北京郵電大學
Beijing University of Posts and Telecommunications

操作系统内核
-基于Linux

第3讲
内存寻址

主讲：杨文川

内容

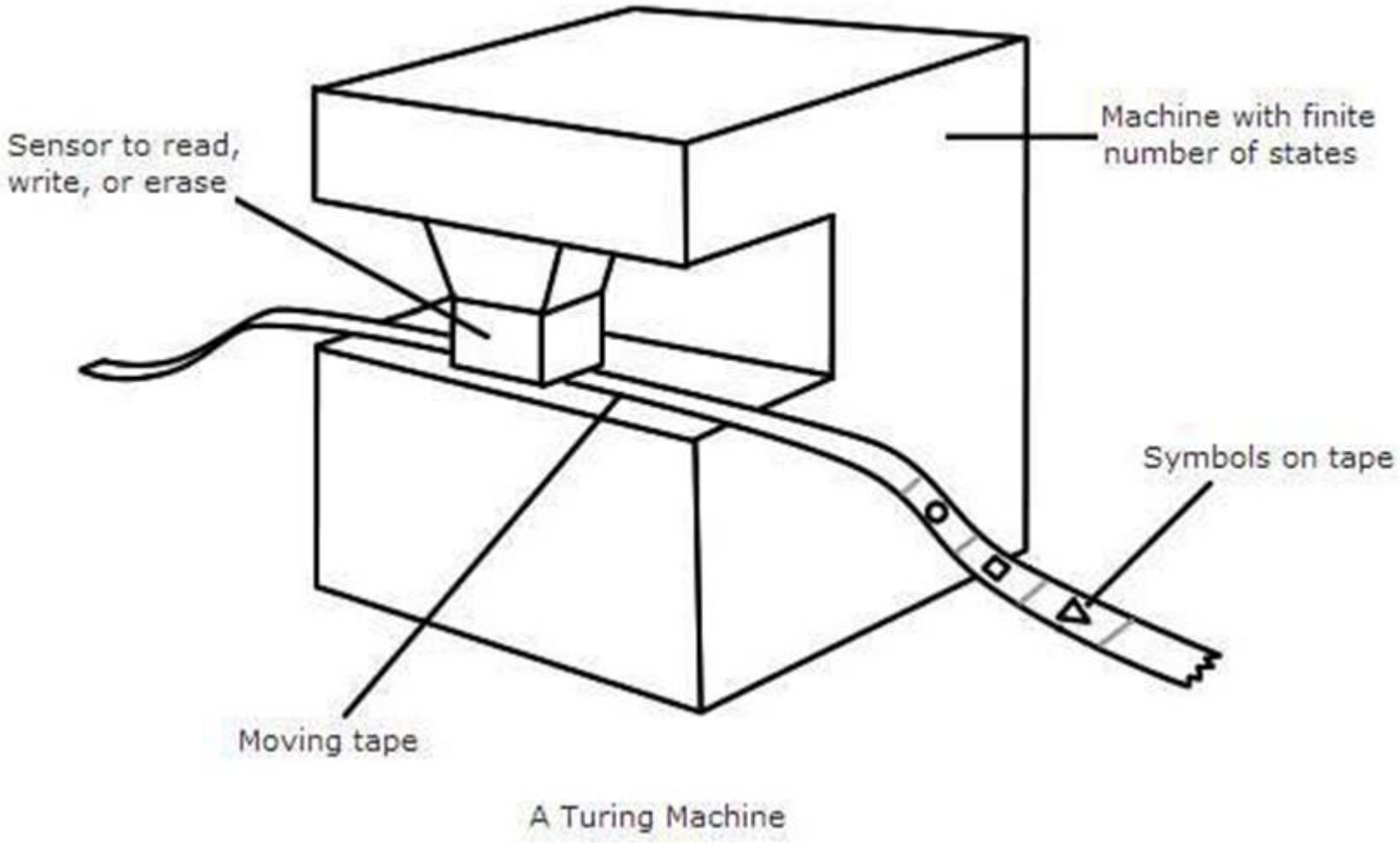
- 1 内存管理之内存寻址
- 2 段机制
- 3 分页机制
- 4 实践-把虚拟地址转换成物理地址

内容导航：

1 内存寻址概述

图灵机-计算力无限的理想机器

- 为什么首先讲内存寻址？
- 首先从 **图灵机**说起，图灵机是一种通用 自动机器模型。
- 其理念是，由一个两端无限 沿伸 的纸带作为存储装置。
- **输入， 输出**和**状态转移函数**是一个机器 的三要素，这三要素组合并变形 可成为一切机器的原型，可解决 一切图灵机能解决的问题。



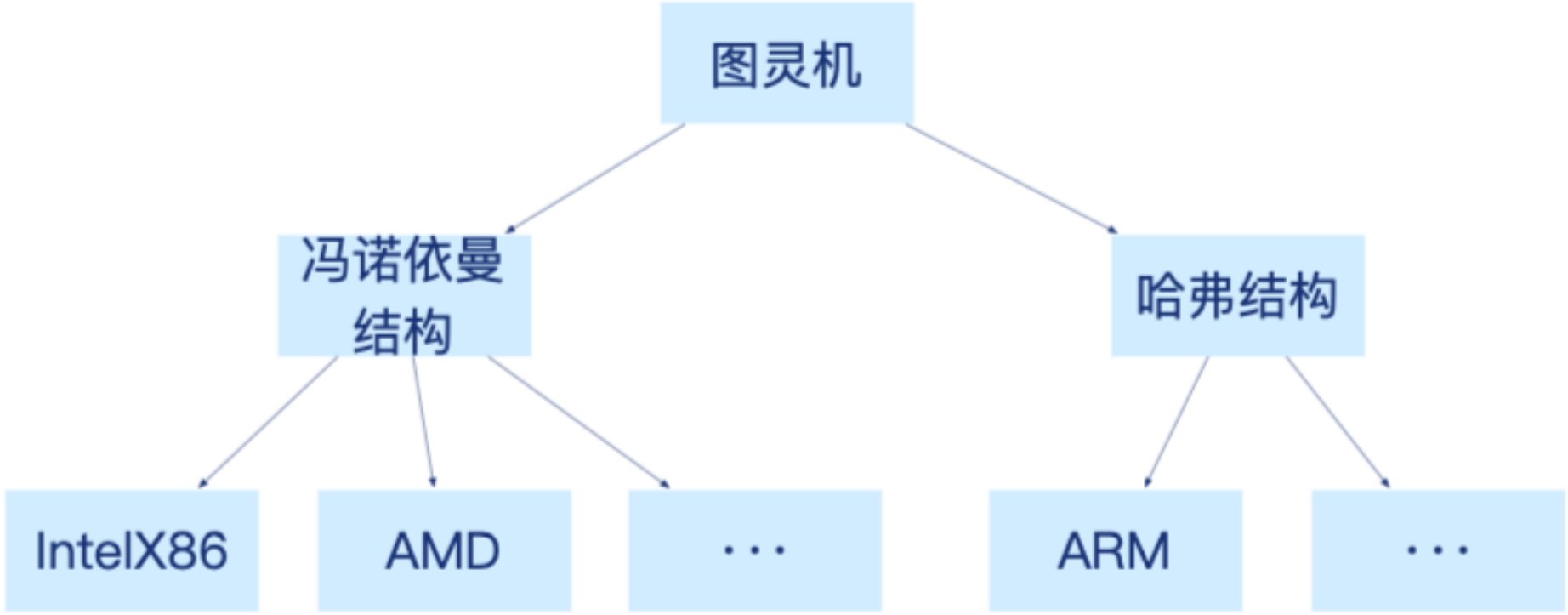
A Turing Machine

冯诺依曼体系结构

- 图灵机听起来是纸上谈兵，但它却是当代冯.诺依曼计算机体系的理论鼻祖。
- 它带来的“数据连续存储和选择读取思想”是目前我们使用的几乎所有机器运行背后的灵魂。
- 计算机体系结构中的核心问题之一，就是如何有效地进行内存寻址，
- 因为所有运算的前提，都是先要从内存中取得数据，所以内存寻址技术，从某种程度上代表了计算机技术。

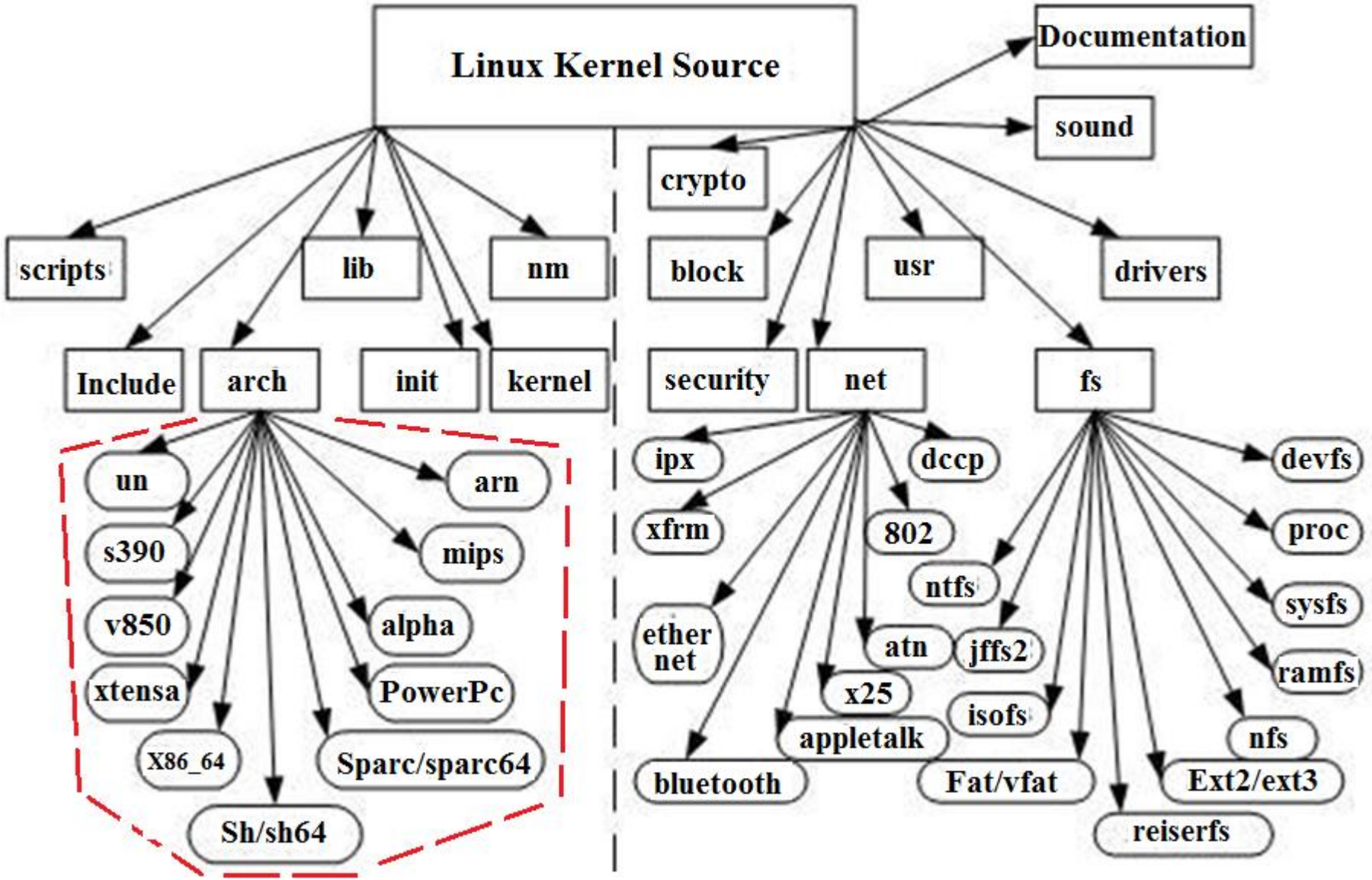
图灵机与冯诺依曼体系结构

- 如图所示，图灵机是冯.诺依曼计算机体系的鼻祖，而目前的大多数CPU体系结构的鼻祖又是冯 诺依曼体系。
- Linux在内核设计中，目前几乎支持所有主流的CPU架构，其设计理念遵循了分离体系结构相关代码的原则。



