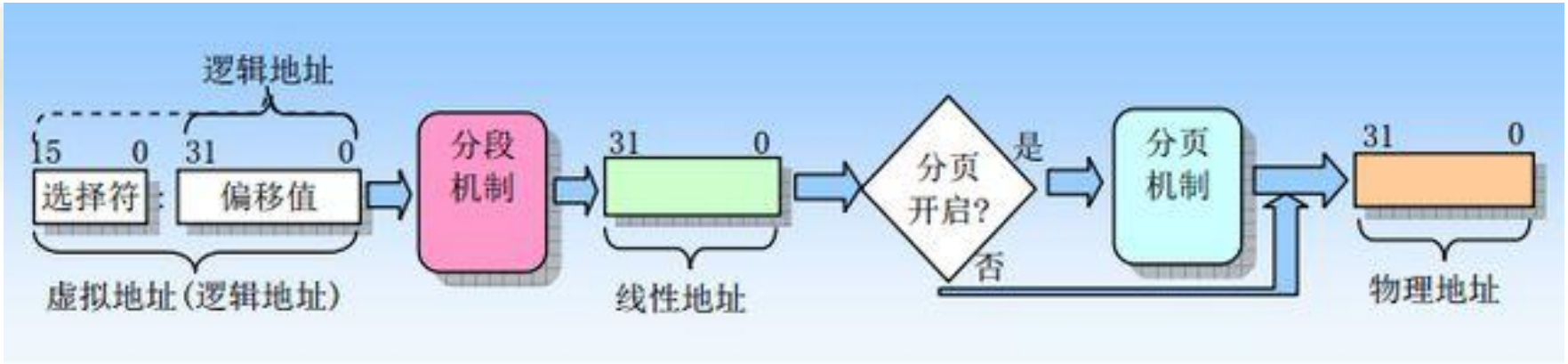
2.3 分页机制

西安邮电大学

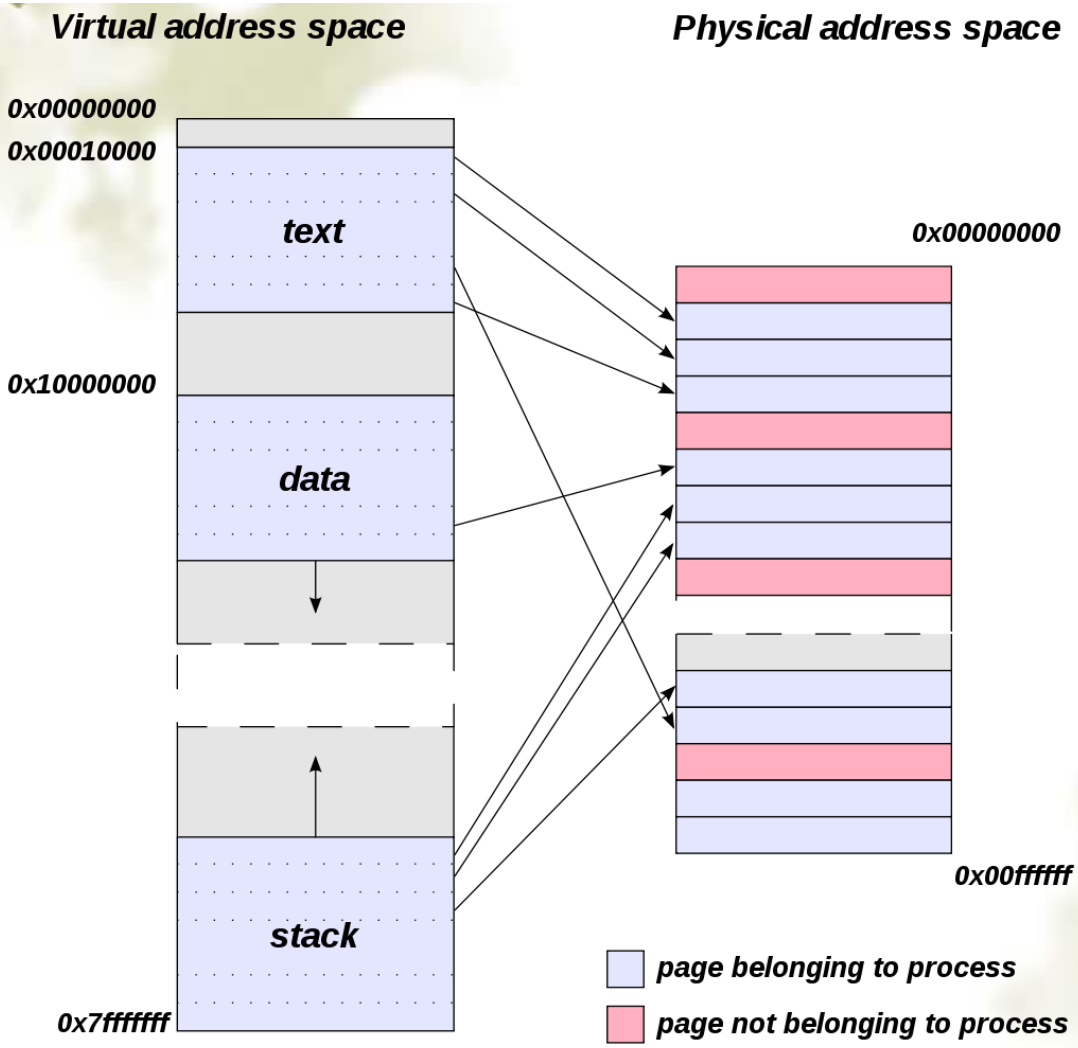
分页机制的引入



我们知道，分页在分段之后进行，以完成线性——物理地址的转换过程。段机制把虚拟地址转换为线性地址，分页机制进一步把该线性地址再转换为物理地址。

X86规定，分页机制是可选的，但这是硬件厂家的一厢情，很多操作系统设计者主要采用分页机制。

分页机制中的页



★ 所谓分页，就是将虚拟地址空间或线性地址空间划分成若干大小相等的片，称为页（Page）

★ 对物理地址空间分成与页大小相等的存储块，称为块或页面（Page Frame）

★ 页的大小为多少？

IA-32：4KB，2MB 和4MB.

