

# 内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
  - 基本原理
  - 地址变换
  - 段共享
  - 与页式管理优缺点对比
  - 段页式管理
- 虚拟存储管理
- 存储管理实例

# 段式存储管理

## 方便编程：

- 通常一个作业是由多个程序段和数据段组成的，用户一般按逻辑关系对作业分段，并能根据名字来访问程序段和数据段。

## 信息共享：

- 共享是以信息的逻辑单位为基础的。页是存储信息的物理单位，段却是信息的逻辑单位。
- 页式管理中地址空间是一维的，主程序，子程序都顺序排列，共享公用子程序比较困难，一个共享过程可能需要几十个页面。

# 段式存储管理

## 信息保护：

- 页式管理中，一个页面中可能装有 2 个不同的子程序段的指令代码，不能通过页面共享实现共享一个逻辑上完整的子程序或数据块。
- 段式管理中，可以以信息的逻辑单位进行保护。

## 动态增长：

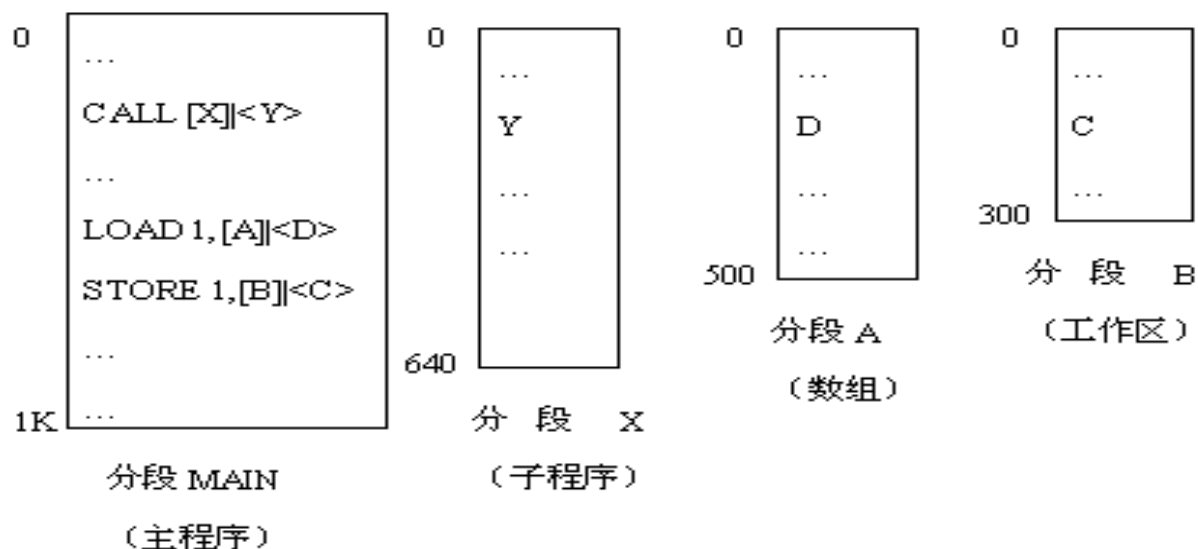
- 实际应用中，某些段（数据段）会不断增长，前面的存储管理方法均难以实现。

## 动态链接：

- 动态链接在程序运行时才把主程序和要用到的目标程序（程序段）链接起来。

# 分段地址空间

一个段可定义为一组逻辑信息，每个作业的地址空间是由一些分段构成的（由用户根据逻辑信息的相对完整来划分），每段都有自己的名字（通常是段号），且都是一段连续的地址空间，首地址为0。



# 地址结构

## 段表

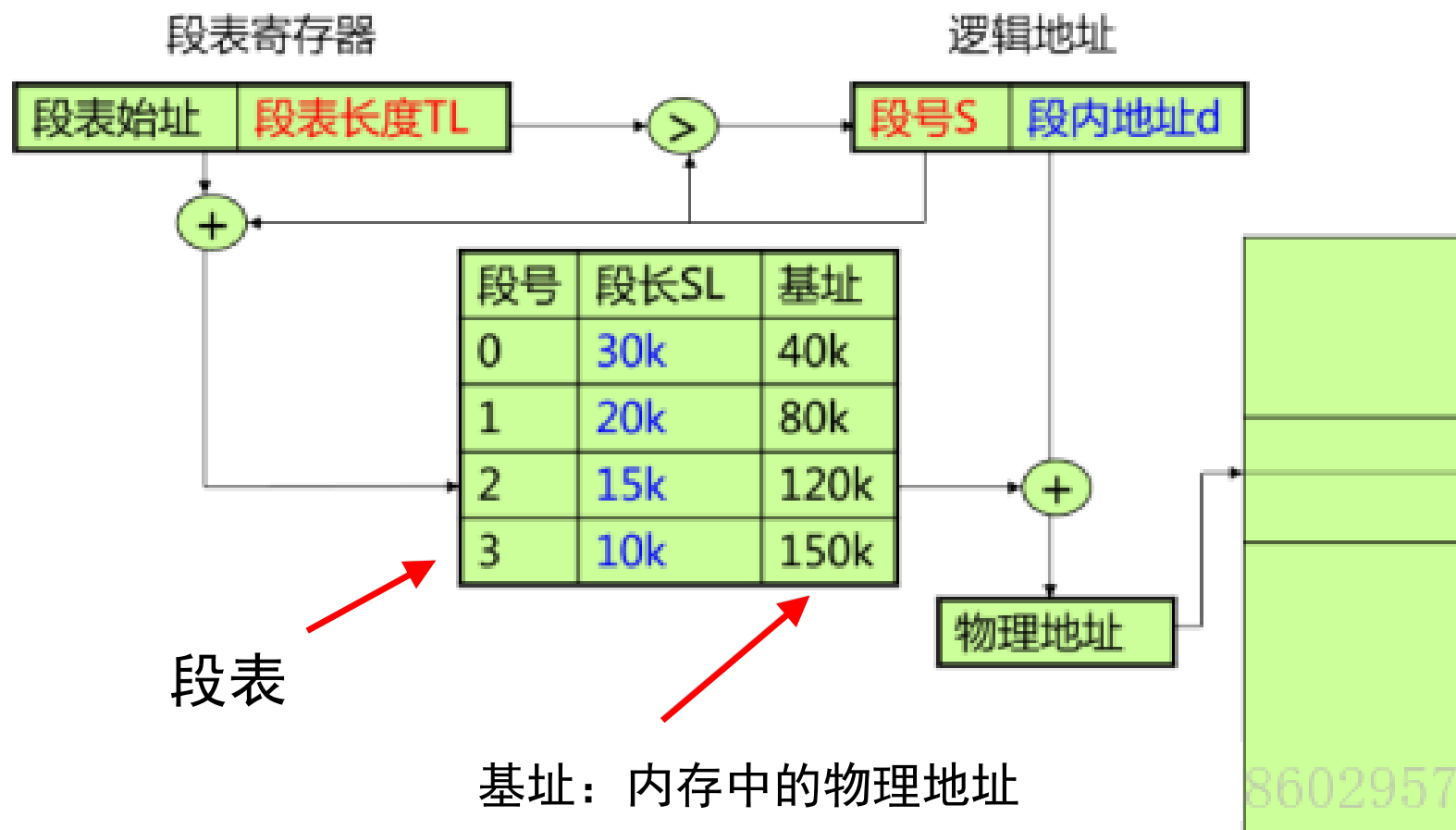
- 段表记录了段与内存位置的对应关系。
- 段表保存在内存中。
- 段表的基址及长度由段表寄存器给出。