Linux内核的可移植性设计

- 如图所示，在Linux支持的众多的CPU体系结构中，与体系结构相关的代码在专门的arch目录下，大家最熟悉的就是X86。
- 因此，我们所介绍的内存寻址也是以此为背景，而且是以32位寻址为主。
- 下面先从X86内存寻址的不同时期说起

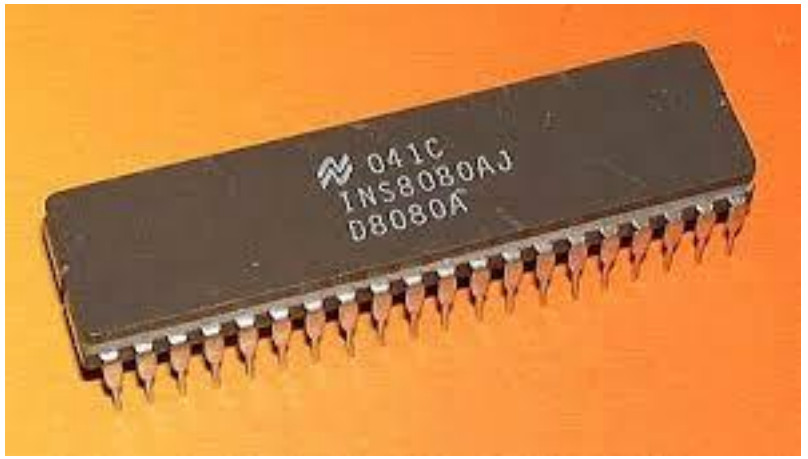


X86内存寻址的不同时期

- 8位时期 - 绝对地址
- 16位时期 - 段的引入
- 24位时期 - 保护模式的引入
- 32、64位时期 -内存寻址的飞跃

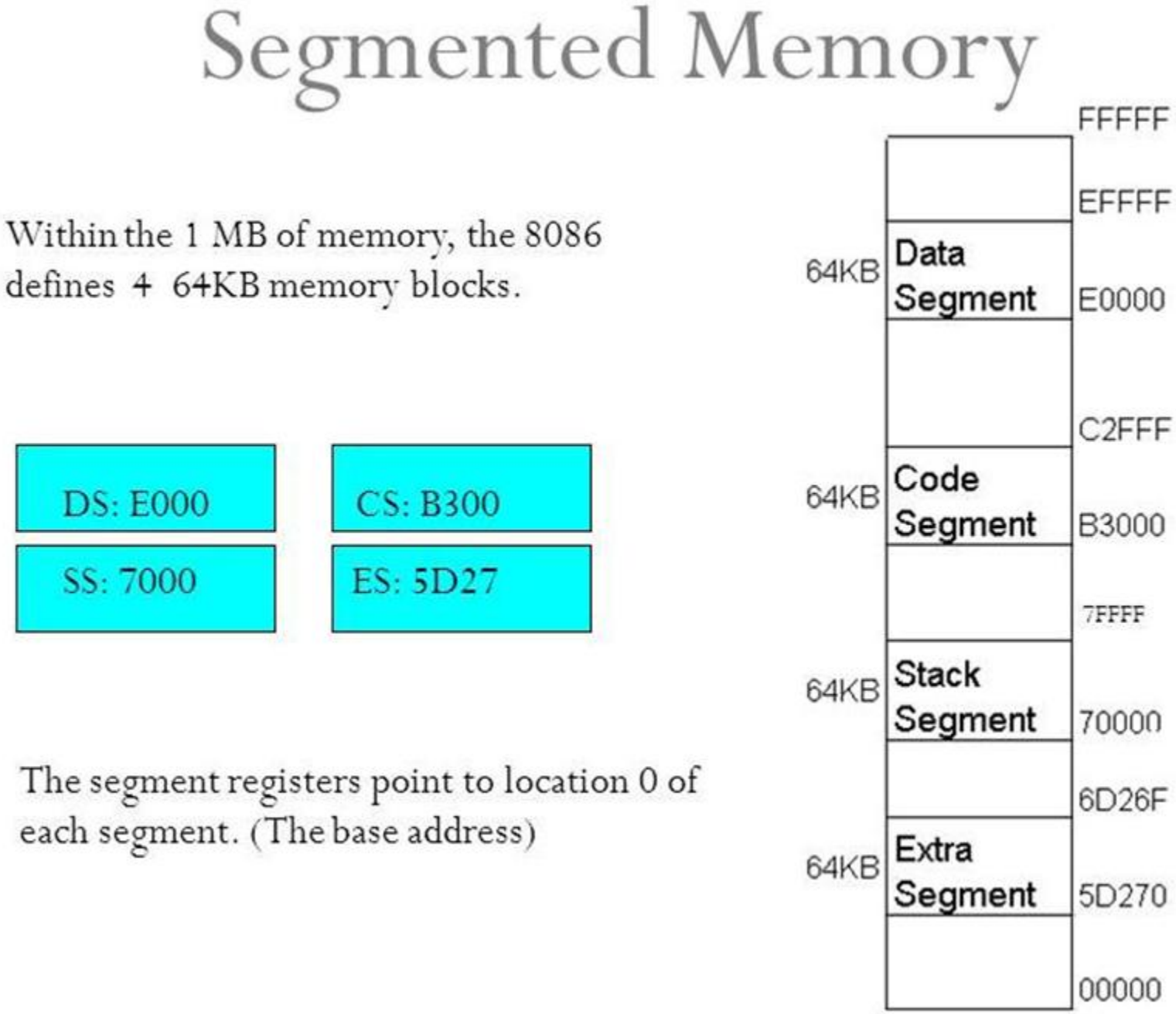
8位时期 - 绝对地址

- 在微处理器的历史上，第一款微处理器芯片4004是由Intel推出的，只有4位；
- 在4004之后，intel推出了一款8位处理器叫8080；
- 那时没有段的概念，访问内存都要通过绝对地址，因此程序中的地址，必须进行硬编码(也就是给出具体地址)，而且也难以重定位。



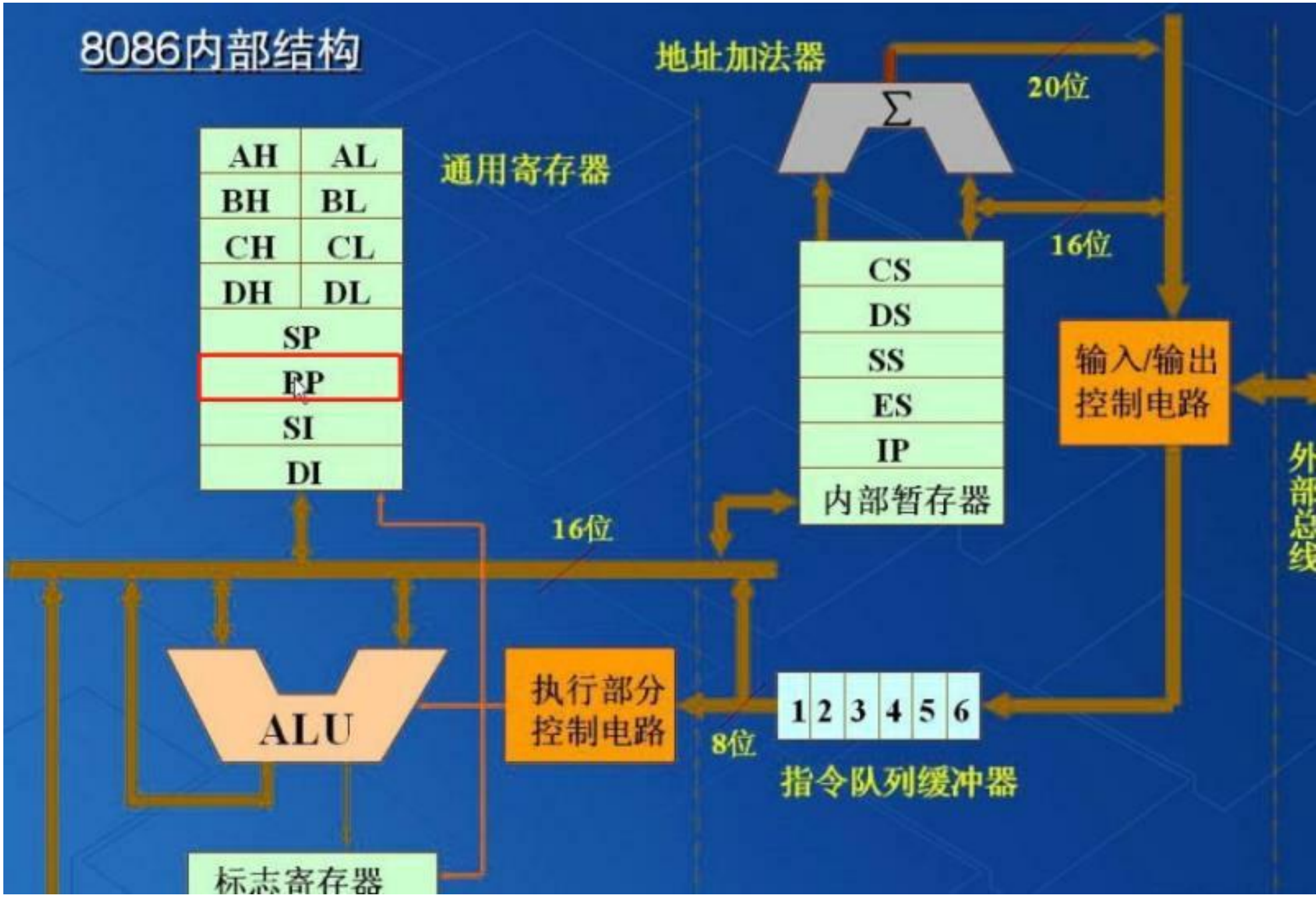
16位时期 - 段的引入

- 8086处理器的目标定位1M，于是它的地址总线扩展到了20位，可是数据总线只有16位。
- 也就是把1M大的空间分成数个64k的段来管理(化整为零了)。
- 段描述了一块有限的内存区域，区域的起始位置存在专门的寄存器(也就是段寄存器中)。
- 如图所示。



X86的寻址方式

- 把16位的段地址左移动4位后，再与16位的偏移量相加，便可获得一个20位的内存地址，
- 也就实现了从16位内存地址，到20位实际地址的转换，或“映射”
- 这种模式也叫“实模式”，
- 如图所示。



24位时期 - 保护模式的引入

- 286地址总线位数增加到了24位。
- 从此开始引进了一个全新理念—保护模式
- 访问内存时不能直接从段寄存器中，获得段的起始地址了，而需要经过额外转换和检查。

32、64位时期 - 内存寻址的飞跃

- 386是一个32位的CPU，其寻址能力达到4GB
- Intel选择了在段寄存器的基础上构筑保护模式，并且保留段寄存器16位，
- 在保护模式下,它的段范围不再受限于64K，可以达到4G
- 这真正解放了软件工程师，他们不必再费尽心思去压缩程序规模，软件功能也因此迅速提升
- 从80386以后，从32位到目前的64位，Intel的CPU经历了各种型号，但基本上属于同一种系统结构的改进与加强，而无本质的变化
- 所以我们把80386以后的处理器统称为x86

