
東北大學秦皇島分校
Northeastern University at Qinhuangdao

# Linux系统与内核分析

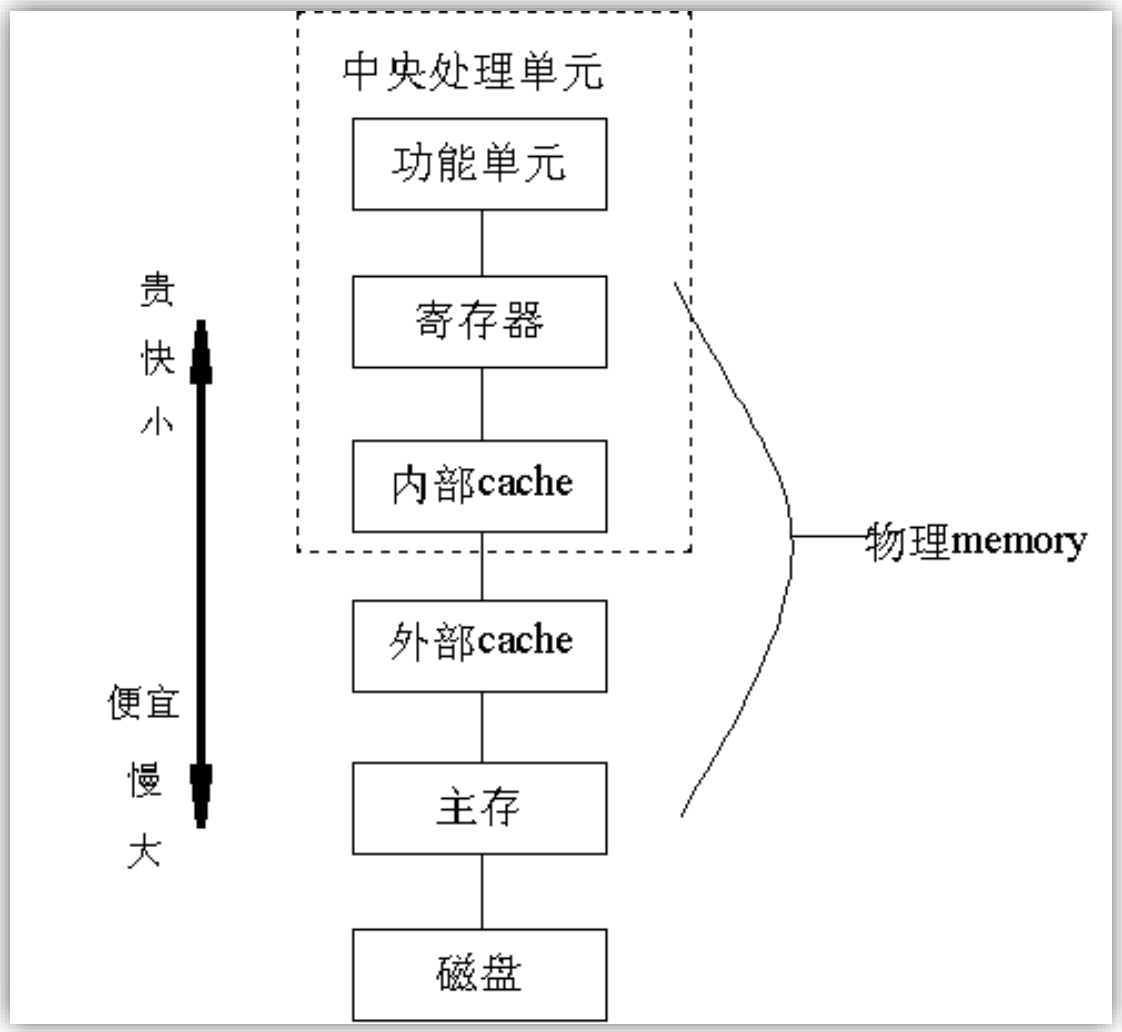
## --内存管理

于七龙

# 目录


# Part 1 Linux内存管理概述

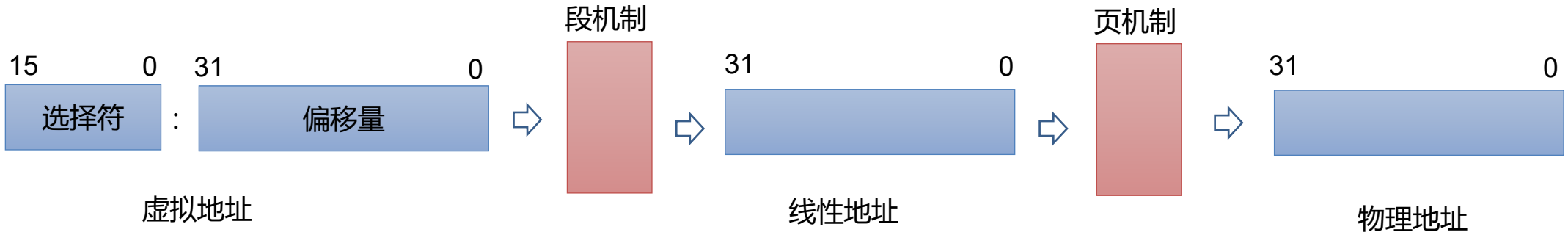
# 内存层次结构




- 内存结构层次

# 地址转换

- 内存管理单元(MMU)用于将虚拟地址转换为物理地址




- 地址转换

# 内存管理目的

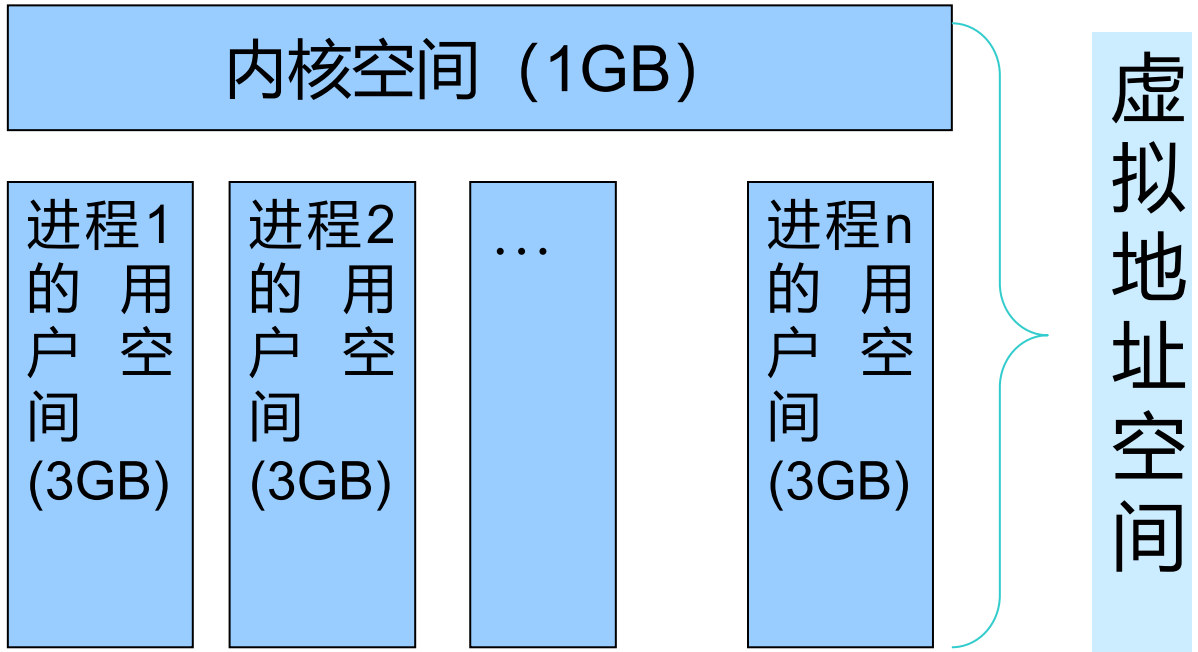
- 内存管理主要解决以下问题
  - 对源程序编译链接后形成的地址空间的管理
  - 程序装入内存过程中，虚拟地址向物理地址的转换
  - 物理内存的管理

# 虚拟地址空间

- Linux中的虚拟地址空间是系统的线性空间
  - 32位平台的线性空间是4G，即虚拟地址空间也为4G
