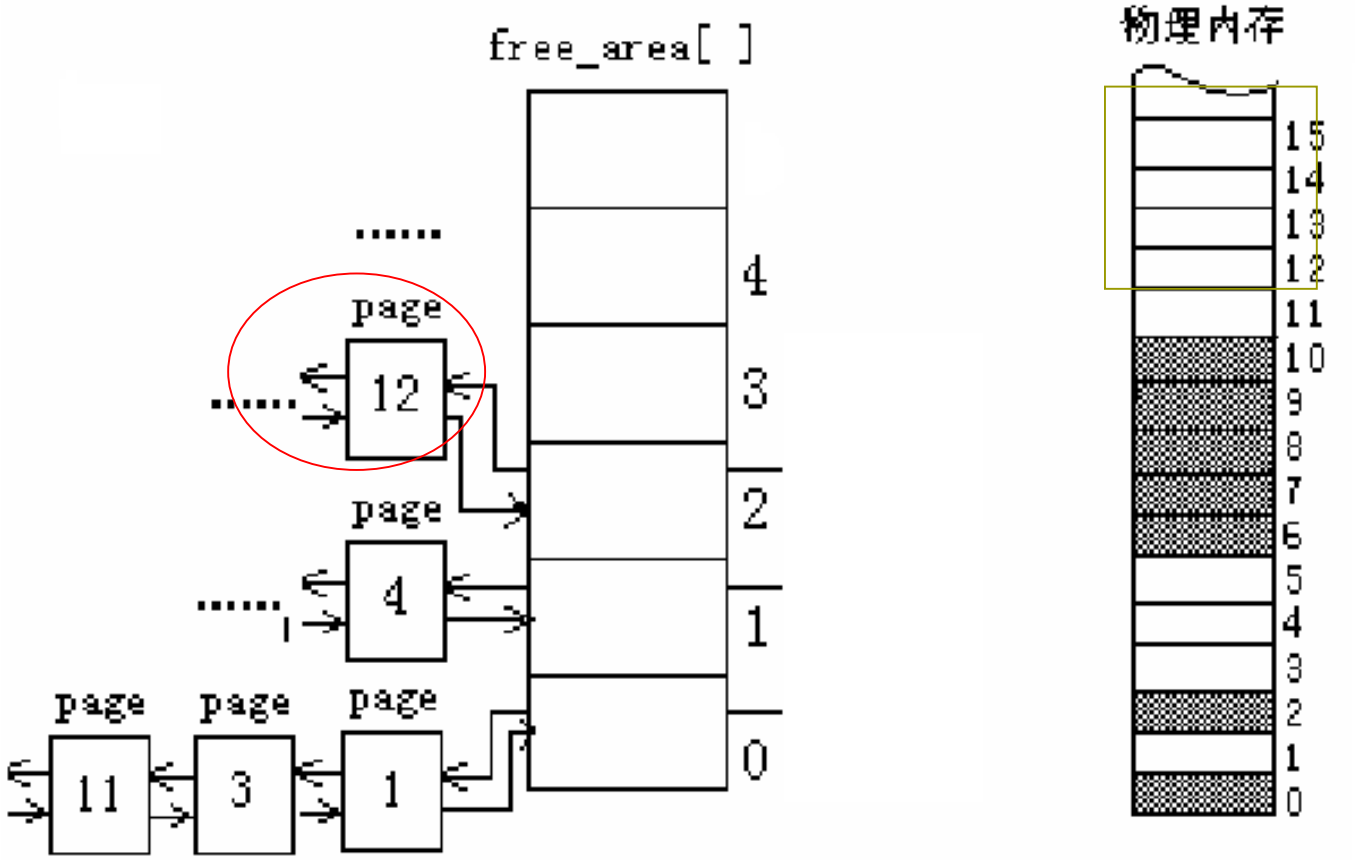


8、9、10页面分配后的示意

内存回收

- 在内存页帧释放时，系统将做为空闲页帧看待。 然后检查是否存在与这些页帧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。
- 内存回收需要一定时间

13页帧回收后图



13号页面释放后

Buddy算法

- 伙伴系统的操作：
  - 申请空间的函数为alloc_pages( );
  - 释放函数为free_pages( );
  - 当在申请内存发现页帧短缺时，还会唤醒kswapd内核线程运行,该线程会腾出一些空间以满足要求。

物理页帧的分配

- 函数alloc_pages( )用于分配物理页帧
- 该函数所做的工作如下：
  - 检查所请求的页块大小是否能够被满足
  - 检查系统中空闲物理页帧的总数是否已低于允许的下界
  - 正常分配。从free_area数组的第order项开始，这是一个mem_map_t链表。
  - 换页。通过下列语句调用函数try_to_free_pages（），启动换页进程

物理页帧的分配

1) 如果该链表中有满足要求的页块，则：

将其从链表中摘下；将free_area数组的位图中该页块所对应的位取反，表示页块已用；修改全局变量nr_free_pages（减去分配出去的页数）；根据该页块在mem_map数组中的位置，算出其起始物理地址，返回。