实模式和保护模式寄存器对比

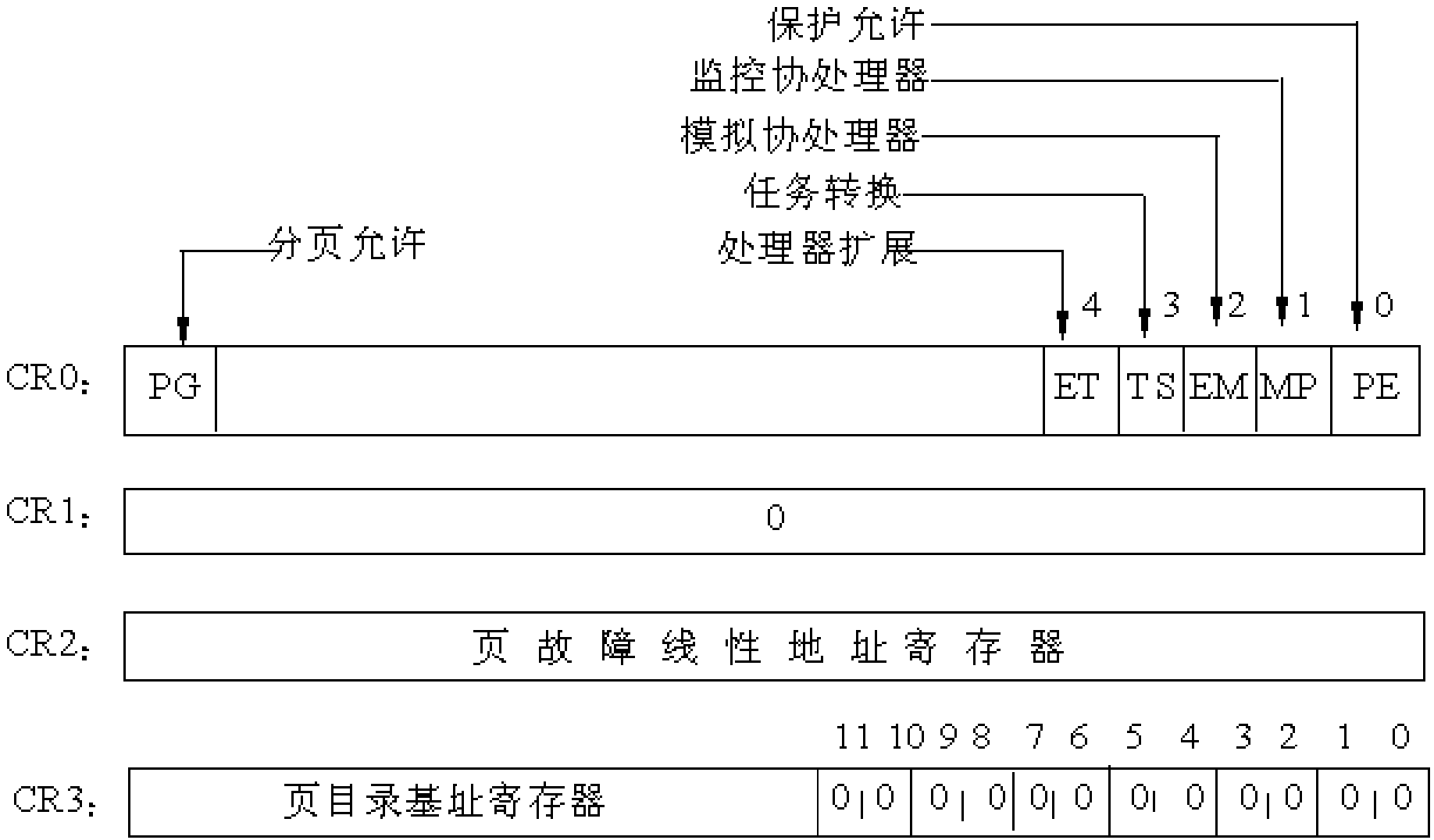
- 保护模式下的寄存器有很大的变化，
- 一些寄存器是专门属于操作系统使用的，比如用于分页的控制寄存器CR0~CR3，一般用户不能使用，
- 一些寄存器是系统地址寄存器，还有7个调试寄存器和7个测试寄存器，都是保护模式所特有的。
- 如图所示。

8086和80386寄存器对比

	8086的寄存器	80386的寄存器
通用寄存器	AX, BX, CX, DX, SP, BP, DI, SI	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
段寄存器	CS, DS, SS, ES	CS, DS, SS, ES, FS, GS
段描述符寄存器	无	对程序员不可见
状态和控制寄存器	FLAGS, IP	EFLAGS, EIP, CR0, CR1, CR2, CR3
系统地址寄存器	无	GDTR, IDTR, TR, LDTR
调试寄存器	无	DR0-DR7
测试寄存器	无	TR0-TR7

保护模式下的页表寄存器

- 这几个寄存器中保存全局性和任务无关的机器状态。
- CR0中包含了6个预定义标志，这里介绍内核中主要用到的0位和31位。



控制寄存器

- CR0的第0位是保护允许位PE(Protected Enable)，用于启动保护模式，如果PE位置1，则保护模式启动，如果PE=0，则在实模式下运行。
- CR0的第31位是分页允许位(Paging Enable)，它表示芯片上的分页部件是否被允许工作。
- 由PG位和PE位定义的操作方式如图所示。
- 其他几个寄存器暂时不详细介绍。

PG	PE	方式
0	0	实模式，8080操作
0	1	保护模式，但不允许分页
1	0	出错
1	1	允许分页的保护模式

内容导航：

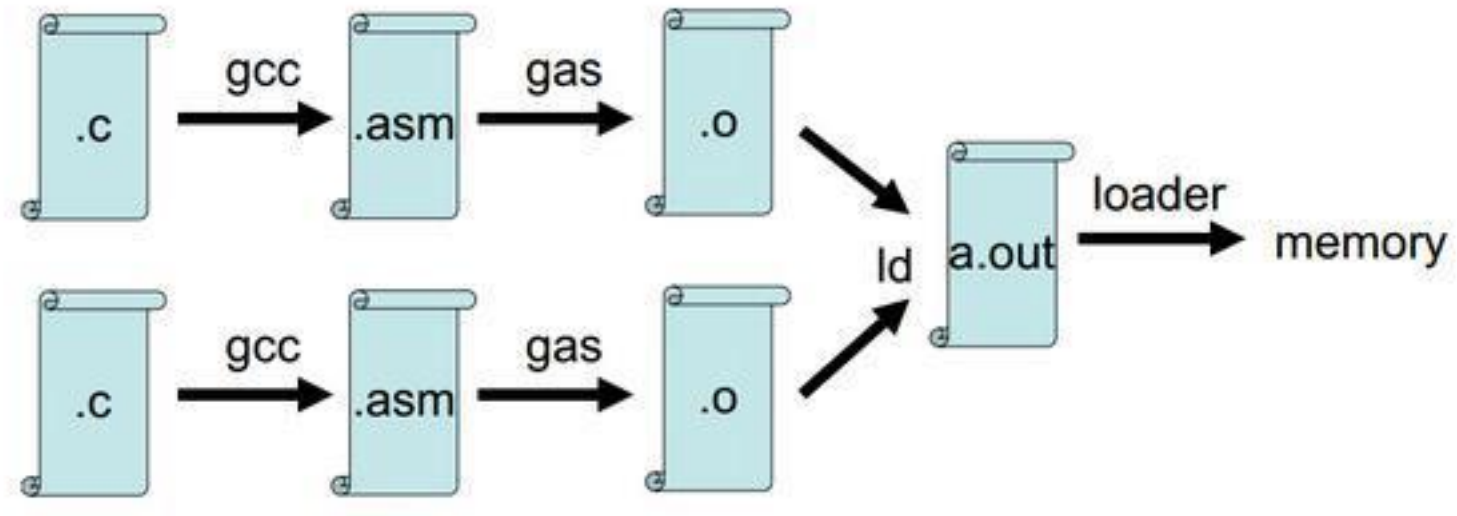
2 保护模式之段机制

从一个简单“Hello World”程序说起

- # include < stdio.h >
- int main(void) {
- printf(" Hello world！ ")；
- return0；
- }
- 注意：GNU 的C语言http://www.faqs.org/docs/learnc/
- 汇编使用的是AT&T的汇编格式，与Intel的汇编格式稍有差异，在C语言中可以嵌入汇编代码，叫GCC嵌入式汇编
- 代码参见samplej/3helloDev，
  - 可从QQ群文件中下载3helloDev解压缩拷贝到虚机samplej下运行

程序的编译、链接和装载

- 1. gcc -S hello.c -o hello.s// 编译
- 2. gcc -c hello.s -o hello.o// 汇编
- 3. gcc hello.c -o a.out// 链接
- 4. ./a.out// 装载并执行
- objdump -d a.out//反汇编



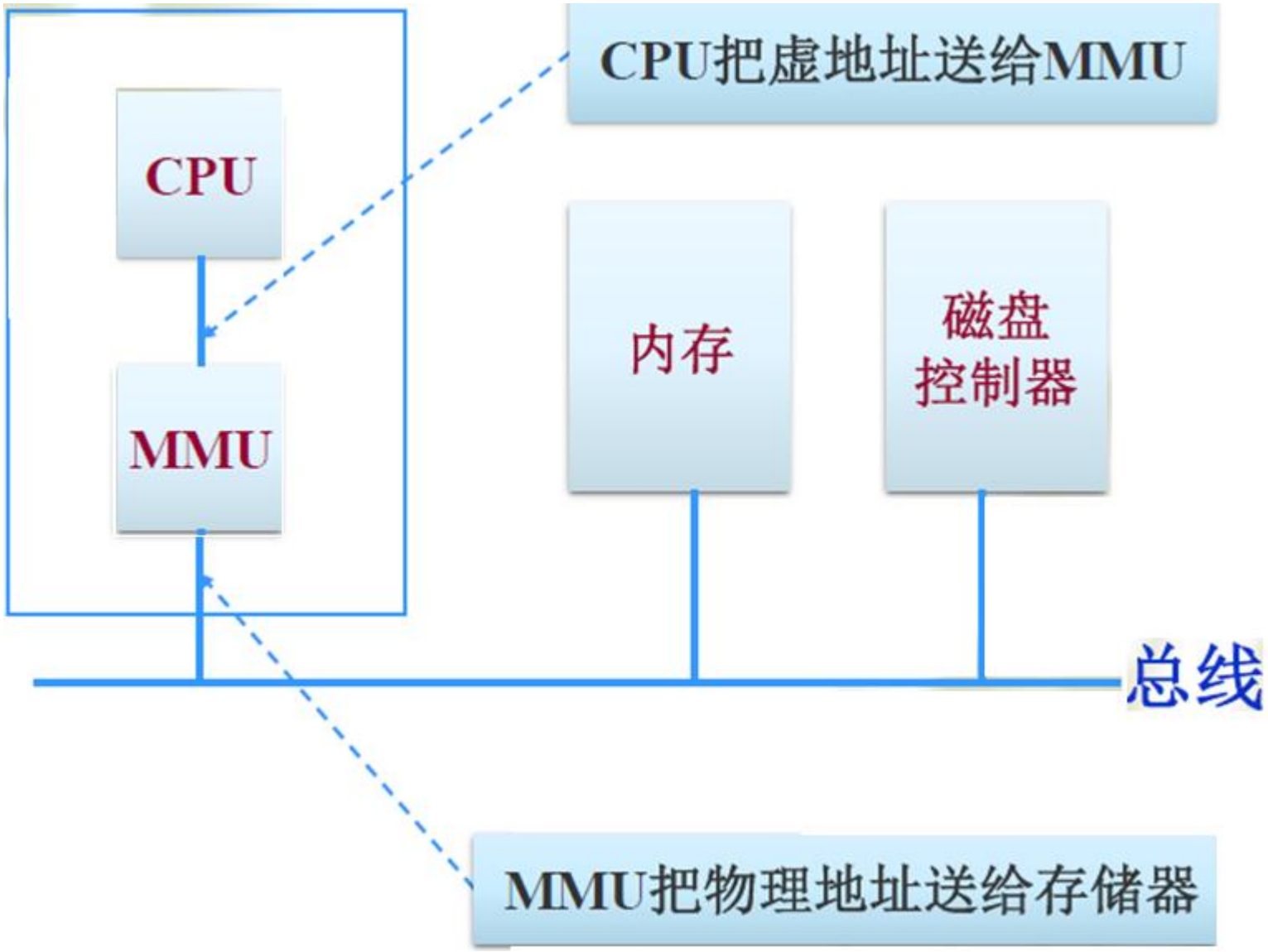
- 一个程序通过编译器GCC将其编译成汇编程序，经过汇编器gas将其汇编成目标代码，经过链接接器ld形成可执行文件a，最后通过装载器装入到内存。
- 那么，问题来了，链接以后形成的地址是虚地址还是实地址，装入程序把可执行代码装入到虚拟内存还是物理内存？CPU访问的是虚地址还是物理地址？