- Linux中虚拟地址空间分为**内核空间**与**用户空间**
  - 内核空间
    - 虚拟地址的最高1G(0xC0000000-0xFFFFFFFF)，供内核使用
  - 用户空间
    - 虚拟地址的低3G(0x00000000-0xBFFFFFFF)，供进程使用

# 虚拟地址空间

- 用户空间(0-3G)由进程私有，最高的1G内核空间由所有进程共享




- 进程的虚拟地址空间

# 虚拟地址空间

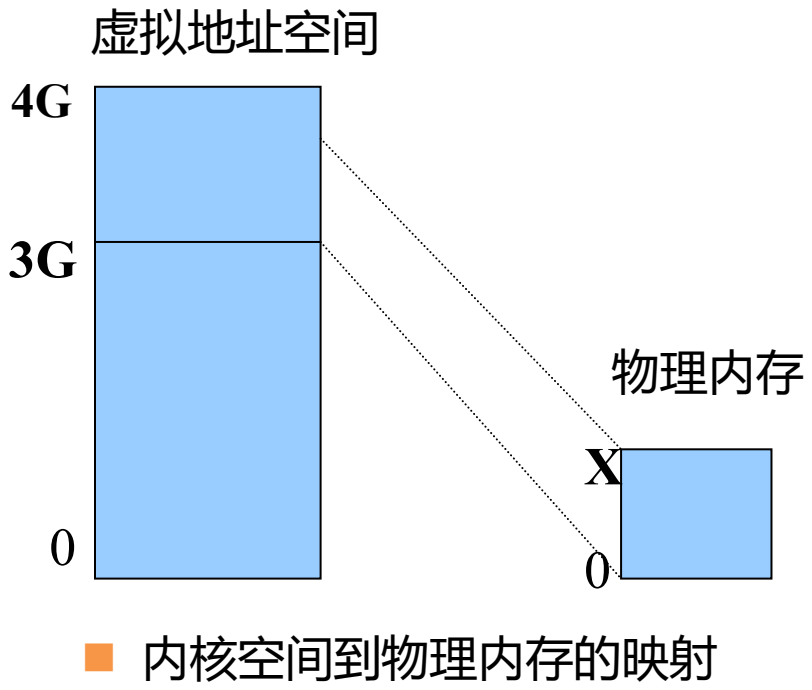
- 虚拟地址空间的变化，随进程切换而变化
  - 虚拟地址空间针对CPU
  - 每个进程有自己的虚拟地址空间
  - 当进程发生切换时，虚拟地址空间也随之切换
- 虚拟地址空间必须映射到物理内存空间中
  - 每个进程都有对应页表，每个进程的虚拟地址空间根据自身需要映射到物理地址空间上
  - 只要有当前进程的页表，CPU就可以实现其虚拟地址到物理地址的转换