段表始址	段表长度
------	------

- 访问一个字节的数/指令需访问内存两次 (段表一次，内存一次)。
- 逻辑地址由段和段内地址组成。

段号	段内地址
----	------

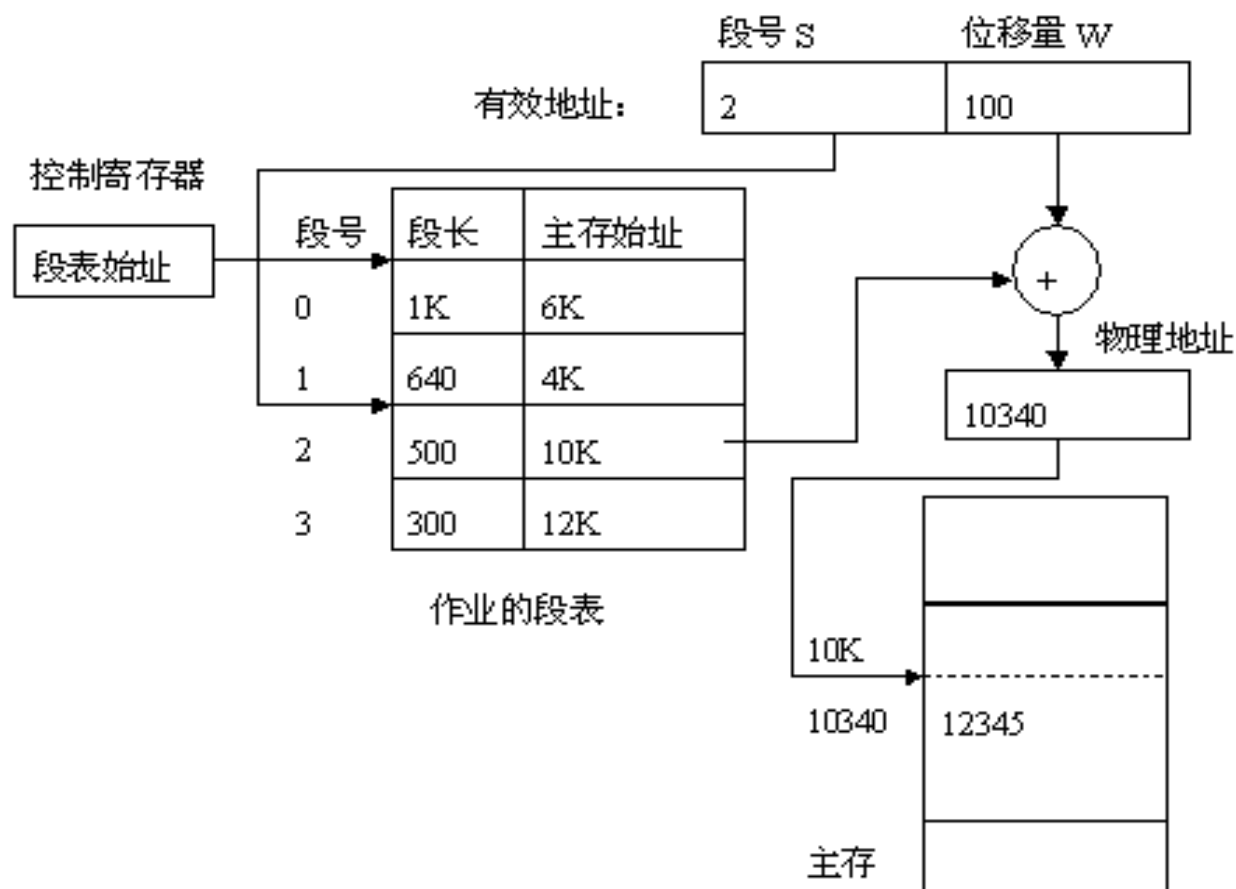
# 地址变换机构



# 地址变换过程

1. 系统将逻辑地址中的段号  $S$  与段表长度  $TL$  进行比较。
  - 若  $S > TL$ ，表示段号太大，是访问越界，于是产生越界中断信号。
  - 若未越界，则根据段表的始址和该段的段号，计算出该段对应段表项的位置，从中读出该段在内存的始址。
2. 再检查段内地址  $d$ ，是否超过该段的段长  $SL$ 。
  - 若超过，即  $d > SL$ ，同样发出越界中断信号。
  - 若未越界，则将该段的基址与段内地址  $d$  相加，即可得到要访问的内存物理地址。

# 地址变换过程





# 信息共享

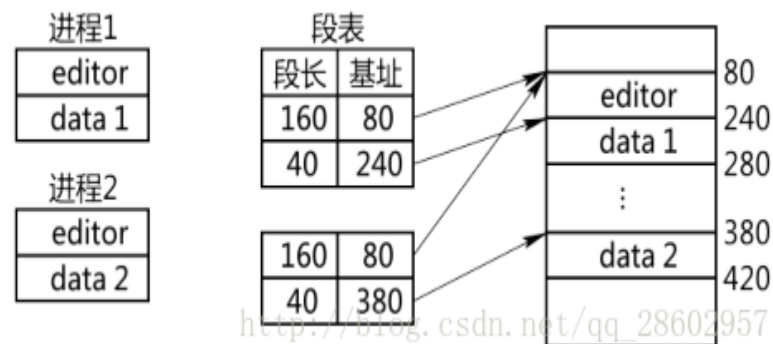
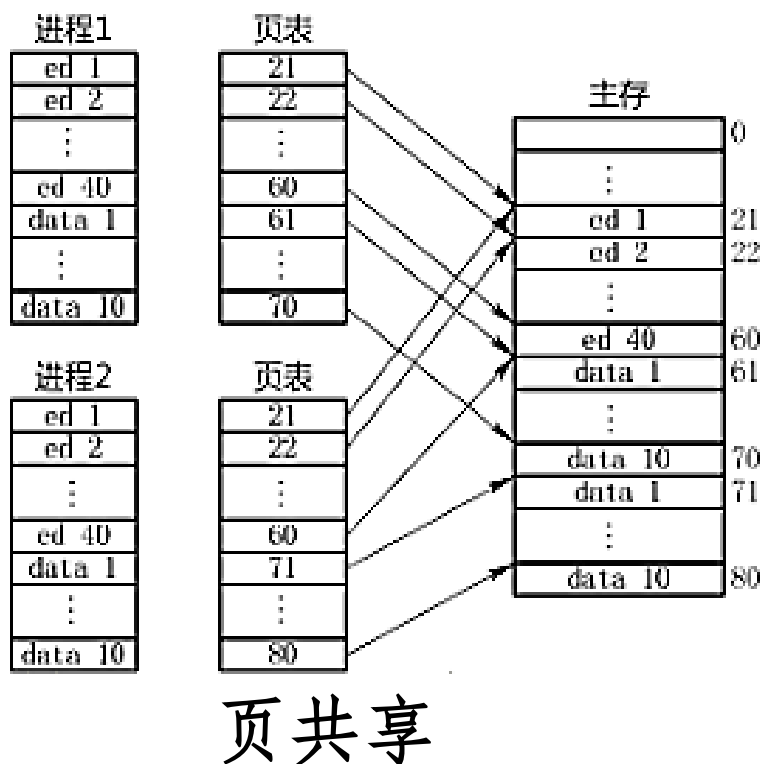
例：一个多用户系统，可同时接纳 40 个用户，都执行一个文本编辑程序 (Text Editor)。如果文本编辑程序有 160KB 的代码和另外 40 KB 的数据区，如果不共享，则总共需有 8 MB 的内存空间来支持 40 个用户。

如果 160 KB 的代码是**可重入**的，则无论是在分页系统还是在分段系统中，该代码都能被共享。因此在内存中只需保留一份文本编辑程序的副本，此时所需的内存空间仅为 1760 KB( $40 \times 40 + 160$ )，而不是  $(160 + 40) \times 40 = 8000$  KB。

可重入代码(Reentrant Code) 又称为“纯代码”(Pure Code)，是一种允许多个进程同时访问的代码。为使各个进程所执行的代码完全相同，绝对不允许可重入代码在执行中有任何改变。因此，可重入代码是一种不允许任何进程对它进行修改的代码。

# 分页与分段共享比较

- 例子中，若采用分页共享，每个进程要使用40个页表项共享160K的editor；在分段系统中，实现共享容易得多，只需在每个进程的段表中为文本编辑程序设置一个段表项。



