Linux源码阅读——内存管理

Linux源码阅读

- 源码版本及下载
  - **Linux-2.6.24.3**
  - ftp://ftp.tsinghua.edu.cn/mirror/kernel.org/linux/kernel/v2.6/linux-2.6.24.3.tar.bz2
  - 课程网站上也会有镜像供下载
- 阅读工具
  - 推荐**SourceInsight**，个人自选

Linux源码阅读任务

- 理解讲义中描述的Linux内核内存管理结构
- 理解伙伴系统的工作原理
- 理解Slab内存分配器的工作原理

- 阅读理解新的Slob和Slub内存分配器的代码，分析其工作原理
  - mm/slob.c, mm/slub.c

Linux 2.6内存管理

Linux中的分段(1)

- 内核代码段__KERNEL_CS：范围 0-4G；可读、执行；DPL=0
- 内核数据段__KERNEL_DS：范围 0-4G；可读、写；DPL=0
- 用户代码段__USER_CS：范围 0-4G；可读、执行；DPL=3
- 内核代码段__USER_DS：范围 0-4G；可读、写；DPL=3

Linux中的分段(2)

- **TSS(**任务状态段**)**：存储进程的硬件上下文，进程切换时使用，每个**CPU**有一个
- **default_ldt**：理论上每个进程都可以同时使用很多段，这些段可以存储在自己的**ldt**段中，但实际**linux**极少利用**x86**的这些功能，多数情况下所有进程共享这个段，它只包含一个空描述符

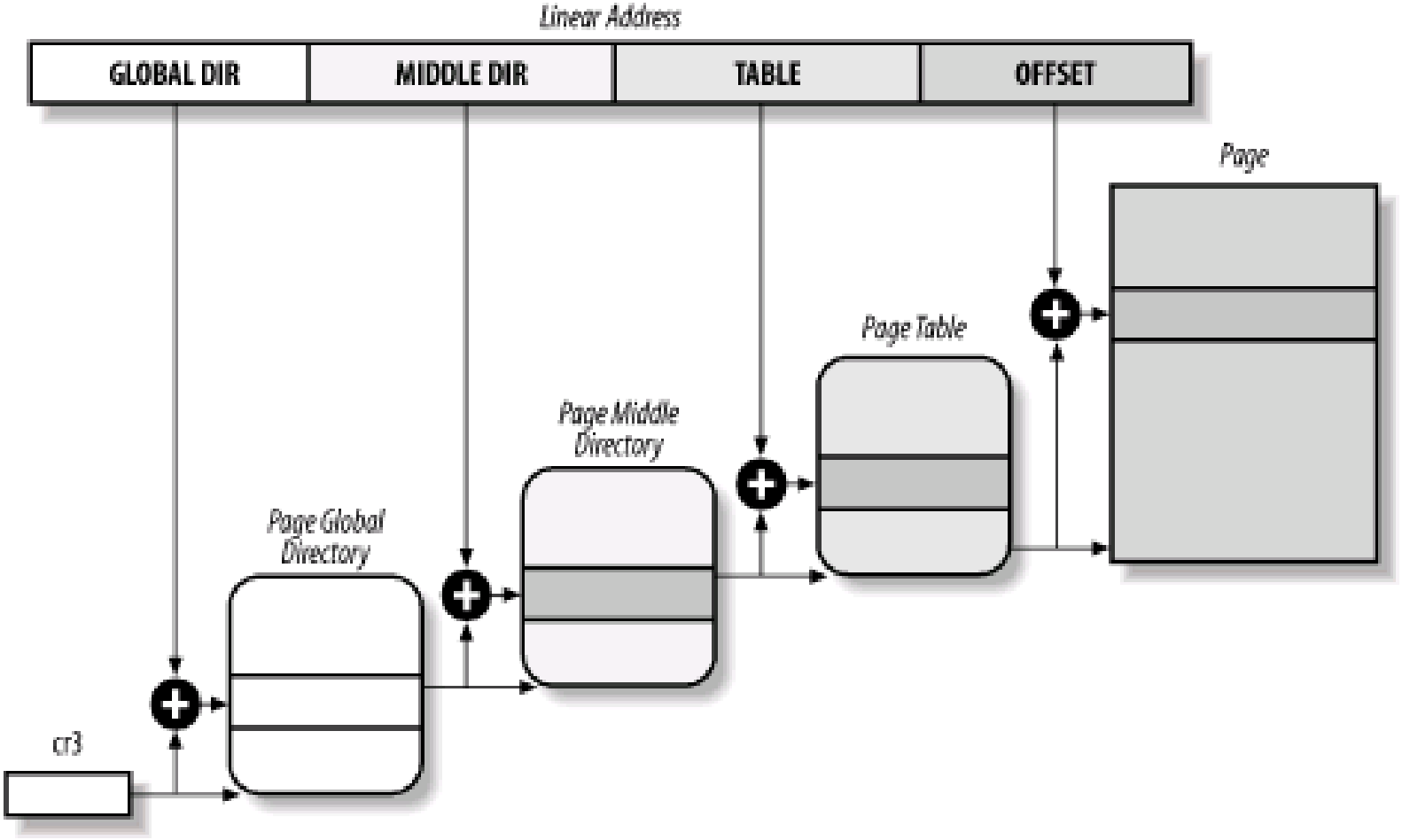
Linux中的分段(3)

- 由于历史原因，IA32体系结构仍然强制使用硬件分段
- 段式映射基地址总是0，逻辑地址与虚拟地址总是一致的

Linux中的分页(1)

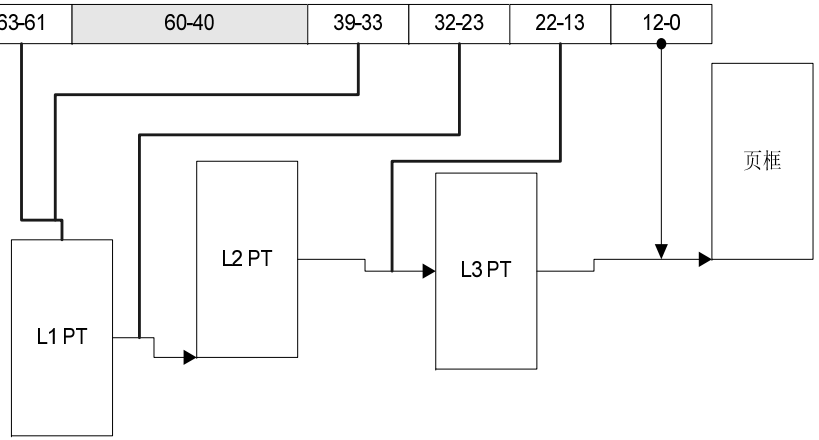
- 与体系结构无关的三级页表模型
  - **pgd**，页目录
  - **pmd**，页中级目录
  - **pte**，页表项
- 为什么使用三级页表
  - 设计目标决定：可移植性
  - 硬件特性决定：**PAE**

Linux中的分页(2)


Linear Address
GLOBAL DIR
MIDDLE DIR
TABLE
OFFSET
Page
Page Table
Page Middle Directory
Page Global Directory
cr3

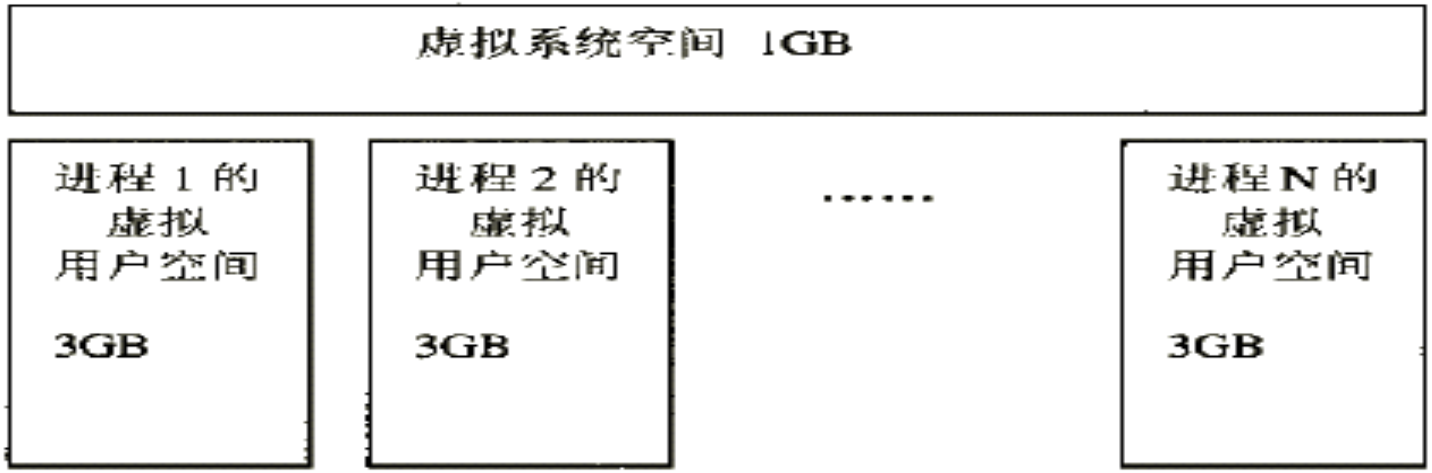
虚拟地址格式

- 采用三级页表结构，每个页框**8KB**，每级目录占用一个页框，每项**8**字节大小，由全局目录**(PGD)**、中间目录**(PMD)**、页表项**(PTE)**组成
- 全局目录索引被分为**pgdh**和**pgdl**，**ar.k7**寄存器指向当前进程的页表树基地址



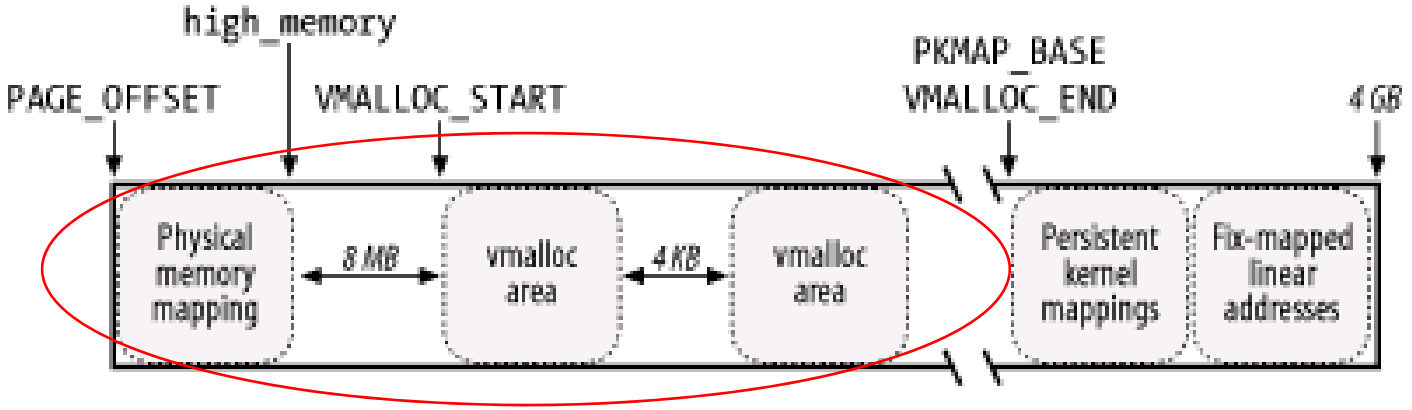
Linux中的分页(3)