# 内核空间

- 内核空间存放内核代码和数据
- 内核空间占据虚拟空间的高地址，但映射到物理内存中以低地址开始
  - Linux内核定义了内核地址到物理地址的转换方法

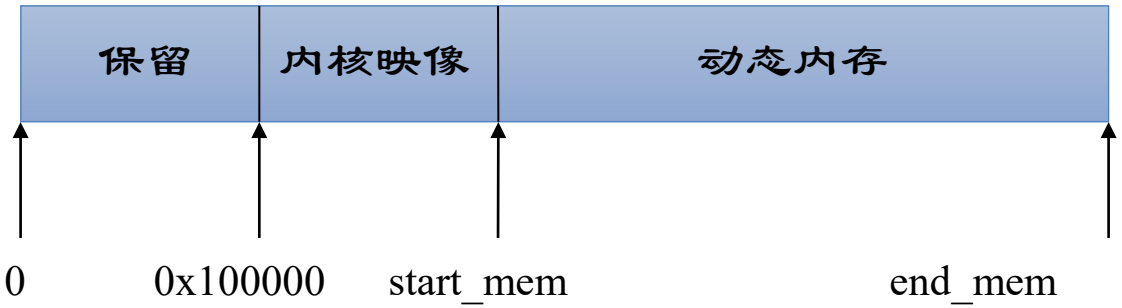
```
//page.h

#define __PAGE_OFFSET          (0xC0000000)
    ……
#define PAGE_OFFSET    ((unsigned long)__PAGE_OFFSET)
#define __pa(x)              ((unsigned long)(x)-PAGE_OFFSET)
#define __va(x)              ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

此定义只适用于内核空间虚地址的映射，用户空间的地址映射通过分页机制完成



# 内核映像

- 内核的代码段和数据段称为内核映像(Kernel Image)
- 内核映像在系统启动时被装入物理地址0x00100000开始的地方，即从1MB开始
  - 0-1MB的空间用于存放与系统硬件相关的代码和数据
  - 内核映像在内核空间中的起始地址是0xC0100000




系统启动后物理内存布局


# Part 2 用户空间管理

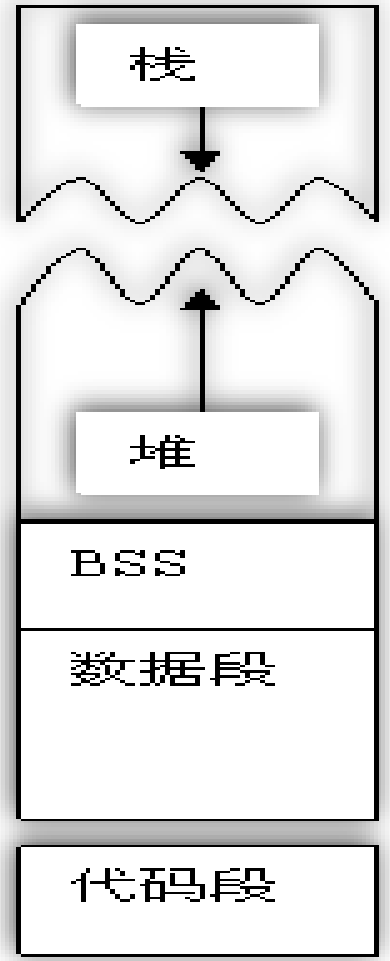
# 用户空间

- 用户空间为进程私有的3G虚拟空间，用于存放用户程序的代码和数据
- 用户空间可分为代码段、数据段、堆栈等区间
- 代码段、数据段、堆栈空间在进程建立时分配

# 用户空间区间划分

## 进程的用户空间区间划分

- 栈：位于用户空间顶部，运行时自上向下延伸
  - 大小固定，由内核自动分配释放 ，存放函数的参数值，局部变量的值等
- 堆：位于栈之下，运行时由下向上延伸
  - 动态分配内存通过malloc()，并添加至堆中
- BSS：未初始化的数据段，存放未初始化的全局变量和静态变量
- 数据段：存放已初始化的全局变量和静态变量
- 代码段：存放执行指令




用户空间划分

# 用户空间区间划分

- 用户空间划分举例
  - 代码详见 prtArea.c