2) 如果该链表中没有满足要求的页块，则在free_area数组中顺序向上查找。其结果有二：

a) 整个free_area数组中都没有满足要求的页块，此次无法分配，返回。

b) 找到一个满足要求的页块，则：

将其从链表中摘下；将free_area数组的位图中该页块所对应的位取反，表示页块已用；修改全局变量nr_free_pages（减去分配出去的页数）；因为页块比申请的页块要大，所以要将它分成适当大小的块。因为所有的页块都由2的幂次的页数组成，所以这个分割的过程比较简单，只需要将它平分就可以：

I. 将其平分为两个伙伴，将小伙伴加入free_area数组中相应的链表，修改位图中相应的位；

II.如果大伙伴仍比申请的页块大，则转I，继续划分；

III.大伙伴的大小正是所要的大小，根据其在mem_map数组中的位置，算出它的起始物理地址，返回。

物理页帧的回收

- 函数free_pages()用于页块的回收
- 该函数所做的工作如下：
  - 根据页块的首地址addr算出该页块的第一页在mem_map数组的索引；
  - 如果该页是保留的（内核在使用），则不允许回收；
  - 将页块第一页对应的mem_map_t结构中的count域减1，表示引用该页的进程数减了1个。若count域的值不为0，有别的进程在使用该页块，不能回收，仅简单返回
  - 清除页块第一页对应的mem_map_t结构中flags域的PG_referenced位，表示该页块不再被引用；
  - 将全局变量nr_free_pages的值加上回收的物理页数
  - 将页块加入到数组free_area的相应链表中

slab分配器

- 伙伴系统是以页帧为基本分配单位，因而对于小对象容易造成内部碎片。
- 不同的数据类型用不同的方法分配内存可能提高效率。比如需要初始化的数据结构，释放后可以暂存着，再分配时就不必初始化了。内核的函数常常重复地使用同一类型的内存区，缓存最近释放的对象可以加速分配和释放。
- 解决办法：基于伙伴系统的slab分配器。
- slab分配器的基本思想：为经常使用的小对象建立缓冲,小对象的申请与释放都通过slab 分配器来管理。slab 分配器再与伙伴系统打交道。
- 好处：其一是充分利用了空间,减小了内部碎片；其二是管理局部化,尽可能少地与伙伴系统打交道,从而提高了效率。

slab分配器的构成

- 为不同的常用对象生成不同的缓冲,每个缓冲存储相同类型的对象。
- 每个slab又由一个或多个连续的物理页帧组成,包含了若干同种类型的对象。
- slab对象用结构slab_t来描述,某种特定对象建立的slab队列都有个队头。kmem_cache_t
- 系统有一个总的slab队列,其对象是其他对象的队头,其队头也是一个kmem_cache_t结构,叫cache_cache。

- 除了上面讨论的特定对象的缓冲外，Linux还提供了13种通用的缓冲区，其存储对象的单位大小分别为32B，64B，128B，256B，512B，1KB，2KB，4KB，8KB，16KB，32KB，64KB和128KB。
- 这些缓冲区用来满足特定对象之外的普通内存需求。单位大小级数增长保证了内部碎片率不超过50%。

slab分配器的相关操作:

(1)kmem_cache_create( )

- 该函数创建一种特定对象的kmem_cache_t结构,并加入cache_cache 所管理的队列。如下代码创建了inode_cache：

```
kmem_cache_create("inode_cache",sizeof(structinode), 0,
    SLAB_HWCACHE_ALIGN, init_once,NULL);
```

- kmem_cache_create( )函数一开始分配一个kmem_cache_t结构,然后进行一系列运算,以确定最佳的slab 构成。包括:每个slab由几个页帧组成,可包含几个对象；slab的控制结构应该在slab外面集中存放还是放在每个slab的尾部。

(2)kmem_cache_alloc( )与kmem_cache_free( )

- 当需要分配一个拥有专用slab队列的对象时,应该通过kmem_cache_alloc( )函数,相反的动作则是kmem_cache_free( )函数。
- 分配的对象来自slab缓冲，释放的对象归还slab缓冲,通过某种标识表明对象是可用还是已被占用。

(3) kmem_cache_grow()与kmem_cache_reap()

- kmem_cache_create()函数只是建立了所需的专用缓冲区队列的基础设施,所形成的slab队列是个空队列。而具体slab的创建则要等需要分配缓冲区时,却发现队列中并无空闲的缓冲区可供分配时,再通过kmem_cache_grow()来进行, kmem_cache_grow()再向伙伴系统申请空间。kswapd会定时调用kmem_cache_reap()来“收割”缓冲区队列。

(4) kmalloc( )函数与kfree( )函数

- 从通用的缓冲区队列中申请和释放空间。

实验作业

- 作业 分析do_page_fault( ) 。