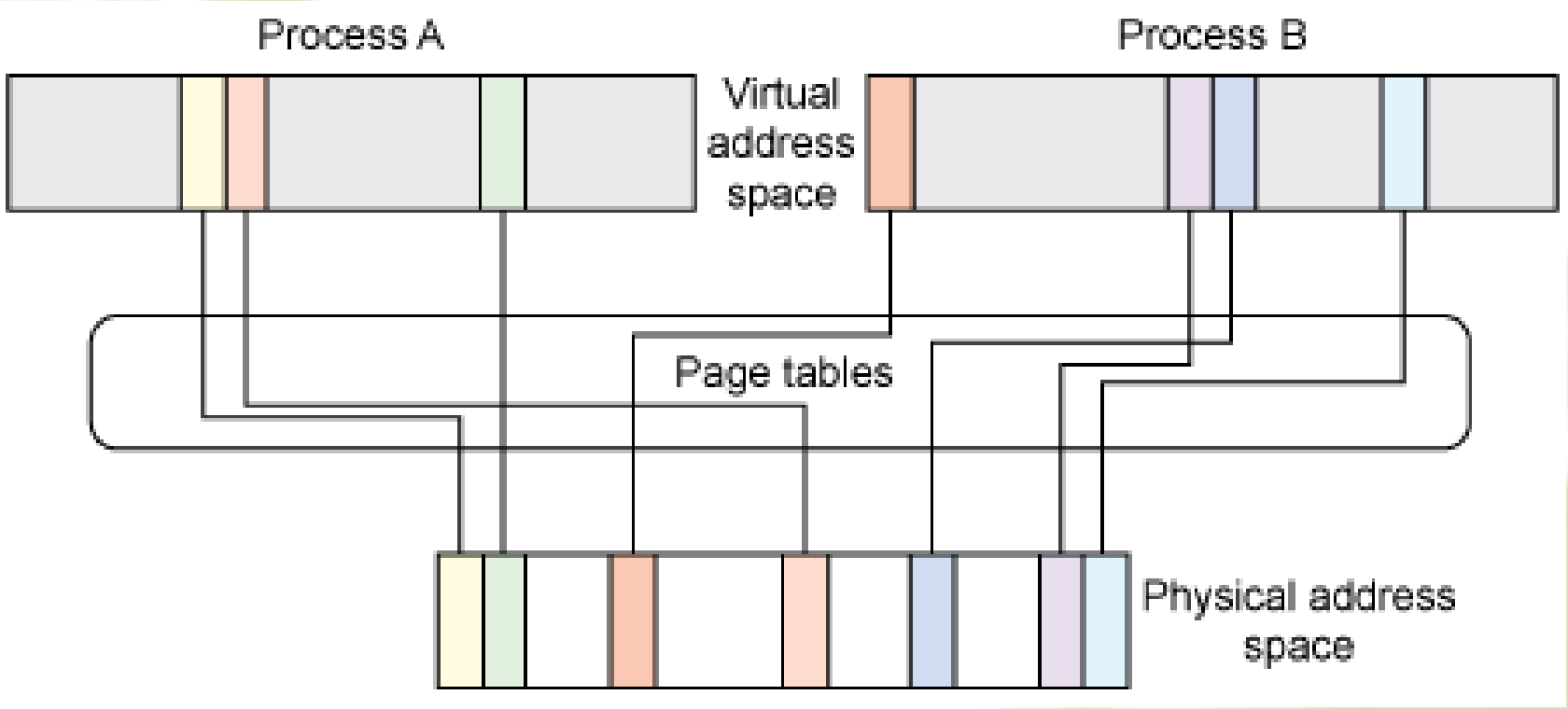
IA-64：4KB，8KB，64KB，256KB，1MB，4MB，16MB 和256MB.（从4KB到256MB有9种选择）