```
root@bogon:/code/prtArea# ./prt
Bellow ar addresses  of types of process's memory:
Text Location:
        Address of main(Code Segment):0x5584ce895165
--------------------------------
Stack Location:
        Initial end of stack:0x7ffd48fcc194
        new end of stack:0x7ffd48fcc190
--------------------------------
Data Location:
        Address of data_var(Data segment): 0x5584ce898048
        new end of data_var(Data Segment):0x5584ce89804c
--------------------------------
BSS Location:
        Address of bss_var:0x5584ce898054
--------------------------------
Heap Location:
        Initial end of heap:0x5584cfb10000
        New end of heap:0x5584cfb10004
```

程序运行结果

# 用户空间申请

- 所谓向内核申请一块空间，实际上指请求内核分配一块虚存空间，以及相应的若干物理页面，并建立映射关系
- 内核在创建进程是并不为整个用户空间都分配好相应的物理空间，而是根据需要才真正分配一些物理页面并建立映射
- 系统利用请页机制避免物理内存过分使用

# 用户空间描述

- 进程用户空间主要由mm_struct和vm_area_struct结构描述
  - mm_struct，内存描述符，用于描述整个用户空间
  - vm_area_struct，**虚存区**，用于描述用户空间中的各个区间
    - 如代码段、数据段等都属于虚存区

# 用户空间描述

```
346
347  struct mm_struct {
348      struct vm_area_struct *mmap;        /* list of VMAs */
349      struct rb_root mm_rb;
350      u32 vmacache_seqnum;                   /* per-thread vmacache */
351  #ifdef CONFIG_MMU
352      unsigned long (*get_unmapped_area) (struct file *filp,
353                  unsigned long addr, unsigned long len,
354                  unsigned long pgoff, unsigned long flags);
355  #endif
356      unsigned long mmap_base;        /* base of mmap area */
357      unsigned long mmap_legacy_base;         /* base of mmap area in bottom-up allocations */
358      unsigned long task_size;        /* size of task vm space */
359      unsigned long highest_vm_end;       /* highest vma end address */
360      pgd_t * pgd;
361      atomic_t mm_users;          /* How many users with user space? */
362      atomic_t mm_count;          /* How many references to "struct mm_struct" (users count as 1) */
363      atomic_long_t nr_ptes;          /* Page table pages */
364      int map_count;              /* number of VMAs */
365
366      spinlock_t page_table_lock;     /* Protects page tables and some counters */
367      struct rw_semaphore mmap_sem;
368
369      struct list_head mmlist;        /* List of maybe swapped mm's.  These are globally strung
370                                   * together off init_mm.mmlist, and are protected
371                                   * by mmlist_lock
372                                   */
373
374
```

- mm_struct(*/incluede/linux/mm_types.h*)

# 用户空间描述

- mm_struct主要域

| 域名 | 说明 |
| --- | --- |
| mmap | 指向线性区对象的链表头 |
| mm_rb | 指向线性区对象的红-黑树的根 |
| pgd | 进程的页目录基地址 |
| mm_user | 表示共享地址空间的进程数目 |
| mm_count | 对mm_struct结构的引用进行计数。 |
| map_count | 在进程的整个用户空间中虚存区的个数 |
| mmap_sem | 线性区的读写信号量 |
| page_table_lock | 线性区的自旋锁和页表的自旋锁 |
| mmlist | 所有mm_struct通过mmlist域链接成双向链表 |
| start_brk，brk<br>start_stack | 堆地址，即用户空间的空洞。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 |

# 用户空间描述

- mm_struct主要域
  - 内核代码中，指向该数据结构的变量通常是mm
  - 每个进程对应一个mm_struct,并由task_struct中的*mm指针指向
  - 每个进程用户空间可能有多个虚存区，当虚存区较少时由链表组织，较多时由红黑树组织
  - 由于进程的用户空间及其虚存区有可能在不同的上下文中受到访问，所以mm_struct中设置P、V操作的信号量mmap_sem

# 虚存区描述

- 虚存区由vm_area_struct定义，简称VMA

```
247  struct vm_area_struct {
248      /* The first cache line has the info for VMA tree walking. */
249
250      unsigned long vm_start;      /* Our start address within vm_mm. */
251      unsigned long vm_end;        /* The first byte after our end address
252                                      within vm_mm. */
253
254      /* linked list of VM areas per task, sorted by address */
255      struct vm_area_struct *vm_next, *vm_prev;
256
257      struct rb_node vm_rb;
258
259      /*
260       * Largest free memory gap in bytes to the left of this VMA.
261       * Either between this VMA and vma->vm_prev, or between one of the
262       * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
263       * get_unmapped_area find a free area of the right size.
264       */
265      unsigned long rb_subtree_gap;
266
267      /* Second cache line starts here. */
```

- vm_area_struct(*/incluede/linux/mm_types.h*)

# 虚存区描述

- vm_area_struct主要域

| 域名 | 说明 |
| --- | --- |
| vm_mm | 指向虚存区所在的mm_struct结构的指针。 |
| vm_start，vm_end | 虚存区的起始地址和终止地址。 |
| vm_page_prot | 虚存区的保护权限。 |
| vm_flags | 虚存区的标志。 |
| vm_next | 构成线性链表的指针，按虚存区基址从小到大排列。 |
| vm_rb | 用于红-黑树结构 |
| vm_ops | 对虚存区进行操作的函数。这些给出了可以对虚存区中的页所进行的操作 |
| vm_pgoff | 映射文件中的偏移量。对匿名页，它等于0、vm_start或PAGE_SIZE |
| vm_file | 指向映射文件的文件对象 |
| vm_private_data | 指向内存区的私有数据 |

# 虚存区的意义

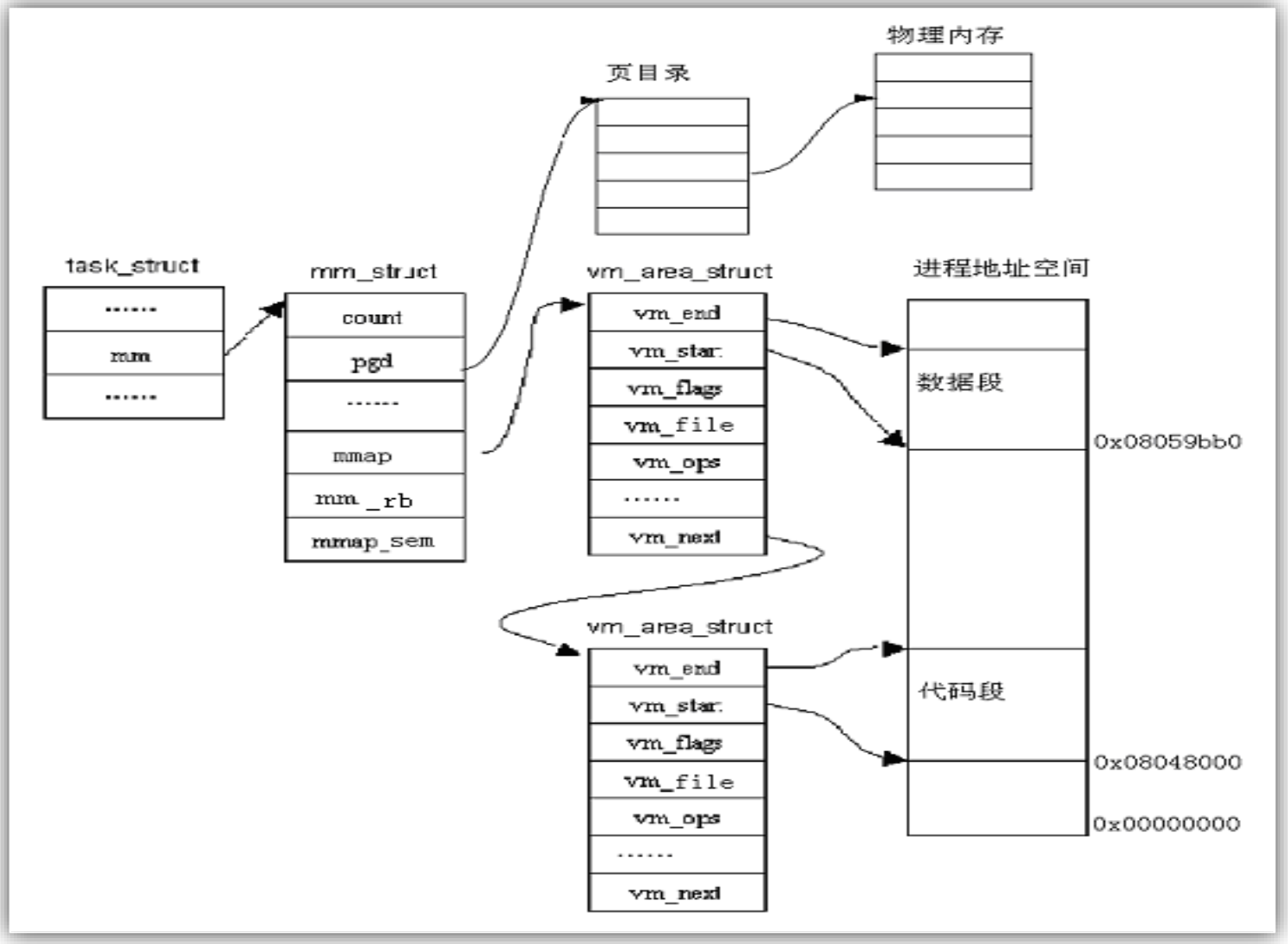
- 虚存区的来源可能不同，有的可能来自可执行映像，有的可能来自共享库，有的可能是动态分配的内存区
- 对不同的区间可能具有不同的访问权限，或可能有不同的操作

# 虚存区的操作

- 系统利用虚存区处理函数（vm_ops）来抽象对不同来源虚存区的处理办法
  - Linux利用了面向对象的思想，即把虚存去看成一个对象，用vm_area_structs描述了该对象的属性
  - vm_area_structs定义了虚存区相关操作函数