- 进程页表
  - 各进程拥有自己的3G用户空间
  - 内核占用最高的1G作为系统空间，系统空间由所有进程共享



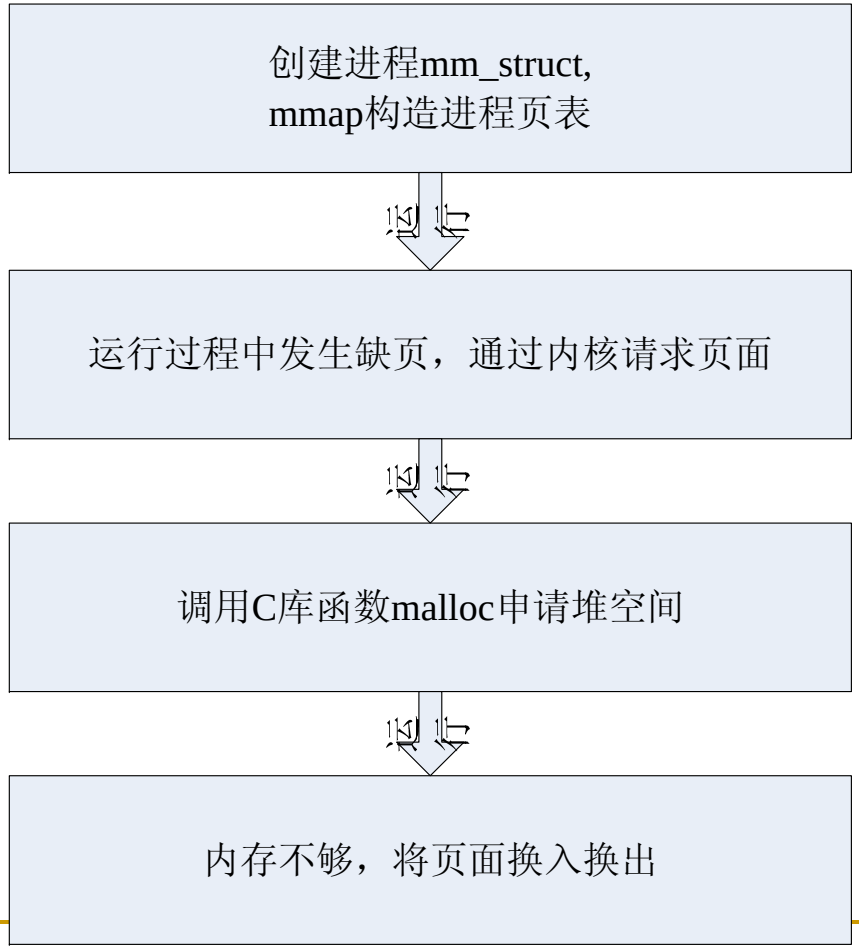
Linux中的分页(4)

- 1G的内核空间
  - 内核空间＝＝物理内存空间
  - 1G的后128MB用作实现非连续内存分配和固定映射的线性地址



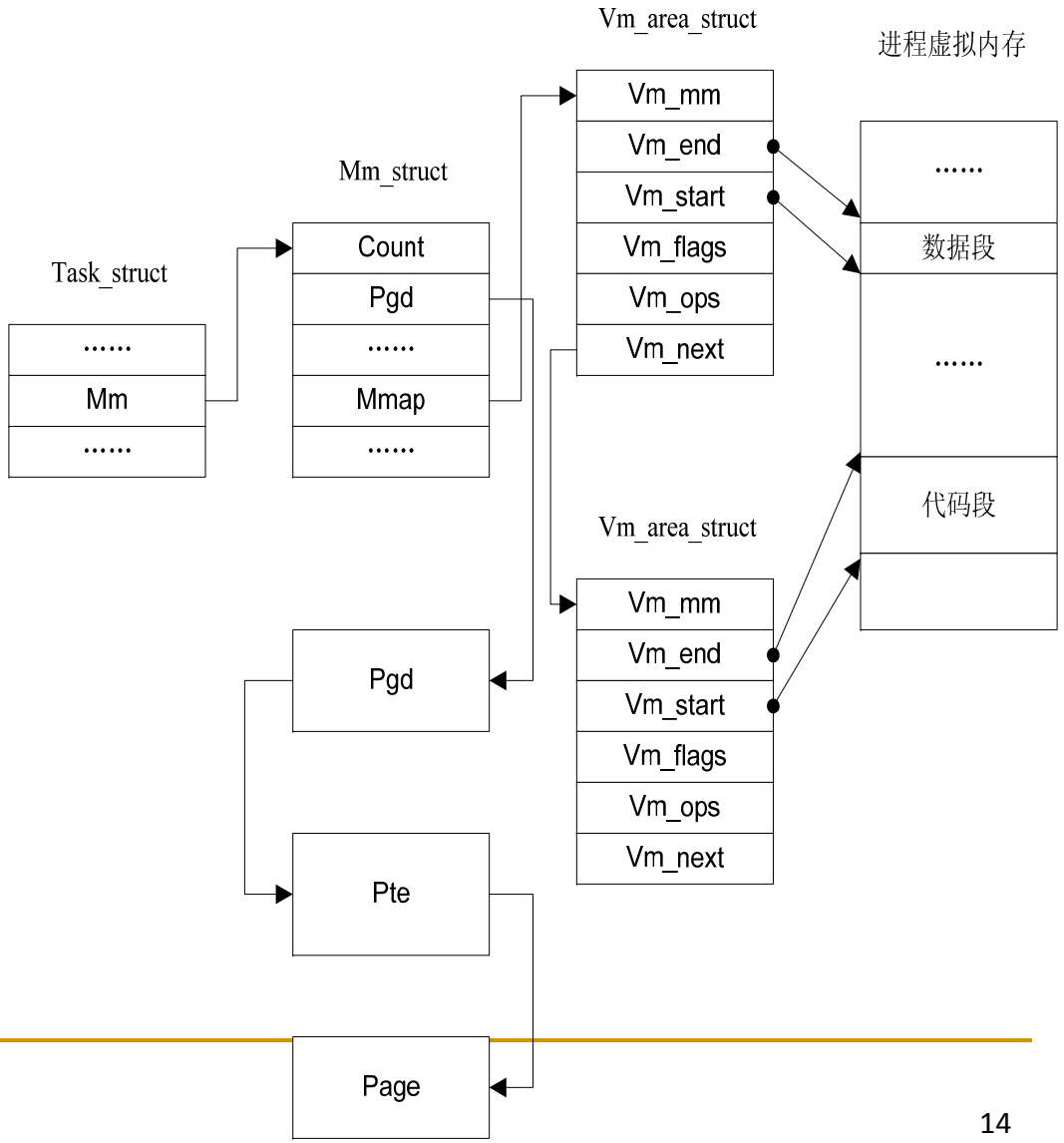
内存管理场景

- **shell参与，利用fork或exec以及mmap系统调用构造进程页表**



进程的内存组织

- 内存管理数据结构
  - 每个进程有一个**mm_struct**结构，描述进程的虚拟内存
  - **vm_area_struct**结构描述进程的虚拟内存地址区域
    - 对页错误处理有同一规则的进程虚拟内存空间部分，如共享库、堆栈
  - **page**结构描述一个物理页，系统保证跟踪到每一个物理页



数据结构-mm_struct

```
struct mm_struct {
    struct vm_area_struct *mmap;
    struct vm_area_struct *mmap_avl;
    struct vm_area_struct *mmap_cache;
    pgd_t * pgd;
    atomic_t count;
    int map_count;
    struct semaphore mmap_sem;
    unsigned long context;
    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;
    unsigned long def_flags;
    unsigned long cpu_vm_mask;
    unsigned long swap_cnt;
    unsigned long swap_address;
    void * segments;
};
```

VMA链，VMA AVL树，和最近使用的VMA Cache

指向PGD的指针

代码，数据，堆栈段的地址范围

- Mm_struct标识了一个进程
- 定义在文件/linux/include/linux/sched.h

数据结构-vm_area_struct

```
struct vm_area_struct {
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;

    struct vm_area_struct *vm_next;

    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_share;
    struct vm_area_struct **vm_pprev_share;

    struct vm_operations_struct * vm_ops;
    unsigned long vm_offset;
    struct file * vm_file;
    unsigned long vm_pte;
};
```