程序的地址空间

- 这是编译链接后的64位的地址空间，最左边是地址，中间是指令码，右边是AT&T格式的汇编指令。

```
00000000004006cd <main>:
  4006cd:       55                      push   %rbp
  4006ce:       48 89 e5                mov    %rsp,%rbp
  4006d1:       48 81 ec a0 00 00 00    sub    $0xa0,%rsp
  4006d8:       89 bd 6c ff ff ff       mov    %edi,-0x94(%rbp)
  4006de:       48 89 b5 60 ff ff ff    mov    %rsi,-0xa0(%rbp)
  4006e5:       83 bd 6c ff ff ff 03    cmpl   $0x3,-0x94(%rbp)
  4006ec:       74 19                   je     400707 <main+0x3a>
  4006ee:       bf 80 08 40 00          mov    $0x400880,%edi
  4006f3:       b8 00 00 00 00          mov    $0x0,%eax
  4006f8:       e8 63 fe ff ff          callq  400560 <printf@plt>
  4006fd:       bf 01 00 00 00          mov    $0x1,%edi
  400702:       e8 b9 fe ff ff          callq  4005c0 <exit@plt>
  400707:       48 8b 85 60 ff ff ff    mov    -0xa0(%rbp),%rax
  40070e:       48 83 c0 08             add    $0x8,%rax
  400712:       48 8b 00                mov    (%rax),%rax
```

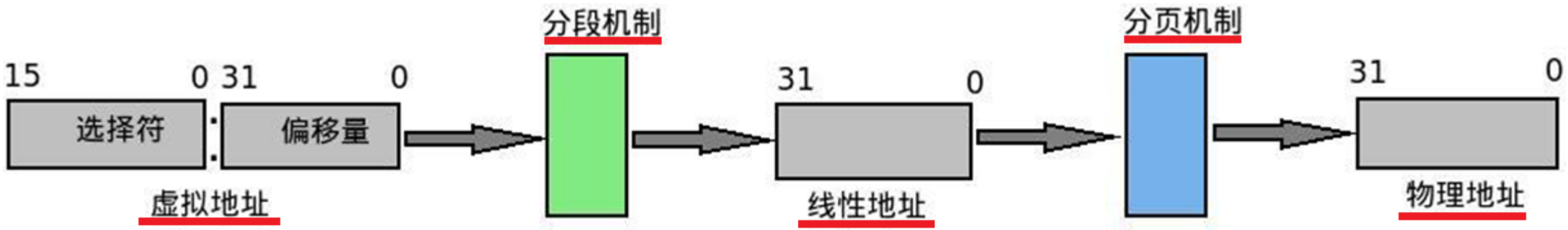
保护模式下的寻址



- 当我们编译链接形成的虚地址，就是CPU要访问的地址，
- CPU把虚地址送给MMU(内存管理单元)，它和CPU在一起的，
- 然后MMU把虚地址，转换成物理地址送给存储器。

MMU的地址转换

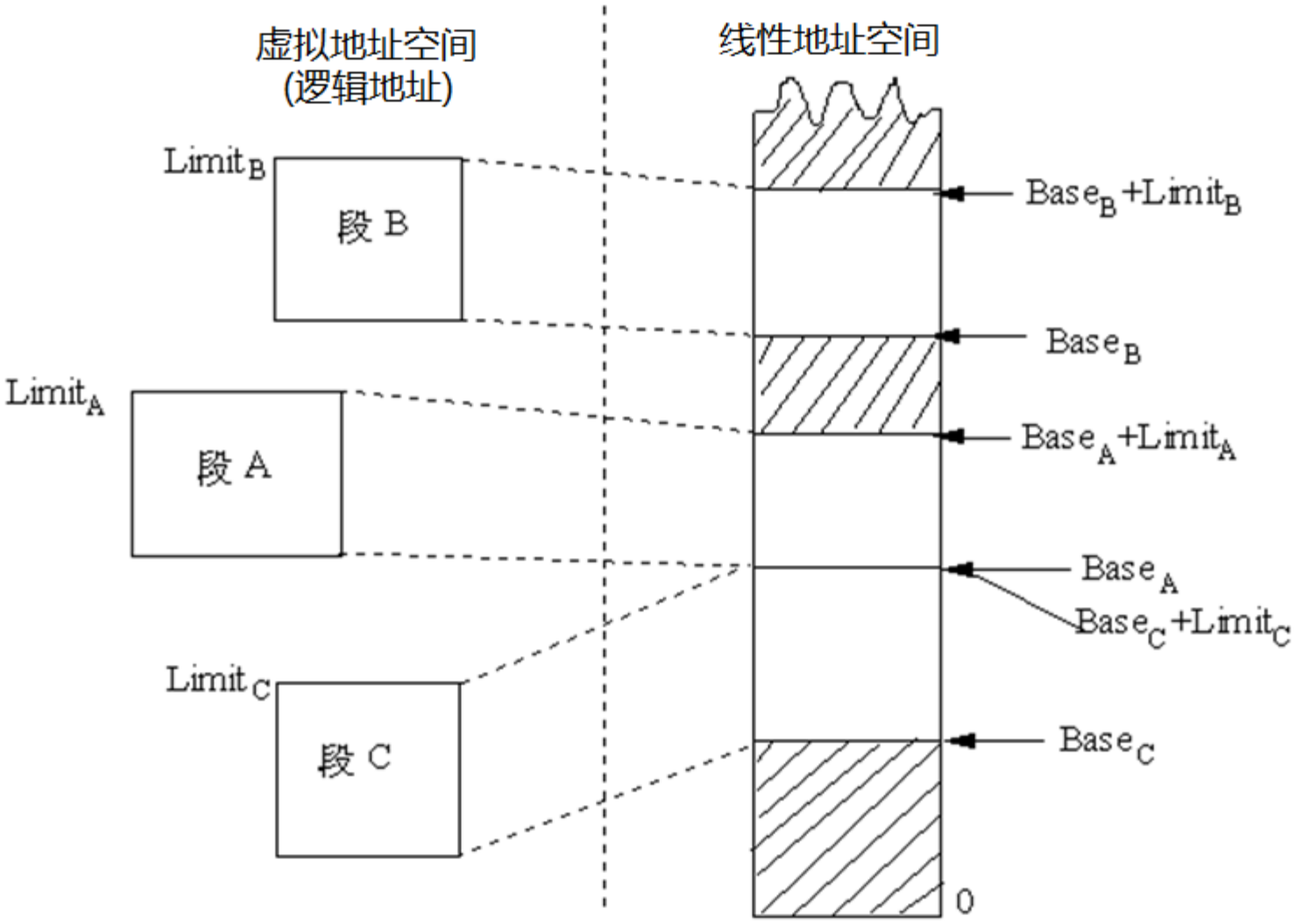
- MMU(Memory Management Unit)的转换分两个阶段，
- **分段机制** 和 **分页机制**，
- 分段 把 虚拟地址 转换为 线性地址，
- 分页 把 线性地址 转换为 物理地址。



MMU把虚拟地址转化为物理地址

虚拟地址 - 线性地址 的转换

- 虚拟地址 空间中偏移量从0到limit范围内的一个段，
- 映射到 线性地址空间中就是从Base到Base+Limit。



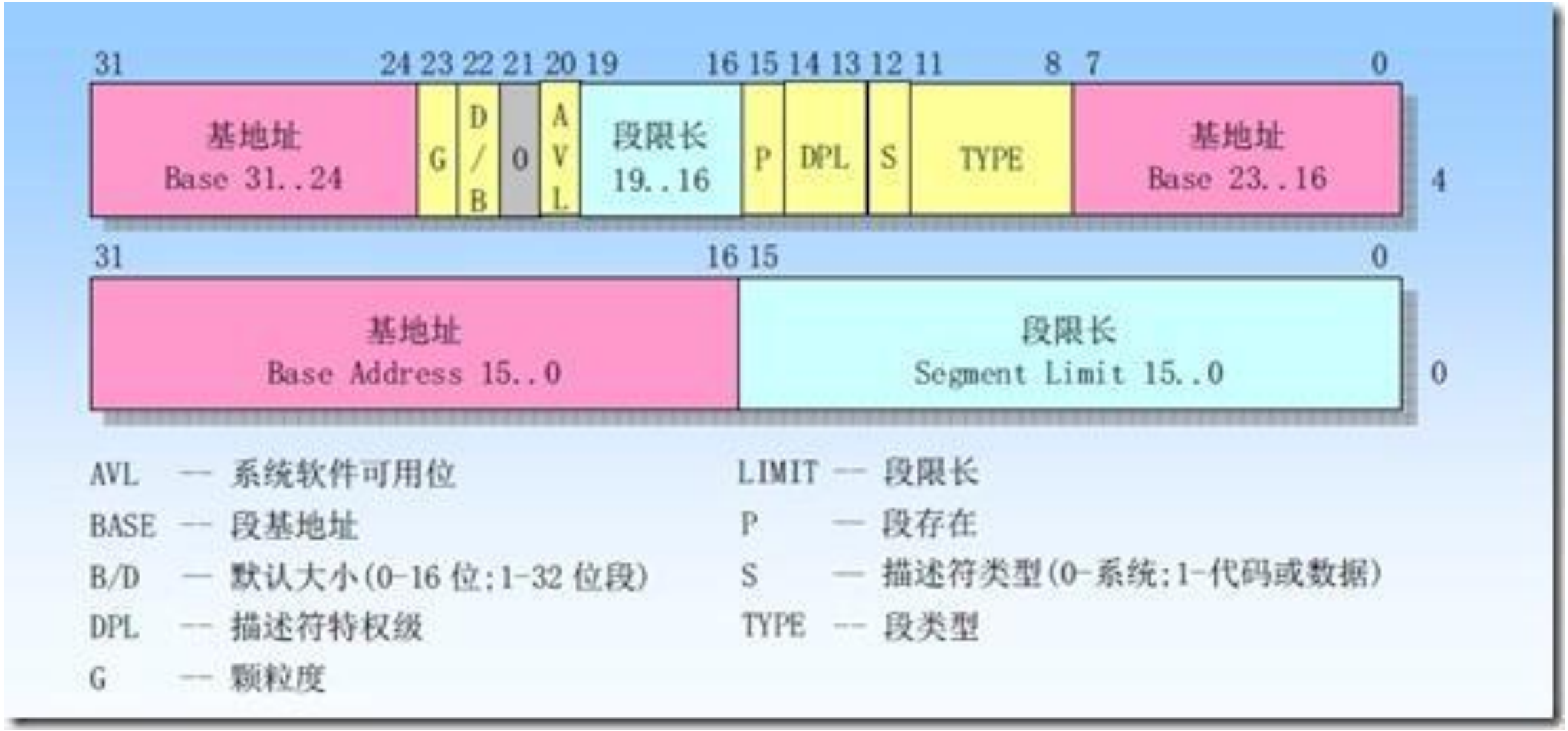
段描述符表 - 段表

- 如图所示的段描述符表(或叫段表)来描述转换关系。
- 段号描述的是虚拟地址空间段的编号，基地址是线性地址空间段的起始地址。
- 段描述符表中的每一个表项，叫做段描述符

索引（段号）	基地址	界限	属性
0	$Base_b$	$Limit_b$	$Attribute_b$
1	$Base_a$	$Limit_a$	$Attribute_e$
2	$Base_c$	$Limit_c$	$Attribute_c$