```
stuct vm_operations_struct{
        void (*open)(struct vm_area_struct * area);
        void (*close)(struct vm_area_struct * area);
        struct page *(*nopage)(struct vm_area_struct *area, unsigned long address, int unused);
        …
};
```

# 相关数据结构关系




- 数据结构关系

# 进程用户空间的创建

- 系统执行fork()系统调用时创建完整的用户空间
  - fork()执行内核调用copy_mm()
  - copy_mm()函数通过建立新进程的所有页表和mm_struct结构创建进程的用户空间
- 按“写时复制”方法，进程用户空间的创建所做工作仅仅是页表、mm_struct、vm_area_struct等结构的建立，并没有真正的复制物理页面

# 虚存映射

- 进程执行时，Linux并不一定将映像(代码段、数据段等)装入物理内存，可执行文件只是被链接到进程的用户空间，在进程的执行过程中，被引用到的程序再由内核装入到物理内存，这种将映像链接到进程空间的方法被称为“虚存映射”
  - 可执行映像映射到进程的用户空间时，由vm_area_struct结构描述
  - 每一个vm_area_struct结构代表可执行映像的一部分

# 虚存映射

## 虚存映射举例

- 例4.1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv)
{
        int i;
        unsigned char *buff;
        buff = (char *)malloc(sizeof(char)*1024);
        printf("My pid is :%d\n", getpid());
        for (i = 0; i < 60; i++)
                sleep(60);
        return 0;
}
```

# 虚存映射

- 虚存映射举例

虚存区地址范围　　权限　偏移量　　映射文件

```
root@bogon:/code/vma_4.2# cat /proc/28678/maps
55a8504d2000-55a8504d3000 r--p 00000000 08:01 1113767                    /code/vma_4.2/vma
55a8504d3000-55a8504d4000 r-xp 00001000 08:01 1113767                    /code/vma_4.2/vma
55a8504d4000-55a8504d5000 r--p 00002000 08:01 1113767                    /code/vma_4.2/vma
55a8504d5000-55a8504d6000 r--p 00002000 08:01 1113767                    /code/vma_4.2/vma
55a8504d6000-55a8504d7000 rw-p 00003000 08:01 1113767                    /code/vma_4.2/vma
55a85248a000-55a8524ab000 rw-p 00000000 00:00 0                          [heap]
7f629150c000-7f6291531000 r--p 00000000 08:01 133810                     /usr/lib/x86_64-linux-gnu/libc-2.30.so
7f6291531000-7f629167b000 r-xp 00025000 08:01 133810                     /usr/lib/x86_64-linux-gnu/libc-2.30.so
7f629167b000-7f62916c5000 r--p 0016f000 08:01 133810                     /usr/lib/x86_64-linux-gnu/libc-2.30.so
7f62916c5000-7f62916c8000 r--p 001b8000 08:01 133810                     /usr/lib/x86_64-linux-gnu/libc-2.30.so
7f62916c8000-7f62916cb000 rw-p 001bb000 08:01 133810                     /usr/lib/x86_64-linux-gnu/libc-2.30.so
7f62916cb000-7f62916d1000 rw-p 00000000 00:00 0
7f62916e6000-7f62916e7000 r--p 00000000 08:01 133805                     /usr/lib/x86_64-linux-gnu/ld-2.30.so
7f62916e7000-7f6291705000 r-xp 00001000 08:01 133805                     /usr/lib/x86_64-linux-gnu/ld-2.30.so
7f6291705000-7f629170d000 r--p 0001f000 08:01 133805                     /usr/lib/x86_64-linux-gnu/ld-2.30.so
7f629170e000-7f629170f000 r--p 00027000 08:01 133805                     /usr/lib/x86_64-linux-gnu/ld-2.30.so
7f629170f000-7f6291710000 rw-p 00028000 08:01 133805                     /usr/lib/x86_64-linux-gnu/ld-2.30.so
7f6291710000-7f6291711000 rw-p 00000000 00:00 0
7ffcbdb29000-7ffcbdb4a000 rw-p 00000000 00:00 0                          [stack]
7ffcbdbd0000-7ffcbdbd3000 r--p 00000000 00:00 0                          [vvar]
7ffcbdbd3000-7ffcbdbd4000 r-xp 00000000 00:00 0                          [vdso]
```

# 用户空间相关系统调用

- 对用户空间与虚存区产生影响的系统调用
  - fork()
  - mmap()：在用户空间内创建一个新的虚存区
  - munmap()：销毁虚存区
  - exec()：执行进程，装入新的可执行文件以代替当前用户空间的内容
  - exit()：销毁进程空间及虚存区


# Part 3 物理内存管理

# 请页机制

- Linux采用请页机制节约物理内存
  - 系统只将用户空间中的少数页装入物理内存，当访问的虚存页面未装入物理内存时，处理器产生缺页异常报告故障
- “请求调页”动态内存分配技术将页面的分配推迟到不能再推迟为止，即进程需要访问的页不在内存引发缺页异常为止
  - 进程开始运行时不需访问地址空间全部地址
  - 程序的局部性原理，程序执行每个阶段只需用到进程页的一小部分
- 因缺页异常必须由内核处理，系统需为请求调页付出额外的开销

# 页描述符

- 内核用struct_page描述内存中的每个物理页面，称为页描述符
  - 大多数32位体系结构页为4K，对于1G的内存，则有2^18个页

```
44  struct page {
45      /* First double word block */
46      unsigned long flags;        /* Atomic flags, some possibly
47                                   * updated asynchronously */
48      union {
49          struct address_space *mapping;  /* If low bit clear, points to
50                                           * inode address_space, or NULL.
51                                           * If page mapped as anonymous
52                                           * memory, low bit is set, and
53                                           * it points to anon_vma object:
54                                           * see PAGE_MAPPING_ANON below.
55                                           */
56          void *s_mem;                /* slab first object */
57      };
58
```