VMA的地址范围

VMA链

VMA保护位和标志

VMA AVL树

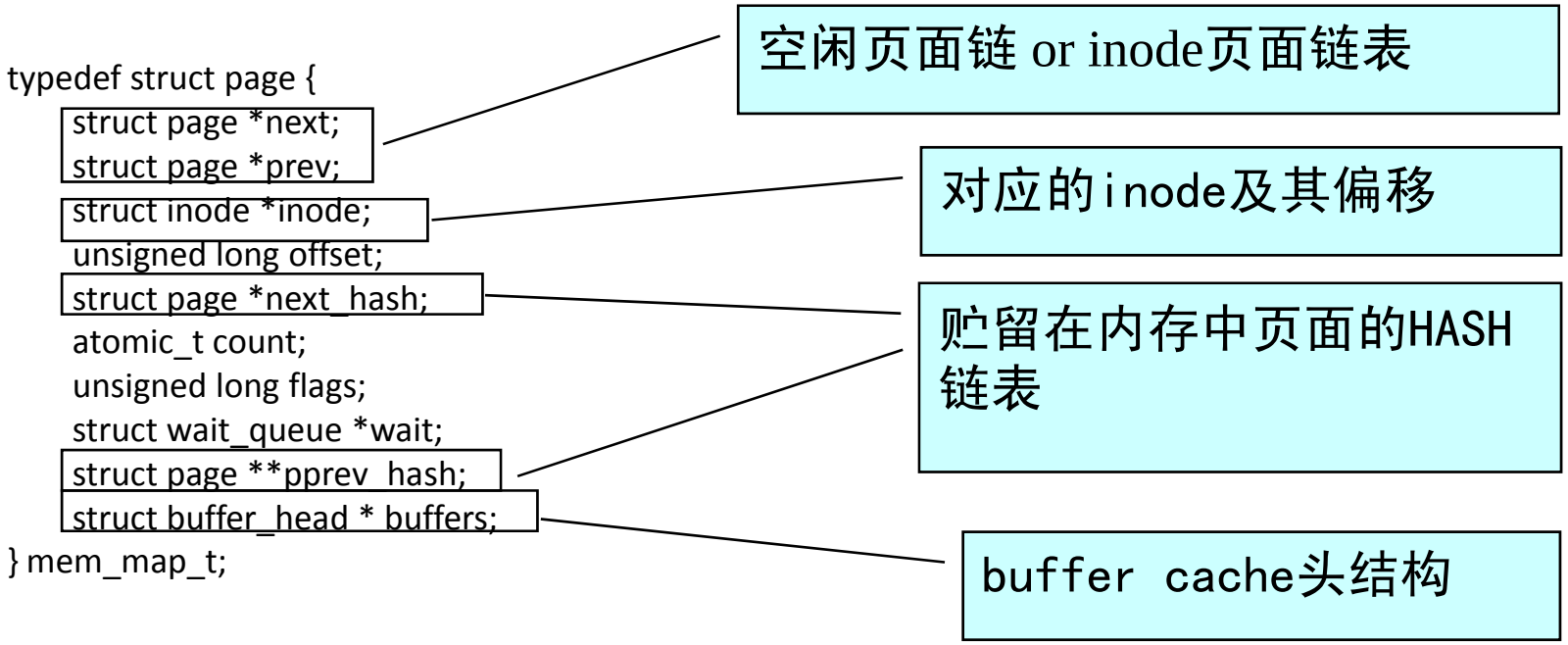
共享VMA链

VMA操作函数表

对应file及其偏移

- Vm_area_struct管理着进程的虚拟空间的一个区域
- 定义在文件/linux/include/linux/sched.h

数据结构-Page

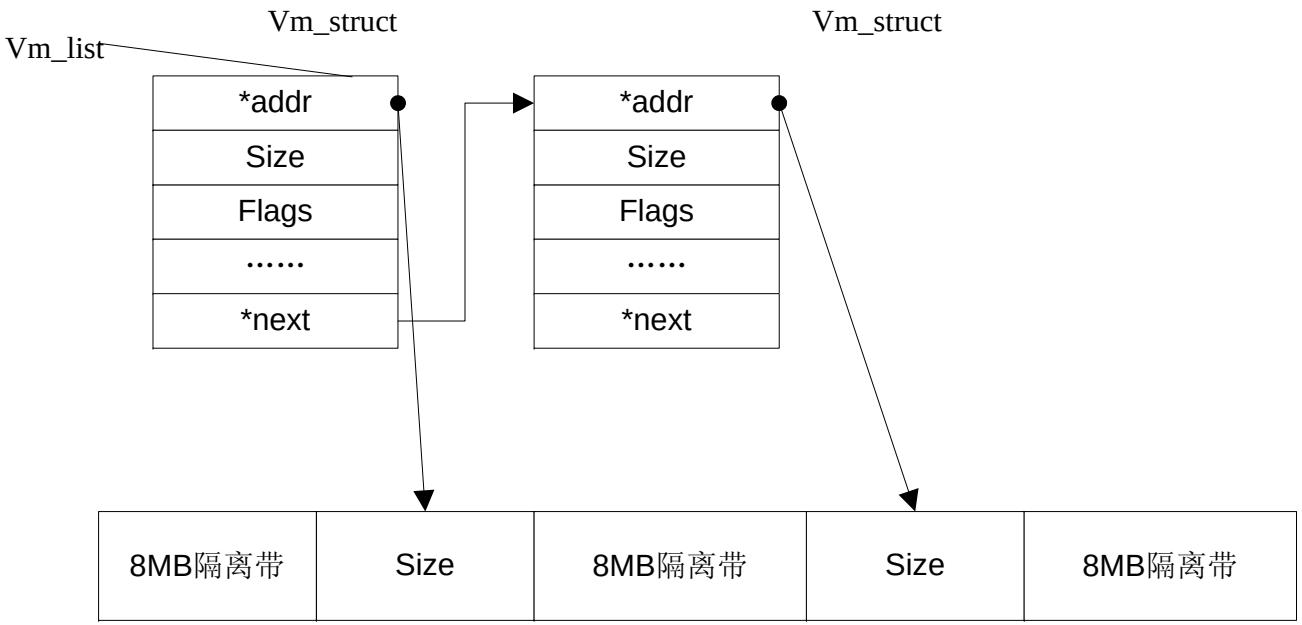


- Page(mem_map_t)代表了物理上的一个页
- 定义在文件/linux/include/mm.h

内核虚拟内存接口—大容量对象缓存

- **kmalloc**和**kfree**分配真实地址已知的实际物理内存块
  - 适合设备驱动使用
- **vmalloc**和**vfree**用于对内核使用的虚拟内存进行分配和释放
  - 位于**3G+high_memory+VMALLOC_OFFSET**以上高端，由**vmlist**链表管理
  - **VMALLOC_OFFSET**是长度为**8MB**的“隔离带”，起越界保护作用

- **vm_struct结构与虚拟内存关系**



- 函数__vmalloc(unsigned longn size, int gfp_mask, pgprot_t prot)
  - **size:** 分配的虚拟空间大小
  - **gfp_mask:** 页面分配器标志
  - **prot:** 一分配页的保护掩码
  - 分配足够页数与**size**相配
  - 页面映射到连续的内核虚拟空间，页面可以不连续

- 函数__vmalloc(unsigned longn size, int gfp_mask, pgprot_t prot)
  - 在vmlist寻找一个大小合适的虚拟内存块，get_vm_area(size)
  - 检查该虚拟块是否可用，如果可用则建立页表项，alloc_area_pte()
  - 建立页目录，set_pgdir()
  - 找到空闲分配给调用进程，get_free_page()
  - 如果不可用，必须释放该虚拟块，vfree()

内存映射

- 文件**mmap.c**的主要函数为**do_mmap**，其功能是把从文件结构中得到的逻辑地址转换为**vm_area_struct**结构所需要的地址

内存映射原理

- 进程形成**vm_area_struct**后，**vm_area_struct**上的操作函数也随之初始化
- 逻辑地址与物理地址由内核与硬件**MMU**共同完成
  - 内核通过页目录和页表告诉**MMU**如何把每个进程的逻辑页面映射到相应的物理页面
  - **MMU**在进程提出内存请求时完成实际转换工作

Mmap系统调用

- 进程可以通过**mmap**，将一个已打开的文件内容映射到它的用户空间，直接调用**do_mmap**完成映射
- 参数：**file**为映射的文件，**addr**为映射的地址，**len**为**VMA**的长度，**prot**指定**vma**段的访问权限，**flag**为**vma**段的属性

- 参数prot取值

名称	值	含义
PROT_READ	0x1	只允许读
PROT_WRITE	0x2	只允许写
PROT_EXEC	0x4	允许执行
PROT_NONE	0	不允许访问

- 参数flag取值

名称	值	含义
MAP_FIXED	0x10	Vma只能从虚拟地址addr映射
MAP_SHARED	0x01	写操作作用在同一组共享页面上，不发生copy on write
MAP_PRIVATE	0x02	Copy on write

Sys_brk系统调用

- 支持**malloc**和**free**的底层操作，**malloc**通过**sys_brk**向内核申请一段虚地址空间**vma**来建立地址映射，**sys_brk**将建立全部的映射，而不必通过缺页中断建立映射

缺页中断

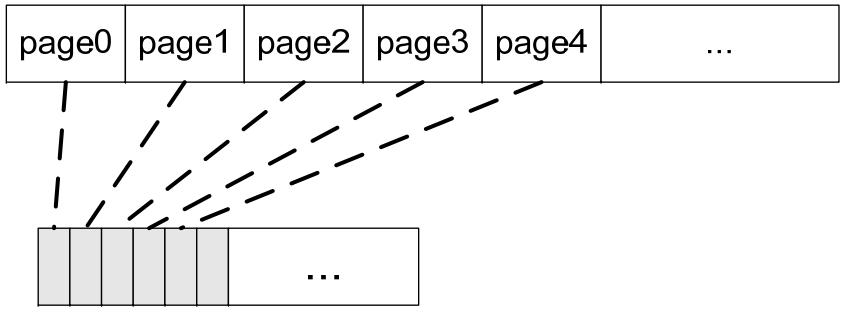
- **Do_no_page (mm/memory.c)**
- 页故障错误的产生有三种原因
  - 程序出现错误，虚拟内存无效，内核向进程发送**SIGSEGV**信号并终止进程运行
  - 虚拟内存有效，但其所有对应的页不再物理内存中，这时操作系统必须从磁盘映象（未分配或共享库）或交换文件（被换出）中将其装入内存
  - 要访问的虚地址被写保护，发生保护错误

缺页中断(续)

- Cpu在地址映射过程中发现页表项或页目录项中的P标志位为0，则表示相应的物理页面不在内存，处理器产生异常，句柄为do_page_fault函数，参数error_code表示错误码，描述了页错误发生的虚地址和访问类型
  - Bit0　0表示没发现页，1表示页保护错误
  - Bit1　0表示读，1表示写
  - Bit2　0表示内核模式，1表示用户模式
- Do_page_fault从CR2寄存器中得到发生错误的虚拟地址，找到相应的vma，然后调用handle_mm_fault函数进行处理

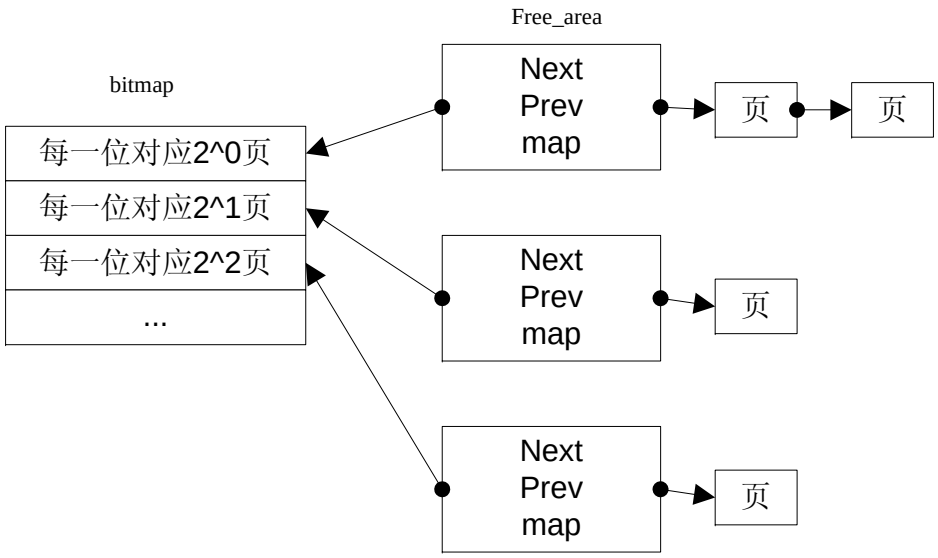
物理内存管理——物理页位图

- 系统中每个节点所有物理页面均可用包含**page**结构的链表**mem_map**描述，每个物理页面用**page**描述



物理内存管理——物理页位图(续)