分页原理是什么

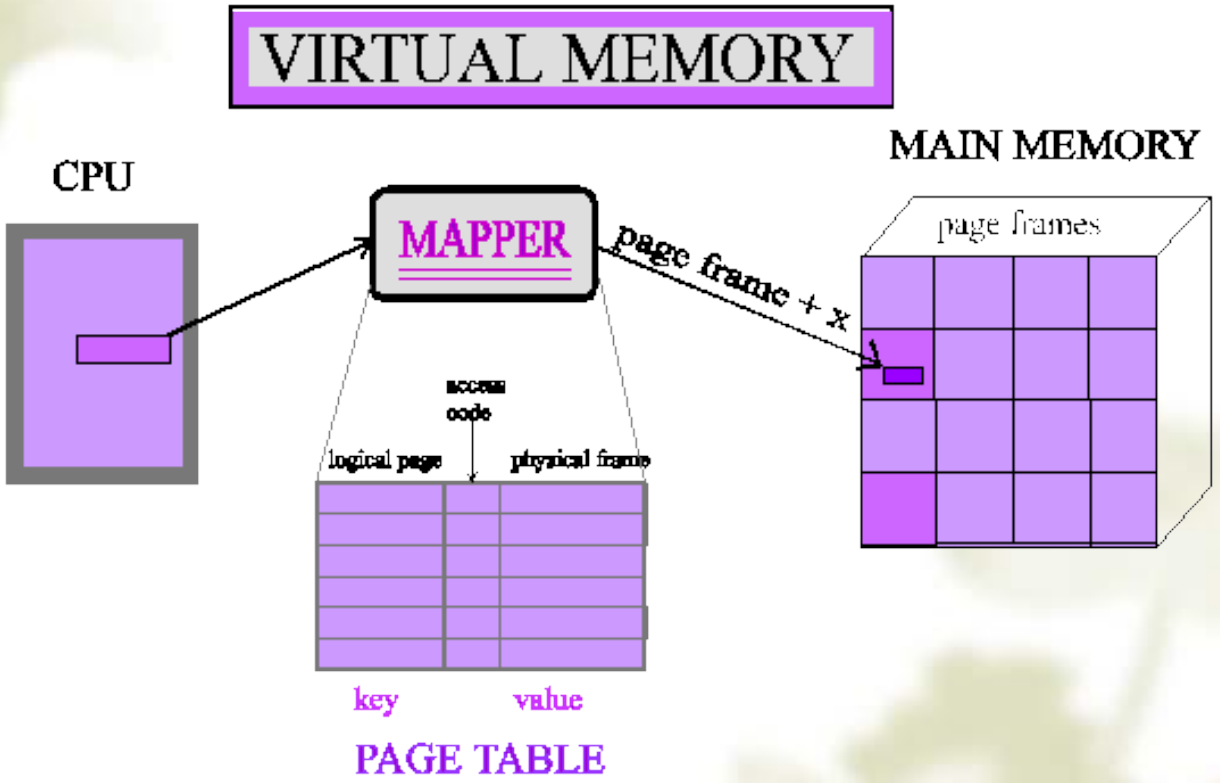
- ★使得每个进程可以拥有自己独立的虚拟内存空间。为了达到这个目的，CPU在访存的时候就需要进行一次地址变换，也就是把虚地址转换为物理地址，于是我们给出映射函数：Pa=f(va)

  这种转换在时间和空间上都要付出代价，因此必须进行优化。
- ★时间的优化。因为访存很频繁，因此，映射函数f一定要简单，否则会效率很低，所以需要简单查表算法，这也就是页表引入的原因。
- ★空间的优化。因为内存空间是按字节编址的，地址一一进行映射的话，效率也很低，于是要按照一定的粒度（也就是页）进行映射，这样，粒度内的相对地址（也就是页内偏移量）在映射时保持不变。