- 页描述符（include/linux/mm_type.h）

# 页描述符

- 系统中每个物理页都需分配struct page结构体
  - 如每个struct page占用40B内存，对于128MB物理内存，则需要1280KB内存空间存储内存信息
- 系统初始化时就建立page结构的数组mem_map，用以管理页面

# 伙伴算法

- Linux采用**伙伴算法（Buddy）**来管理物理内存中的页面
  - 用户程序运行过程中需不断申请和释放物理页面资源，频繁的请求和释放不同大小的一组连续页面，必然导致在内存中产生大量内存碎片
  - 因内存碎片的存在，导致无法满足大块的连续页面需求
    - 如DMA技术需连续内存，连续内存可提高读写速度

# 伙伴算法

## 伙伴算法原理

- 伙伴算法将空闲页面统一管理，连续的页面被定义为“块”
- 把规格相同的“块” 组成链表
- 通过数组对链表进行管理

# 伙伴算法

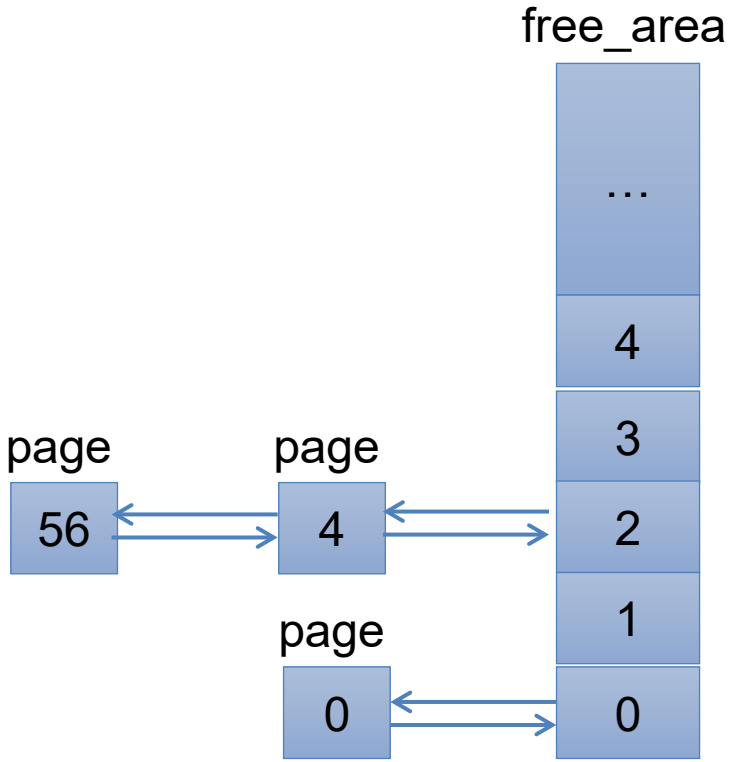
- 伙伴算法把所有空闲页面分为11个块链表，每个链表中的一个块含有2的次幂个页面
  - 第0个链表中块的大小都是2^0个页面，第1个链表中块大小是2^1个页面...

# 伙伴算法

- 伙伴算法采用free_area数组描述

```
struct free_area  free_area[MAX_ORDER];
```

```
struct free_area
{
        struct list_head free_list[MIGRATE_TYPES];
        unsigned long nr_free;        //空闲页块个数
 };
```




伙伴算法

# 伙伴算法

## 伙伴算法对空闲页面的分配

- 假如进程需要的块为64个页面，该算法先在64个页面的链表中查找是否有满足要求的空闲块，如果有，则直接分配，如果没有，则查找128个页面的链表是否有空闲块，如果有，则将该块分为两部分，一份分配出去，另一份则插入块大小为64的链表……
- 假如进程需要的块为100个连续页面，则查找128个页面的链表……

# 伙伴算法

## 伙伴算法对空闲页面的回收

- 伙伴算法对空闲页面分配的逆过程为对页面的回收
- 对满足以下要求的块进行合并
  - 两个块大小相同
  - 两个块的物理地址连续
- 块合并算法采用迭代操作，如果合并后还可以和相邻的块合并，则继续进行合并操作

# 物理页面回收

- 回收函数 free_pages （）
  - 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1。若count域的值非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

# 物理页面分配

- 当系统中空闲内存数量太少时，内核会唤醒守护进程kswapd
  - kswapd时内核交换守护进程，用于将内核中的某些页交换到外存

- freepages在系统初始化时被赋予初值，其界限值根据物理实际内存计算
- 系统中的物理页面数不能少于freepages.min
- 空闲页面数少于freepages.low时开始加强交换
- 空闲页面数少于freepages.high时启动后台交换

```
struct freepages_v1
{
        unsigned  int  min;
        unsigned  int  low;
        unsigned  int  high;
 }freepages_t;

freepages-t  freepages;
```