- 物理页分配采用链表和位图结合的方法，内核定义**free_area**的数组，该数组的第**k**个元素描述了**2^k**页块的信息，每项包含两个元素：**list**和**map**，**list**是一个双向链表的头指针，该双向链表的每个节点包含空闲页块的起始物理页框编号；而**map**则是记录这种页块分配情况的位示图，如果位示图的第**N**位为**0**，则表明第**N**个页块是空闲的



- 物理内存是以伙伴关系机制进行管理的，无论内存中的占用块或是空闲块，其大小均是**2^k**，申请和释放的内存块大小也是**2^k**
- 空闲块分裂时，由同一大块分裂出来的小块互称伙伴，释放内存时先判断其伙伴是否空闲，若否则简单加入**free_area**,否则先在**free_area**中删除伙伴然后合并，重复以上步骤
- 算法简单速度快，缺点是只能以页为单位

基于区的伙伴系统：基本数据结构

- 页框
  - 页描述符**page**
  - 页描述符数组**mem_map**
- 内存区
  - 区描述符**zone_struct**
    - **zone_mem_map:**指定对应的**mem_map**中的第一个元素
    - **Size**：元素个数
- 节点
  - 节点描述符**pg_data_t**
  - 链表，首元素为**pgdat_list**
  - **80x86**下，只有一个节点，其描述符存在**contig_page_data**变量中

基于区的伙伴系统：伙伴系统(续)

- 伙伴系统使用的数据结构
  - 页描述符数组**mem_map**
  - **free_area**数组,第**k**项管理大小为2^k的块： **typedef struct free_area_struct**

    {

    struct list_head free_list;//指向大小为2^k的块的双向循环链表，每个元素指向该块的第一个页描述符

    unsigned long* map;//每一位描述大小为2^k个页框的两个伙伴块的状态

    }

Slab分配器：缘起

- 伙伴系统解决了外碎片的问题，当请求的内存小于一个页框时怎么办？
- 早期Linux的做法：
  - 提供大小为2,4,8,16,...,131056字节的内存区域
  - 需要新的内存区域时，内核从伙伴系统申请页面，把它们划分成一个个区域，取一个来满足需求
  - 如果某个页面中的内存区域都释放了，页面就交回到伙伴系统

Slab分配器：缘起（续）

- 在伙伴算法之上运行存储器区(memory area)分配算法没有显著的效率
  - 不同的数据类型用不同的方法分配内存可以提高效率。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使用同一类型的内存区，缓存最近释放的对象可以加快分配和释放
  - 对内存的请求可以按照请求频率来分类，频繁使用的类型使用专门的缓存，很少使用的可以使用类似2.0中的取整到2的幂次的通用缓存
  - 使用2的幂次大小的内存区域时高速缓存冲突的概率较大，可以通过仔细安排内存区域的起始地址来减少高速缓存冲突
  - 缓存一定数量的对象可以减少对buddy系统的调用，从而节省时间并减少由此引起的高速缓存“污染”
- Slab分配器的设计体现了以上的思想

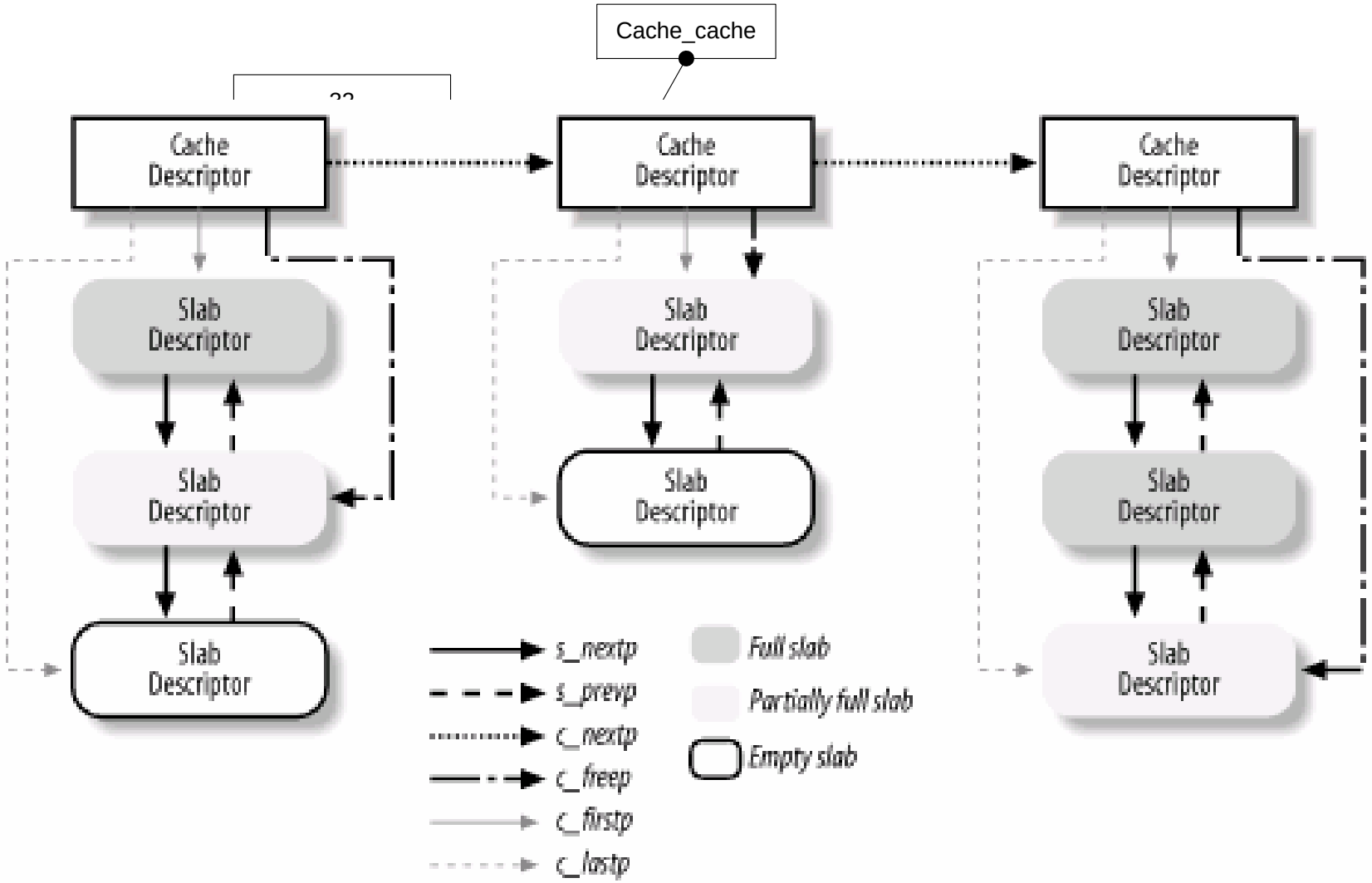
Slab分配器：基本组成

- Slab分配器把存储器区看作对象(object)
- Slab分配器把对象按照类型分组成不同的高速缓存
- 每个Slab由一个或多个连续的页框组成，这些页框中包含已分配的对象，也包含空闲的对象
- Slab分配器通过伙伴系统分配页框

Slab分配器：数据结构

- 高速缓存描述符struct kmem_cache_s
  - 链表结构
  - 包含三种slab双向循环链表
    - slabs_full
    - slabs_partial
    - slabs_free
- Slab描述符struct slab_s
  - 在高速缓存中根据类型不同分别存在不同的slab链表中
  - 可存放在slab外，也可存放在slab内

