第三次实例分析

第三部分：页表项的建立和换页

胡康智
胡呈睿

1. mappages()函数中PGROUNDDOWN()函数的作用是什么？为何需要调用该函数?

```
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

PGSIZE = 4096（10） = 1000000000000（2） = 1000（16）
~(PGSIZE-1) = fffff000（16）

作用：将a低12位置0
调用原因：页表一共4K（即2的12次方），这里不需要考虑页表偏移量

2. mappages()函数中“*pte = pa | perm | PTE_P;”代码的含义是什么?

- PTE_P = 0x001 // Present 存在
- PTE_W = 0x002 // Writeable 可写
- PTE_U = 0x004 // User 可用

- pa 物理地址
- perm 标志位（通常用来写入一些标志，比如是否可写）

- *pte = pa | perm | PTE_P;
- 含义：pa为传入的物理地址，perm为标志位，用来写入一些标志，比如是否可写，PTE_P用来表示pde地址指向的内容存在

3. walkpgdir()函数中“*pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;”代码的含义是什么?

- *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;

- 含义：V2P(pgtab)为将pgtab转化为物理地址后的值
- PTE_P | PTE_W | PTE_U表示其存在/可写/可用

4. virtual address与physical address的映射关系是什么？对虚拟内存是如何实现的?

- V2P(a) (((uint) (a)) - KERNBASE)
- 虚拟地址转物理地址：地址 - 0x80000000

- P2V(a) ((void *)(((char *) (a)) + KERNBASE))
- 物理地址转虚拟地址：地址 + 0x80000000

mappages:

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

mappages:将每个页表的物理地址和其部分标志存入一个pte

- 传入参数：pde_t（uint）型指针*pgdir、void指针*va、uint型变量size和pa、int型变量perm

- a = (char*)PGROUNDDOWN((uint)va); //a为页表起始地址
- last = (char*)PGROUNDDOWN(((uint)va) + size - 1); //last为页表尾地址，size为页表大小

- if((pte = walkpgdir(pgdir, a, 1)) == 0) //即在walkpgdir中((pgtab = (pte_t*)kalloc()) == 0)，
  - return -1； //或者说申请页表地址失败，则返回-1

- if(*pte & PTE_P) //若pte地址指向的内容存在
  - panic("remap"); //则输出remap并冻结CPU（自旋锁）

- *pte = pa | perm | PTE_P; //将每个页表的物理地址和其标志存入一个pte

- if(a == last)　　//遍历结束
- 　　break;

- a += PGSIZE;　　//a每循环+PGSIZE(4096)，即a前20位的末位+1
- pa += PGSIZE;　　//同上

walkpgdir:

```
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

walkpgdir:根据传入的虚拟地址找到对应的pte

- 传入参数：uint（pde_t）型指针*pgdir、void常量指针*va、int型变量alloc

- pde = &pgdir[PDX(va)]; //pgdir的[PDX(va)]位的地址（取va高10位）

- 若(*pde & PTE_P) //若pde地址指向的内容存在
- 则pgtab = (pte_t*)P2V(PTE_ADDR(*pde)); //将pde的高20位地址转化为虚拟地址赋给pgtab

- 否则
- 若(!alloc || (pgtab = (pte_t*)kalloc()) == 0) //若传入参数alloc为0或者申请页表地址失败则返回0
- 则返回0
- memset(pgtab, 0, PGSIZE); //将从pgtab地址开始的PGSIZE（4096）位全部置0
- *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U; //将pgtab转化为物理地址并将其后三位置1赋给pde地址指向的内容