# 分段管理的优缺点

## ■ 优点：

- 分段系统易于实现段的共享，对段的保护也十分简单。

## ■ 缺点：

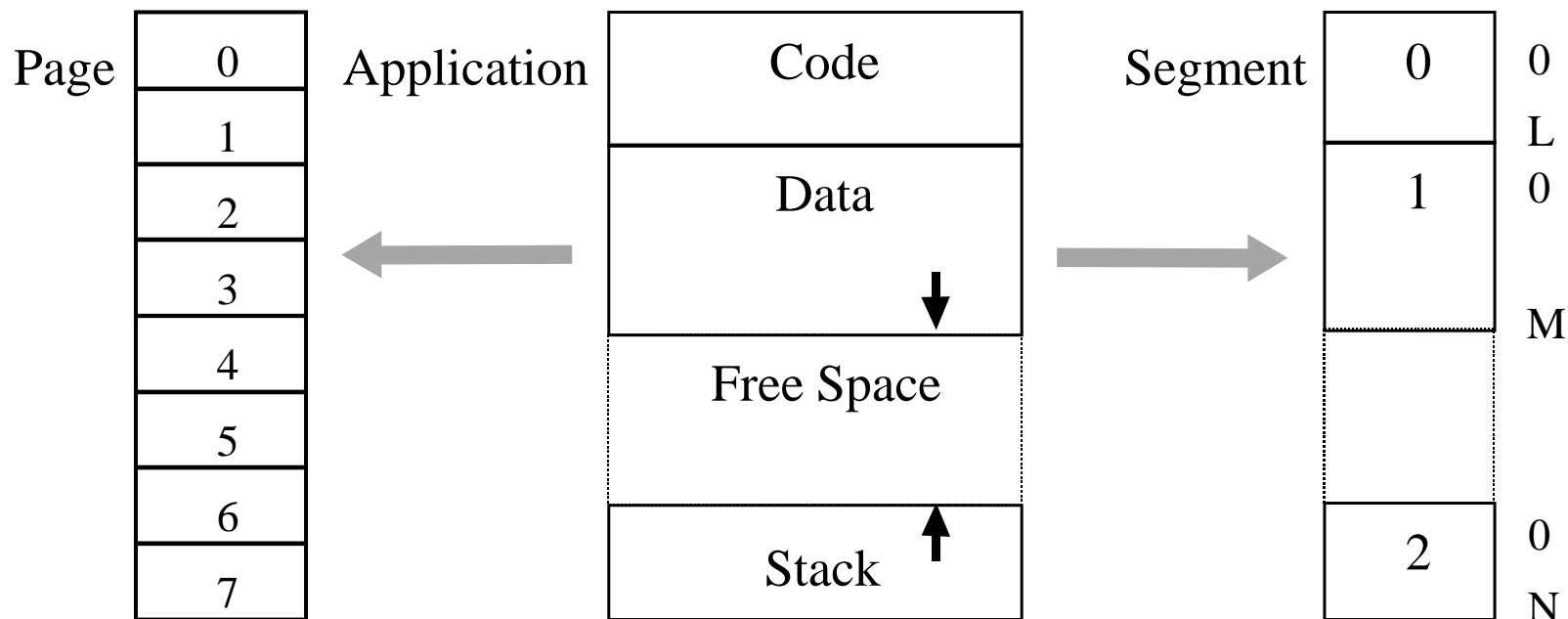
- 处理机要为地址变换花费时间；要为表格提供附加的存储空间。
- 为满足分段的动态增长和减少外零头，要采用拼接手段。
- 在辅存中管理不定长度的分段困难较多。
- 分段的最大尺寸受到主存可用空间的限制。

# 分页与分段的比较

- 分页的作业的地址空间是单一的线性地址空间，分段作业的地址空间是二维的。
- “页”是信息的“物理”单位，大小固定。  
“段”是信息的逻辑单位，即它是一组有意义的信息，其长度不定。
- 分页活动用户是看不见的，而是系统对于主存的管理。分段是用户可见的（分段可以在用户编程时确定，也可以在编译程序对源程序编译时根据信息的性质来划分）。

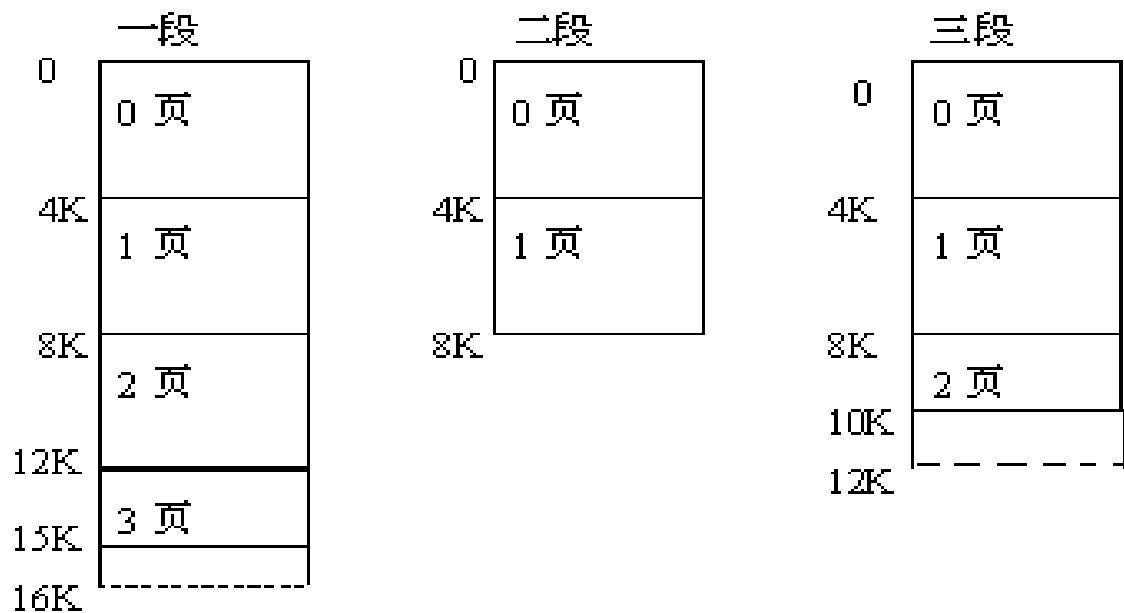
# 分页与分段的比较

	页式存储管理	段式存储管理
目的	实现非连续分配，解决碎片问题	更好地满足用户需求
信息单位	页（物理单位）	段（逻辑单位）
大小	固定（由系统定）	不定（由用户程序定）
内存分配单位	页	段
作业地址空间	一维	二维
优点	有效解决了碎片问题（没有外碎片，每个内碎片不超过页大小）；有效提高内存的利用率；程序不必连续存放。	更好地实现数据共享与保护；段长可动态增长；便于动态链接