Slab分配器：数据结构（续）

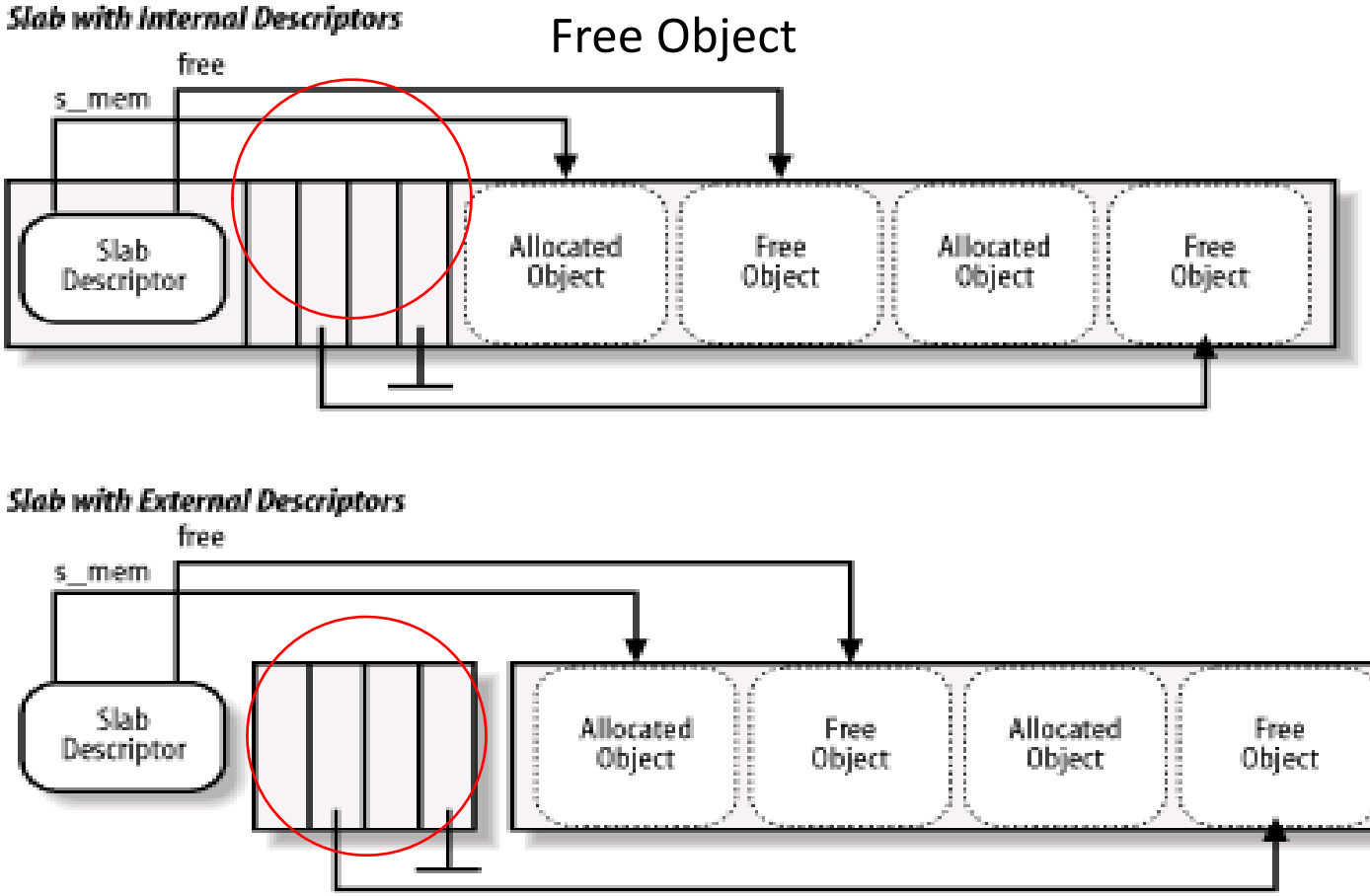


Slab分配器：数据结构（续）

- 对象描述符**kmem_bufctl_t**
  - 存放在数组中，位于相应的**slab**描述符之后
  - 只在对象空闲时有意义，包含的是下一个空闲对象在**slab**中的下标

Slab分配器：数据结构（续）

Slab Descriptor中的free数据域指向第一个Free Object



Slab分配器：与伙伴系统的接口

void * kmem_getpages(kmem_cache_t *cachep, unsigned long flags)

功能：为创建的slab分配一组空闲连续的页框

参数：

cachep指向需要额外页框的高速缓存描述符

flags说明如何请求页框

数据结构：

cachep->gfporder单独slab包含的连续页框数的对数

其它函数：

__get_free_pages(gft_mask, order)

类似于alloc_pages()，返回第一个所分配页的线性地址

Slab分配器：与伙伴系统的接口(续)

void kmem_freepages(kmem_cache_t *cachep, void *addr)

功能：释放分配给slab分配器的页框

参数：

cachep指向需要释放页框的高速缓存描述符

addr从addr开始释放页框

数据结构：

cachep->gfporder请求页框的个数

mem_map页描述符数组

其它函数：

free_pages(addr, order)：从线性地址addr起释放页框

PageClearSlab(page)：清除page中的PG_Slab标志，PG_Sl
示该页框包含在Slab中

- kmem_cache_create创建一个cache，分配合适的slab大小，在slab存储对象个数限制下，让slab尽量大，然后挂到slab中的链表下
- 涉及许多计算
  - 对齐计算
  - 根据对象大小决定是On slab方式还是off slab方式
  - 计算空间浪费，根据对象大小调整合适的slab大小
  - slab着色使得对象均匀分布在页面内，避免集中在页边界附近，导致cpu cache频繁失效（回忆cpu cache的映射机制常采用简单的地址映射）
  - 代码出现递归，在分配kmem_cache_t数据结构时，恰恰还是使用slab分配器分配kmem_cache_t对象，解决办法是第一个kmem_cache_create采用kmalloc分配对象

- **Shrink_dcache_memory**将释放缓冲区内全部空闲的**slab**块
- **Kmem_cache_reap**释放缓冲区内超过**80**％空闲的**slab**块

Slab分配器： slab的分配与释放(续)

- 分配时机
  - 已发出分配新对象的请求，并且
  - 高速缓存不包含任何空闲对象
- **Kmem_cache_grow()**
  - **kmem_getpages()为slab分得页框**
    - **kmem_cache_slabmgmt()获得新slab描述符**
  - **kmem_cache_init_objs()为新的slab中的对象申请构造方法**
  - 扫描所有页框描述符
    - 高速缓存描述符地址存页描述符中的**list->next**
    - **slab描述符地址存页描述符中的list->prev**
    - 设置页描述符的**PG_slab**标志
  - 将**slab**描述符加到全空**slab**链表中

Slab分配器： slab的分配与释放(续)

- 释放时机
  - 伙伴系统不能满足新请求的一组页框，并且
  - slab为空
- 内核查找其他空闲页框时，调用**try_to_free_pages()**
  - 调用**kmem_cache_reap()**选择至少包含一个空**slab**的高速缓存
  - 调用**kmem_slab_destroy()**从完全空闲的**slab**链表中删除**slab**并撤销它

Slab分配器： slab的分配与释放(续)

void kmem_slab_destroy(kmem_cache_t *cachep, slab_t *slabp)

数据结构：

slabp->s_mem:指向slab内第一个对象

cachep->num:挤进一个单独slab中的对象个数

cachep->objsize:高速缓存中包含对象的大小

其它函数：

cachep->dtor:对象的析构函数

kmem_freepages(kmem_cache_t *cachep, void *addr):释放分配给slab的页框

OFF_SLAB宏：判断slab描述符是否存放在slab的外面

kmem_cache_free (kmem_cache_t *cachep, void *objp):释放slab描述符

Slab分配器：高速缓存中的对象管理（续）

- **kmem_cache_alloc()为高速缓存分配对象**

kmem_cache_alloc()中的代码片段：

```
void * objp;
slab_t * slabp;
struct list_head * entry;
local_irq_save(save_flags);
entry = cachep->slabs_partial.next;
if (entry == & cachep->slabs_partial) {
    entry = cachep->slabs_free.next;
    if (entry == & cachep->slabs_free)
        goto alloc_new_slab;
    list_del(entry);
    list_add(entry, & cachep->slab_partials);
}
slabp = list_entry(entry, slab_t, list);
```

查找空闲对象的slab,若不存在，则分配一个新的slab

Slab分配器：高速缓存中的对象管理（续）

kmem_cache_alloc()中的代码片段（接上）：

```
slabp->inuse++;
            ...
objp = & slabp->s_mem[slabp->free * cachep->objsize];
            ...
slabp->free = ((kmem_bufctl_*)(slabp+1))[slabp->free];
if (slabp->free == BUFCTL_END) {
   list_del(&slabp->list);
   list_add(&slabp->list, &cachep->slabs_full);
}
            ...
local_irq_restore(save_flags);
return objp;
```

数据结构：

slabp->inuse:当前已分配的对象个数

slabp->s_mem:指向slab内第一个对象

cachep->objsize:高速缓存中包含对象的大小

slabp->free:指向slab内第一个空闲对象

对象描述符数组紧挨着slab描述符，表项指向下一个空闲对象

Slab分配器：高速缓存中的对象管理（续）

- **kmem_cache_free()**释放由slab分配器以前所获得的对象

kmem_cache_free()中的代码片段：

```
slab_t * slabp;
unsigned int objnr;
local_irq_save(save_flags);
slabp = (slab_t *) mem_map[__pa(objp) >> PAGE_SHIFT].list.prev;
            ...
objnr = (objp - slabp->s_mem) / cachep->objsize;
((kmem_bufctl_t *)(slabp+1))[objnr] = slabp->free;
slabp->free = objnr;
```

计算出slab描述符地址，在为slab分配页框时，描述符地址存于页框描述符的list.prev中

导出对象描述符，将对象追加到空闲对象链表的首部
数据结构：
slabp->free:指向slab内第一个空闲对象
对象描述符数组紧挨着slab描述符，表项指向下一个空闲对象

Slab分配器：
高速缓存中的对象管理（续）

kmem_cache_free()中的代码片段（接上）：

```
if (--slabp->inuse == 0) { /* slab is now fully free */
    list_del(&slabp->list);
    list_add(&slabp->list, &cachep->slabs_free);
} else if (slabp->inuse+1 == cachep->num) { /* slab was full */
    list_del(&slabp->list);
    list_add(&slabp->list, &cachep->slabs_partial);
}
local_irq_restore(save_flags);
```

最后检查slab是否需要移动另一个链表中

Slab分配器：
高速缓存中的通用对象管理

- 如果对存储区的请求不频繁，就用一组普通高速缓存来处理
  - **kmalloc()**：得到通用对象
  - **Kfree()**：释放通用对象

Slab分配器：
高速缓存中的通用对象管理（续）

void * kmalloc(size_t size, int flags)

数据结构：

```
cache_sizes_t cache_size[26];          //26个几何分布的高速缓存
typedef struct cache_sizes {
        size_t          cs_size;
        kmem_cache_t    *cs_cachep;//用于DMA分配
        kmem_cache_t    *cs_dmacachep;//用于常规分配
} cache_sizes_t;
```

其它函数：

__kmem_cache_alloc (kmem_cache_t *cachep, int flags)：在高速缓存中分配对象

Slab分配器：
高速缓存中的通用对象管理（续）

void kfree(const void *objp)

功能：释放通用对象

数据结构：

mem_map页描述符数组

在为slab分配页框时，高速缓存描述符地址存于页框描述符的list.next中

其它函数：

__kmem_cache_free (kmem_cache_t *cachep, void* objp)：释放高速缓存中的对象

Q&A

附录

- 讲义中提到的函数细节

80386段页式管理机制

- **Linux**内存管理的硬件基础
- **Linux**内存管理的概念基础
- 通过**386**的硬件机制，区分内存管理中硬件与软件的分工

硬件分段(1)