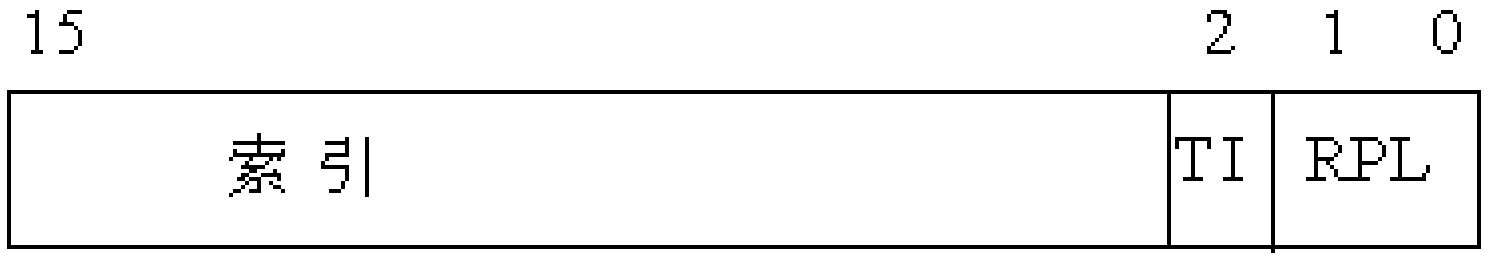
段描述符—描述段的结构

- 从图可以看出，一个段描述符指出了段的32位基地址和20位段界限(即段长)，1.5个字节用于描述段的属性。



保护模式下段寄存器中存放什么

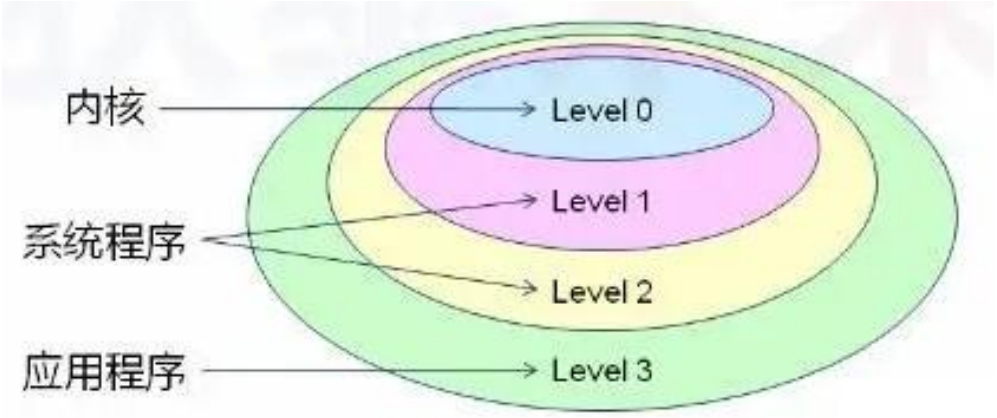
- 存放索引或叫段号，因此，这里的段寄存器也叫选择符，即从描述符表中选择某个段。
- 选择符(段寄存器)的结构：



- RPL表示请求者的特权级(Requestor Privilege Level)
- TI(Table Index)

保护模式下的特权级

- 保护模式提供了四个特权级，用0~3四个数字表示
- 很多操作系统(包括Linux , Windows)只使用了其中的最低和最高两个，即
  - 0表示最高特权级，对应内核态；
  - 3表示最低特权级，对应用户态。
- 保护模式规定，高特权级可以访问低特权级，而低特权级不能随便访问高特权级。



保护模式下的其他描述符表简介

- 全局描述符表GDT(Global Descriptor Table)
- 中断描述符表IDT(Interrupt Descriptor Table)
- 局部描述符表LDT(Local Descriptor Table)
- 为了加快对这些表的访问，Intel设计了专门的寄存器，以存放这些表的基地址，及表的长度界限。
- 这些寄存器只供操作系统使用。
- 有关这些表的详细内容请参看有关保护模式的参考书。

Linux中的段

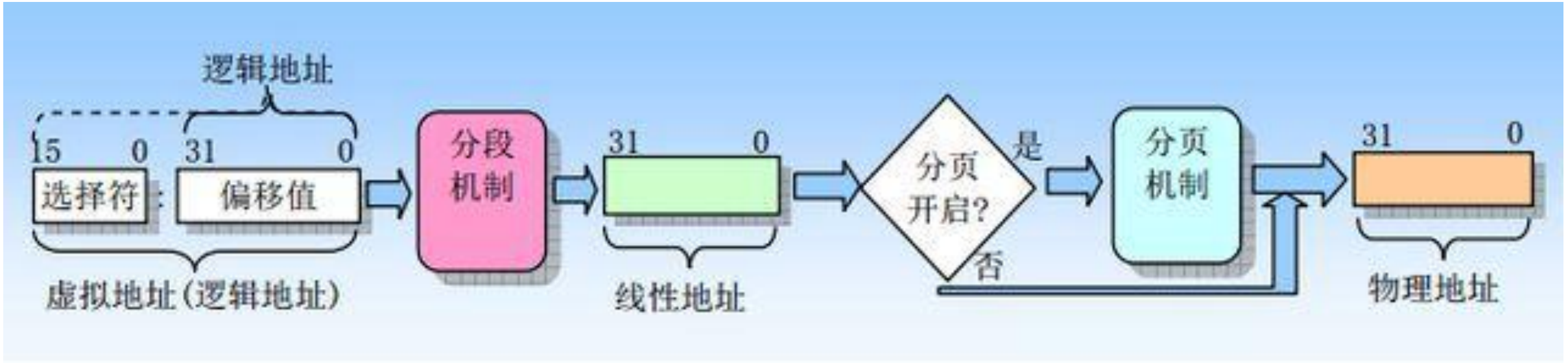
- 线性地址 = 段的起始地址+偏移量
- **Linux**在启动的过程中，设置了段寄存器的值和全局描述符表**GDT**的内容，内 核代码中可以这样定义段：
  - #define __KERNEL_CS 0x10
    - / *内核代码段，index=2,TI=0,RPL=0* /
  - #define __KERNEL_DS 0x18
    - / *内核数据段, index=3,TI=0,RPL=0* /
  - #define __USER_CS 0x23
    - / *用户代码段, index=4,TI=0,RPL=3* /
  - #define __USER_DS 0x2B
    - / *用户数据段, index=5,TI=0,RPL=3* /

内容导航：

3 分页机制

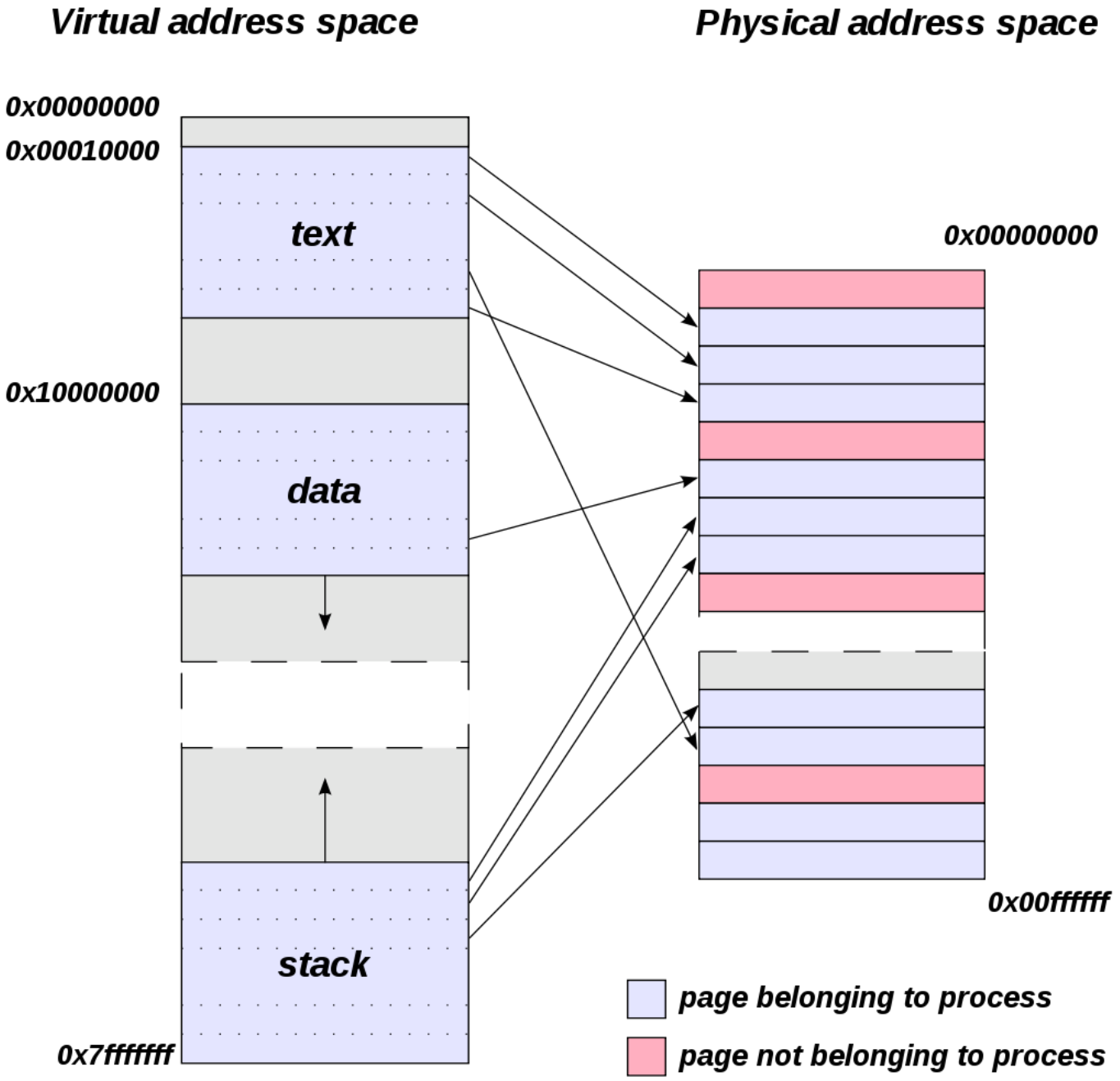
分页机制的引入

- 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** 再把 线性地址 转换为 物理地址。
- X86规定，分页机制是可选的，但很多操作系统设计者，主要采用分页机制。