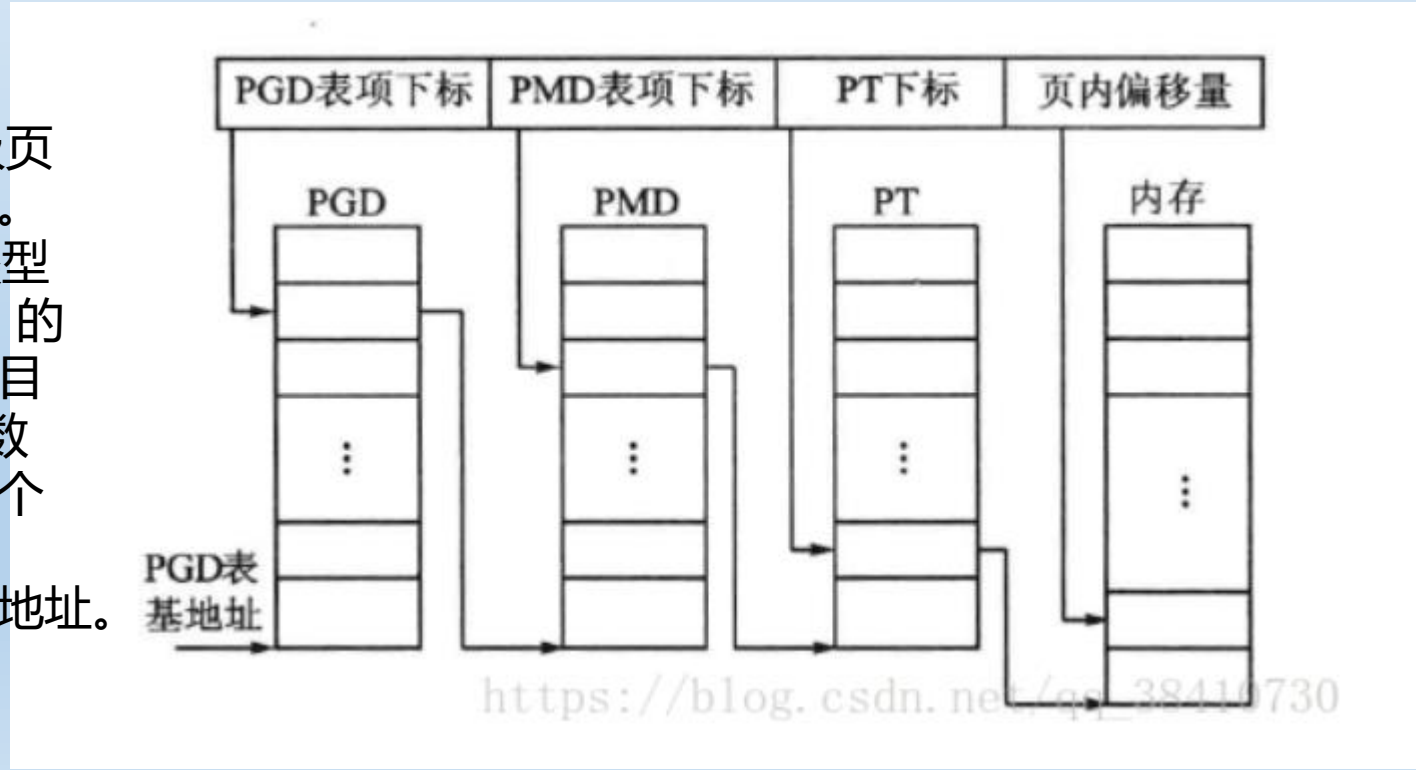
- 返回&pgtab[PTX(va)] //将pgtab的[PTX(va)]位的地址返回（取va第22位到13位）

5. linux代码中virtual address与physical address是如何映射的？如何实现了虚拟内存?

- 参考资料：linux内核虚拟内存之物理内存
  https://blog.csdn.net/heliangbin87/article/details/77989953
  Linux的虚拟内存详解（MMU、页表结构）
  https://blog.csdn.net/qq_38410730/article/details/81036768