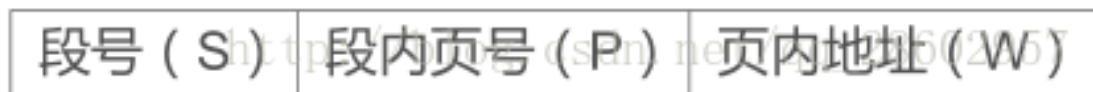
# 段页式存储管理

- 基本思想：用分段方法来分配和管理虚拟存储器，而用分页方法来分配和管理实存储器。



# 实现原理

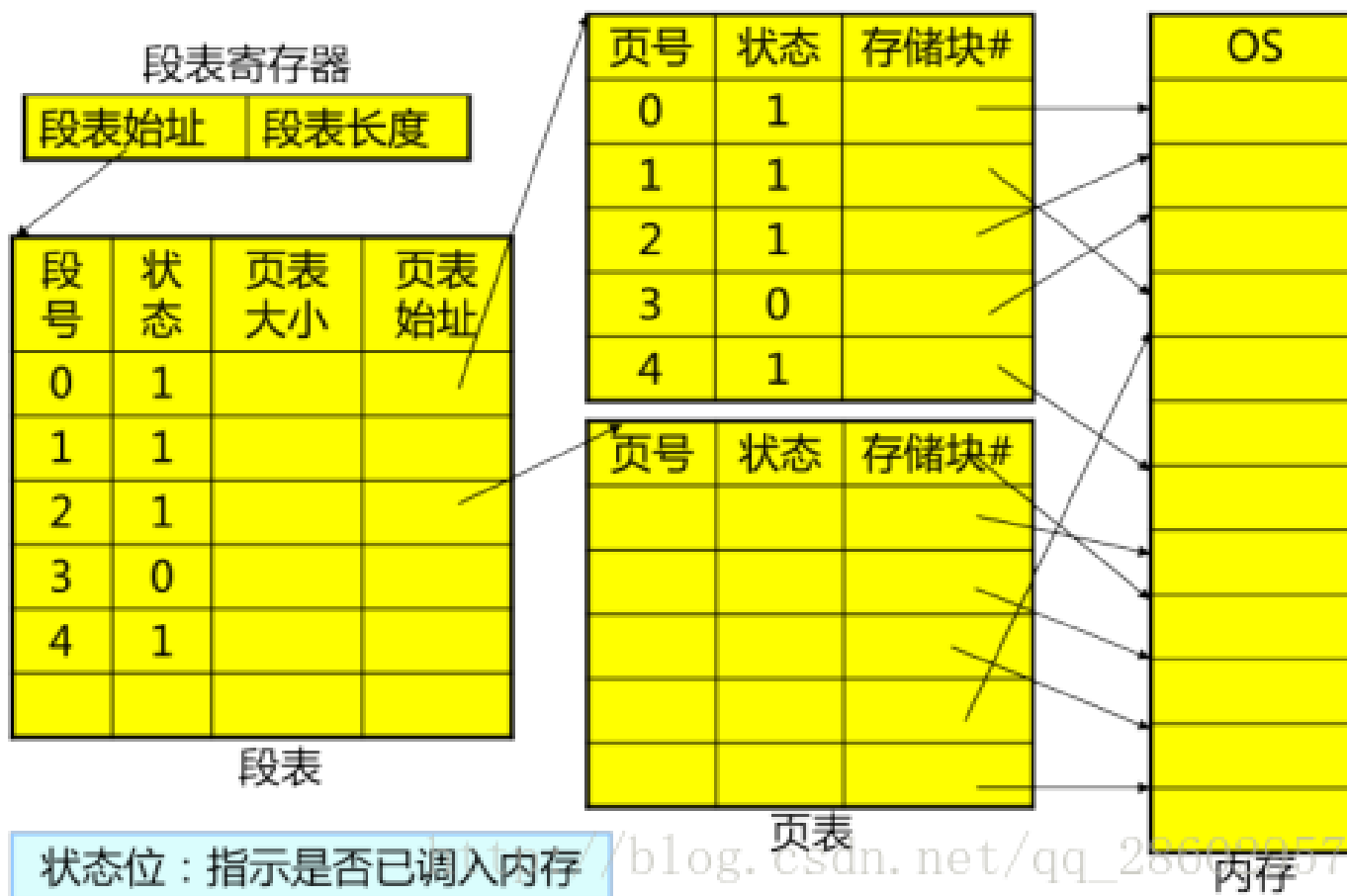
- 段页式存储管理是分段和分页原理的结合，即先将用户程序分成若干个段（段式），并为每一个段赋一个段名，再把每个段分成若干个页（页式）。
- 其地址结构由段号、段内页号、及页内位移三部分所组成。



- 系统中设段表和页表，均存放于内存中。读一字节的指令或数据须访问内存三次。为提高执行速度可增设高速缓冲寄存器。
- 每个进程一张段表，每个段一张页表。
- 段表含段号、页表始址和页表长度。页表含页号和块号。