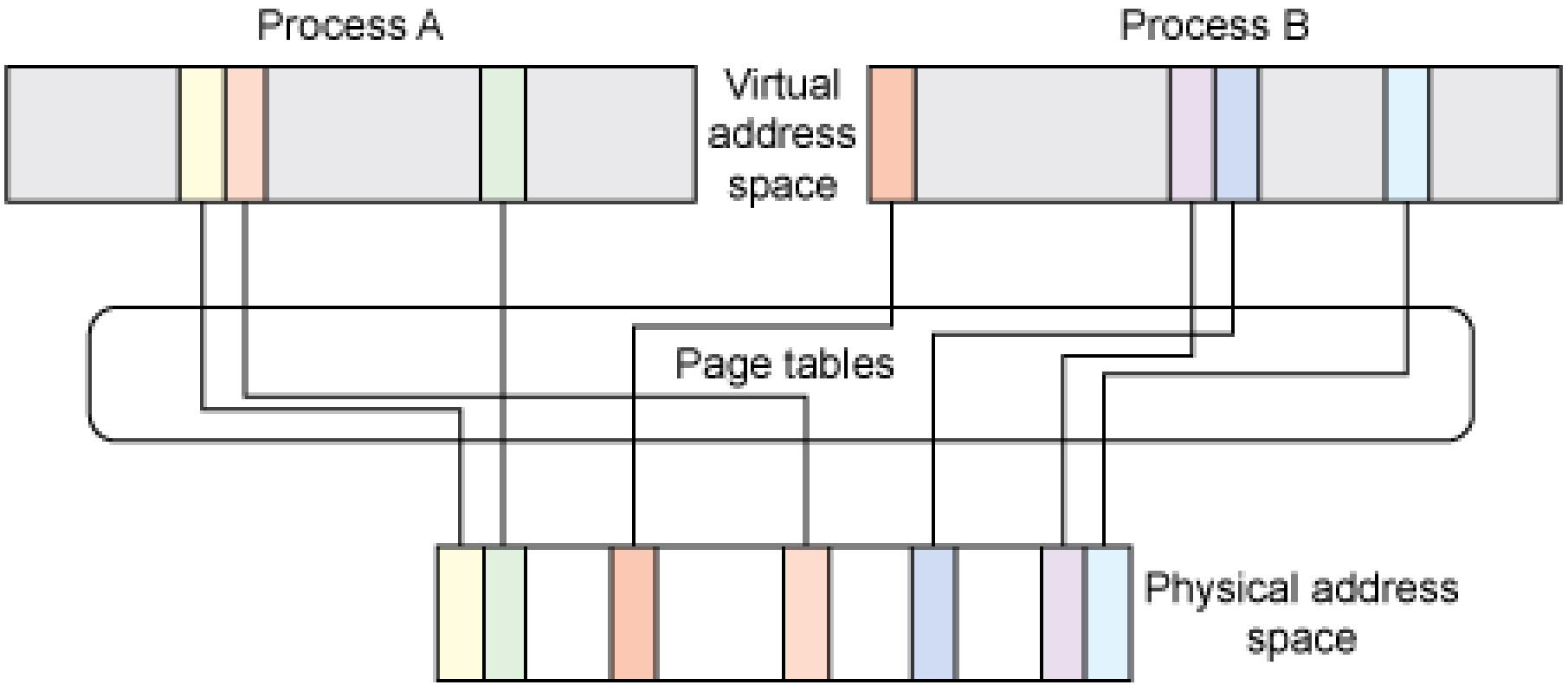
分页机制中的页

- 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)
- 把 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)

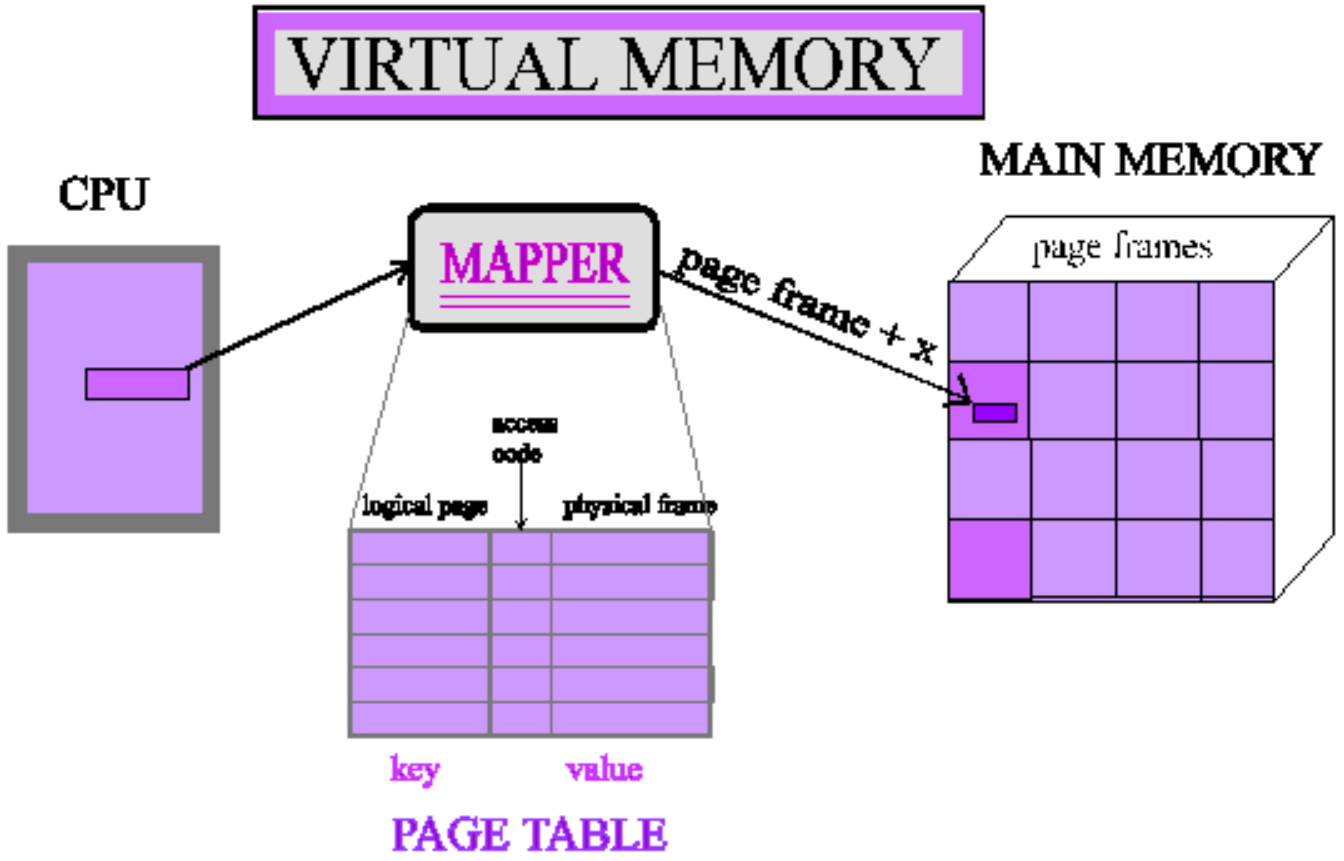


优化

- 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。

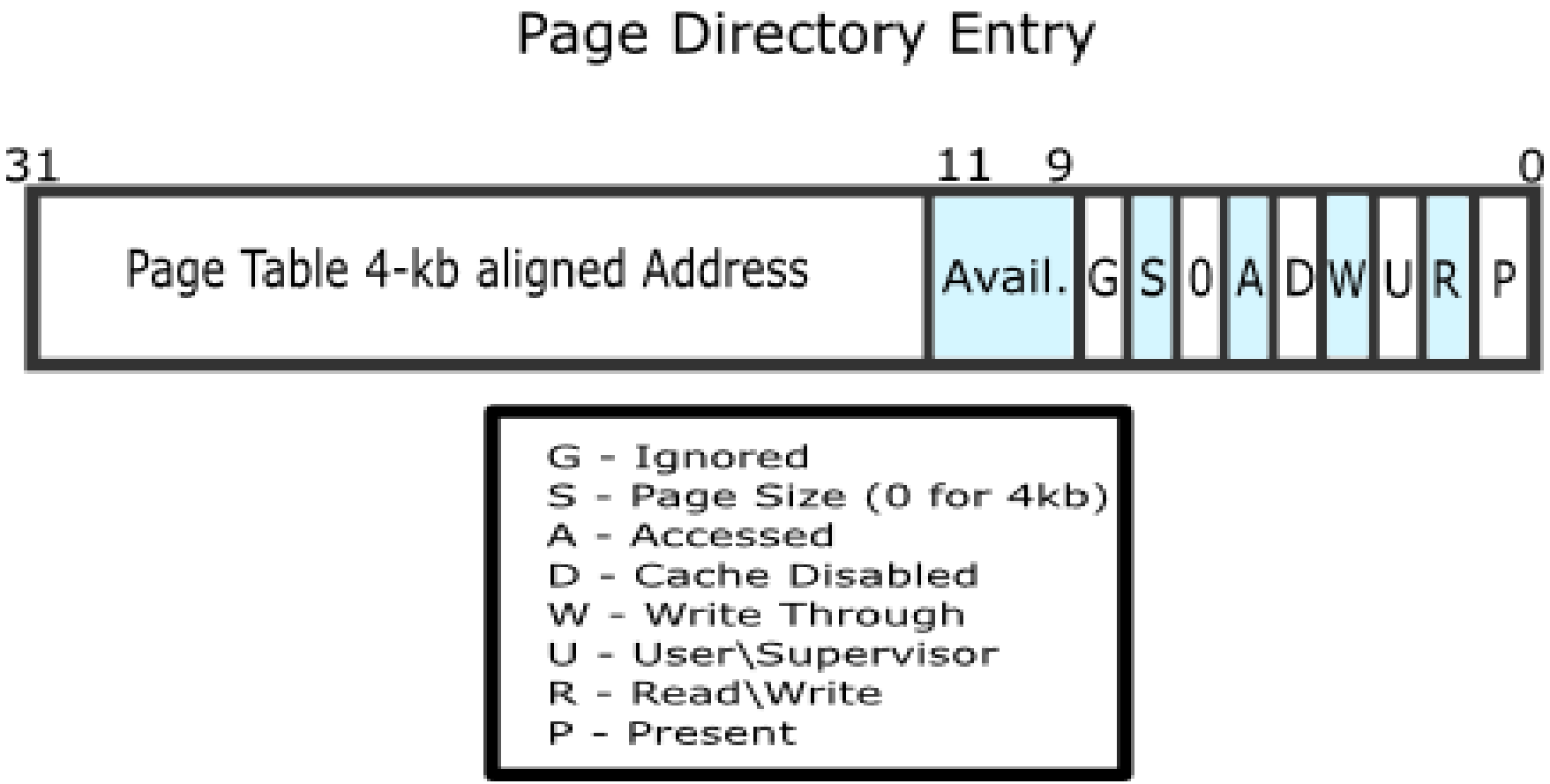
什么是页表

- 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**)