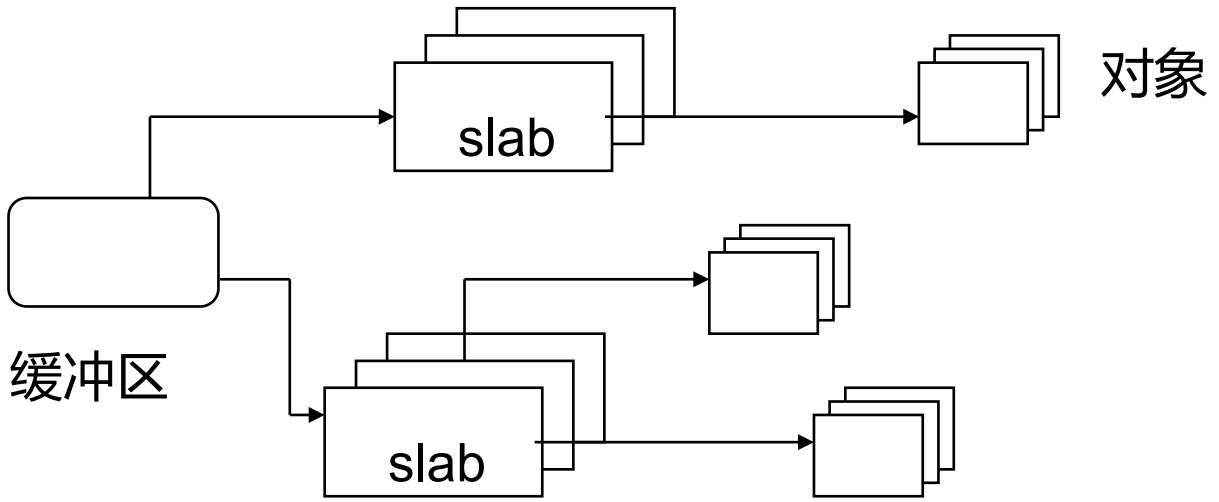
# Slab分配机制

## ◆ Slab机制用以提高物理内存利用率

- 当进程仅需几K空间时，按伙伴算法，仍需分配4K的页面，导致产生大量页内碎片
- 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进程，就要为该进程PCB分配内存区，当进程结束是又收回内存区，因为进程的创建和撤销非常频繁，导致系统花费大量时间反复分配或回收这些内存
- 伙伴算法的每次调用都会“弄脏”硬件高速缓存，导致增加了对内存的平均访问次数

# Slab分配机制

- 对于预期频繁使用的内存区，可在内存中创建一组专用缓冲区
  - 缓冲区被划分为多个slab，每个slab由一个或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




Slab的组成

# Slab分配机制

- 对于预期频繁使用的内存区，可在内存中创建一组专用缓冲区
  - slab分配器将相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，slab分配器就从一个slab列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片
  - slab分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中，当以后又要请求新的对象时，就可以从内存直接获取而不用重复初始化

# 物理内存分配函数

- Linux物理内存分配有kmalloc()与vmalloc函数
  - vmalloc()与 kmalloc()都可用于分配内存
  - kmalloc()分配的内存处于3GB～high_memory之间，这段内核空间与物理内存的映射一一对应，而vmalloc()分配的内存在VMALLOC_START～4GB之间，这段非连续内存区映射到物理内存也可能是非连续的
  - vmalloc() 分配的物理地址无需连续，而kmalloc() 确保页在物理上是连续的

# 物理内存分配函数

```
[13283.951027] pagemem = 0xffff914e73ce7000
[13283.951028] kmallocmem = 0xffff914e6de69700
[13283.951054] vmallocmem = 0xffffadd981901000
```

*#malloc_test.c*

```
pagemem = __get_free_page(GFP_KERNEL);
if(!pagemem)
        goto fail3;
printk(KERN_INFO "pagemem = 0x%lx\n", pagemem);

kmallocmem = kmalloc(100, GFP_KERNEL);
if(!kmallocmem)
        goto fail2;
printk(KERN_INFO "kmallocmem = 0x%lx\n", kmallocmem);

vmallocmem = vmalloc(1000000);
if(!vmallocmem)
        goto fail1;
printk(KERN_INFO "vmallocmem = 0x%lx\n", vmallocmem);
```

- 内存分配函数


# Part 4 交换机制

# 交换机制

- 交换机制用于内存不足时释放内存空间
  - 交换机制由守护进程kswapd完成
  - 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交换空间，以便留出空闲区来调入需要使用的页面
  - 交换的基本原理：当触发交换机制时，就执行换出操作（包括把进程的整个地址空间拷贝到交换空间）。反之，当调度算法选择一个进程运行时，整个进程又被从交换空间中交换进来
  - 交换空间是在外部存储设备中开辟的空间，用于临时存放从内存中调出的页面

# 交换策略

- 需要时才交换：当缺页异常发生时，如果没有空闲的页面可供分配，就将一个或多个内存页面换出到磁盘上
- 系统空闲时交换：系统空闲时，预先换出一些页面
- 换出但并不立即释放：缓存
- 把页面换出推迟到不能再推迟为止：“干净”页面，可以一直缓冲到必要时才加以回收