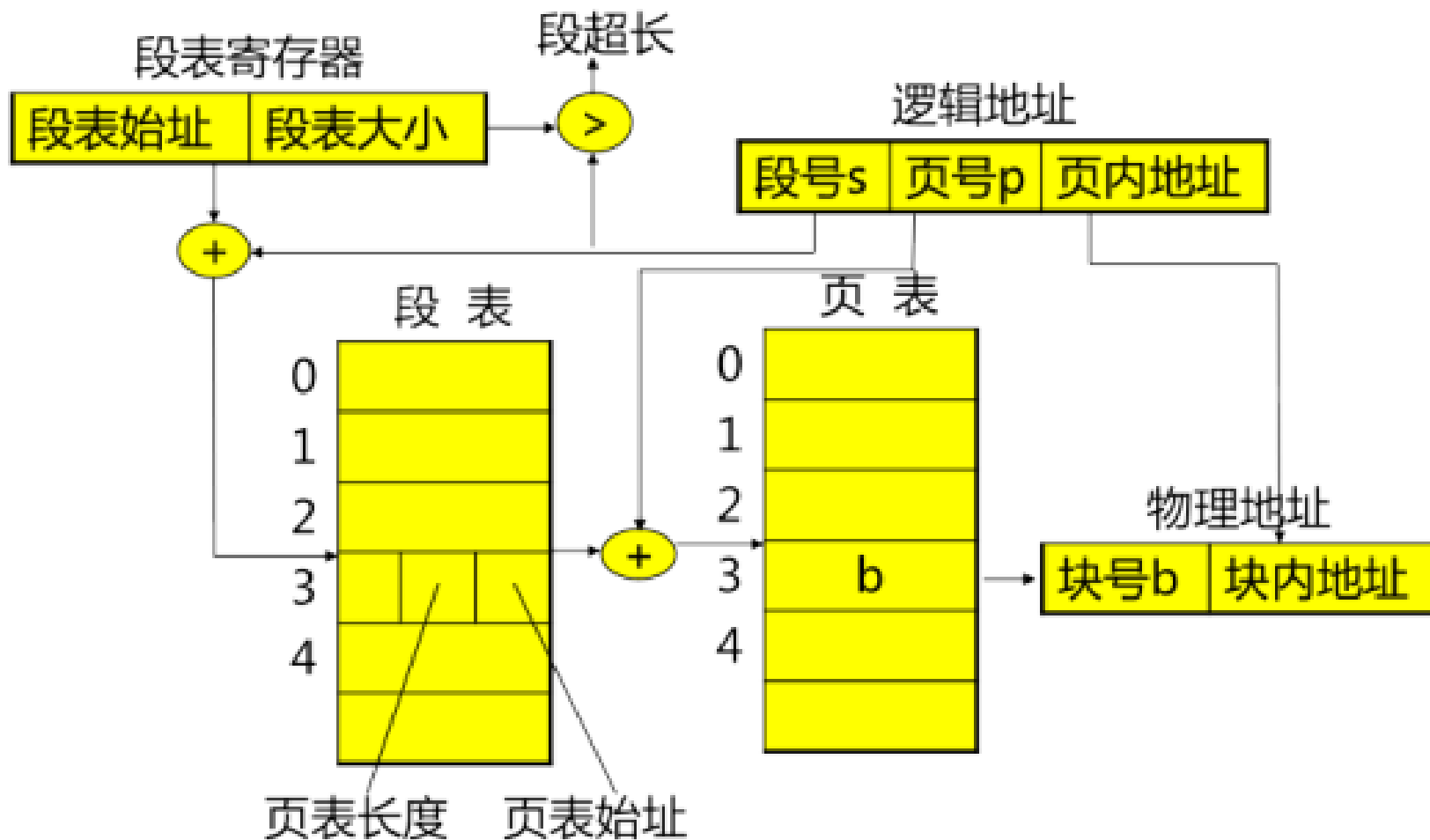
# 利用段表和页表实现地址映射



# 段页式存储管理的地址变换

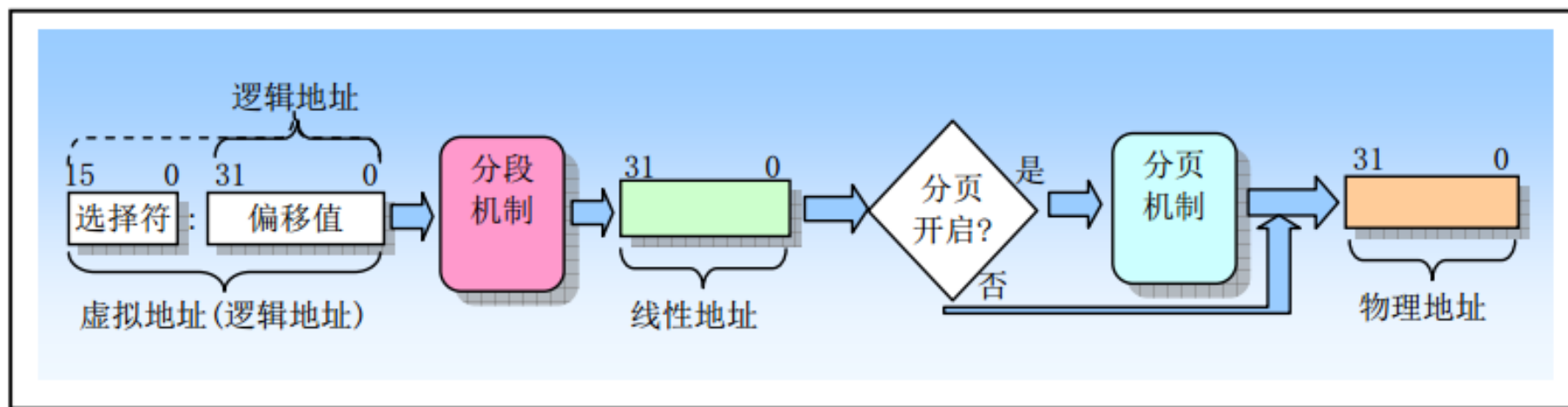
- 从 PCB 中取出段表始址和段表长度，装入段表寄存器。
- 将段号与段表长度进行比较，若段号大于或等于段表长度，产生越界中断。
- 利用段表始址与段号得到该段表项在段表中的位置。取出该段的页表始址和页表长度。
- 将页号与页表长度进行比较，若页号大于或等于页表长度，产生越界中断。
- 利用页表始址与页号得到该页表项在页表中的位置。
- 取出该页的物理块号，与页内地址拼接得到实际的物理地址。

# 段页式存储管理的地址变换

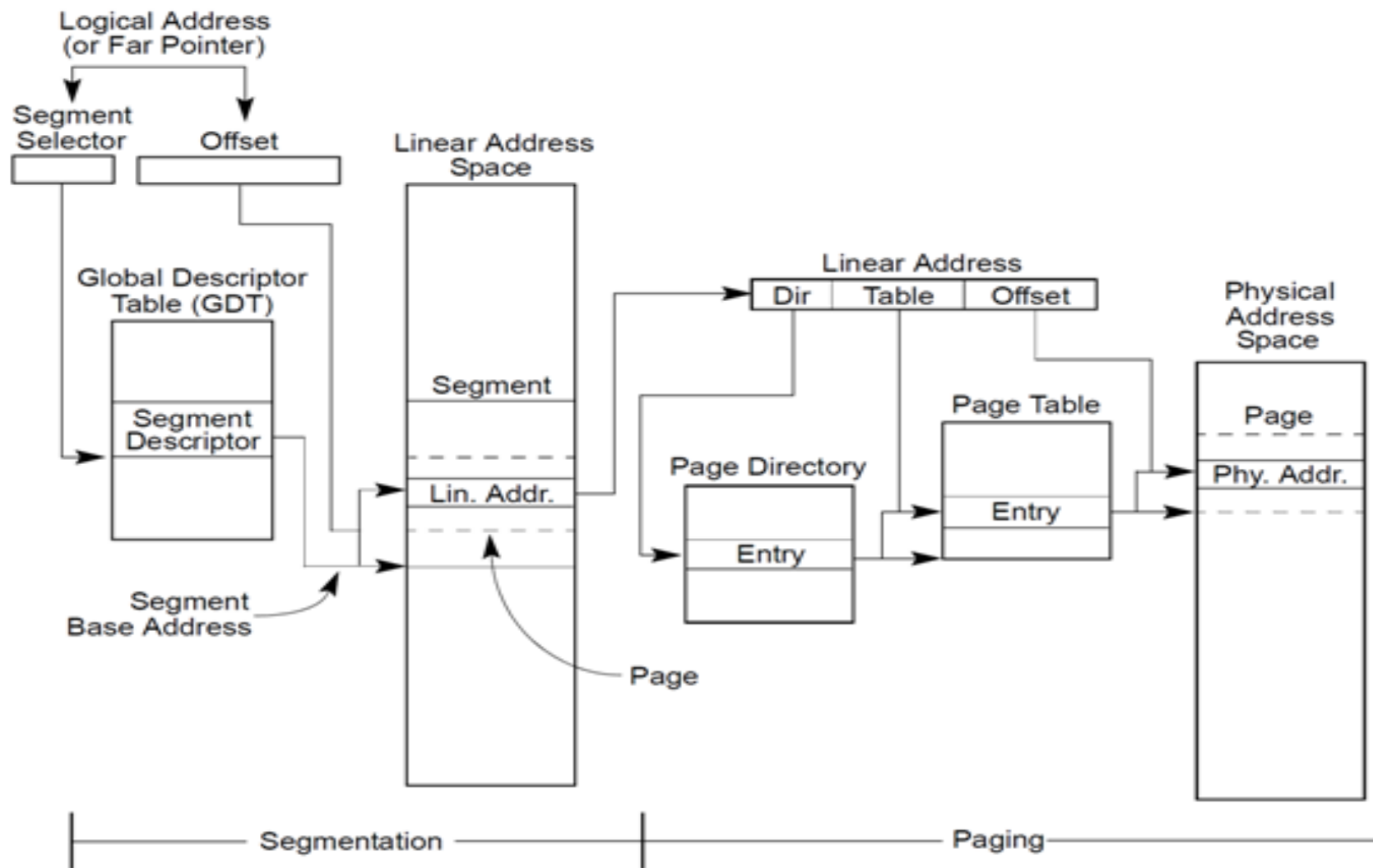


# 实例：X86的段页式地址映射

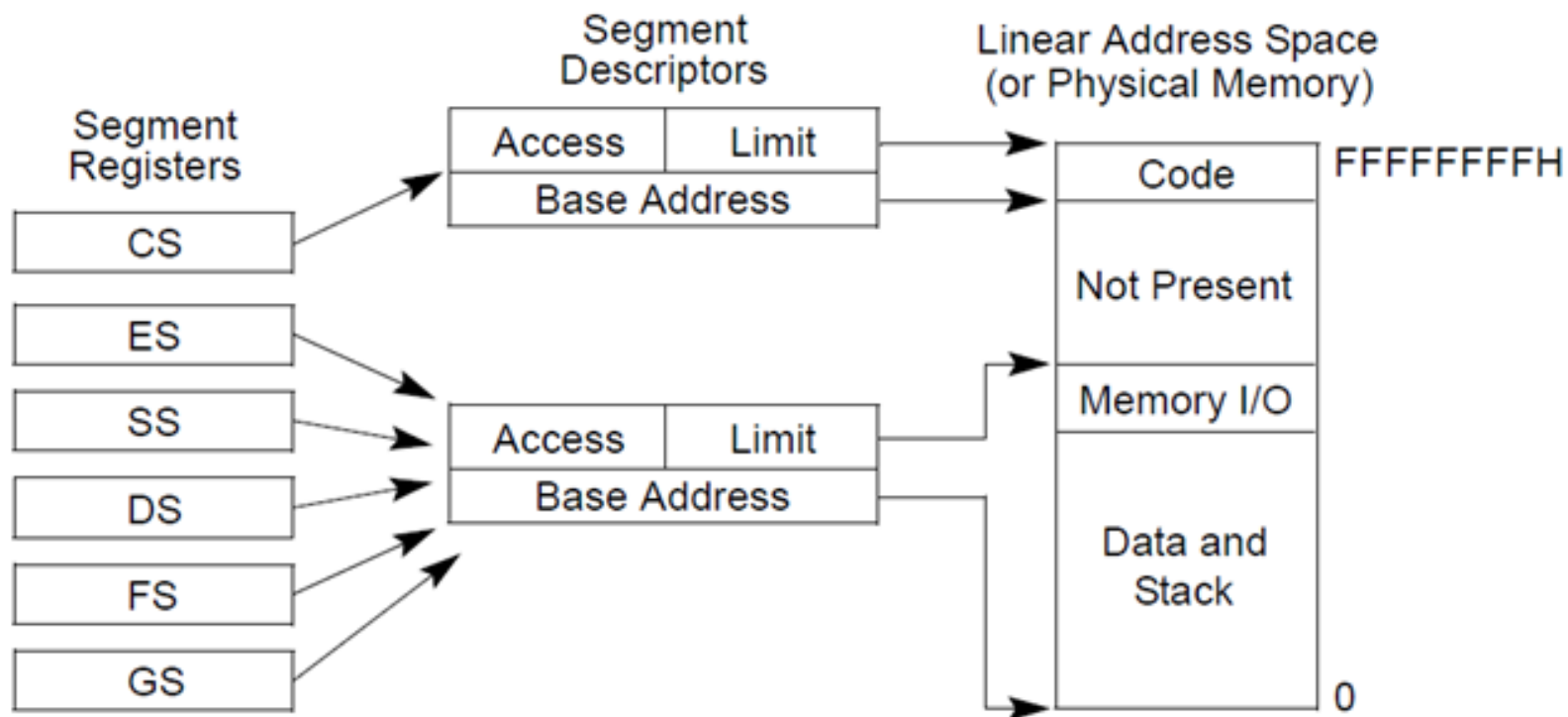
- X86的地址映射机制分为两个部分：
  - 段映射机制，将逻辑地址映射到线性地址；
  - 页映射机制，将线性地址映射到物理地址。



# X86的段页式地址映射



# 第一阶段：段式地址映射

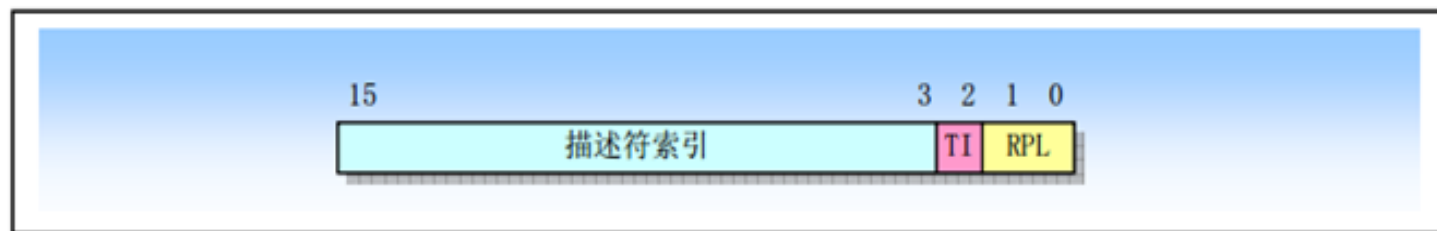


# 段式地址映射过程

1. 根据指令的性质来确定应该使用哪一个**段寄存器**（Segment Selector），例如转移指令中的地址在代码段，而取数据指令中的地址在数据段；
2. 根据段寄存器的内容，找到相应的“**地址段描述结构**”（Segment Descriptor），段描述结构都放在一个表（Descriptor Table）中（**GDT**或**LDT**等），而表的起始地址保存在GDTR、LDTR等寄存器中。
3. 从地址段描述结构中找到基地址（Base Address）；
4. 将指令发出的地址作为位移，与段描述结构中规定的段长度相比，看看是否越界；
5. 根据指令的性质和段描述符中的访问权限来确定是否越权；
6. 将指令中发出的地址作为位移，与基地址相加而得出线性地址（Linear Address）。

# Segment Selector

- 80386之后的处理器共有6个段选择子，
  - CS寄存器：程序指令段起始地址；
  - DS寄存器：程序数据段起始地址；
  - SS寄存器：栈起始地址；
  - ES, FS, GS寄存器：额外段寄存器。



段选择符结构

TI（加载指示）：值为0处理器从GDT中加载；1则处从LDT中加载。

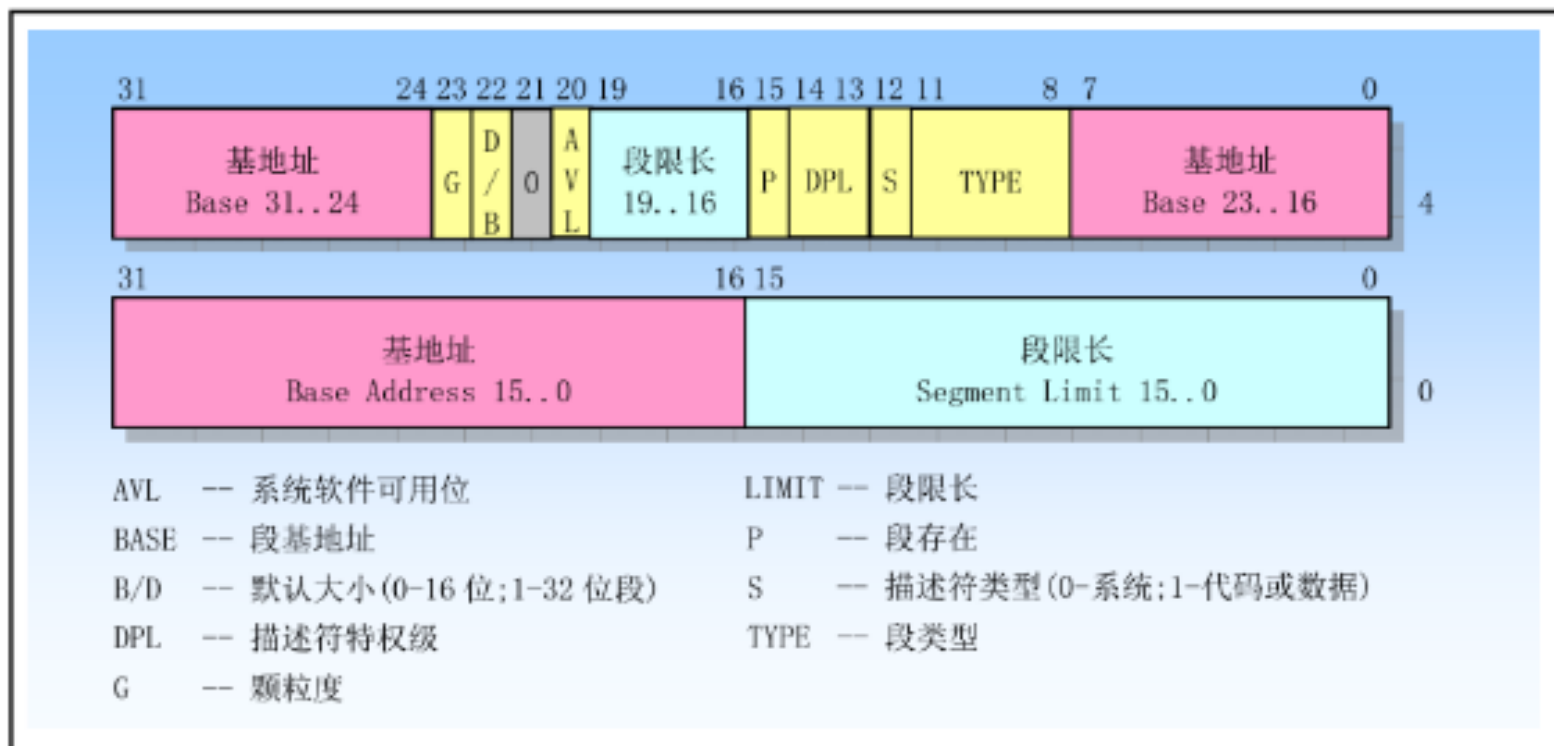
RPL（请求优先级）：00最高，11最低。



# GDT及LDT

- GDT (Global Descriptor Table): 全局描述符表，是全局性的，为所有的任务服务，不管是内核程序还是用户程序，我们都是把段描述符放在GDT中。
- LDT (Local Descriptor Table) : 局部描述符表，为了有效实施任务间的隔离，处理器建议每个任务都应该有自己的描述符表，并且把专属于这个任务的那些段描述符放到LDT中。

# Segment Descriptor



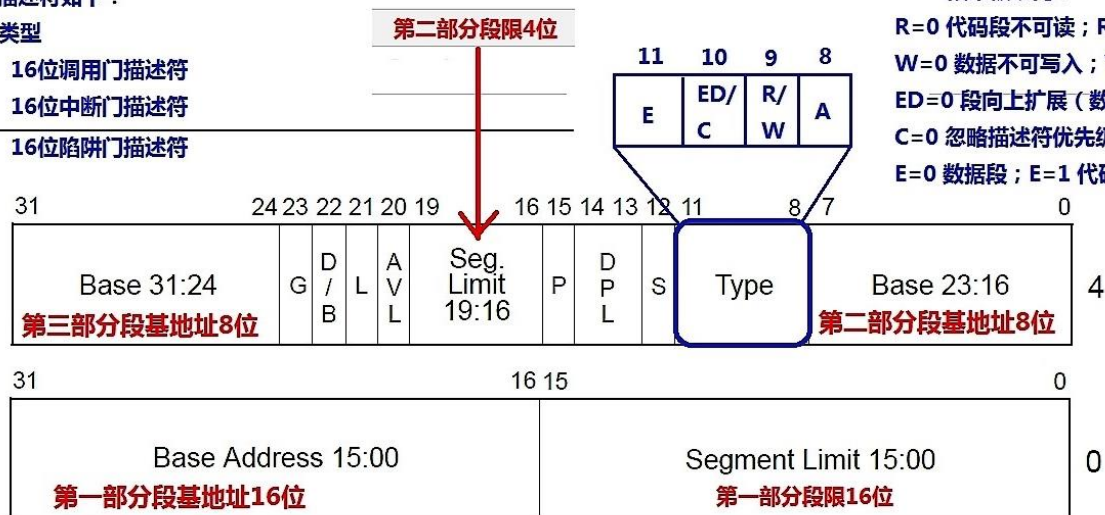
段描述符通用格式

# Segment Descriptor

Type类型域4个bits的特定组合组合表示的门描述符如下：

bit11	bit10	bit9	bit8	门描述符类型
0	1	0	0	16位调用门描述符
0	1	1	0	16位中断门描述符
0	1	1	1	16位陷阱门描述符
1100	32位调用门描述符			
1110	32位中断门描述符			
1111	32位陷阱门描述符			

所有类型门描述符的bit12位，即S位都是'0'，属于系统描述符  
调用门描述符存储在全局描述符表（GDT）中；中断门，陷阱门描述符存储在中断描述符表（IDT）中



A=0 段未被访问；A=1 段已被访问

R=0 代码段不可读；R=1 代码段可读

W=0 数据不可写入；W=1 数据可写入

ED=0 段向上扩展（数据段）；ED=1 段向下扩展（堆栈段）

C=0 忽略描述符优先级；C=1 遵循描述符优先级

E=0 数据段；E=1 代码段

L — 64-bit code segment (IA-32e mode only)

AVL — Available for use by system software

BASE — Segment base address 用于计算最终线性地址（关闭分页功能）的32位段基地址，由3部分组成

D/B — D=0 16位指令模式，使用16位偏移地址和16位寄存器；D=1 32位指令模式，使用32位偏移地址和32位寄存器

DPL — Descriptor privilege level 描述符优先级 / 描述符特权级，00=ring0，01=ring1，10=ring2，11=ring3

G — Granularity 粒度位，G=0时，段限为20位，即00000~FFFF，即1B~1MB；G=1时，段限乘以4KB，即4KB~4MB，即

LIMIT — Segment Limit 20位段限，由2部分组成

00000FFF~FFFFFFFF  
段限为32位

P — Segment present

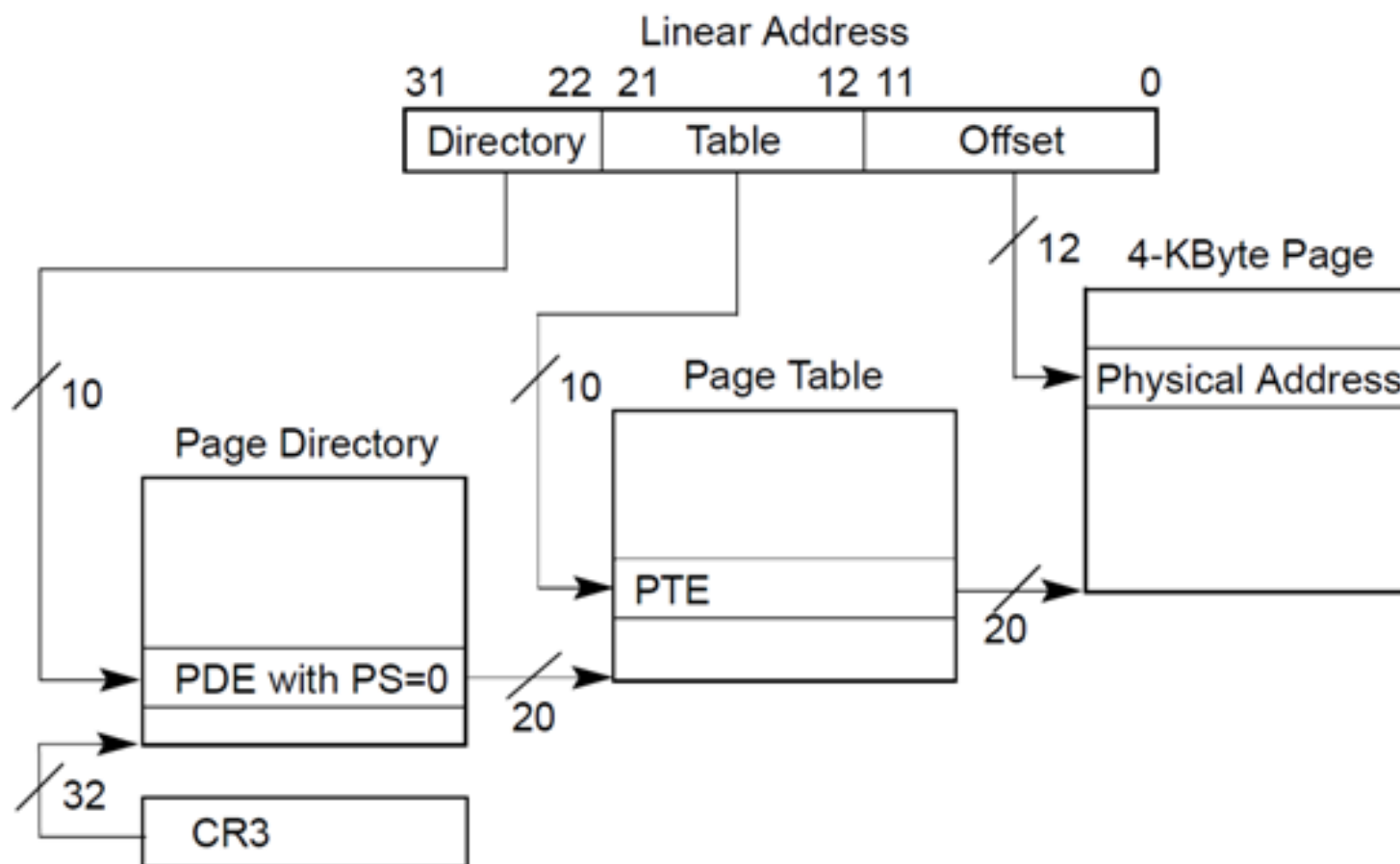
S — Descriptor type (0 = system; 1 = code or data) 段描述符类型，S=1为代码和数据段描述符，

TYPE — Segment type

S=0为系统段描述符

**Segment Descriptor 64位段描述符，分成低32位和高32位**

# 第二阶段：页式地址映射



# 页式地址映射过程

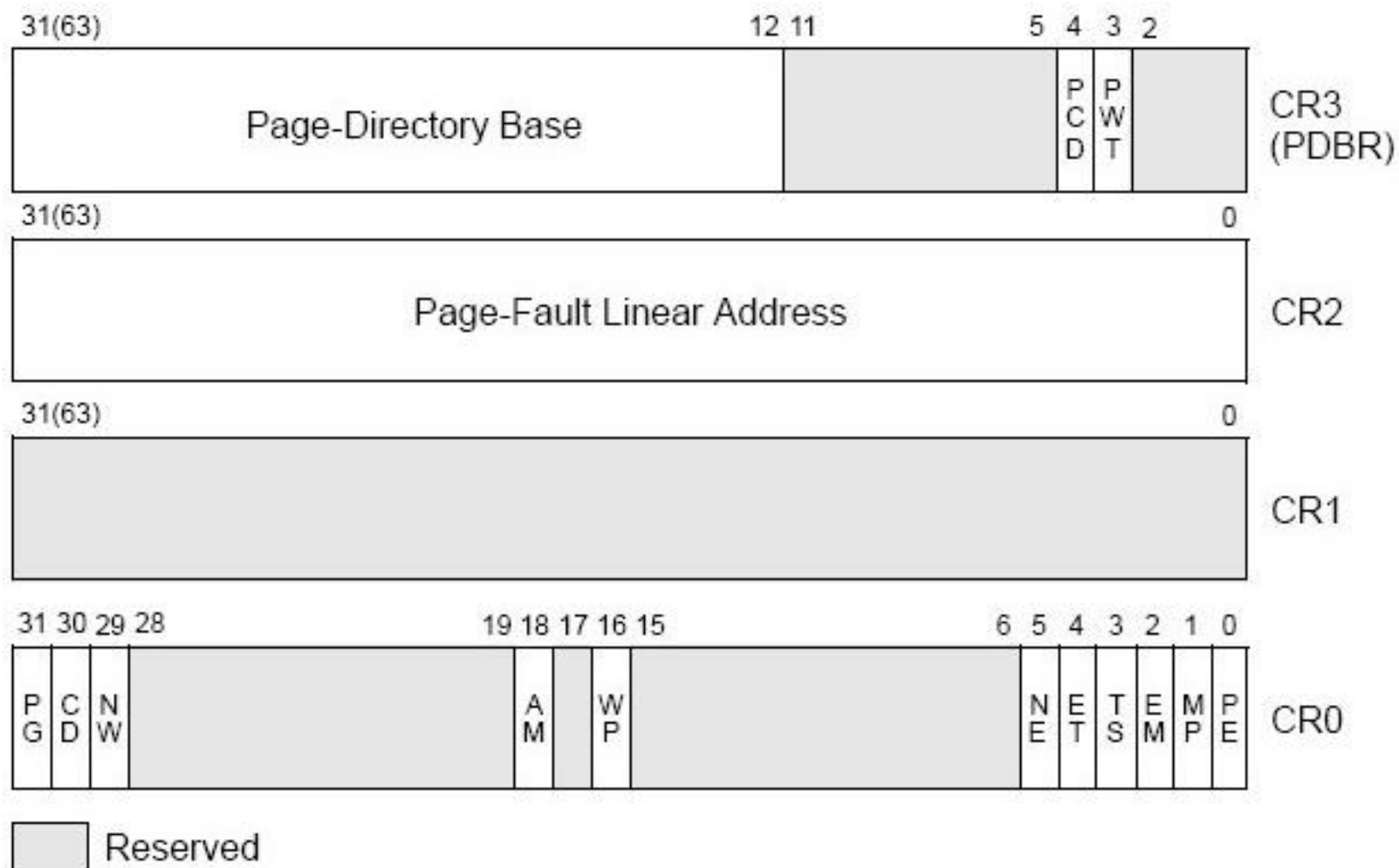
1. 从CR3寄存器中获取页面目录表（Page Directory）的基地址；
2. 以线性地址的Directory位段为下标，在目录（Page Directory）中取得相应页面表（Page Table）的基地址；
3. 以线性地址中的Table位段为下标，在所得到的页面表中获得相应的页面描述项；
4. 将页面描述项中给出的页面基地址与线性地址中的offset位段相加得到物理地址。

# X86的控制寄存器

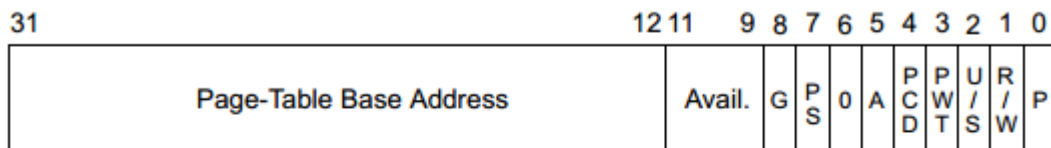
控制寄存器（CR0 ~ CR3）用于控制和确定处理器的操作模式以及当前执行任务的特性：

- CR0中含有控制处理器操作模式和状态的系统控制标志；
- CR1保留不用；
- CR2含有导致页错误的线性地址；
- CR3中含有页目录表物理内存基地址，因此该寄存器也被称为页目录基地址寄存器PDBR（Page-Directory Base address Register）。

# X86的控制寄存器



## Page-Directory Entry (4-KByte Page Table)



Available for system programmer's use

Global page (Ignored)

Page size (0 indicates 4 KBytes)

Reserved (set to 0)

Accessed

Cache disabled

Write-through

User/Supervisor

Read/Write

Present

# 页目录项PDE

## Page-Table Entry (4-KByte Page)



Available for system programmer's use

Global page

Reserved (set to 0)

Dirty

Accessed

Cache disabled

Write-through

User/Supervisor

Read/Write

Present

# 页表项PTE



- **【P】：**存在位。为1表示页表或者页位于内存中。否则，表示不在内存中，必须先予以创建或者从磁盘调入内存后方可使用。
- **【R/W】：**读写标志。为1表示页面可以被读写，为0表示只读。当处理器运行在0、1、2特权级时，此位不起作用。页目录中的这个位对其所映射的所有页面起作用。
- **【U/S】：**用户/超级用户标志。为1时，允许所有特权级别的程序访问；为0时，仅允许特权级为0、1、2的程序访问。页目录中的这个位对其所映射的所有页面起作用。
- **【PWT】：**Page级的Write-Through标志位。为1时使用Write-Through的Cache类型；为0时使用Write-Back的Cache类型。当CR0.CD=1时（Cache被Disable掉），此标志被忽略。
- **【PCD】：**Page级的Cache Disable标志位。为1时，物理页面是不能被Cache的；为0时允许Cache。当CR0.CD=1时，此标志被忽略。
- **【A】：**访问位。该位由处理器固件设置，用来指示此表项所指向的页是否已被访问（读或写），一旦置位，处理器从不清这个标志位。这个位可以被操作系统用来监视页的使用频率。
- **【D】：**脏位。该位由处理器固件设置，用来指示此表项所指向的页是否写过数据。
- **【PS】：**Page Size位。为0时，页的大小是4KB；为1时，页的大小是4MB（for normal 32-bit addressing）或者2MB（if extended physical addressing is enabled）。
- **【G】：**全局位。如果页是全局的，那么它将在高速缓存中一直保存。当CR4.PGE=1时，可以设置此位为1，指示Page是全局Page，在CR3被更新时，TLB内的全局Page不会被刷新。
- **【AVL】：**被处理器忽略，软件可以使用。

# Intel X86

- 分段不能禁用
- 使用分页需要设置CR0的PG位
- Linux使用的“扁平化内存管理”

Name	Description	Base	Limit	<a href="#">DPL</a>
__KERNEL_CS	Kernel code segment	0	4 GiB	0
__KERNEL_DS	Kernel data segment	0	4 GiB	0
__USER_CS	User code segment	0	4 GiB	3
__USER_DS	User data segment	0	4 GiB	3

# 小结

- 基本原理
  - 多个独立的逻辑地址空间：一维→二维
- 地址变换：段表
- 段共享：与页共享的区别
- 与页式管理优缺点对比
- 段页式管理
  - 基于分段的地址空间管理
  - 基于分页的内存分配管理