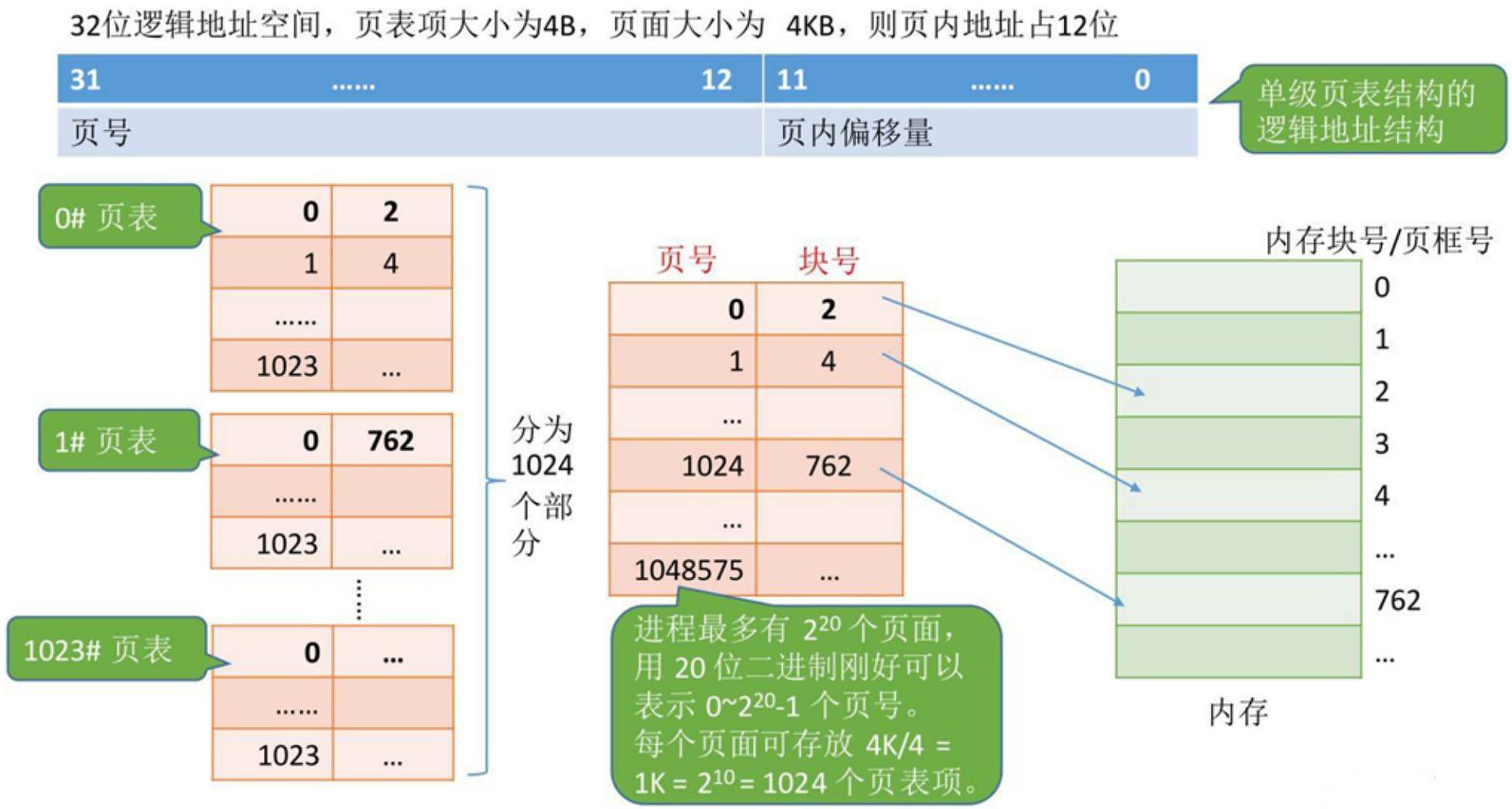
页表项结构

- 如图为32位x86页表项的具体结构。



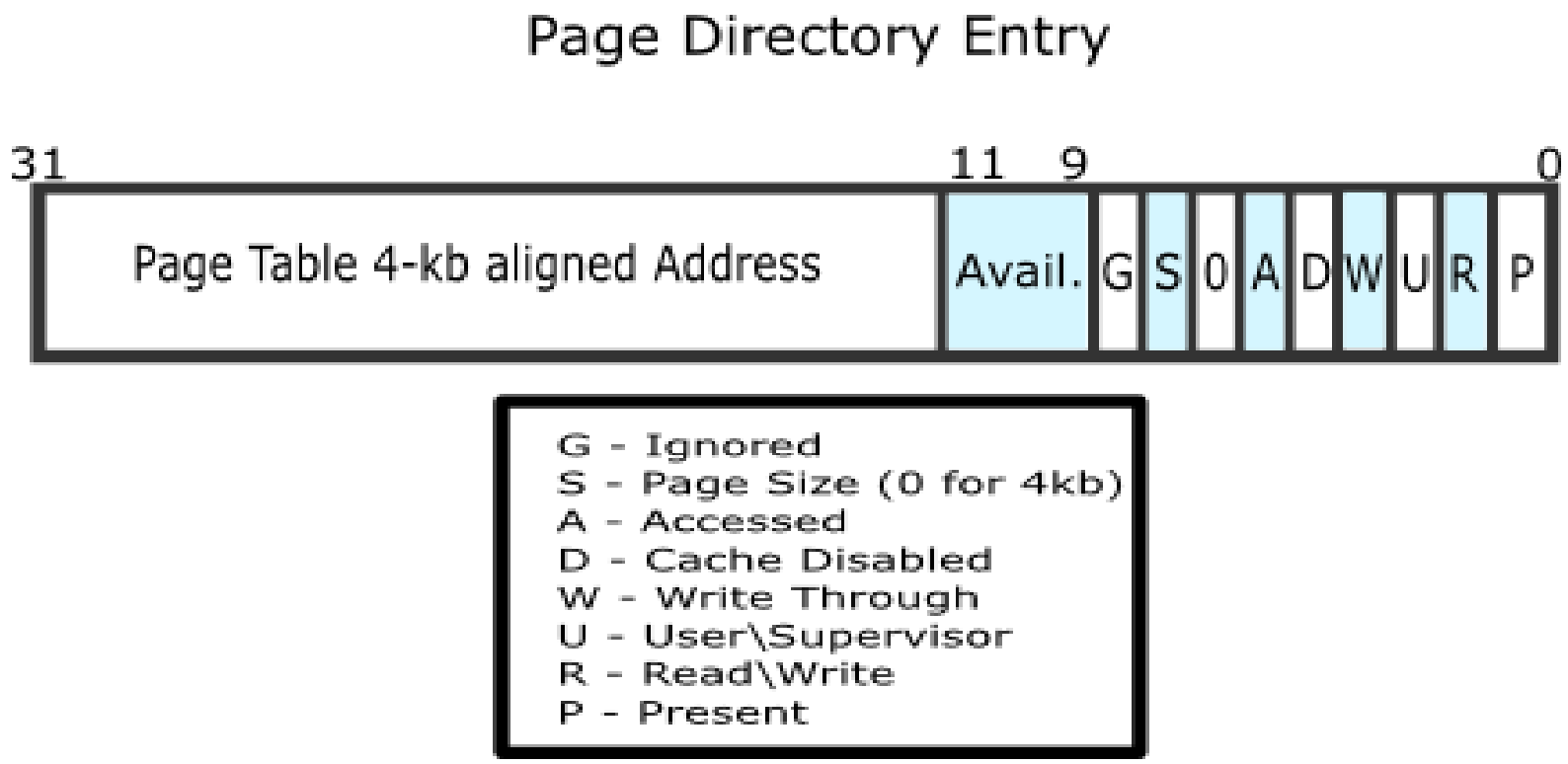
单级页表结构

- 在32位的地址空间，通常页的大小为4KB，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。



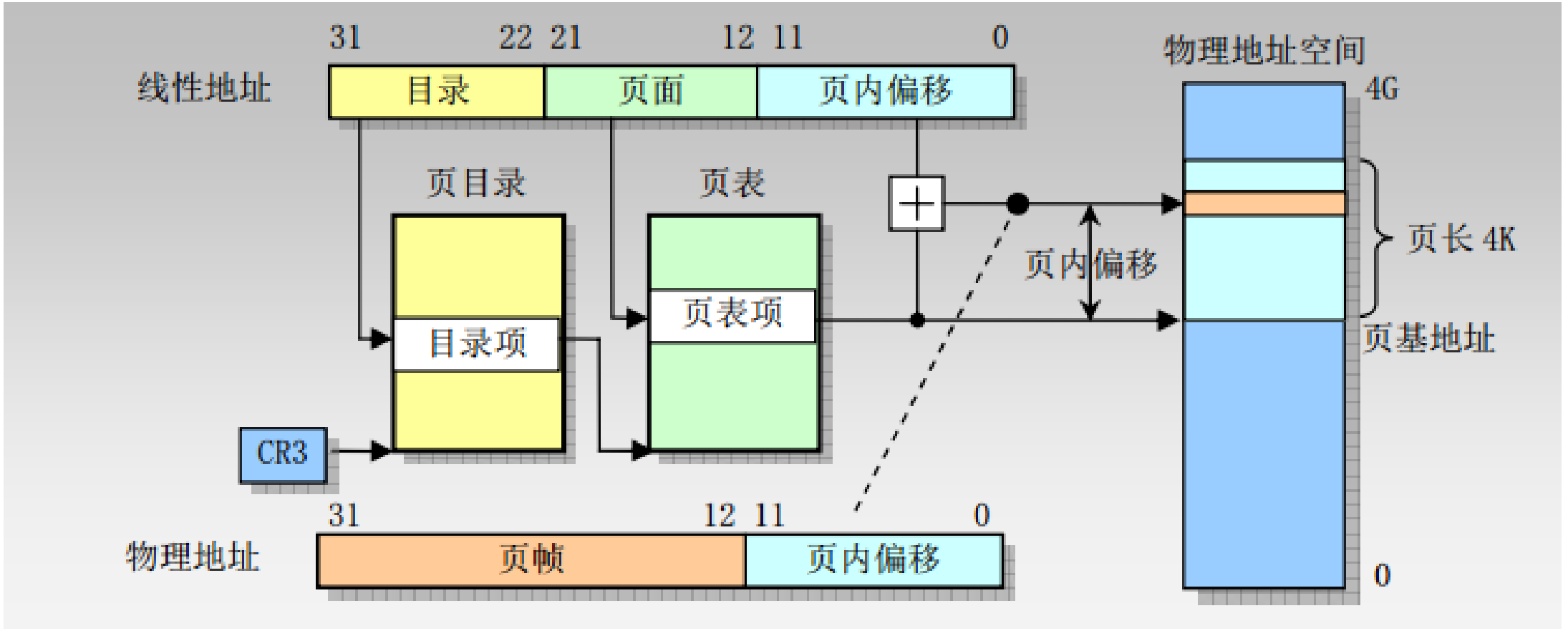
页表项结构

- 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。



两级页表

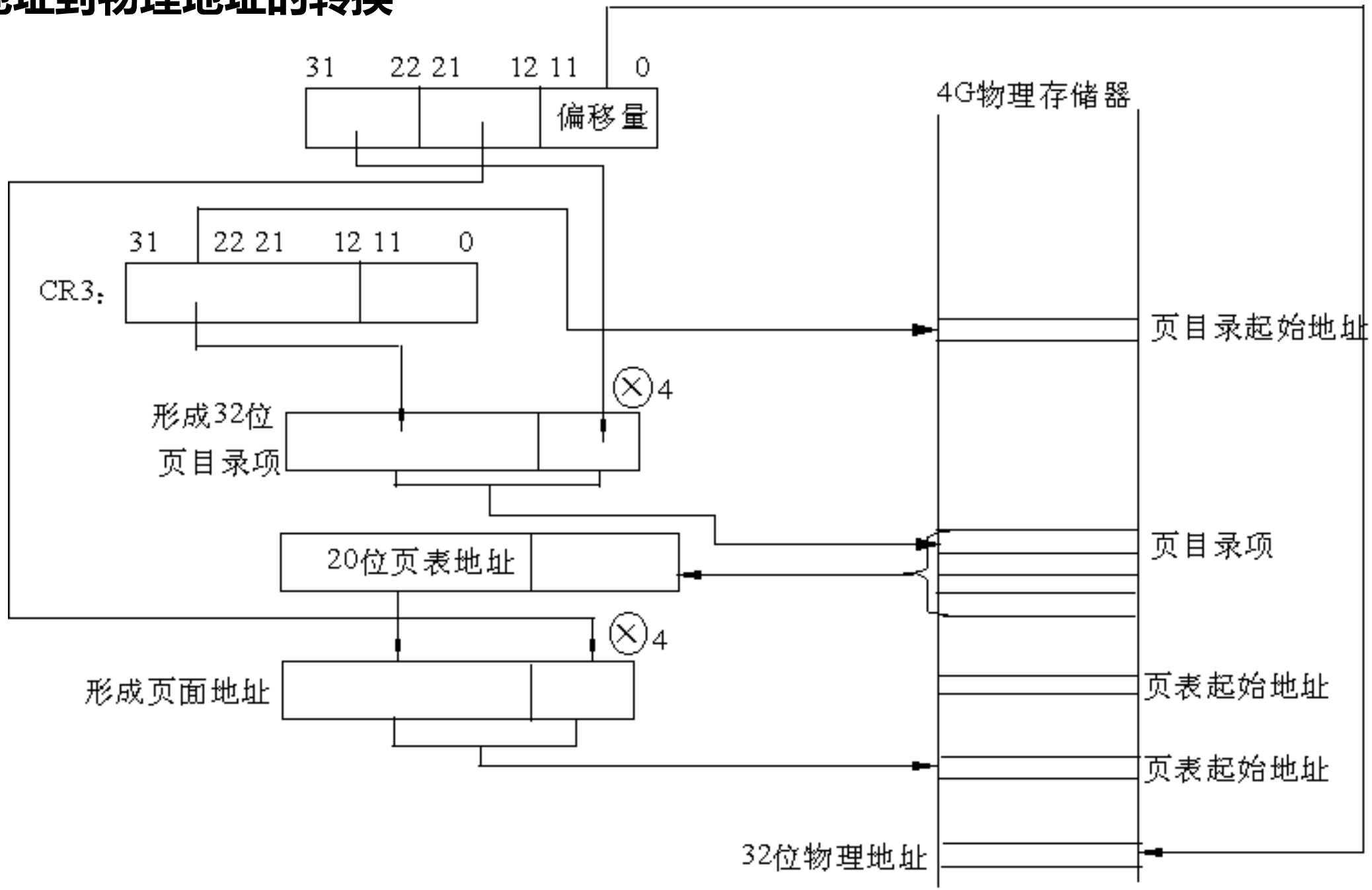
- 如果只用一级页表，因为每个页表最大可占4MB的空间，而且必须连续，这就为内存的分配带来困难，可以采用分而治之的原则，将高20位分为两部分，分别占10位，形成两级页表。
- 两级页表进行地址转换的方法如下图所示。



地址转换过程

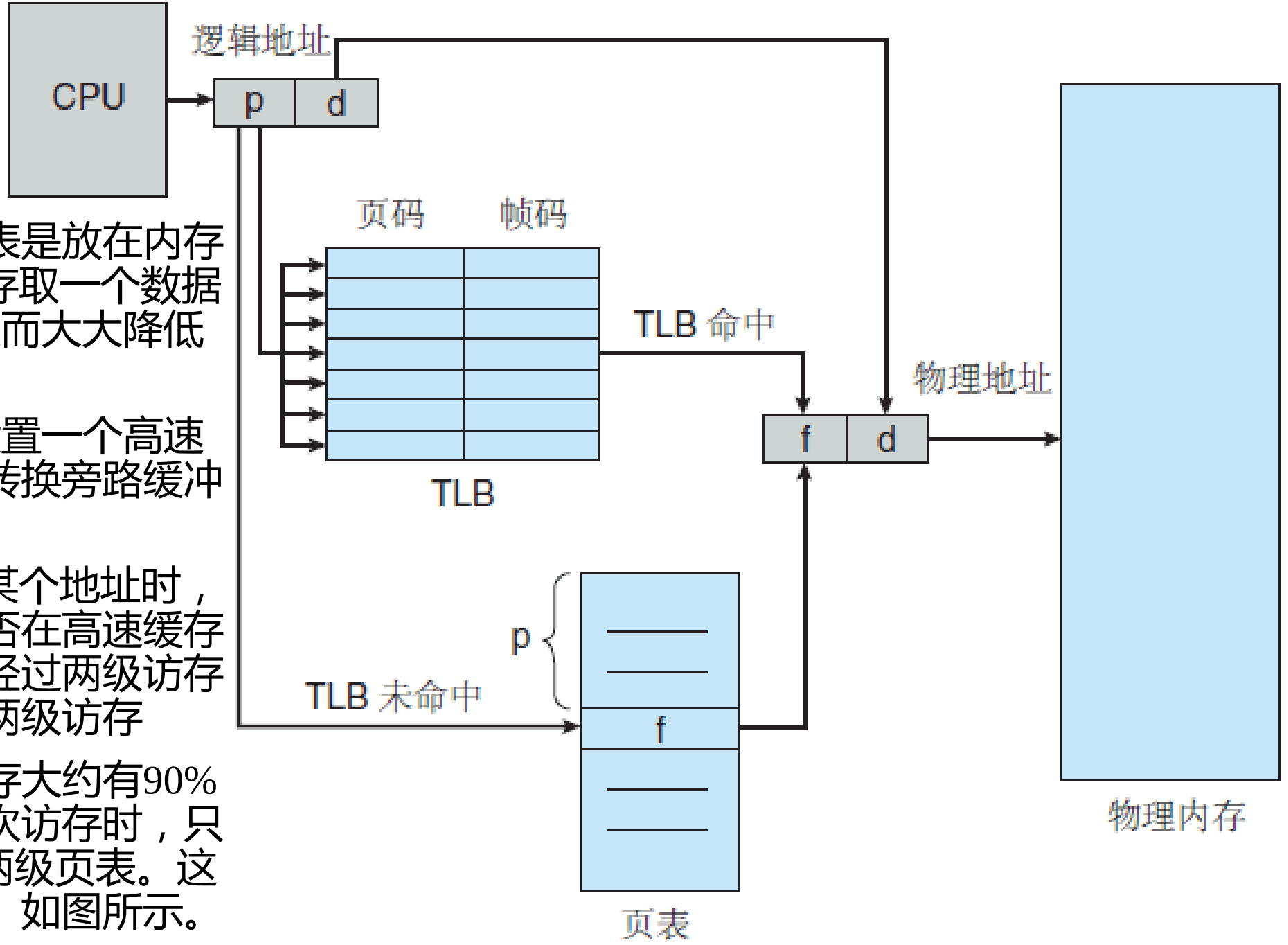
- 第一步，用最高10位作为页目录项的索引，将它乘以4，与CR3中的页目录的起始地址相加，获得相应目录项在内存的地址。
- 第二步，从这个地址开始读取32位页目录项，取出其高20位，再给低12位补0，形成页表在内存的起始地址。
- 第三步，用中间的10位作为页表中页表项的索引，将它乘以4，与页表的起始地址相加，获得相应页表项在内存的地址。
- 第四步，从这个地址开始读取32位页表项，取出其高20位，再将线性地址的第11~0位放在低12位，形成最终32位页面物理地址。
- 如后图所示。

线性地址到物理地址的转换



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。
- 为提高速度，在x86中设置一个高速缓存硬件机制，也叫“转换旁路缓冲器”TLB。
- 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。如图所示。

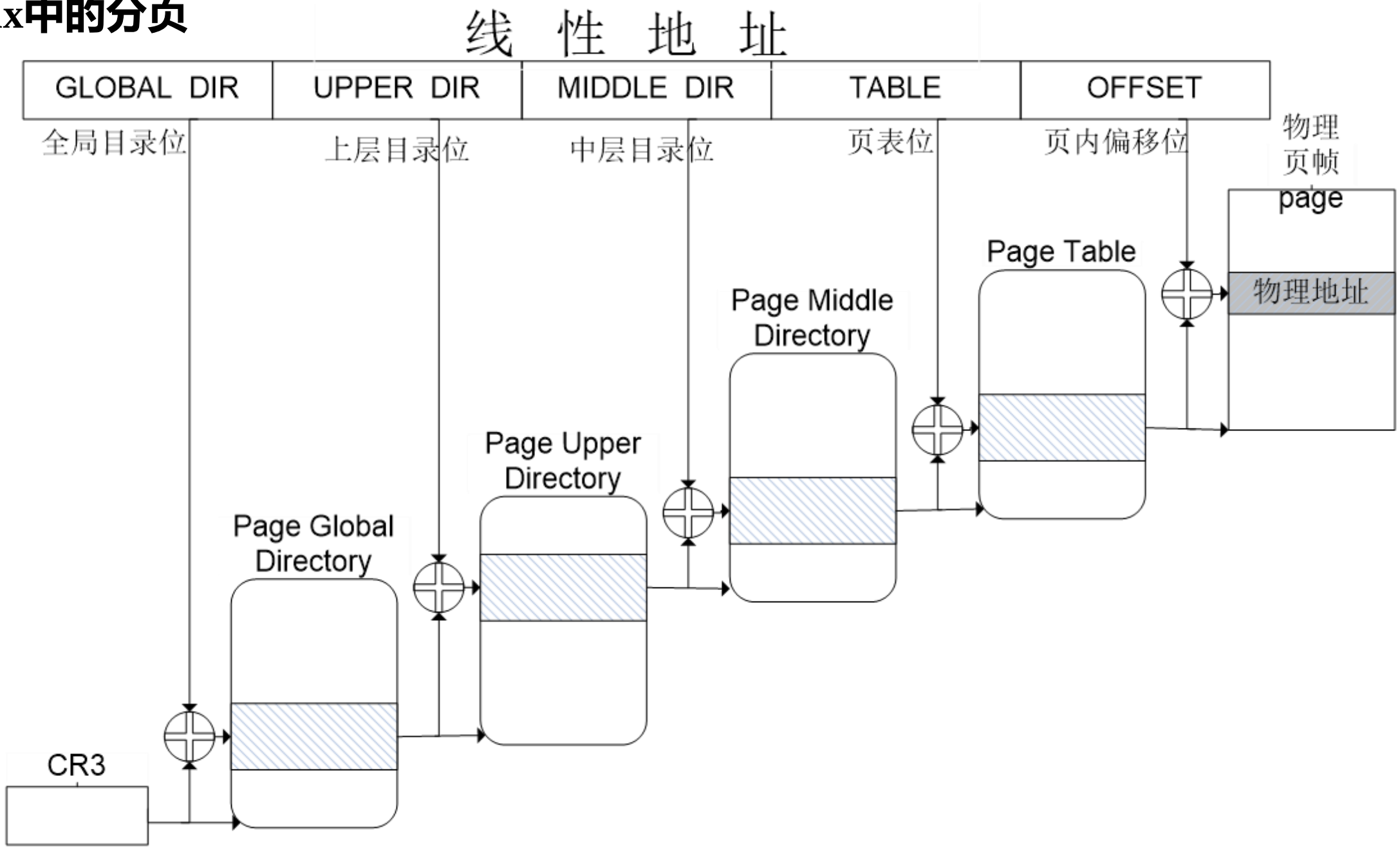


Linux中的分页

- Linux主要采用分页机制，来实现虚拟存储器管理，
- 这是因为以下两个原因：
- (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，Linux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 Directory)
  - 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


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

- 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

课程思政

课程思政

运用科学、辩证的观点和思想方法分析问题

20世纪90年代初期，基于MINIX和UNIX思想而研发的开源Linux系统面市，其是一款支持多用户、多任务、多线程和多内核的操作系统，不仅能够运行UNIX工具软件、应用程序和网络协议，还具有稳定的系统性能。发展至今，Linux已有上百种不同的发行版本。

在Linux系统稳步发展过程中，Windows系统亦不分昼夜地进行着功能完善、界面美化以及版本更新等工作。进入21世纪后，微软公司的Windows系统在个人计算机领域基本占领了垄断地位。

由垄断所导致的潜在安全问题是各国相关部门尤为关心的核心问题，而解决该问题（即去微软公司化）的主流途径便是采用开源Linux系统。

运用科学、辩证的观点和思想方法分析问题

但是，要想简单地通过采用Linux系统实现去微软公司化，实属不易。例如：2004年，德国慕尼黑政府宣布将政府办公计算机中所采用的Windows系统换为Linux系统，然而10年之后试验并未获得预期的效果。

从此类案例中可以看出，单纯地通过采用Linux系统+开源软件的模式来降低运维信息化成本，效果并不理想，后期维护工作量大，原因是采用Linux系统实现去微软公司化缺乏相应的产业基础。

但我们也绝对不应放弃反垄断！在此情形下，我们知识分子与科技人才更应运用科学、辩证的观点和思想方法分析问题：究竟应当如何解决操作系统垄断问题，以及如何在解决该问题的过程中，通过发挥我们自身的价值，助力研发自主可控的国产操作系统？

课后练习题

1、图灵机是一种通用自动机器模型，请对图灵机的理念做一个简述。
2、请简述8位时期的X86内存寻址的不同时期有何特点？
3、请简述X86内存寻址的四个不同时期是什么，各时期有何特点？
4、请简述相对于之前的模式，保护模式下的寄存器有什么大的变化？
5、请按顺序对程序从编译到执行过程中的各步骤做一个介绍。
6、MMU的转换分为两个阶段，请对其中的分段机制做一个介绍。
7、请简述分页机制中的页与块。
8、页表是一种映射机制，其中存放的是什么之间的映射？
9、请简述两级页表如何进行地址转换？
10、页面高速缓存有什么作用，请简述。
11、Linux定义了四种类型的页表，请简述是哪四种？

谢 谢！

THANKS