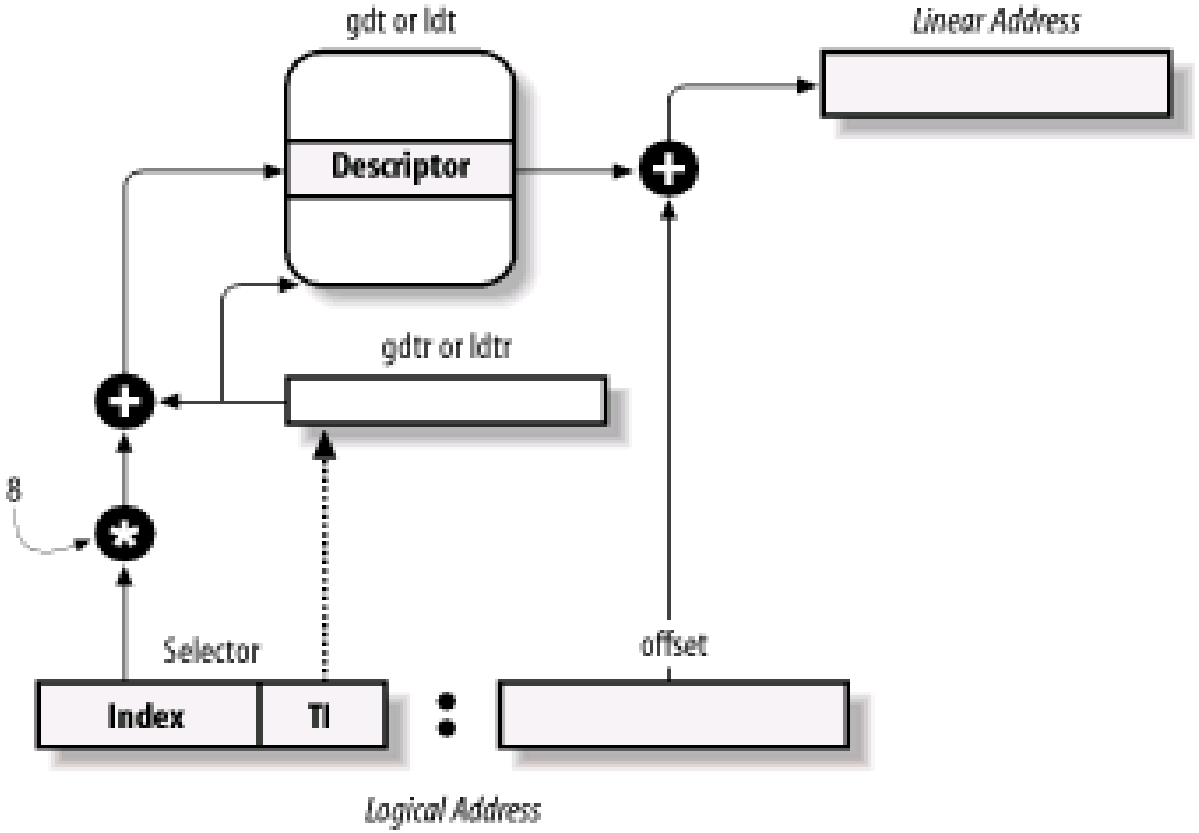
- 为支持保护模式，段改由段描述符来描述
  - 段基址
  - 段限长
  - 类型
  - 访问该段所需的最小特权级
  - …
- 段描述符存放在**GDT**或**LDT**中（全局/局部描述符表）中
- **GDT/LDG**的基址存放在寄存器**GDTR/LDTR**中

硬件分段(2)

- 段寄存器
  - **CS：代码段寄存器**
  - **DS：数据段寄存器**
  - **SS：堆栈段寄存器**
  - **ES、FS、GS…**
- 每个段寄存器存放段选择符
  - **GDT/LDT的索引**
  - **TI位：指定描述符是GDT还是LDT**
  - **RPL：CPU的当前特权级**

硬件分段(3)

- 逻辑地址到线性地址的转换

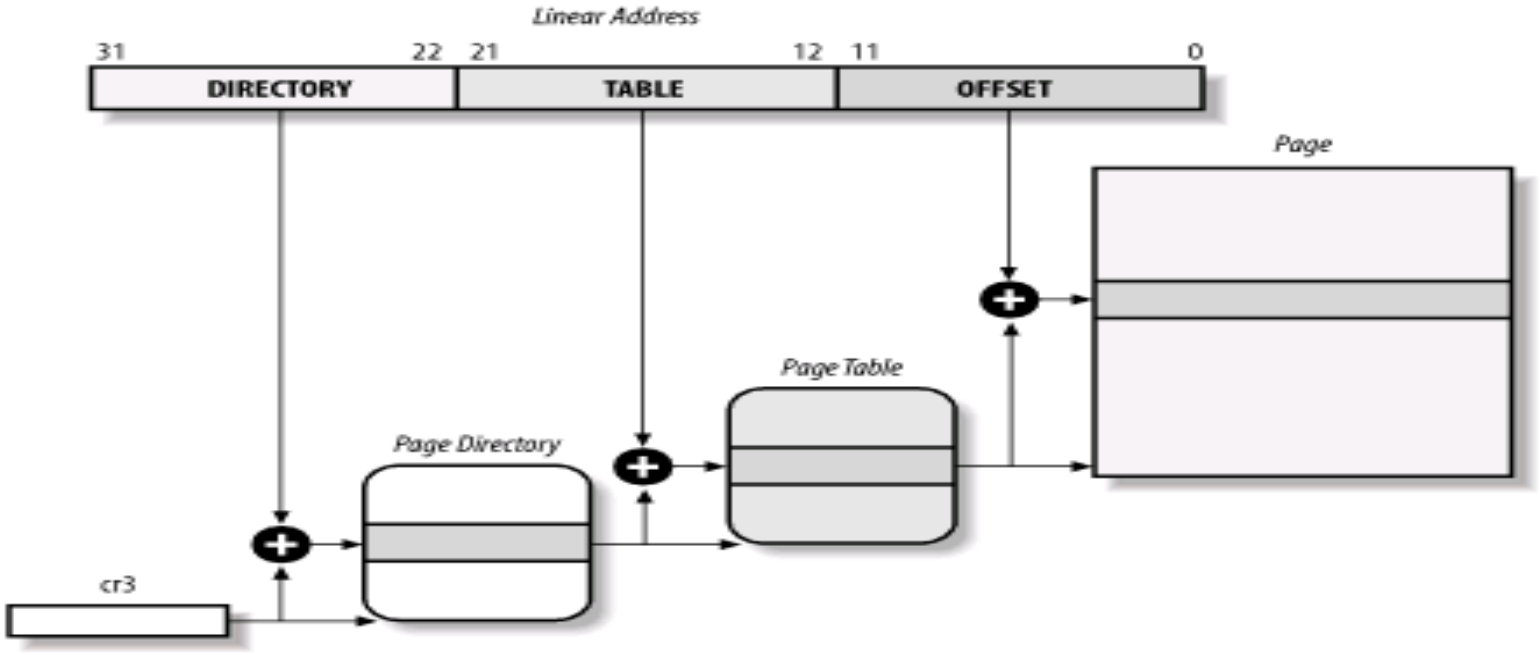


硬件分页(4)

- 页 vs. 页框
  - 页：数据块，可以在主存也可以在磁盘
  - 页框：固定长度的RAM块
- 分页
  - Intel CPU处理4KB的页(why 4KB?)
  - 通过设置CR3的PG位开启分页
- 数据结构
  - 页目录
    - 当前使用的页目录的物理地址在CR3中
  - 页表

硬件分页(5)

- 硬件的分页过程
  - 发生在物理地址送地址线之前
  - **MMU**中进行



硬件分页(6)

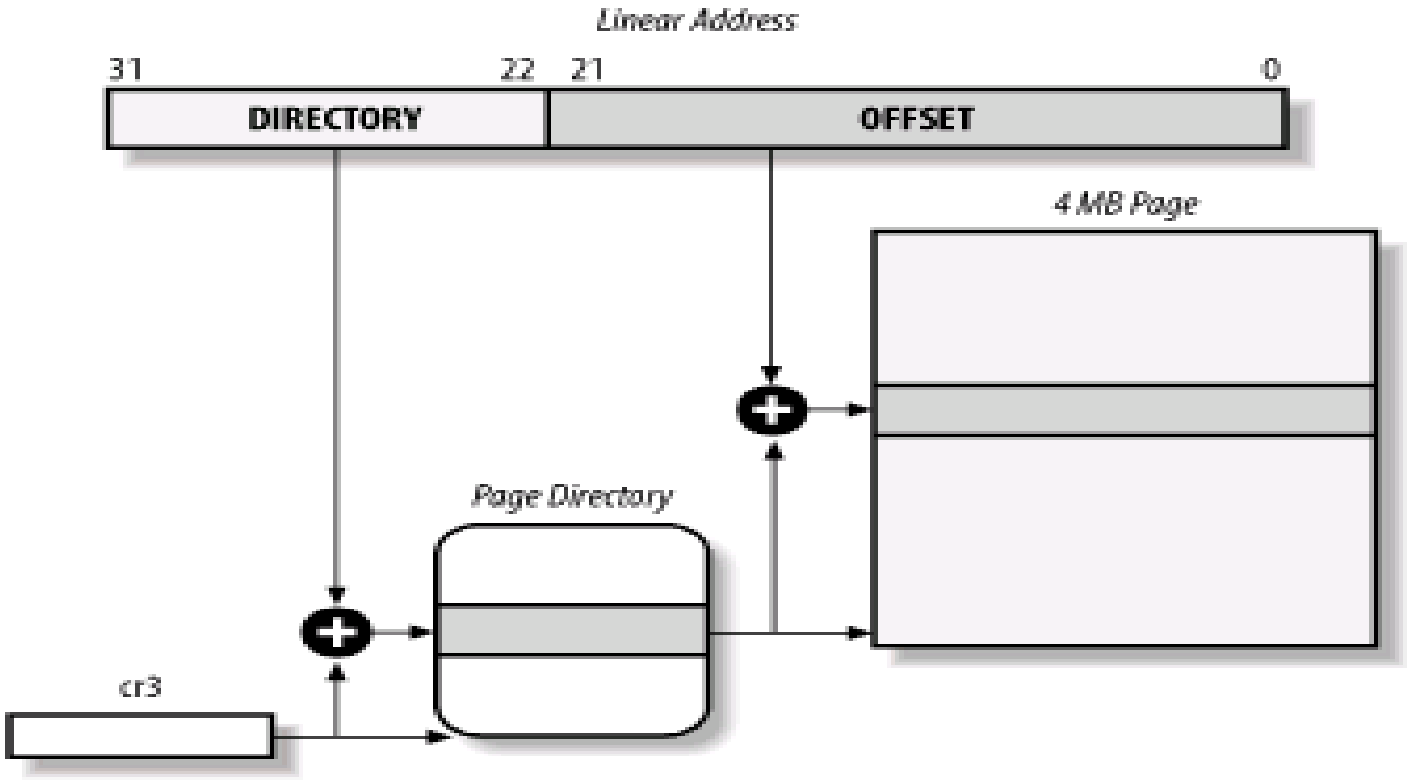
- **TLB(Translation Lookaside Buffer)**
  - 硬件，用来加速页表查找
  - 关键的一点：如果操作系统更改了页表内容，它必须相应的刷新**TLB**以使**CPU**不误用过时的表项
  - **CR3**发生改变时，硬件自动更新**TLB**中所有的表项

硬件保护机制

- 段描述符
  - **type标志：读/写/执行**
- 页表项
  - **User/Supervisor标志**
  - **Read/Write标志**
- 硬件产生**page fault**

扩展分页(1)

- 允许页框为4MB



扩展分页——物理地址扩展分页（PAE）

- 内核不能直接对1G以上的RAM进行寻址
- 市场需求刺激下，Intel进行了修补：
  - 在Pentiume Pro体系结构，地址线36位，但仍必须把32位线性地址转换为36位
  - 办法：页表项长度改为64位（因为32位已不够用），再增加一级页表，该页表4个表项，基址存在CR3中

地址类型(1)

- 用户虚拟地址
  - 用户空间的程序使用的地址
  - 根据硬件体系结构，可以是32位或64位
  - 每个进程拥有自己独立的虚拟地址空间

地址类型(2)

- 物理地址
  - 物理内存地址
  - 处理器和系统内存之间使用
  - 根据硬件体系结构，可以是32位或64位
- 总线地址
  - 总线寄存器地址
  - 外设总线和内存之间使用

地址类型(3)

- 内核逻辑地址
  - 常规的内核地址空间
  - 映射了大部分主存，可以当作物理内存使用
  - 内核逻辑地址与相应的物理地址相差一个常数偏移量
  - 逻辑地址使用硬件特有的指针大小，在配置大量内存的32位系统上可能无法直接访问所有的物理内存
  - 可将内核逻辑地址与物理地址相互转换，定义在<asm/page.h>中

地址类型(4)

- 内核虚拟地址
  - 由函数**vmalloc**和**kmap**返回的地址
  - 不能直接对应物理内存
  - 需要内存分配和地址转换才能与逻辑地址联系起来
  - 通常保存在指针变量中

地址类型(5)

- 低端内存与高端内存
  - 均指物理内存
  - 低端内存代表存在于内核空间的与逻辑地址相应的物理内存
  - 高端内存是那些不存在逻辑地址的内存，通常保留给用户空间的进程使用
  - 在i386体系结构下，低端和高端之间的界限通常为1GB
  - 用户进程可访问4GB虚拟线性空间，0到3GB为用户空间，3GB到4GB为内核空间
  - 所有进程内核空间均相同，具有相同页目录和页表
  - 内核虚拟空间3GB到3GB+8MB映射到物理内存0到8MB，对应内核启动和设备内存映射等

内存保护(1)

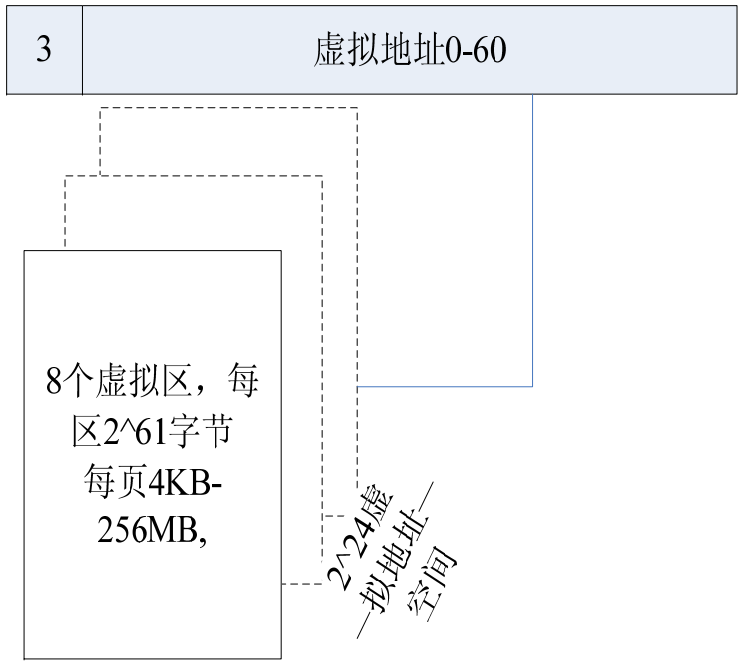
- 不同任务之间的保护
  - **MMU**实现任务虚拟地址空间的隔离
  - 每个任务拥有不同的虚拟地址到物理地址的映射
  - 操作系统的保护通过任务共享地址空间实现
  - 地址空间分为全局地址空间和局部地址空间

内存保护(2)

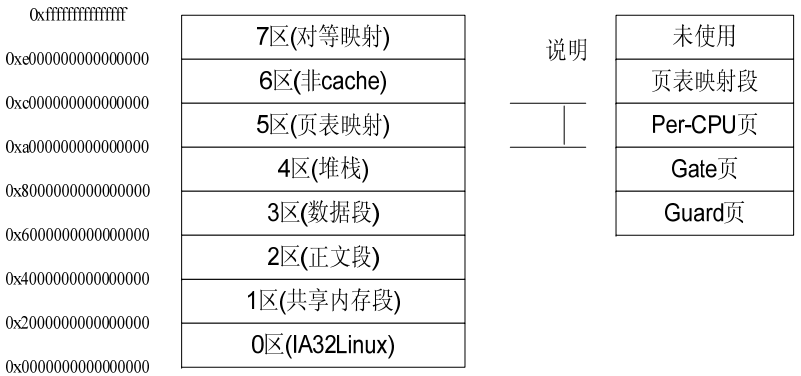
- 同一任务的保护
  - 定义4种特权级别，限制内存访问
  - 0级是操作系统内核，处理IO，内存管理及其他关键操作
  - 1级系统调用处理程序
  - 2级库过程
  - 3级用户程序
  - Linux只用0级和3级，1级和2级归到0级中

IA-64Linux地址空间划分

- 64位虚地址空间划分为8个相等的区，高3位表示区号
- 0-4区作为用户空间，5-7区作为核心空间
- 内核空间可进一步划分为页表映射段和对等映射段
- 页表映射段用于实现内核的vmalloc区域，实现虚地址连续的大内存块分配
- 对等映射段包含Linux内核，该段虚拟地址可与物理地址直接映射，虚地址减去一个基地址得到物理地址



- **IA-64中使用0-4区作为用户空间，5-7区作为核心空间**



区号	使用	页大小	范围	映射方式
7	Cache	256MB	全局	对等
6	Cache	256MB	全局	对等
5	Vmalloc,，guard, gate	8KB	全局	页表
4	堆栈	8KB	进程	页表
3	数据段	8KB	进程	页表
2	正文段	8KB	进程	页表
1	共享内存	8KB	进程	页表
0	IA-32模拟	8KB	进程	页表

查看进程内存布局

- 进程内存布局实例

```
[root@serv3 ~]# cat /proc/1/maps
起始          结束          权限 偏移       主从设备号  inode号   文件映像
001be000-001bf000 r-xp 001be000 00:00 0          [vdso]
001e7000-00201000 r-xp 00000000 03:01 4471690    /lib/ld-2.3.6.so
00201000-00202000 r-xp 00019000 03:01 4471690    /lib/ld-2.3.6.so
00202000-00203000 rwxp 0001a000 03:01 4471690    /lib/ld-2.3.6.so
00205000-00328000 r-xp 00000000 03:01 4471692    /lib/libc-2.3.6.so
00328000-0032a000 r-xp 00122000 03:01 4471692    /lib/libc-2.3.6.so
0032a000-0032c000 rwxp 00124000 03:01 4471692    /lib/libc-2.3.6.so
0032c000-0032e000 rwxp 0032c000 00:00 0
00330000-00343000 r-xp 00000000 03:01 4471803    /lib/libsepol.so.1
00343000-00344000 rwxp 00013000 03:01 4471803    /lib/libsepol.so.1
00344000-0034c000 rwxp 00344000 00:00 0
069d5000-069e5000 r-xp 00000000 03:01 4471735    /lib/libselinux.so.1
069e5000-069e6000 rwxp 00010000 03:01 4471735    /lib/libselinux.so.1
08048000-0804f000 r-xp 00000000 03:01 4798259    /sbin/init
0804f000-08050000 rw-p 00007000 03:01 4798259    /sbin/init
08b33000-08b54000 rw-p 08b33000 00:00 0          [heap]
b7f10000-b7f11000 rw-p b7f10000 00:00 0
b7f1d000-b7f1e000 rw-p b7f1d000 00:00 0
bfd09000-bfd1e000 rw-p bfd09000 00:00 0          [stack]
```

```
void *__vmalloc(unsigned long size, int gfp_mask, pgprot_t prot)
{
	struct vm_struct *area;
	struct page **pages;
	unsigned int nr_pages, array_size, i;
 //将要分配的内存按页对齐
	size = PAGE_ALIGN(size);
//分配的空间大于所有物理页面数
	if (!size || (size >> PAGE_SHIFT) > num_physpages)
		return NULL;
//分配vma
	area = get_vm_area(size, VM_ALLOC);
	if (!area)
		return NULL;

	nr_pages = size >> PAGE_SHIFT;
	array_size = (nr_pages * sizeof(struct page *));

	area->nr_pages = nr_pages;
```

fastcall void do_page_fault(struct pt_regs *regs, unsigned long error_code)

- **Handle_mm_fault**函数在申请页表项后，调用函数**handle_pte_fault**处理缺页错误

int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct * vma, unsigned long address, int write_access)

- 对有效的虚拟地址，如果是“缺页”错误的话，**Linux**必须区分页所在的位置，判断在交换文件中还是在可执行映象中，通过页表项中的信息区分页所在的位置，如果页表项无效但是非空，则在交换文件中，否则在可执行映象中
- **Handle_pte_fault**函数处理不同的缺页情况

static inline int handle_pte_fault(structmm_struct *mm,struct vm_area_struct * vma, unsigned long address,int write_access, pte_t *pte, pmd_t *pmd)

- 注意最后必须更新**TLB**，否则会出现逻辑错误

```
 //分配page数据结构内存空间
area->pages = pages = kmalloc(array_size, (gfp_mask & ~__GFP_HIGHMEM));
//可能内存不够
      if (!area->pages) {
            remove_vm_area(area->addr);
            kfree(area);
            return NULL;
      }
//初始化page数据结构
      memset(area->pages, 0, array_size);
//分配物理页
      for (i = 0; i < area->nr_pages; i++) {
            area->pages[i] = alloc_page(gfp_mask);
            if (unlikely(!area->pages[i])) {
                  /* 成功分配i页，在__vunmap()中释放 */
                  area->nr_pages = i;
                  goto fail;
            }
      }
//为申请到的内存更改页目录和页表
      if (map_vm_area(area, prot, &pages))
            goto fail;
      return area->addr;

fail:
      vfree(area->addr);
      return NULL;
}
```