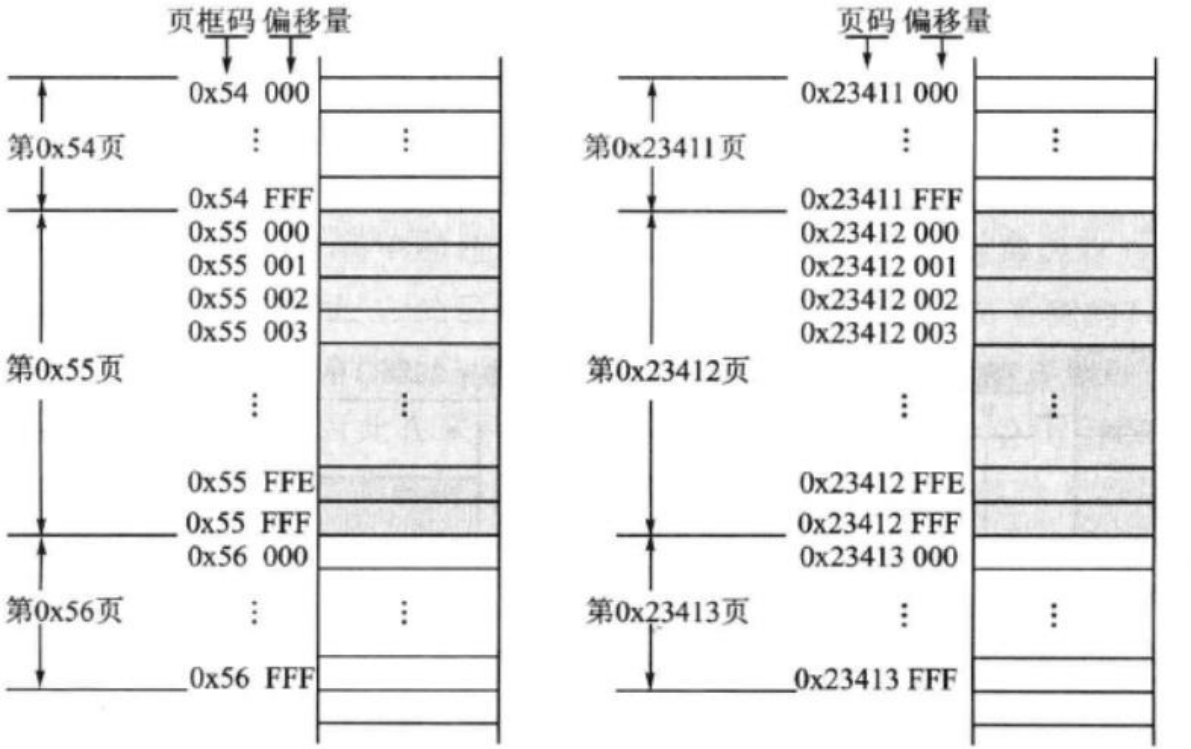
Linux页表结构

为了通用，Linux系统使用了三级页表结构：页目录、中间页目录和页表。PGD为顶级页表，是一个pgd_t数据类型（定义在文件linux/include/page.h中）的数组，每个数组元素指向一个中间页目录；PMD为二级页表，是一个pmd_t数据结构的数组，每个数组元素指向一个页表；PTE则是页表，是一个pte_t数据类型的数组，每个元素中含有物理地址。