分页原理是什么

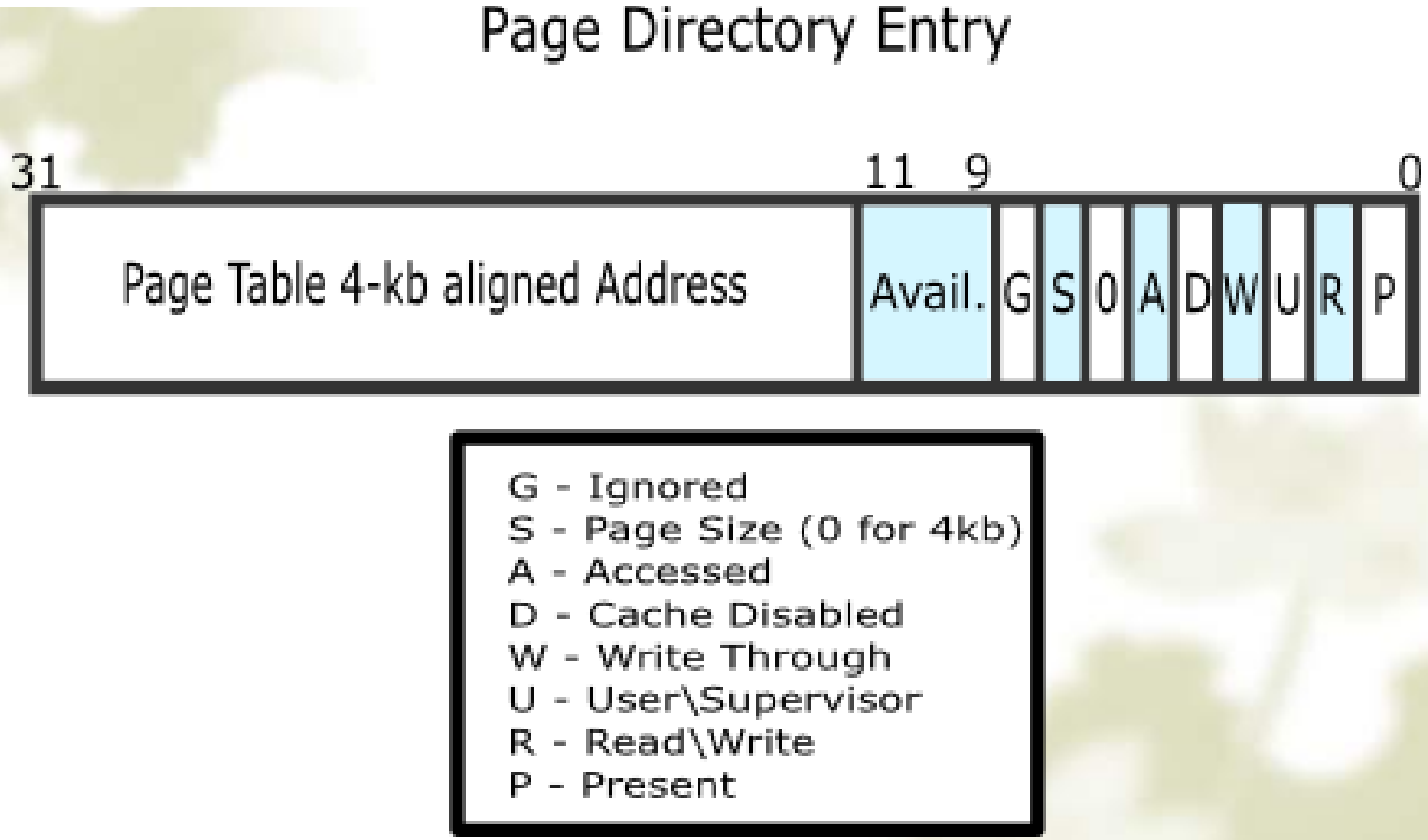

Process A
Process B
Virtual address space
Page tables
Physical address space

什么是页表



页表是一种映射机制，存放的是虚拟地址空间与物理地址空间的映射关系（也就是页号对应块号）

页表项结构



如图为32位x86页表项的具体结构。

单级页表结构

在32位的地址空间，通常页的大小为4KB，这样的话，每一页起始地址的最低12位就为0了，用高20位就可以表示页的起始地址。低12位就可以用来表示页的属性了，也就是一个页表项占4字节，那么，4KB大小的页就可存放1K个页表项。

单级页表结构

32位逻辑地址空间，页表项大小为4B，页面大小为 4KB，则页内地址占12位

31 12	11 0
页号	页内偏移量

单级页表结构的逻辑地址结构

0# 页表

0	2
1	4
.....	
1023	...

1# 页表

0	762
.....	
1023	...

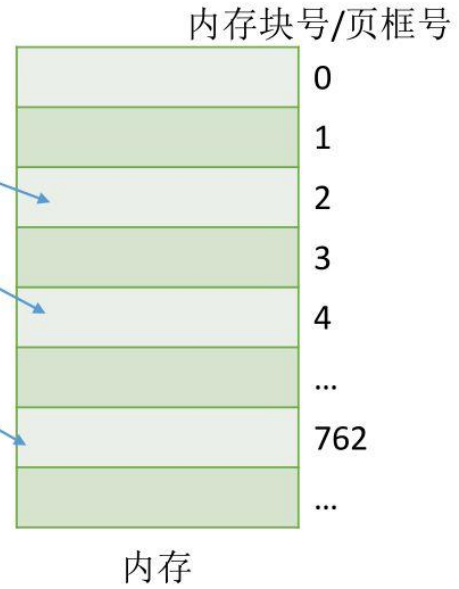
1023# 页表

0	...
.....	
1023	...

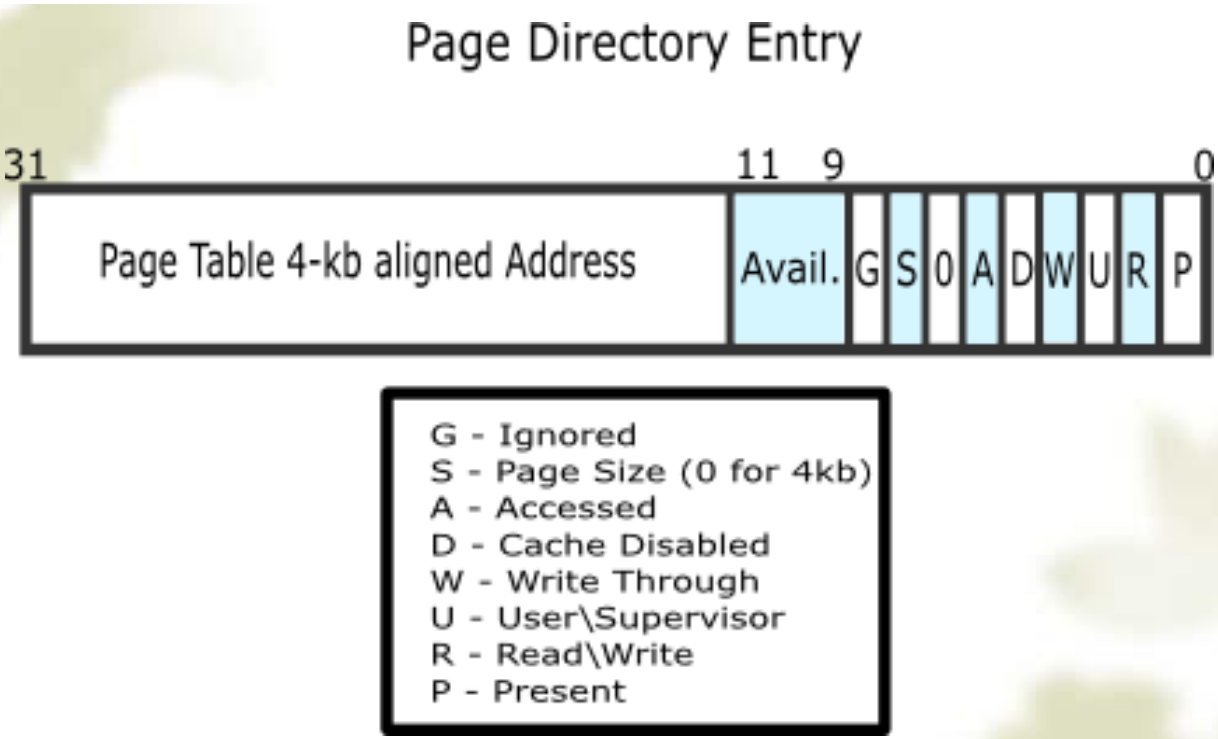
分为1024个部分

页号	块号
0	2
1	4
...	
1024	762
...	
1048575	...

进程最多有 2^{20} 个页面，用 20 位二进制刚好可以表示 0~2^{20}-1 个页号。
每个页面可存放 4K/4 = 1K = 2^{10} = 1024 个页表项。

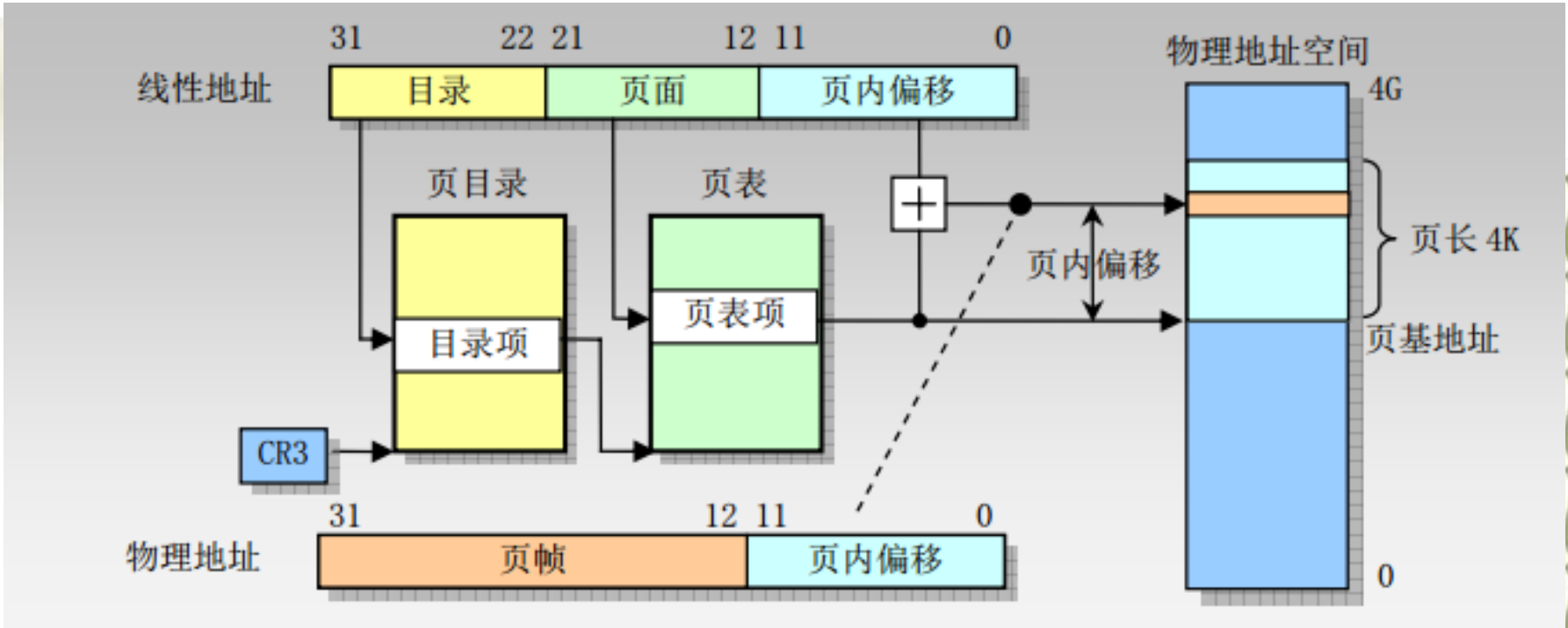


页表项结构



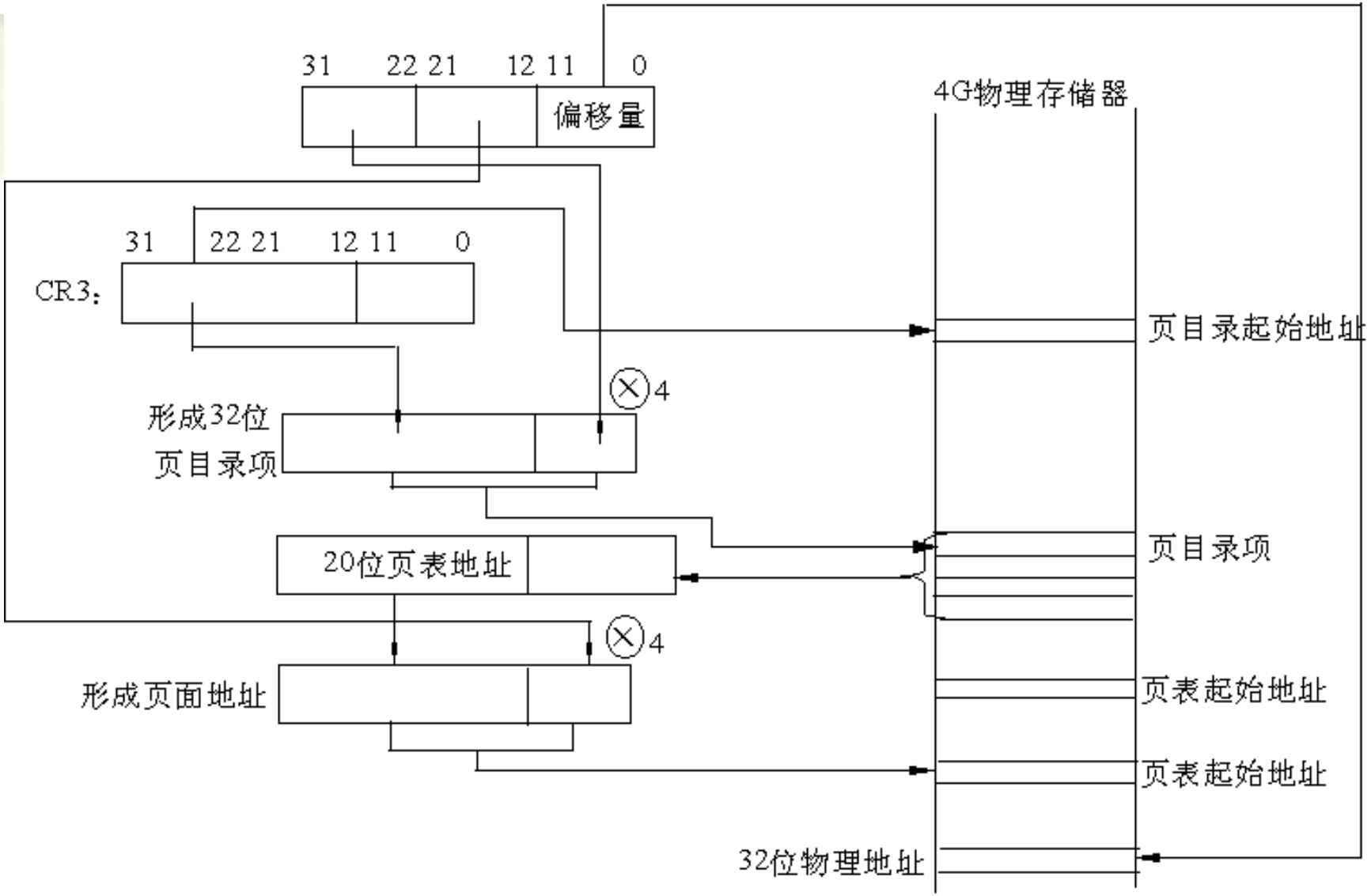
这里我们特别关注一下页表属性中的P位，也就是存在位，这是判别缺页的重要标志，更具体的信息Intel手册都有特别详细的描述，手边务必有Intel的手册。那么问题来了，这种结构是由是来确定的？仅仅是硬件厂家还是与OS的设计者共同商定的？

两级页表



前面给出的页表项是一级页表结果，如果只用一级页表，因为每个页表最大可占4MB的空间，而且必须连续，这就为内存的分配带来困难，怎么办，依然采用分而治之的原则，于是我们将高20位分为两部分，分别占10位，形成两级页表。那么，两级页表如何进行地址转换？

线性地址到物理地址的转换



地址转换过程

第一步，用最高10位作为页目录项的索引，将它乘以4，与CR3中的页目录的起始地址相加，获得相应目录项在内存的地址。

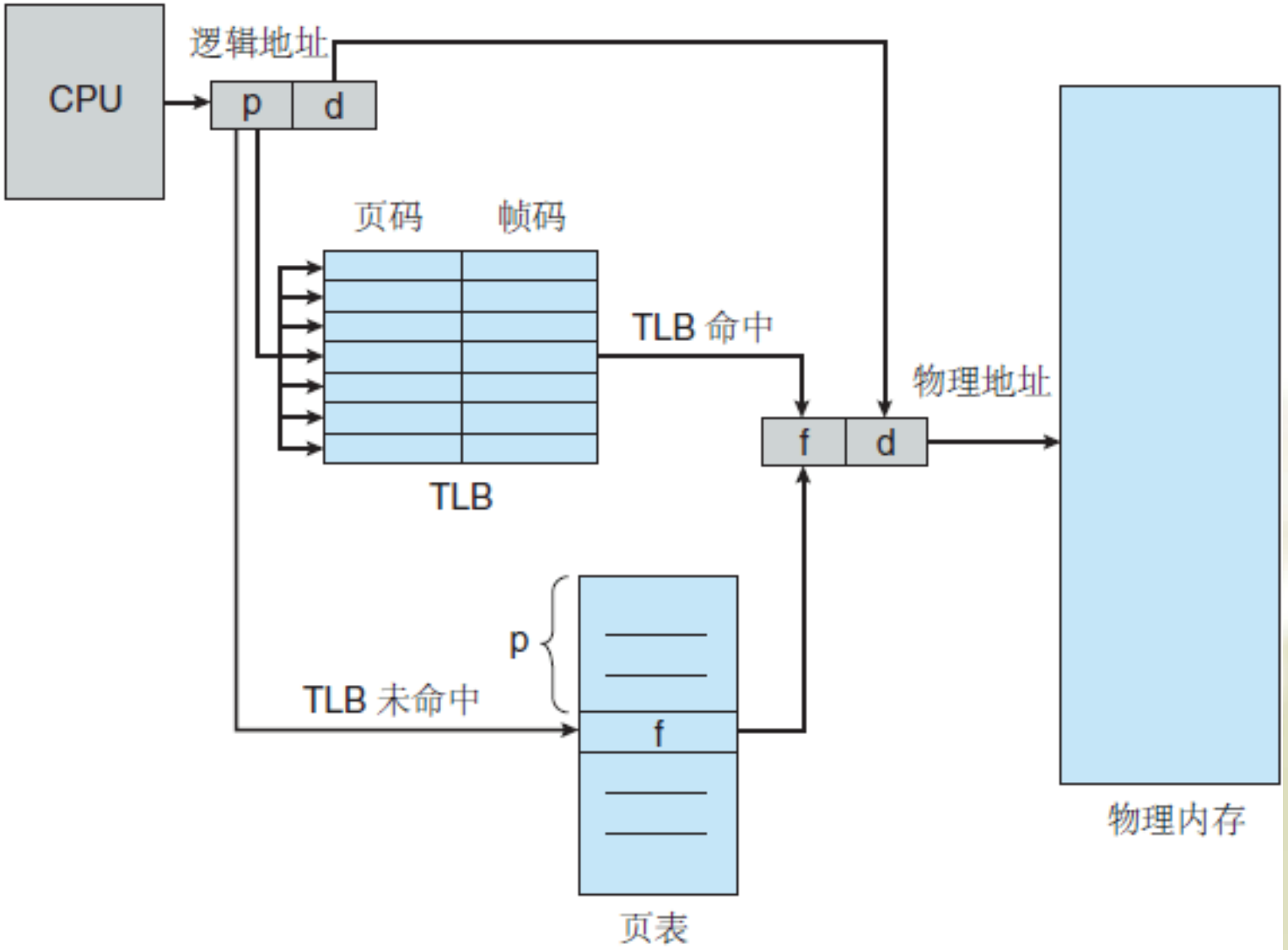
第二步，从这个地址开始读取32位页目录项，取出其高20位，再给低12位补0，形成页表在内存的起始地址。

第三步，用中间的10位作为页表中页表项的索引，将它乘以4，与页表的起始地址相加，获得相应页表项在内存的地址。

第四步，从这个地址开始读取32位页表项，取出其高20位，再将线性地址的第11~0位放在低12位，形成最终32位页面物理地址。

这个转换过程听起来很复杂，到底是谁来完成？硬件还是操作系统？

页面高速缓存

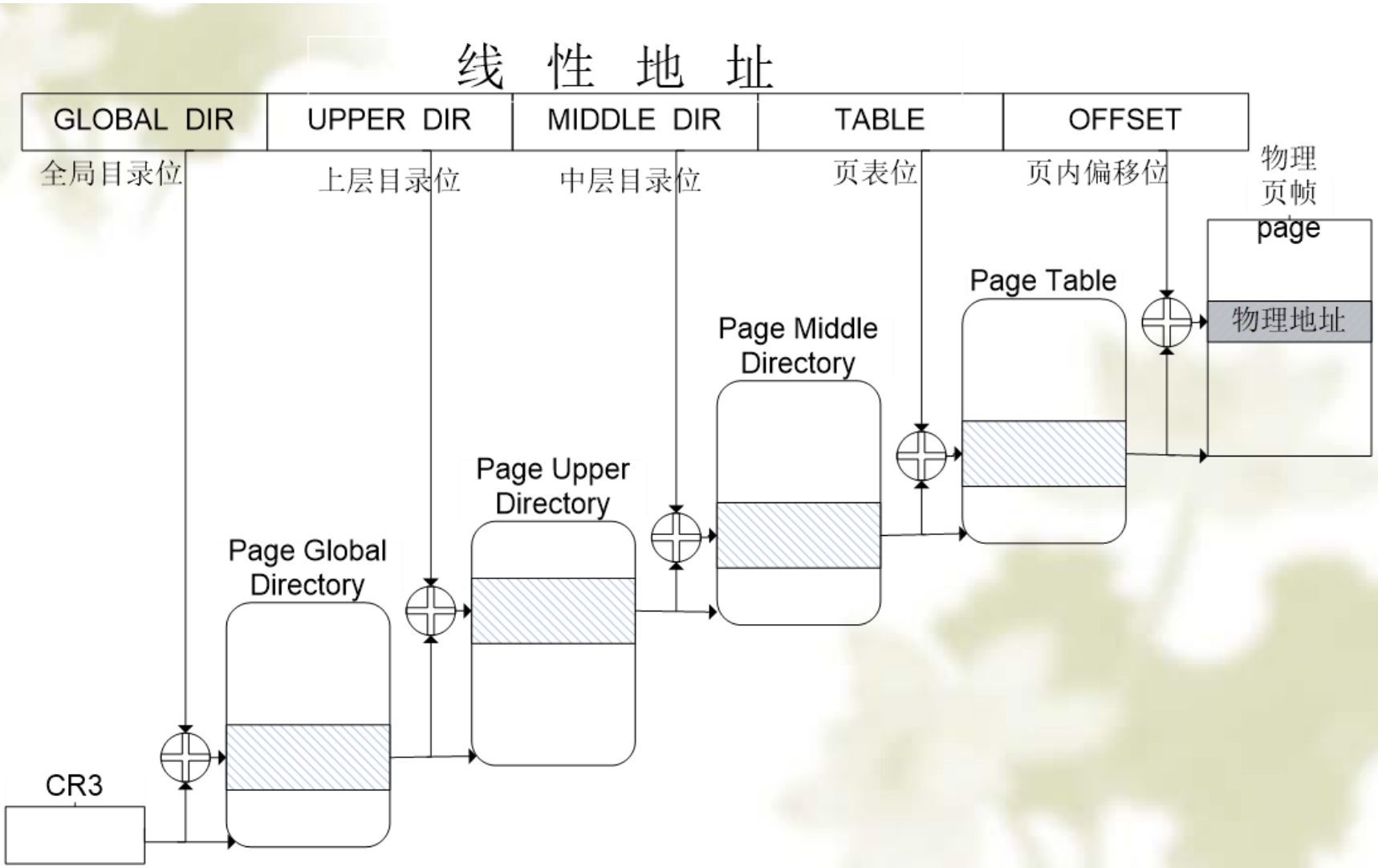


页面高速缓存

由于在分页情况下，页表是放在内存中的，这使CPU在每次存取一个数据时,都要至少两次访存,从而大大降低了访问速度。所以，为了提高速度，在x86中设置一个高速缓存硬件机制，也叫“转换旁路缓冲器”。当CPU访问地址空间的某个地址时，先检查对应的页表项是否在高速缓存中，如果命中，就不必经过两级访存了，如果失败，再进行两级访存。平均来说，页面高速缓存大约有90%的命中率，也就是说每次访存时，只有10%的情况必须访问两级页表。这就大大加快了访存速度。

那么，Linux中如何分页？

Linux中的分页


线 性 地 址
GLOBAL DIR
UPPER DIR
MIDDLE DIR
TABLE
OFFSET
全局目录位
上层目录位
中层目录位
页表位
页内偏移位
物理
页帧
page
物理地址
Page Table
Page Middle
Directory
Page Upper
Directory
Page Global
Directory
CR3

Linux中的分页

★Linux主要采用分页机制来实现虚拟存储器管理,这时因为以下两个原因:

- （1）Linux巧妙地绕过了段机制。
- （2）Linux设计目标之一就是具有可移植性，但很多CPU并不支持段。

目前许多处理器都采用64位结构的，为了保持可移植性，Linux目前采用四级分页模式，为此，定义了四种类型的页表:

★页总目录PGD（Page Global Directory）

★页上级目录PUD（Page Upper Directory）

★页中间目录PMD（Page Middle Derectory）

★页表PT（Page table）

★四级或者三级页表如何与二级页表兼容。Linux内核代码中进行了巧妙的处理，请参考源代码。

Linux中的分页

源代码中与页表相关的头文件如下：

include/asm-generic/pgtable-nopud.h

include/asm-generic/pgtable-nopmd.h

arch/x86/include/asm/pgtable-2level*.h

arch/x86/include/asm/pgtable-3level*.h

arch/x86/include/asm/pgtable_64*.h

分页初始化举例

```
1  # define NR_PGT 0X4
2  # define PGD_BASE (unsigned int *)0X1000
3  # define PAGE_OFFSET (unsigned int ) 0X2000
4
5  # define PTE_PRE 0X01
6  # define PTE_RW 0X02
7  # define PTE_USR 0X04
8
9  void page_init()
10 {
11     int pages = NR_PGT;
12
13     unsigned int page_offset = PAGE_OFFSET;
14     unsigned int * pgd = PGD_BASE;                    //页目录位于物理内存的第二个页框内
15     unsigned int phy_add = 0X0000;                    //在物理地址的最底端建立页机制所需要的表格
16     //页表从物理内存的第三个页框处开始
17     //物理内存的头8kb没有通过页表映射
18     unsigned int * pgt_entry = (unsigned int *)0X2000;
19     while(page--)
20     {
21         *pgd++ = page_offset|PTE_USR|PTE_RW|PTE_PRE;
22         page_offset += 0X1000;
23     }
```

分页初始化举例

```
24      pgd = PGD_BASE;
25      //在页表中填写页到物理地址的映射关系
26      //映射了4MB大小的物理内存
27      while(phy_add < 0X1000000)
28      {
29          *pgt_entry++ = phy_add|PTE_USR|PTE_RW|PTE_PRE;
30          phy_add += 0X1000;
31      }
32
33      _asm_ _volatile_("movl %0, %%eax"
34                       "movl %%cr0, %%eax;"
35                       "orl $0X80000000, %%eax"
36                       "movl %%eax, %%cr0;"::"r"(pgd):"memory","% eax"
37                       );
38  }
```

分页举例

在了解页表基本原理之后，通过代码来模拟内核初始化页表的过程：

第1-7行 是页目录基地址、偏移量以及页表属性的定义

第14行在物理内存的第二个页设置了页目录，然后第19~23行的循环初始化了页目录中的四个页目录项，即四个页表。第27~31行循环初始化了四个页表中的第一个页表，映射了4MB的物理内存，到此页表已初始化好，剩下的工作就是将页目录的地址传递给cr3寄存器，这是由第33~38行的嵌入式汇编代码完成，并且设置了cr0寄存器中的分页允许。关于嵌入式汇编请参见2.5.3节。

虽然上述代码比较简单，但却描述了页表初始化的过程，以此为模型我们可以更容易理解Linux内核中关于页表的代码

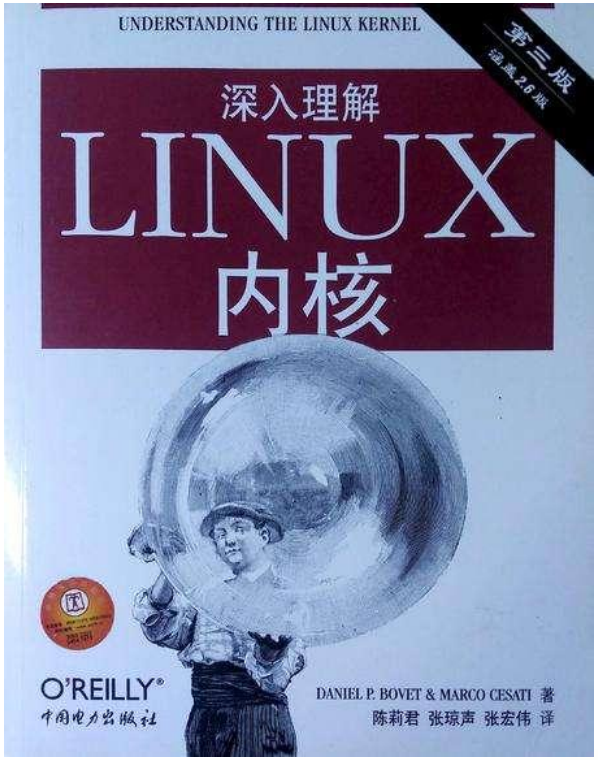
动手实战

在Linux内核之旅网站：http://www.kerneltravel.net/

“电子杂志” 栏目 第二期 “《i386体系结构》下”，实现一个最短小的可启动内核，一是为了加深对x86体系的了解，二是演示系统开发的原始过程。

- 下载代码进行调试

参考资料



深入理解Linux内核第二章

Intel® 64 and IA-32 Architectures Software Developer Manuals

在海量存储中如何使用大页内存？

页的大小不仅只有4KB，还可以是2M等更大的，那么在海量存储中，如何使用大页内存？

谢谢大家！

THANK YOU