内存区

- 受体系结构的制约，不同的存储器页有不同的使用方式，根据内存的不同使用类型划分为内存区**(Memory Zone)**
- **Linux**中的内存区（**Memory Zone)**
  - **ZONE_DMA:**低于**16MB**的存储器页
  - **ZONE_NORMAL:16MB-896MB**的存储器页
  - **ZONE_HIGHMEM:**高于**896MB**的存储器页

内存管理相关高速缓存(1)

- 缓冲区高速缓存
  - 包含块设备使用的数据缓冲区
  - 由设备标志号和块标号索引
  - 如果在缓冲区高速缓存中，则不必进行块设备读取操作

内存管理相关高速缓存(2)

- 页高速缓存
  - 加速磁盘上的映像和数据访问，缓存文件逻辑内容
  - 由文件inode和偏移量索引
  - 当页从磁盘读入物理内存时，就缓存在页高速缓存中，可以提高文件访问速度

内存管理相关高速缓存(3)

- 交换高速缓存
  - 修改后的脏页才写入交换文件中，利用交换高速缓存可以减少磁盘写操作
  - 交换出的页的页表项包含交换文件信息，以及该页在交换文件中的位置信息
  - 如果页面没有被修改则页表项非零

内存管理相关高速缓存(4)

- 硬件高速缓存
  - 对页表项缓存，由处理器完成
  - 减少地址转换的时间
  - 需要由操作系统管理

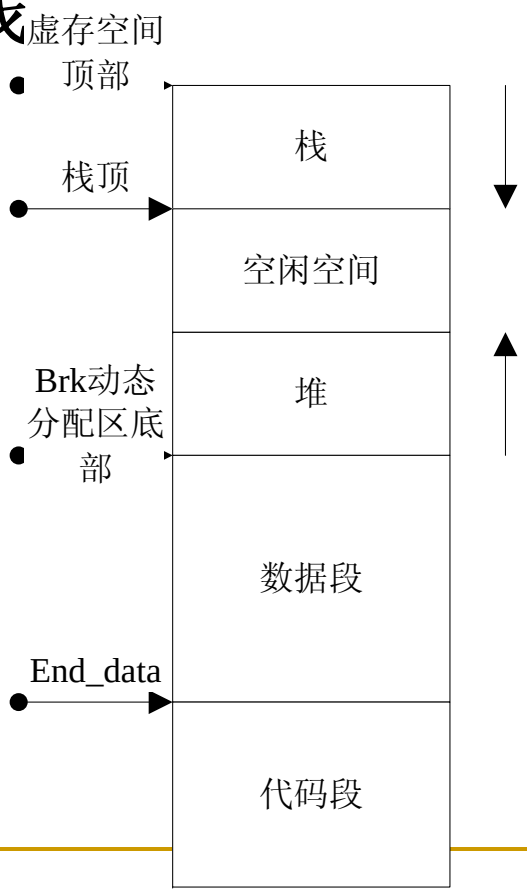
交换空间

- **Linux**采用两种方式保存换出页面：交换设备，使用整个设备，如硬盘分区；交换文件，使用文件系统中固定长度的文件。统称交换空间
- 交换空间的内部格式相同，前**4096**字节是一个以字符串“**SWAP-SPACE**”结尾的位图，位图的每一位对应一个交换空间的页面，置位表示对应的页面可用于换页操作，**4096**字节之后才是真正存放页面的空间
- 交换设备比交换文件更有效率，因为交换设备中属于同一页面的数据总是连续存放，而交换文件中数据块位置可能是零散的

守护进程kswapd

- 内存不足时，linux通过kswapd释放部分物理内存页
- Kswapd是内核线程，在内核初始化时启动，周期性的运行
- Kswapd按照下面三种方法减少系统使用的物理页
  - 减少缓冲区和页缓存的大小
    - 页缓存包含内存映射文件的页，其中包含一些系统不在需要的页面
  - 将system V共享内存页交换出物理内存
  - 将页交换出物理内存或丢弃，kswapd首先选择可交换的进程，或者其中某些页面可以交换出的进程。可执行映象的大部分内容可从磁盘获取，因此可以丢弃。Linux采用LRU算法将进程的一小部分页面换出，被锁定的页面不能被换出

- 代码数据和堆栈



Sys_brk系统调用

asmlinkage unsigned long sys_brk(unsigned long brk)

- 需要检查进程的资源限制，以及参数是否合法

unsigned long
do_brk(unsigned long addr, unsigned long len)

- 检查参数有效性
- 如果内存区域被锁定，那么分配的内存不能超过锁定区域，并且分配的内存也必须被锁定

unsigned long
do_brk(unsigned long addr, unsigned long len)

- 清除旧映射，如果必要创建新的**vma**结构
- 匿名映射是指进程运行过程中产生的内存区域，如堆空间，堆栈空间，反之命名区域指可执行文件中的相应区域
- 为新增区间建立全部映射，**make_pages_present**

- **Do_mmap_pgoff**函数完成具体的映射工作，如果**file**不为空，将调用**file_operations**中的映射函数，以后发生缺页中断时将通过**file_operations**中的相应操作读取页面并完成映射

文件预读

- 文件预读机制是一种流水线式运行的数据读取机制，在应用程序从内存页读取数据时，IO从硬盘中读取数据到内存页
- 预读代码管理两个窗口——“当前窗口”和“前一个窗口”，当前窗口读完时，用前一个窗口代替当前窗口，并且前一个窗口失效，当IO完成时，提交新的IO批次，创建一个新的前一个窗口
- 如果读请求的一页正好是随后的下一页，则发生预读命中，如果预读发生时，窗口会增大，如果发生预读缺失则窗口减小，发生足够多的预读缺失，那么预读将完全被禁止

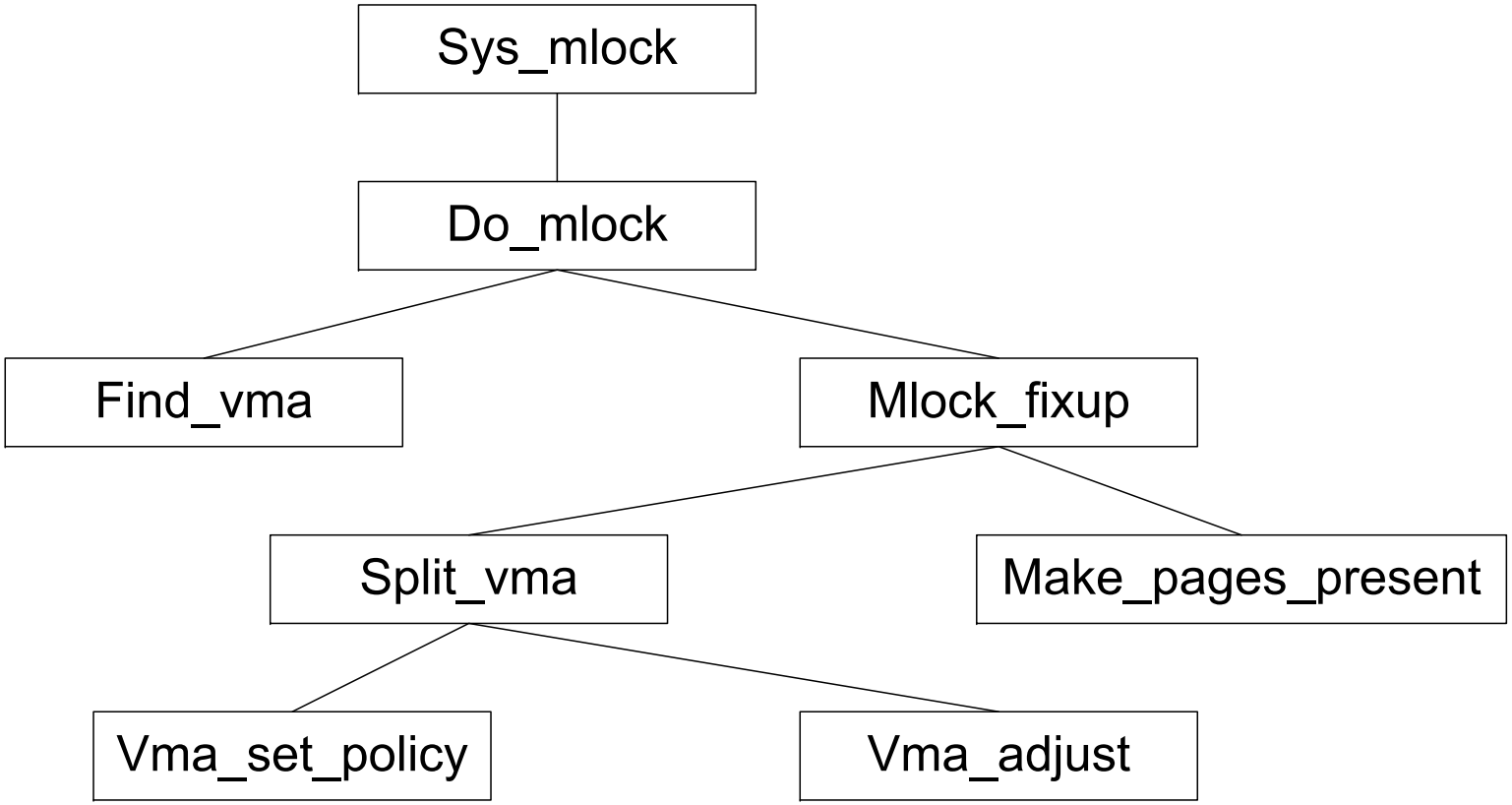
虚拟内存加锁和保护

- **Linux**可以对虚拟内存中的任何一段加锁和保护
  - 实质是对**vma**段的**vm_flags**或上属性**VM_LOCKED**
  - 加锁后，虚存对应的物理页面驻留内存，不再被换出
  - 调用**mlock**的进程终止或者调用**exec**执行其他程序，被锁页面才被释放

虚拟内存加锁和保护

- 实现中的问题
  - 对虚拟地址空间加锁及保护时，如果给定地址及长度与**vma**大小不一致，需要重新分割**vma**，使各部分属性一致
  - 通过地址找到**vma**，将**vma**终点与长度进行比较，决定**vma**的分割，然后设置分割后的**vma**的标志，并加入**vma**链表中

sys_mlock调用关系



- **Kfree**用来释放对象，实际上释放的对象空间并没有立即返回给系统，而是链接到缓冲区空闲对象表，真正的释放对象空间是由守护进程**kswapd**发现内存紧张时，将选中的某个缓冲区进行缩减，此时通过释放整个空闲**slab**块的空间将没有使用的页面交还系统