虚拟内存的页、物理内存的页框及页表

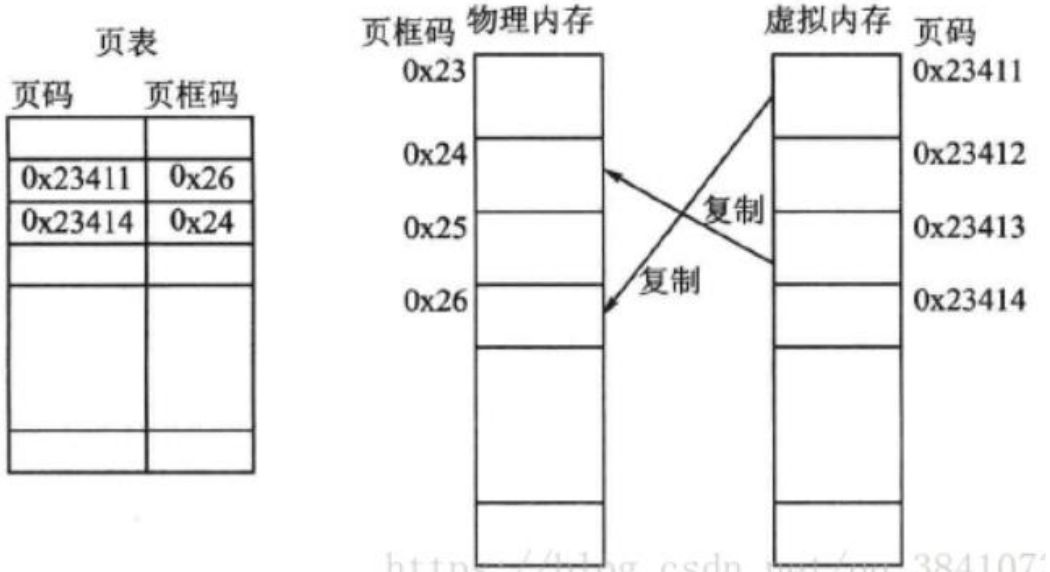
两段虚拟内存和物理内存分页之后的情况

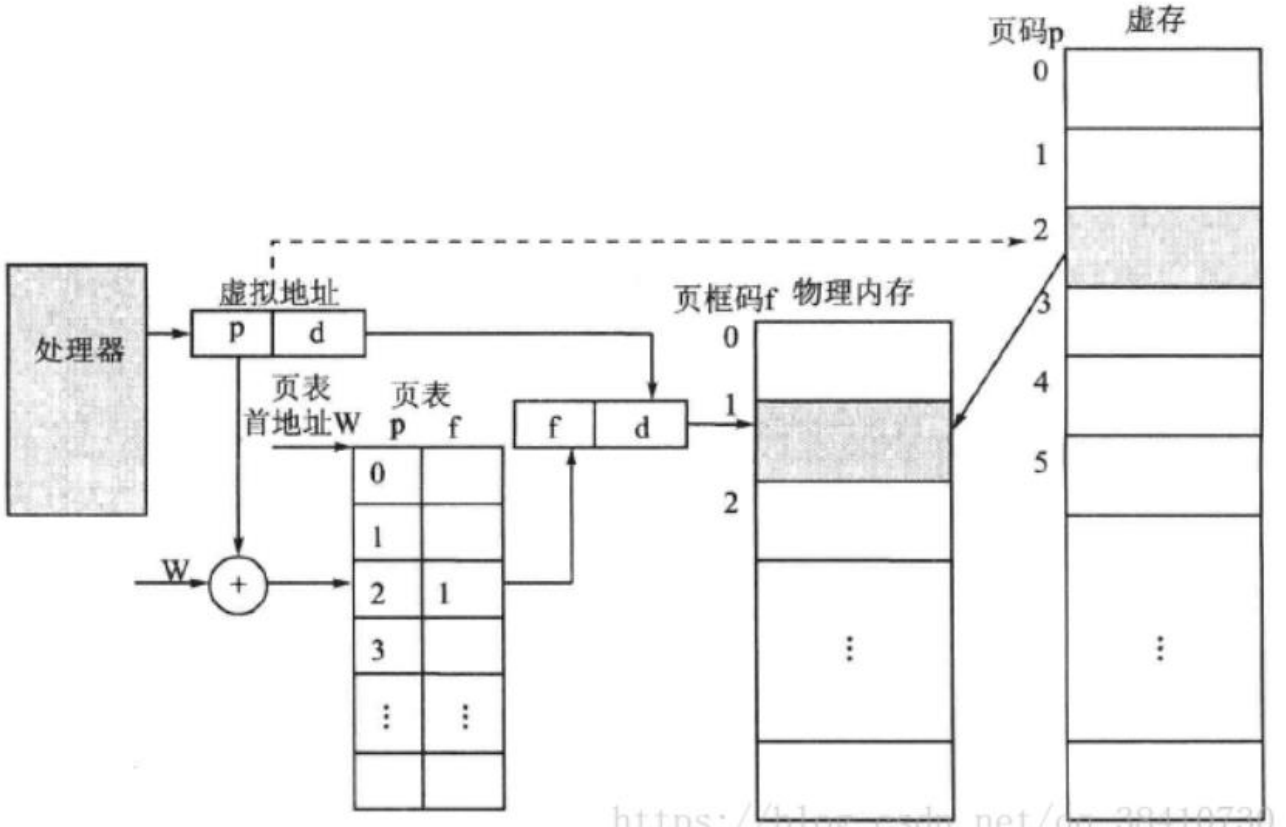


物理内存和虚拟内存在被分成也和页框之后，其地址也被分成了两段：高位段分别叫做页框码和页码，它们是识别页框和页的编码；低位段分别叫做页框偏移量和页内偏移量，它们是存储单元在页框和页内的地址编码。

映射关系

页模式下，虚拟地址、物理地址转换关系的示意图如下所示：





映射关系

- 当处理器试图访问一个虚存页面时，首先到页表中去查询该页是否已映射到物理页框中，并记录在页表中。如果在，则MMU会把页码转换成页框码，并加上虚拟地址提供的页内偏移量形成物理地址后去访问物理内存；如果不在，则意味着该虚存页面还没有被载入内存，这时MMU就会通知操作系统：发生了一个页面访问错误（页面错误），接下来系统会启动所谓的“请页”机制，即调用相应的系统操作函数，判断该虚拟地址是否为有效地址。

- 如果是有效的地址，就从虚拟内存中将该地址指向的页面读入到内存中的一个空闲页框中，并在页表中添加上相对应的表项，最后处理器将从发生页面错误的地方重新开始运行；如果是无效的地址，则表明进程在试图访问一个不存在的虚拟地址，此时操作系统将终止此次访问。

伙伴系统

- 在实际应用中，经常需要分配一组连续的页框，而频繁地申请和释放不同大小的连续页框，必然导致在已分配页框的内存块中分散了许多小块的空闲页框。这样，即使这些页框是空闲的，其他需要分配连续页框的应用也很难得到满足。
- 为了避免出现这种情况，Linux内核中引入了伙伴系统算法(buddy system)。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续页框的页框块。最大可以申请1024个连续页框，对应4MB大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍。