

内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
- 存储管理实例

内容提要

- 存储管理基础
- 基础内存管理
- 页式内存管理
 - 基本原理
 - 基本概念：页表、地址变换、多级页表、快表
 - 页表类型：哈希页表、反置页表
 - 页共享
- 段式内存管理
- 虚拟存储管理
- 存储管理实例

程序、进程和作业

- 程序程序是静止的，是存放在磁盘上的可执行文件
- 进程是动态的。进程包括程序和程序处理对象（数据集），是一个程序对某个数据集的执行过程，是分配资源的基本单位。通常把进程分为系统进程和用户进程两大类：
 - 完成操作系统功能的进程称为系统进程；
 - 完成用户功能的进程则称为用户进程。
- 作业是用户需要计算机完成的某项任务，是要求计算机所做工作的集合。

程序与进程之间的区别

1. 进程是竞争计算机系统有限资源的基本单位。进程更能真实地描述并发，而程序不能。
2. 程序是静态的概念；进程是程序在处理器上一次执行的过程，是动态的概念。
3. 进程有生存周期，有诞生有消亡。是短暂的；而程序是相对长久的。
4. 一个程序可以作为多个进程的运行政序；一个进程也可以运行多个程序。
5. 进程具有创建其他进程的功能；而程序没有。

作业与进程的区别

1. 一个作业的完成要经过作业提交、作业收容、作业执行和作业完成4个阶段。而进程是对已提交完毕的程序所执行过程的描述，是资源分配的基本单位。
2. 作业是用户向计算机提交任务的任务实体。在用户向计算机提交作业后，系统将它放入外存中的作业等待队列中等待执行。而进程则是完成用户任务的执行实体，是向系统申请分配资源的基本单位。任一进程，只要它被创建，总有相应的部分存在于内存中。
3. 一个作业可由多个进程组成，且必须至少由一个进程组成，反过来则不成立。
4. 作业的概念主要用在批处理系统中，像UNIX这样的分时系统中就没有作业的概念。而进程的概念则用在几乎所有的多道程序系统中。

作业、进程和程序之间的联系

- 一个作业通常包括程序、数据和操作说明书3部分。
- 每一个进程由进程控制块PCB、程序和数据集合组成。这说明程序是进程的一部分，是进程的实体。因此，一个作业可划分为若干个进程来完成。

分页式存储管理的基本思想

- 连续存储-碎片浪费空间，紧缩影响性能
- 如果可以把一个逻辑地址连续的的程序分散存放到若干不连续的内存区域内，并保证程序的正确执行，则既可充分利用内存空间，又可减少移动带来的开销。这就是页式管理的基本思想。
- 页式管理首先由英国Manchester大学提出并使用。支持页式管理的硬件部件通常称为MMU（Memory Management Unit）



动态地址翻译：用户进程发出的虚拟地址由MMU翻译成物理地址

Frame
Number

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	


0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	
8	
9	
10	
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	


D

纯分页系统 (Pure Paging System)

- 在分页存储管理方式中，如果不具备页面对换功能，不支持虚拟存储器功能，这种存储管理方式称为纯分页或基本分页存储管理方式。
- 在调度一个作业时，必须把它的所有页一次装到主存的页框内；如果当时页框数不足，则该作业必须等待，系统再调度另外作业。
- 优点：没有外碎片，每个内碎片不超过页大小；一个程序不必连续存放；便于改变程序占用空间的大小。
- 缺点：程序全部装入内存。

基本概念

- **页**：在分页存储管理系统中，把每个作业的地址空间分成一些**大小相等的片**，称之为**页面（Page）**或页，各页从0开始编号。
- **存储块**：在分页存储管理系统中，把主存的存储空间也分成与**页面相同大小的片**，这些片称为存储块，或称为**页框（Frame）**，同样从0开始编号。
- **页表**：为了便于在内存找到进程的每个页面所对应块，分页系统中为每个进程配置一张页表，进程逻辑地址空间中的每一页，在页表中都对应有一个页表项

页面的大小

- 页大小（与块大小一样）是由硬件来决定的。通常为2的幂。
 - 选择页的大小为2的幂可以方便的将逻辑地址转换为页号和页偏移。
 - 如果逻辑地址空间为 2^m ，且页大小为 2^n 单元，那么逻辑地址的高 $m-n$ 位表示页号（页表的索引），而低 n 位表示页偏移。每页大小从512B到16MB不等。
- 现代操作系统中，最常用的页面大小为4KB。

页面的大小

若页面较小

- 减少页内碎片和总的内存碎片，有利于提高内存利用率。
- 每个进程页面数增多，使页表长度增加，占用内存较大。
- 页面换进换出速度将降低。

若页面较大

- 每个进程页面数减少，页表长度减少，占用内存较小。
- 页面换进换出速度将提高。
- 增加页内碎片，不利于提高内存利用率。

练习

- 一个机器有32位的地址空间，页面大小8KB，页表完全由硬件实现，每个页表项是一个32位的字。当进程启动的时候，页表从内存复制到硬件中，复制速度每100ns一个32位字。那么，加载页表占用多少比例的CPU时间？

练习

- 一个机器有32位的地址空间，页面大小8KB，页表完全由硬件实现，每个页表项是一个32位的字。当进程启动的时候，页表从内存复制到硬件中，复制速度每100ns一个32位字。如果每个进程运行100ms，那么，加载页表占用多少比例的CPU时间？
- $1\text{ms} = 1000000\text{ns}$
- 页表包含 $2^{32}/2^{13}$ 项，即 524,288。加载页表花费52ms。如果进程执行100ms，那么52ms用来加载页表，48ms用来运行，52%的时间都花在页表加载上。

地址结构



例：地址长为 32 位，其中 0-11 位为页内地址，即每页的大小为 $2^{12}=4\text{KB}$ ；
12-31 位为页号，地址空间最多允许有 $2^{20}=1\text{M}$ 页。



例：地址长为 22 位，其中 0-11 位为块内地址，即每块的大小为 $2^{12}=4\text{KB}$ ，与页相等；
12-21 位为块号b，内存地址空间最多允许有 $2^{10}=1\text{K}$ 块。

地址结构

已知逻辑地址求页号和页内地址

- 给定一个逻辑地址空间中的地址为 A ，页面的大小为 L ，则页号 P 和页内地址 d （从 0 开始编号）可按下列式求得：

$$P = \text{INT} \left[\frac{A}{L} \right], d = [A] \bmod L$$

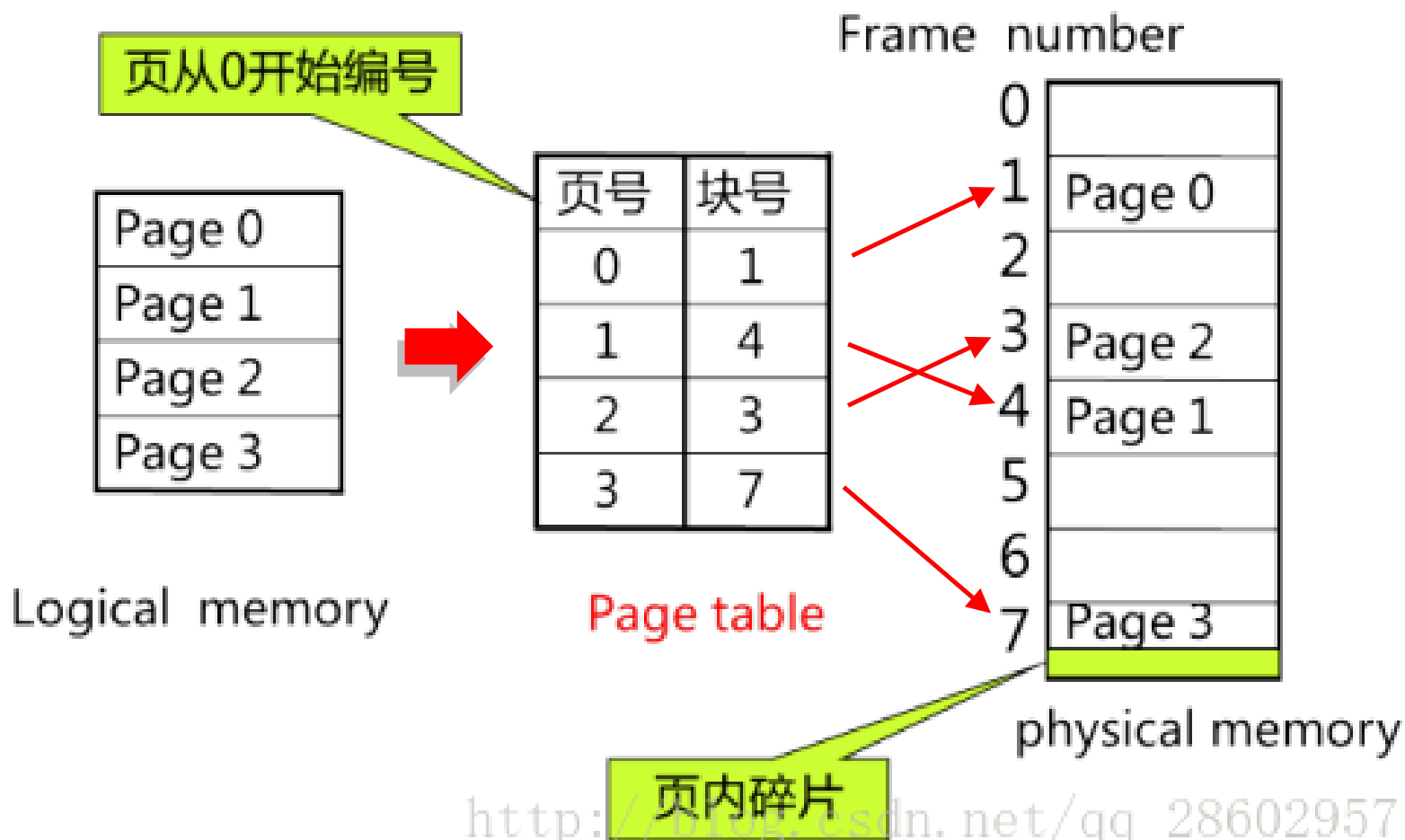
其中，INT 是整除函数，mod 是取余函数。

关于页表

- 页表存放在内存中，属于进程的现场信息。
- 用途：
 1. 记录进程的内存分配情况
 2. 实现进程运行时的动态重定位。
- 访问一个数据需访问内存 2 次 (页表一次，内存一次)。
- 页表的基址及长度由页表寄存器给出。

页表始址	页表长度
------	------

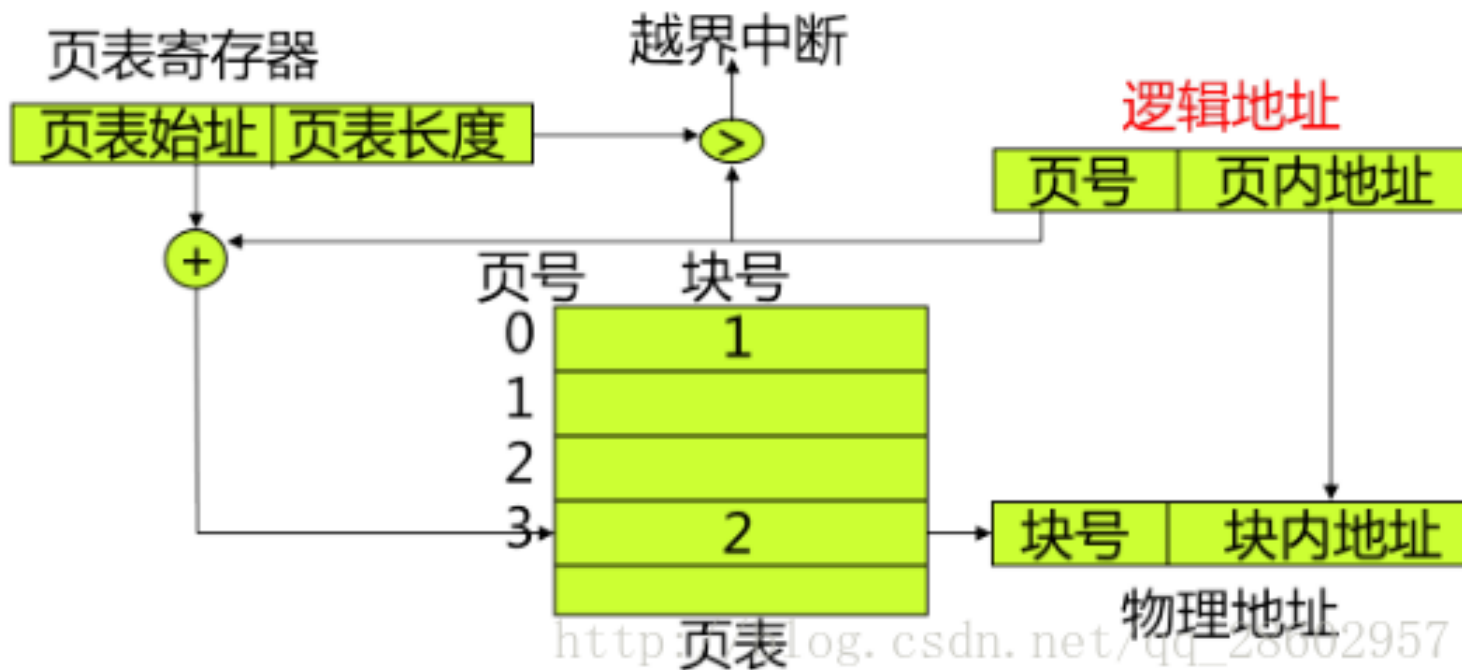
地址变换——页表查找



地址变换机构

- 当进程要访问某个逻辑地址中的数据时，分页地址变换机构会自动地将有效地址（相对地址）分为页号和页内地址两部分。
- 将页号与页表长度进行比较，如果页号大于或等于页表长度，则表示本次所访问的地址已超越进程的地址空间，产生地址越界中断。
- 将页表始址与页号和页表项长度的乘积相加，得到该表项在页表中的位置，于是可从中得到该页的物理块号，将之装入物理地址寄存器中。
- 将有效地址寄存器中的页内地址送入物理地址寄存器的块内地址字段中和块号地址拼接。

地址转换机构：页号到块号的函数



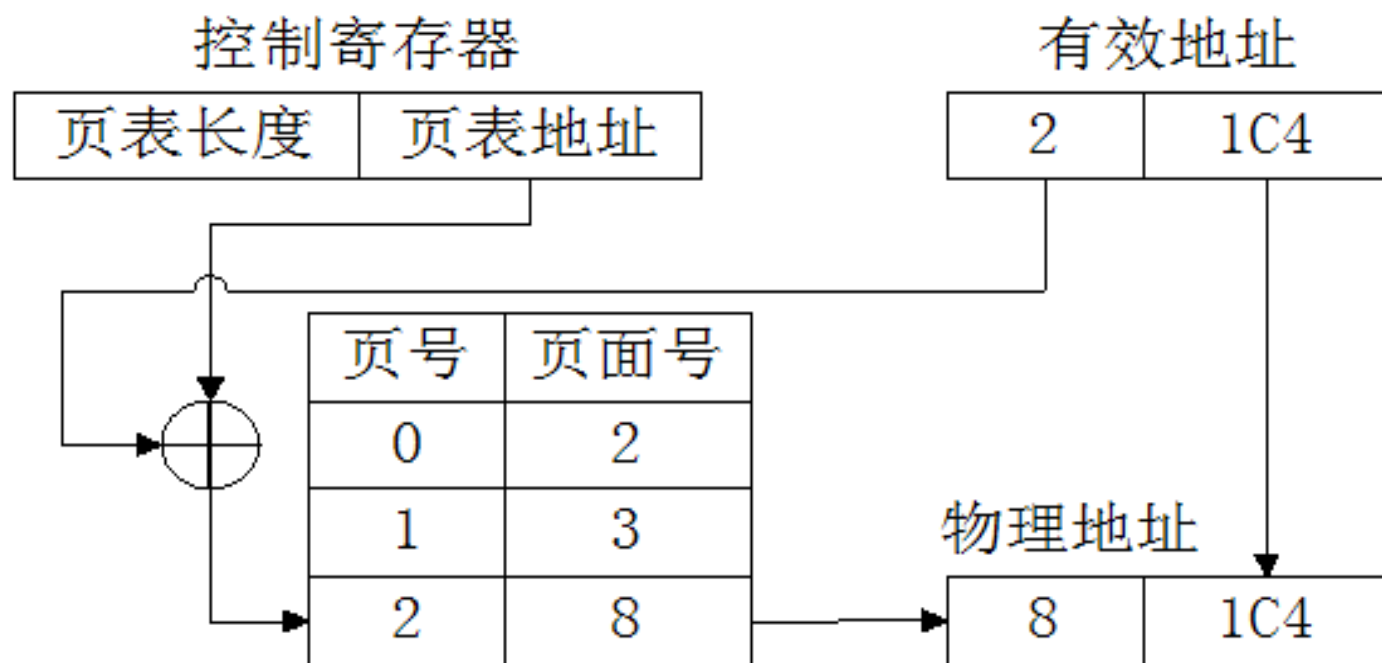
逻辑地址：把相对地址分为页号和页内地址两部分。

页表定位：页表始址 + 页号 × 页表项长度。

查询页表：读出块号。

物理地址：块号 **拼接** 块内地址。（块内地址 = 页内地址）

地址转换例 (21C4→81C4)



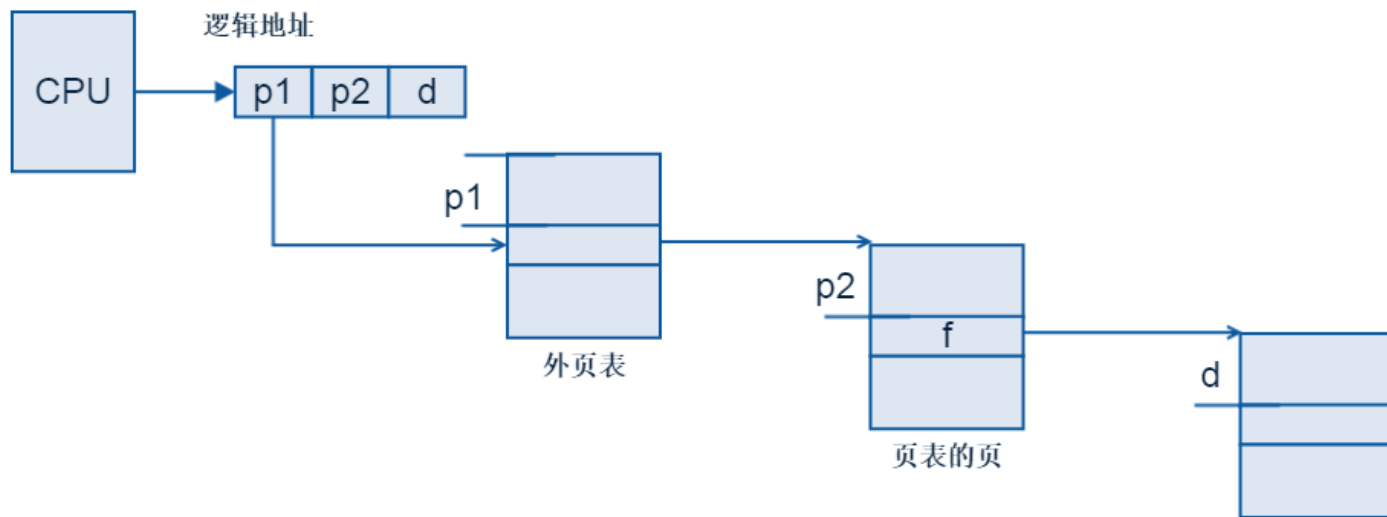
- ① 逻辑地址21C4分为页号2和页内位移1C4
- ② 根据寄存器所指页表地址与页号2找到对应的页面号8
- ③ 将8与页内位移1C4合并成物理地址81C4

一级页表的问题

- 若逻辑地址空间很大 ($2^{32} \sim 2^{64}$)，则划分的页比较多，页表就很大，占用的存储空间大（要求连续），实现较困难。
- 例如，对于 32 位逻辑地址空间的分页系统，如果规定页面大小为 4 KB 即 2^{12} B，则在每个进程页表就由高达 2^{20} 页组成。设每个页表项占用 4 个字节，**每个进程** 仅仅页表就要占用 4 MB 的内存空间。
- 解决问题的方法
 - 动态调入页表: 只将当前需用的部分页表项调入内存，其余的按需调入。
 - 引入多级页表

两级页表

- 将页表再进行分页，离散地将各个页表页面存放在不同的物理块中，同时也再建立一张外部页表用以记录页表页面对应的物理块号。
- 正在运行的进程，必须把一级页表（页表的页表）调入内存，而动态调入二级页表。只将当前所需的一些二级页表装入内存，其余部分根据需要再陆续调入。



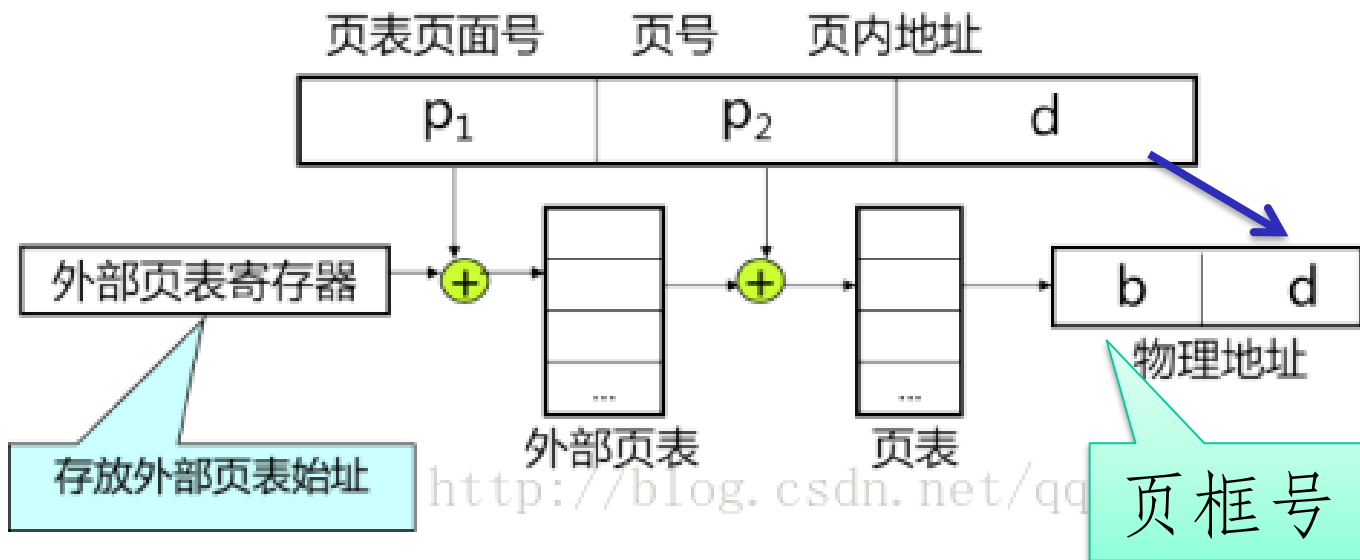
两级页表

对于32位地址空间，页面大小4K，
共有4M个页表， $p_1=10$, $p_2=10$, $d=12$

■ 逻辑地址

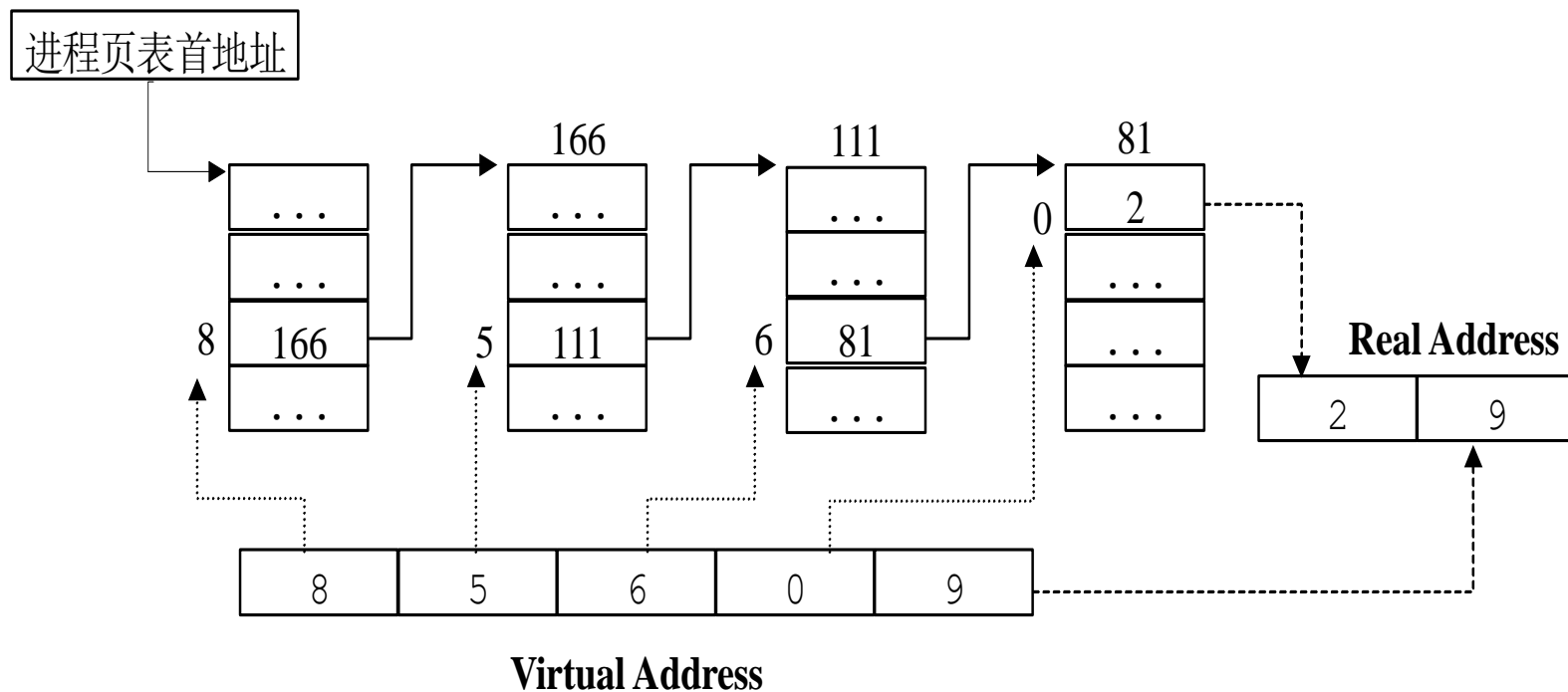
页表页面号	页号	页内地址
p_1	p_2	d

■ 地址变换

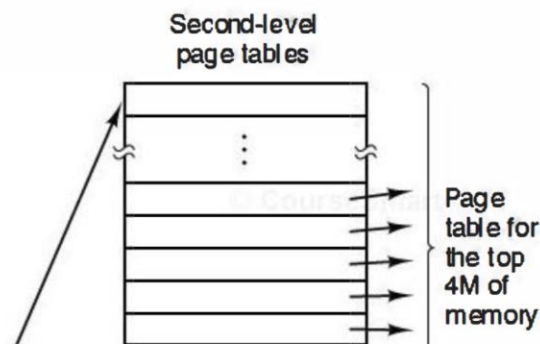


多级页表

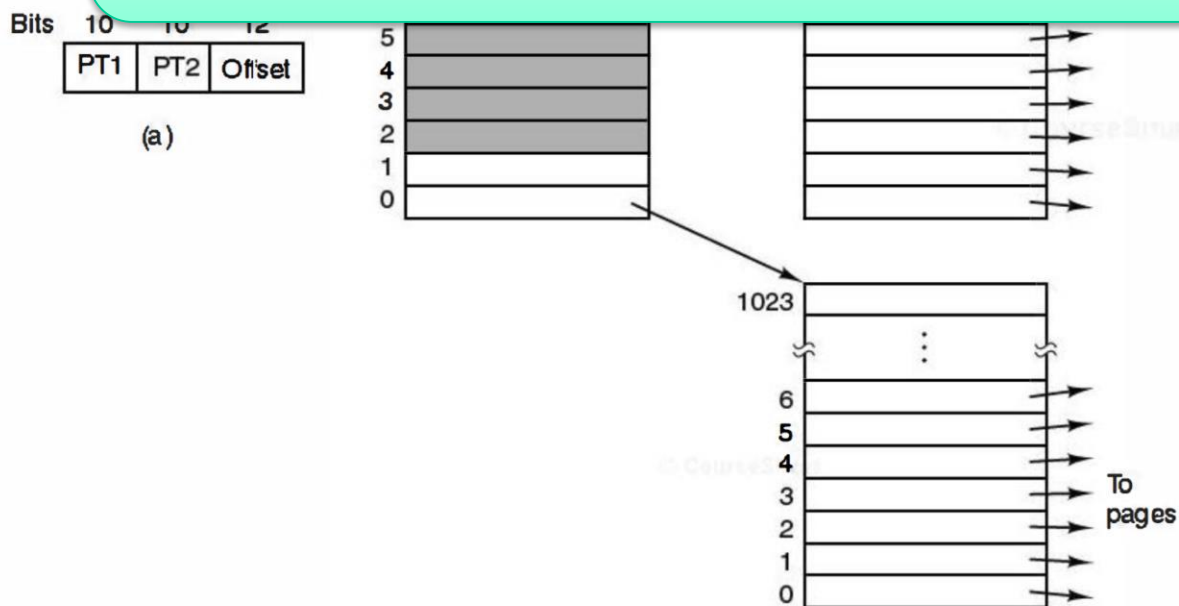
- 多级页表结构中，指令所给出的地址除**偏移地址**之外的各部分全是各级页表的页表号或页号，而**各级页表中记录的全是物理页号**，指向下级页表或真正的被访问页。



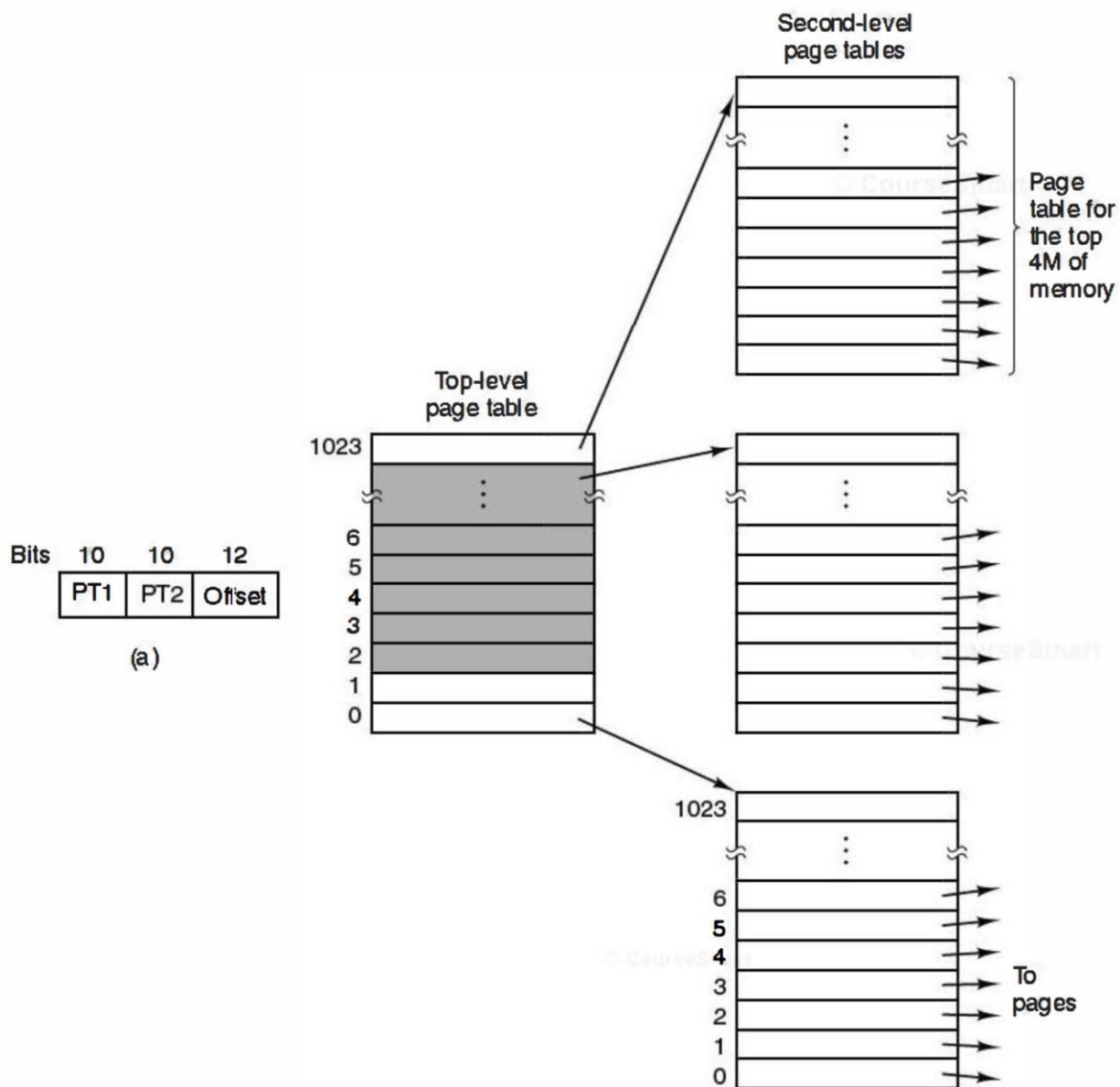
二级页表节省内存空间



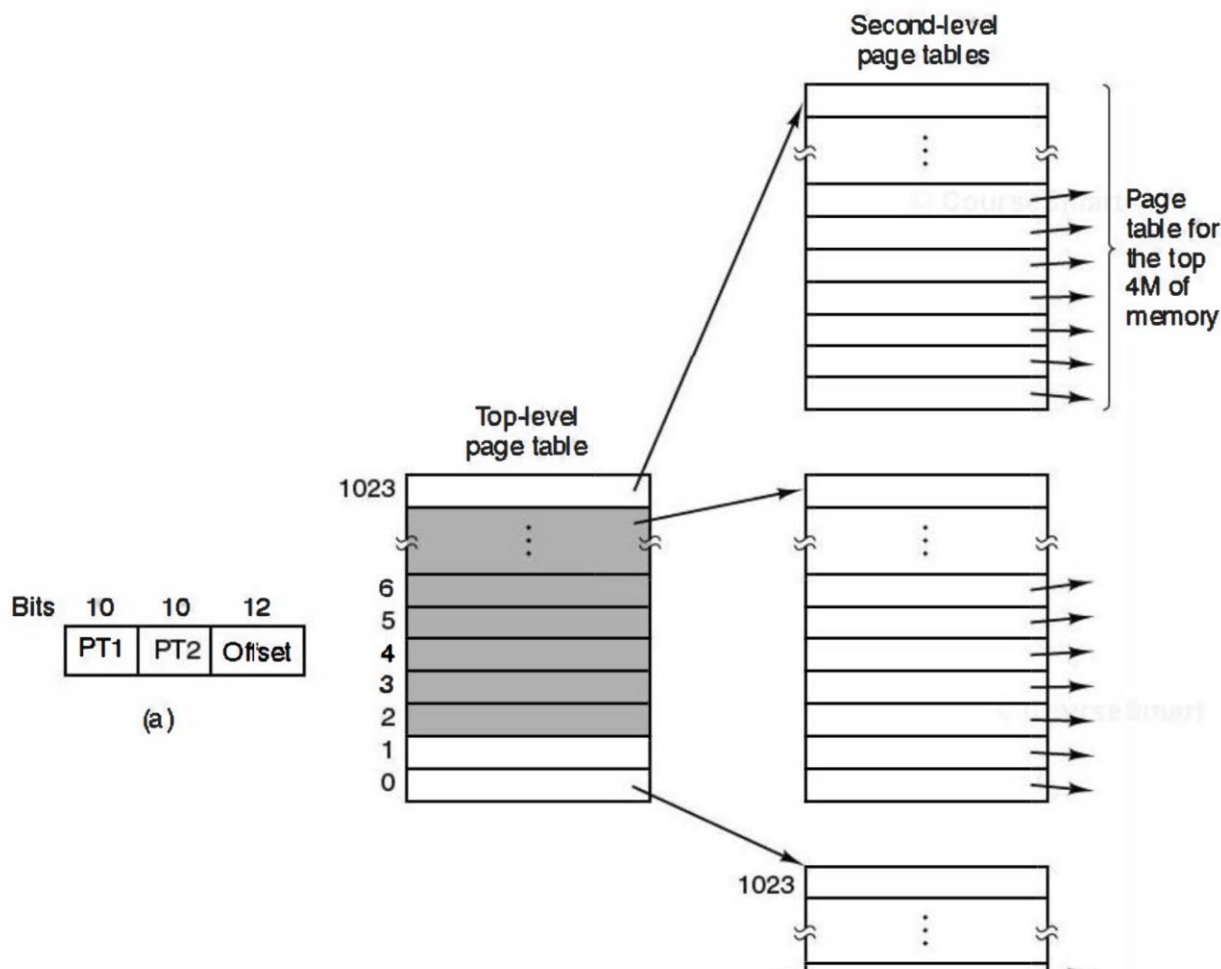
一级页表管理：32位地址，页面大小4K，需要一百万个（1M）页表在内存中，4M



二级页表节省内存空间

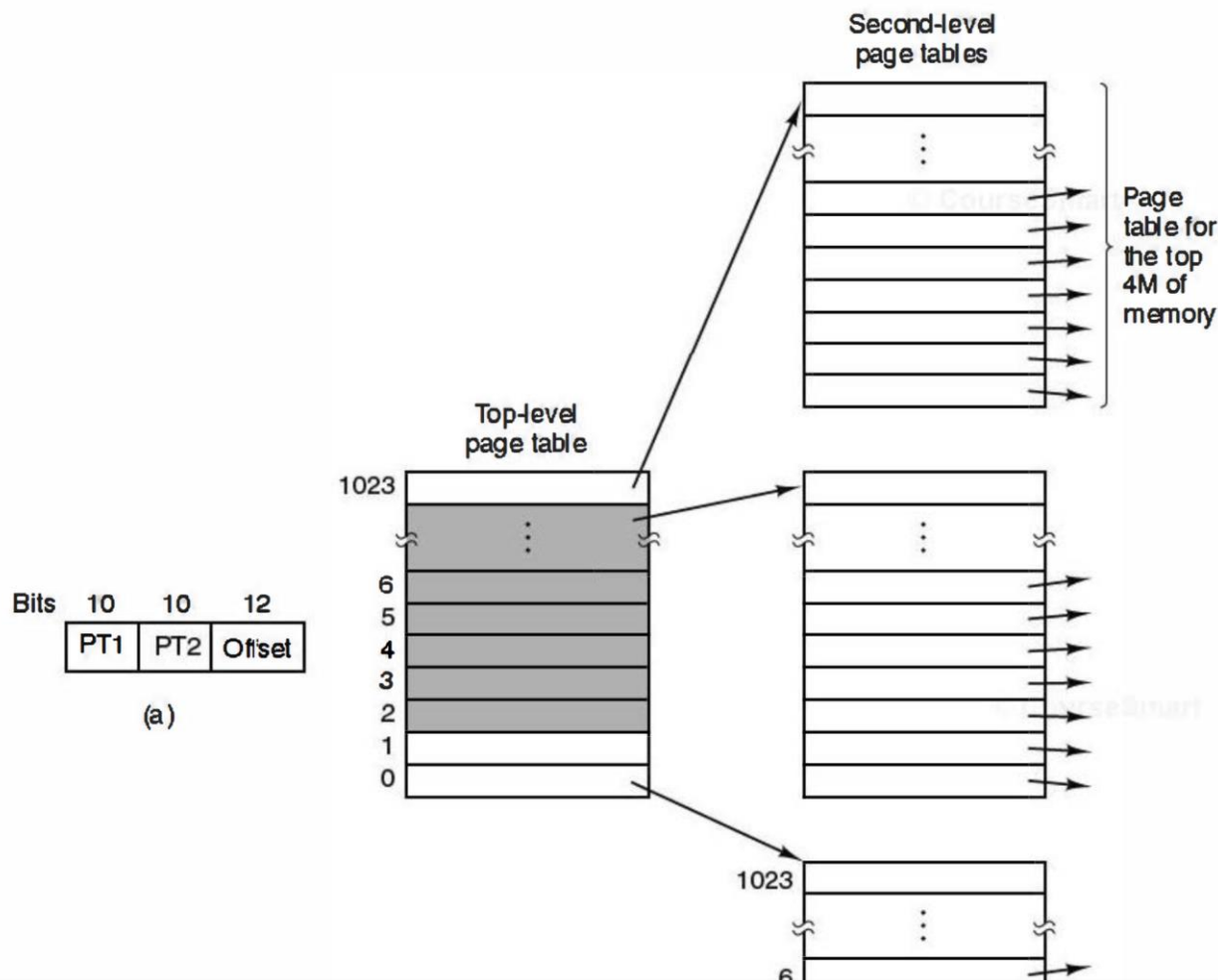


二级页表节省内存空间



二级页表：4G空间，页面大小4K。一级页表1K项，每项代表4M空间。每个二级页表1K项，每项代表4K页面。

二级页表节省内存空间



二级页表管理：假设进程12M，一级页表只需要占用3项
所以二级页表只需要3个在内存，只需要4个页表(100万中)在内存

页表系统需要解决的问题

- 页表解决逻辑地址空间到物理地址空间的映射
- 理想的页表系统需要处理两个问题:
 - 1. 从逻辑地址到物理地址的映射必须很快
 - 指令和数据都来自内存
 - 每个指令需要多次访存，每次都要查页表
 - 页表查询如过很慢，访存时间会翻倍

页表系统需要解决的问题

- 页表解决逻辑地址空间到物理地址空间的映射
- 理想的页表系统需要处理两个问题:
 - 1. 从逻辑地址到物理地址的映射必须很快
 - 指令和数据都来自内存
 - 每个指令需要多次访存，每次都要查页表
 - 页表查询如过很慢，访存时间会翻倍
 - 2. 如果逻辑地址空间很大，页表也会很大
 - 32位逻辑地址，4K页大小，需要一百万 (2^{20}) 个页表项
 - 64位逻辑地址，4K页大小，需要 2^{52} 个页表
 - 更糟糕的是，每个进程需要一个页表

页表系统需要解决的问题

- 页表解决逻辑地址空间到物理地址空间的映射
- 理想的页表系统需要处理两个问题:
 - 1. 从逻辑地址到物理地址的映射必须很快
 - 指令和数据都来自内存

如何让页表查询快速?

- 2. 如果逻辑地址空间很大, 页表也会很大
 - 32位逻辑地址, 4K页大小, 需要一百万 (2^{20}) 个页表项
 - 64位逻辑地址, 4K页大小, 需要 2^{52} 个页表
 - 更糟糕的是, 每个进程需要一个页表

如何让页表查询快速？

- 方案：使用独立的高速硬件缓存，进程切换时，将页表从内存全部读入cache。
 - 页表查询过程无需访问内存
 - 但是。。。

如何让页表查询快速？

- 使用独立的高速硬件缓存，进程切换时，将页表从内存全部读入cache。
 - 页表查询过程无需访问内存
 - 但是。。。
 - 全部放入cache过于昂贵，特别是64位时
 - 进程切换执行，性能损失大

如何让大页表成本可控？

- 另一个极端方案
 - 页表全部进入内存
 - 寄存器存放首地址
 - 问题一 速度太慢

如何让大页表成本可控且查询快速？

- 使用独立的高速硬件缓存，进程切换时，将页表从内存全部读入cache。
 - 页表查询过程无需访问内存
 - 但是。。。- 全部- 进程
- 另一个极端方案
 - 页表全部进入内存
 - 寄存器存放首地址
 - 问题一 速度太慢

采用折衷的方案— TLB

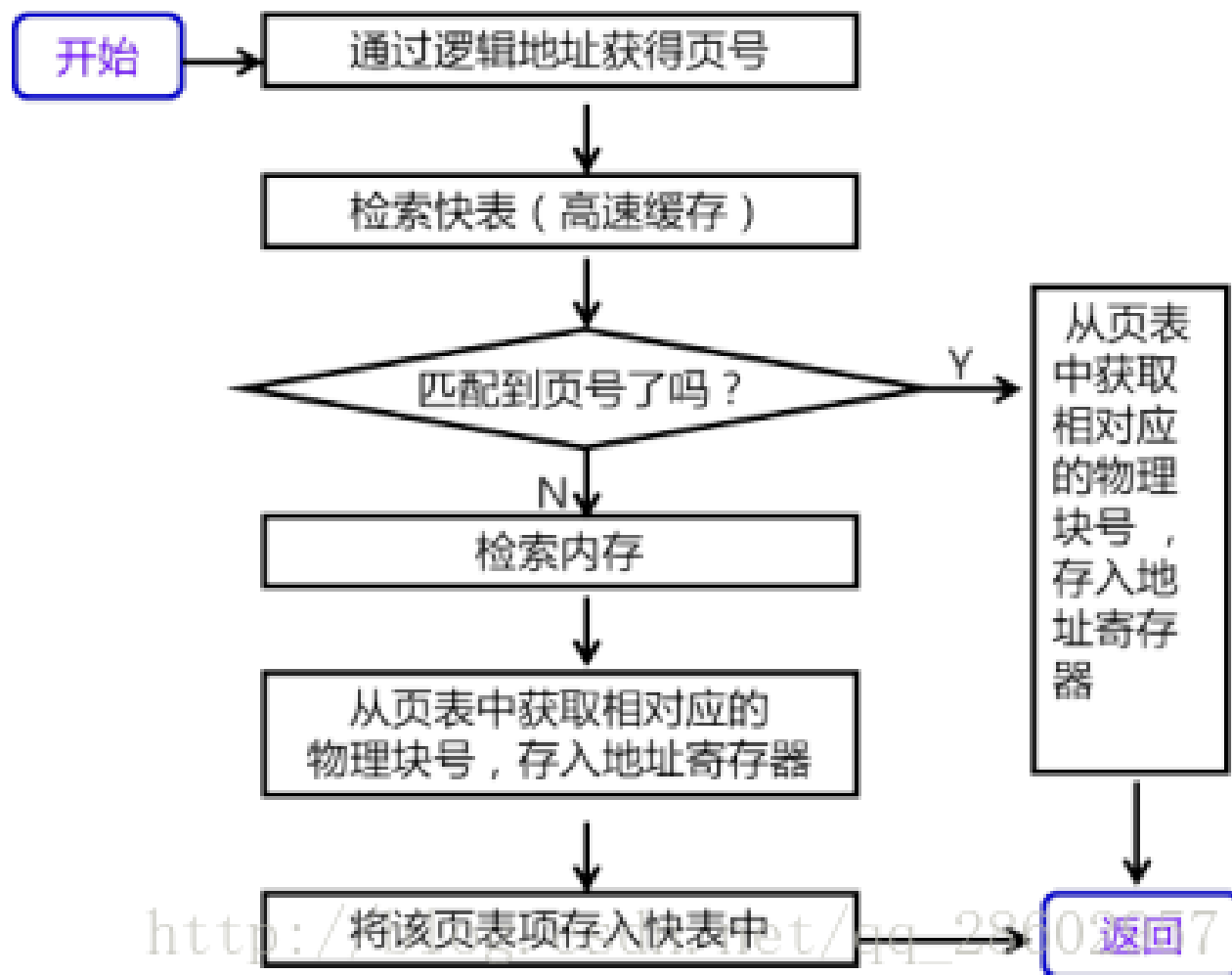
具有快表的地址变换机构

- 快表又称联想存储器 (Associative Memory)、TLB (Translation Lookaside Buffer) 转换表查找缓冲区，IBM最早采用TLB。
- 快表是一种特殊的高速缓冲存储器 (Cache)，内容是页表中的**一部分内容**。
- 快表只是页表的一部分
 - 容量小，成本可以接受
 - 程序访存的局部性原理，大部分访存集中在少量的页表项

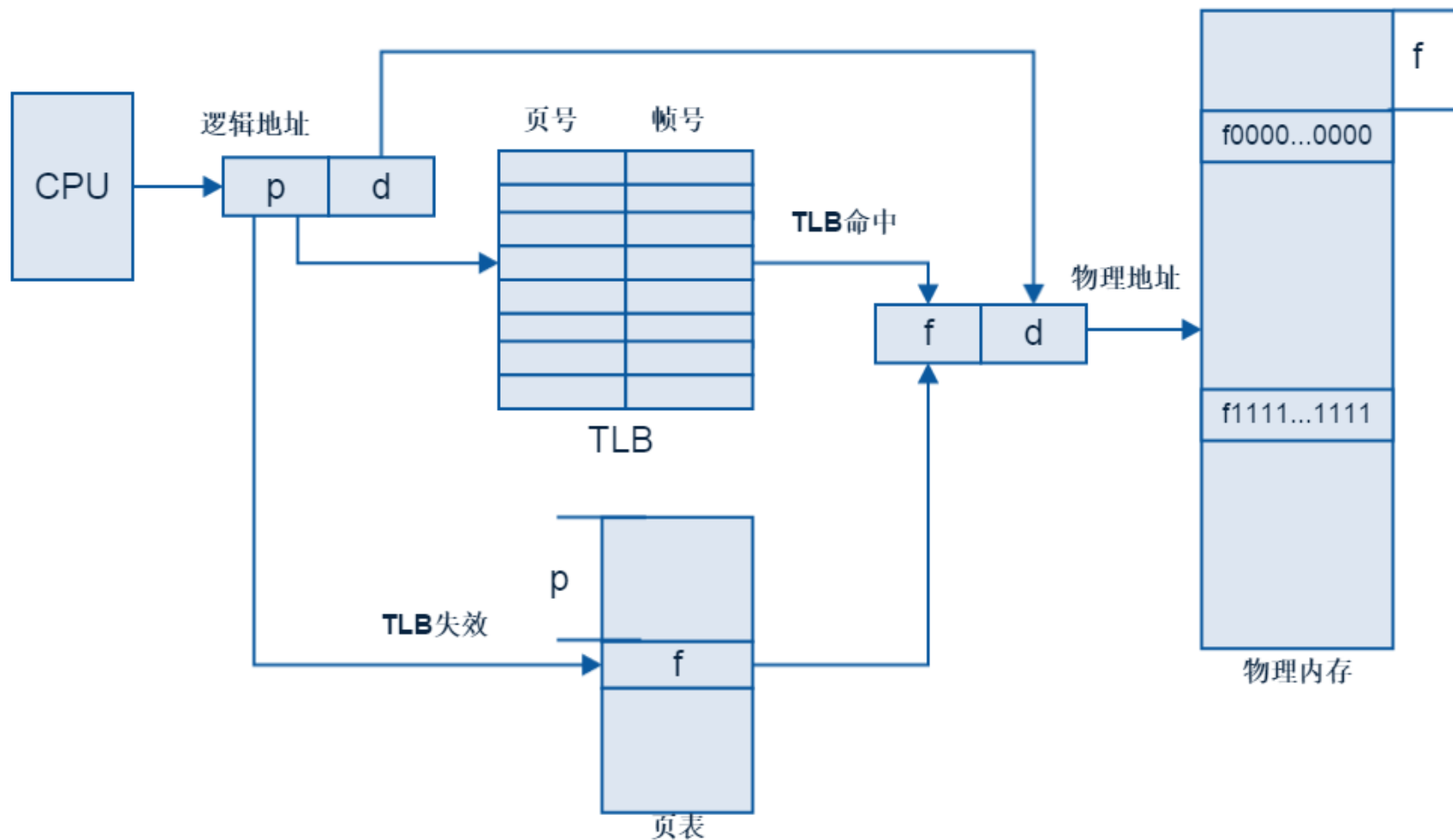
具有快表的地址变换机构

- 快表又称联想存储器 (Associative Memory)、TLB (Translation Lookaside Buffer) 转换表查找缓冲区, IBM最早采用TLB。
- CPU 产生逻辑地址的页号, 首先在快表中寻找, 若命中就找出其对应的页框; 若未命中, 再到页表中找其对应的物理块, 并将之复制到快表。若快表中内容满, 则按某种算法淘汰某些页。
- 通常, TLB中的条目数并不多, 在64~1024之间。

快表 (TLB)



快表 (TLB)



快表 (TLB) 的结构

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

页号到页框号的映射

快表 (TLB)

TLB的性质和使用方法与Cache相同：

- TLB只包括也表中的一小部分条目。当CPU产生逻辑地址后，其页号提交给TLB。如果页码不在TLB中（称为TLB失效），那么就需要访问页表。将页号和帧号增加到TLB中。
- 如果TLB中的条目已满，那么操作系统会选择一个来替换。替换策略有很多，从最近最少使用替换（LRU）到随机替换等。
- 另外，有的TLB允许有些条目固定下来。通常内核代码的条目是固定下来的。

快表 (TLB)

- 页号在TLB中被查找到的百分比称为命中率。
 - 80%的命中率意味着有80%的时间可以在TLB中找到所需的页号。
 - 假如查找TLB需要20ns，访问内存需要100ns，如果访问位于TLB中的页号，那么采用内存映射访问需要120ns。如果不能在TLB中找到（20ns），那么必须先访问位于内存中的页表得到帧号（100ns），并进而访问内存中所需字节（100ns），这总共需要220ns。为了得到有效内存访问时间，必须根据概率对每种情况进行加权。

快表 (TLB)

- 页号在TLB中被查找到的百分比称为**命中率**。
 - 80%的命中率意味着有80%的时间可以在TLB中找到所需的页号。
 - 假如查找TLB需要20ns，访问内存需要100ns，如果访问位于TLB中的页号，那么采用内存映射访问需要120ns。如果不能在TLB中找到（20ns），那么必须先访问位于内存中的页表得到帧号（100ns），并进而访问内存中所需字节（100ns），这总共需要220ns。为了得到有效内存访问时间，必须根据概率对每种情况进行加权。
- 有效内存访问时间 = $0.80 * 120 + 0.2 * 220 = 140$ (ns)
 - 对于这种情况，现在内存访问速度要慢40%（100ns~140ns）
- 如果命中率为98%，那么
 - 有效内存访问时间 = $0.98 * 120 + 0.02 * 220 = 122$ (ns)
 - 由于提高了命中率，内存访问时间只慢了22%

快表 (TLB) : 练习

- 一台计算机的地址空间有1024个页存放在内存中。从页表中读取一个字的时间是5ns。为了降低性能损失，计算机使用TLB，容量为32个<页号，物理页框号>项，并且TLB的查找时间为1ns。为了将平均页表读取时间降低为2ns，需要实现多高的命中率？

快表 (TLB) : 答案

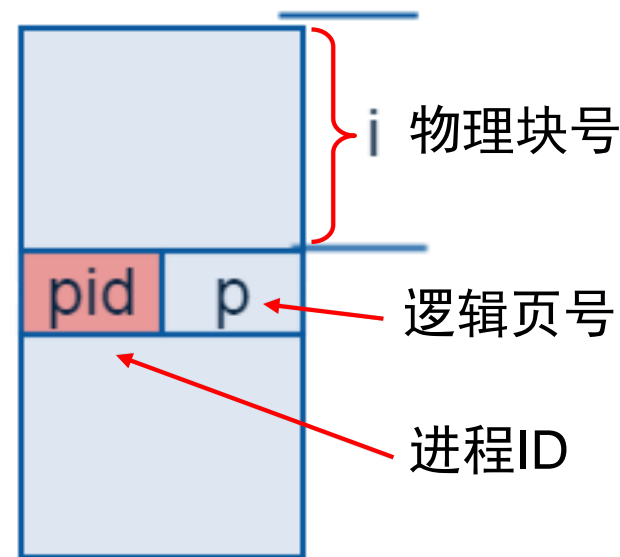
- 一台计算机的地址空间有1024个页存放在内存中。从页表中读取一个字的时间是5ns。为了降低性能损失，计算机使用TLB，容量为32个<页号，物理页框号>项，并且TLB的查找时间为1ns。为了将平均页表读取时间降低为2ns，需要实现多高的命中率？
- 假设 h 为命中率，平均的读取时间为 $1h + 5(1 - h)$ ，将读取时间等于2并求解 h ，则 $h=0.75$ 。

反置页表(Inverted page table)

- 每个进程都有一个相关页表。
- 32位逻辑地址，可以采用多级页表减少页表空间
- 64位地址，地址空间为 2^{64} ，如果每页4KB，页表需要 2^{52} 页表项，如果每项8字节，需要占用3千万G（30PB），这样的页表过大了。即使使用多级页表，多次访问牺牲性能。
- 为了解决这个问题，可以使用反向页表（inverted pagetable）
 - 每个页框一个页表项
 - 而不是每个页一个页表项

反置页表(Inverted page table)

- 反置页表不是依据进程的逻辑页号来组织，而是依据物理页面号来组织（即：按物理页框号排列），其表项的内容是逻辑页号 P 及隶属进程标志符 pid。
- 反置页表的大小只与物理内存的大小相关，与逻辑空间大小和进程数无关。
- 如64位的PowerPC, UltraSparc等处理器。



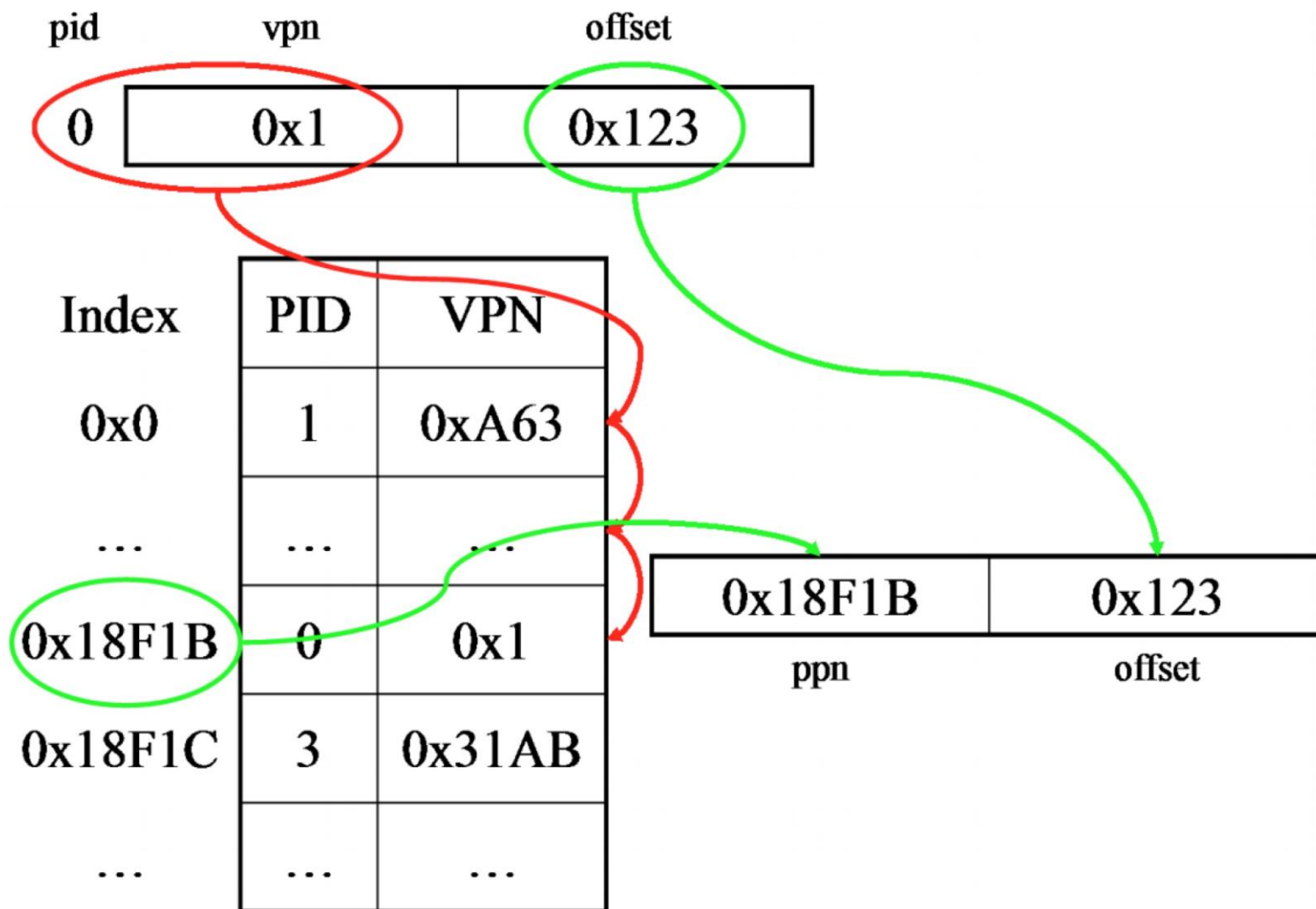
反置页表

反置页表

利用反置页表进行地址变换：

- 用进程标志符和页号去检索反置页表。
- 如果检索完整个页表未找到与之匹配的页表项，表明此页此时尚未调入内存，对于具有请求调页功能的存储器系统产生请求调页中断，若无此功能则表示地址出错。
- 如果检索到与之匹配的表项，则表项的序号 i 便是该页的物理块号，将该块号与页内地址一起构成物理地址。

反置页表(Inverted page table)



反置页表(Inverted page table)

- 反向页表按照物理地址排序，而查找依据虚拟地址，所以可能需要查找整个表来寻找匹配。
- 优点：页表存储空间大大减少
 - 64位，4KB页表，1G RAM， $1G/4KB = 256K$ 表项
- 缺点：
 - 不能用页号索引查找
 - 只能遍历整个页表项查找 (n, p)
 - 每次内存访问都查找256K速度非常慢

反置页表(Inverted page table)

■ 缺点：

- 不能用页号索引查找
- 只能遍历整个页表项查找 (n, p)
- 每次内存访问都查找256K速度非常慢

如何加速？ 方法一：使用快表TLB

反置页表(Inverted page table)

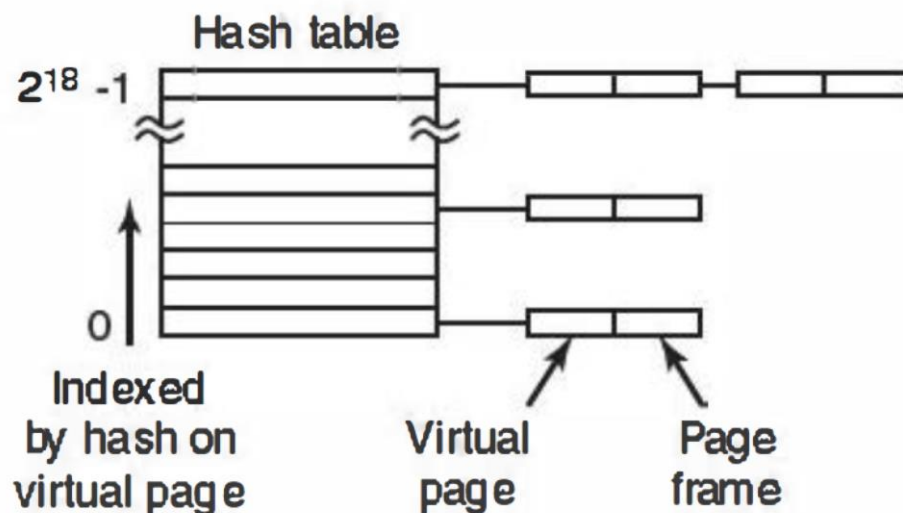
- 反置页表结合块表：
 - 将经常访问的反置页表页面放入TLB
- 如果TLB miss 该怎么办？
 - 仍然需要遍历查询反置页表

如何改进反置页表的查询效率？

哈希页表 (hashed page table)

- 处理超过32位地址空间的常用方法是使用哈希页表 (hashed page table)，并以虚拟页号作为哈希值。哈希页表的每一条目都包括一个链表，链表的节点哈希成同一位置（要处理碰撞）。每个元素有3个域：
 1. 虚拟页号
 2. 所映射的页框号
 3. 指向下一个元素的指针。

哈希页表多大？



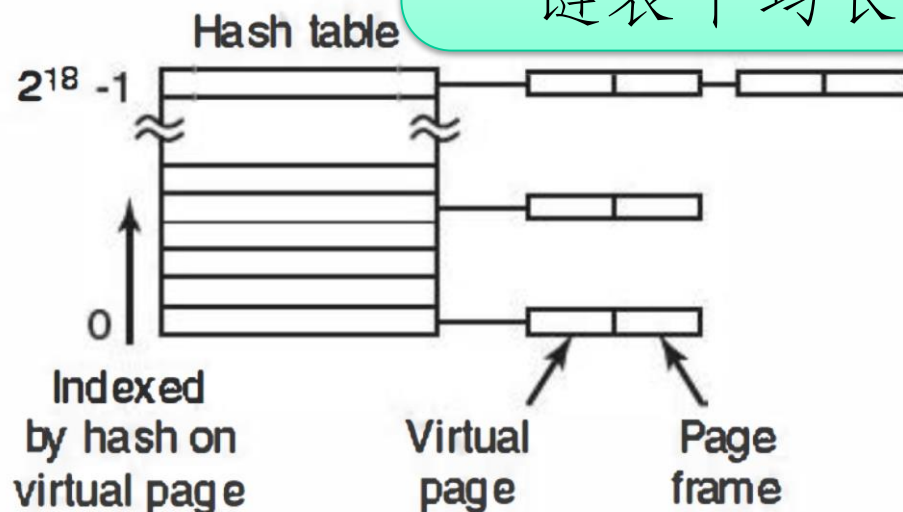
哈希页表 (hashed page table)

- 处理超过32位地址空间的常用方法是使用哈希页表 (hashed page table)，并以虚拟页号作为哈希值。哈希页表的每一条目都包括一个链表，链表的节点哈希成同一位置（要处理碰撞）。每个元素有3个域：

1. 虚拟页号
2. 所映射的页框号
3. 指向下一个元素的指针。

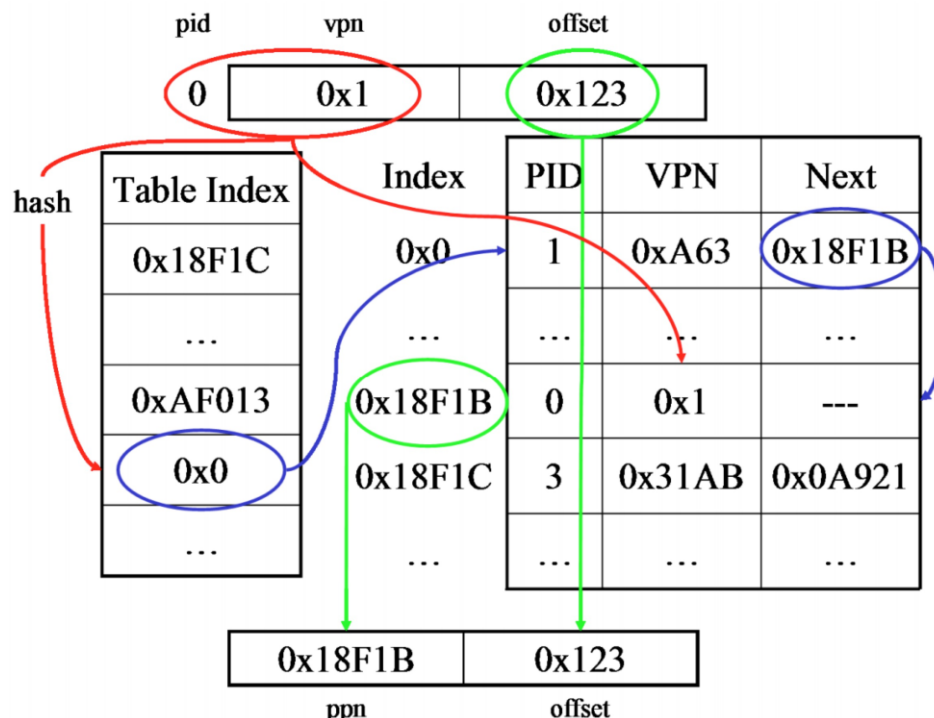
哈希页表多大？

- 如果等于物理页框个数
- 链表平均长度：1



哈希页表 (hashed page table)

- 该算法按照如下方式工作：
 - 计算虚拟页号和pid的哈希值
 - 用虚拟页号与链表中的每一个元素的第一个域相比较
 - 如果匹配，则取出页框号
 - 如果不匹配，沿着链表继续查找



练习-哈希页表

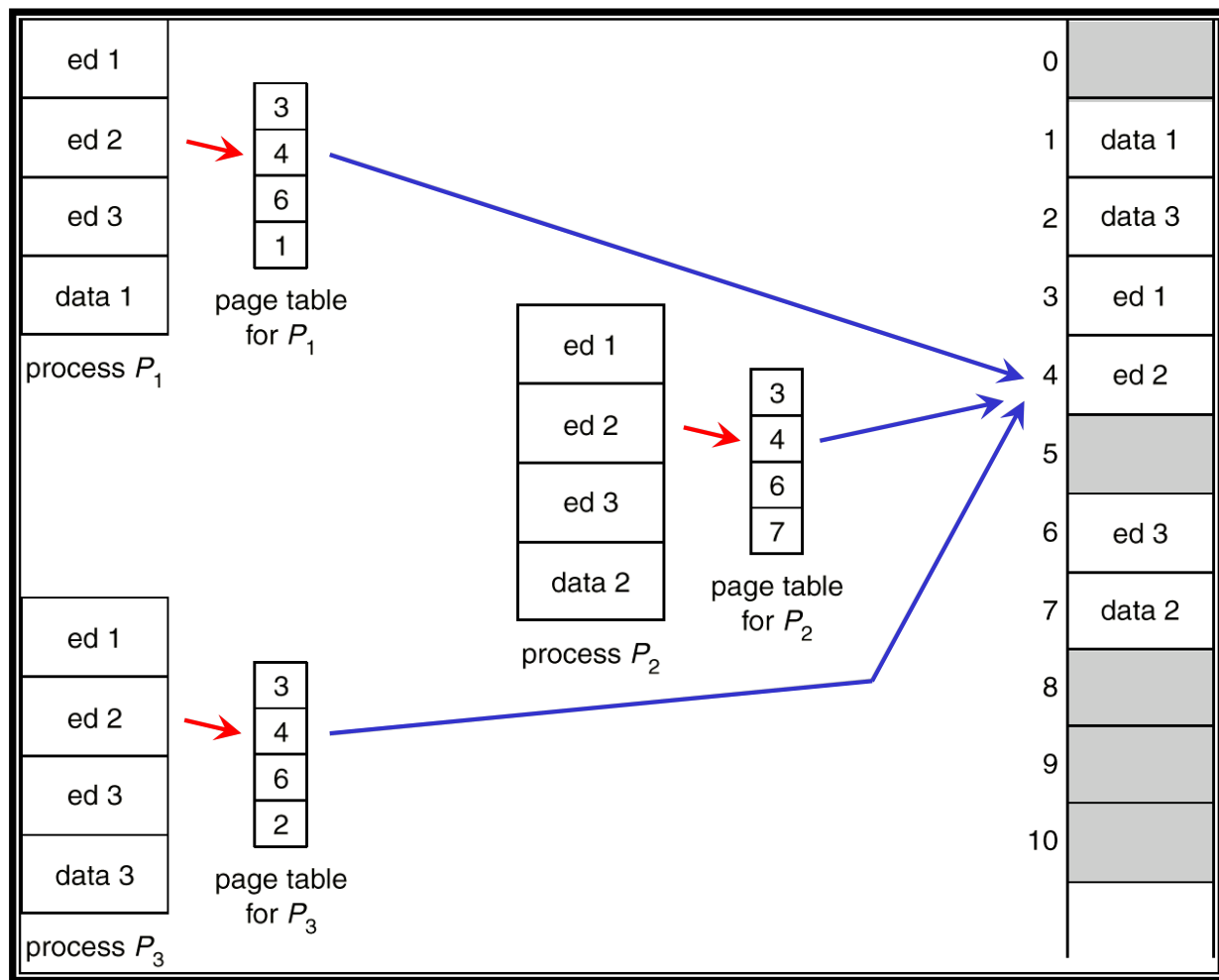
- 一台计算机页面大小8K, 主存256KB, 虚存64GB, 使用哈希页表机制。哈希页表需要多大才能让哈希链表平均长度小于1? 假设哈希表大小是2的幂。

练习-哈希页表

- 一台计算机页面大小8K, 主存256MB, 虚存64GB, 使用哈希页表机制。哈希页表需要多大才能让哈希链表平均长度小于1? 假设哈希表大小是2的幂。
- 页面数为 $256\text{M}/8\text{K}=32\text{K}$ 。所以哈希表32K大小时, 平均链表长度为1. 当哈希表大小64K大小时, 平均长度0.5, 搜索加倍。

页共享与保护

- 各进程把需要共享的数据/程序的相应页指向相同物理块。



页保护

页的保护

- 页式存储管理系统提供了两种方式：
 - 地址越界保护
 - 在页表中设置保护位（定义操作权限：只读，读写，执行等）

页式存储的优缺点

■ 优点

- 不要求连续存储
- 解决了碎片问题，提高了内存利用率

■ 缺点

- 仍然存在内存空间浪费
 - 不同应用大小不同，但是页面大小固定
 - 最后一页会有碎片
- 不适用于大型的应用
 - 虚存解决

页式存储的优缺点

■ 优点

- 不要求连续存储
- 解决了碎片问题，提高了内存利用率

■ 缺点

- 仍然存在内存空间浪费

虽然分页解决了碎片，提供了连续的逻辑地址空间
但是分页真的方便用户使用么？

- 不适合于大型的应用
 - 虚存解决

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。

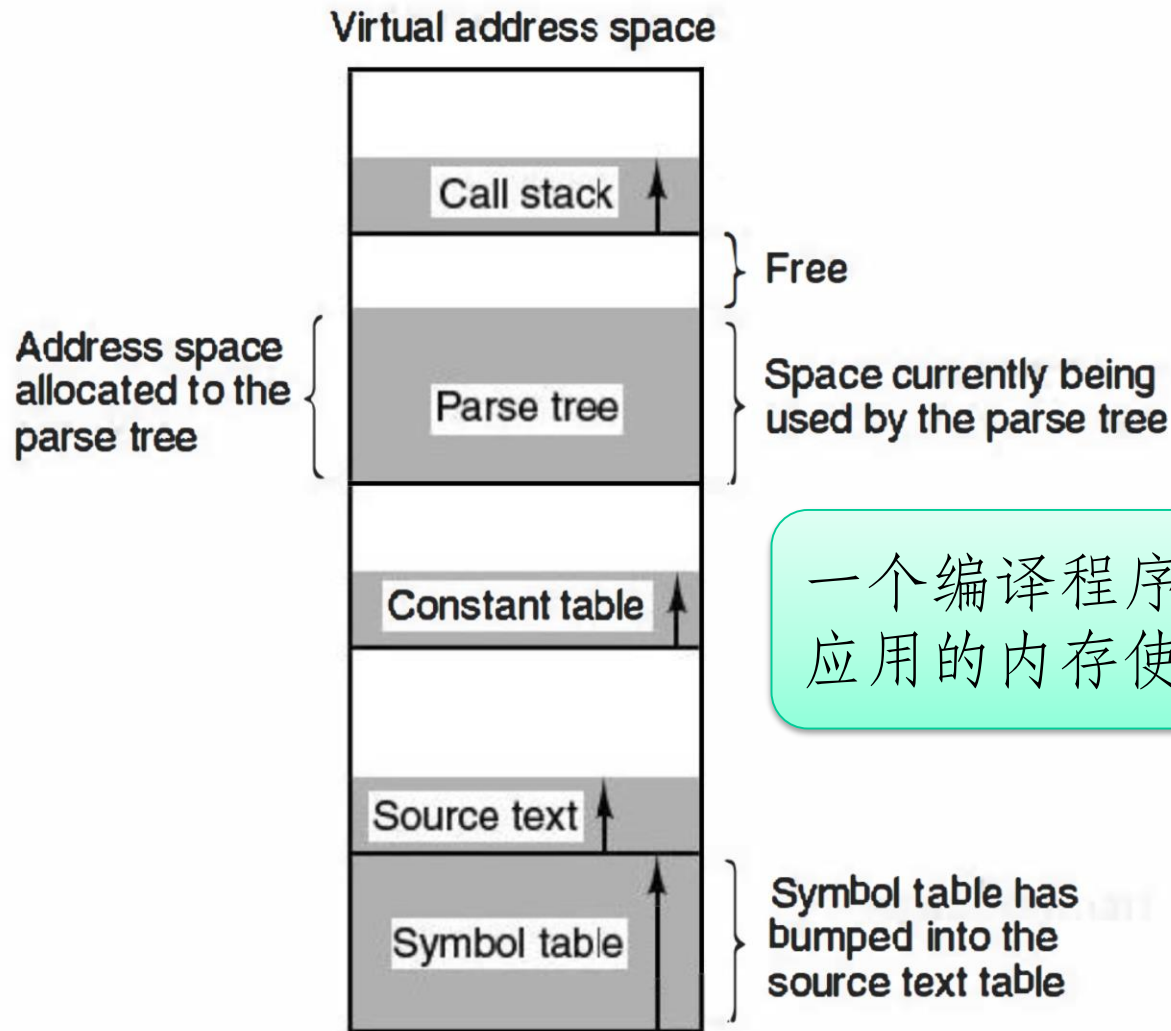
动态增长：

- 实际应用中，某些段（数据段）会不断增长，分页难以实现。

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。

段式存储管理—内存动态增长的例子



一个编译程序的一维地址空间，应用的内存使用可能动态增长

Figure 3-31. In a one-dimensional address space with growing tables, one table may bump into another.

分段地址空间

一个段可定义为一组逻辑相关信息，每个作业的地址空间是由一些分段构成的（二维空间），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。

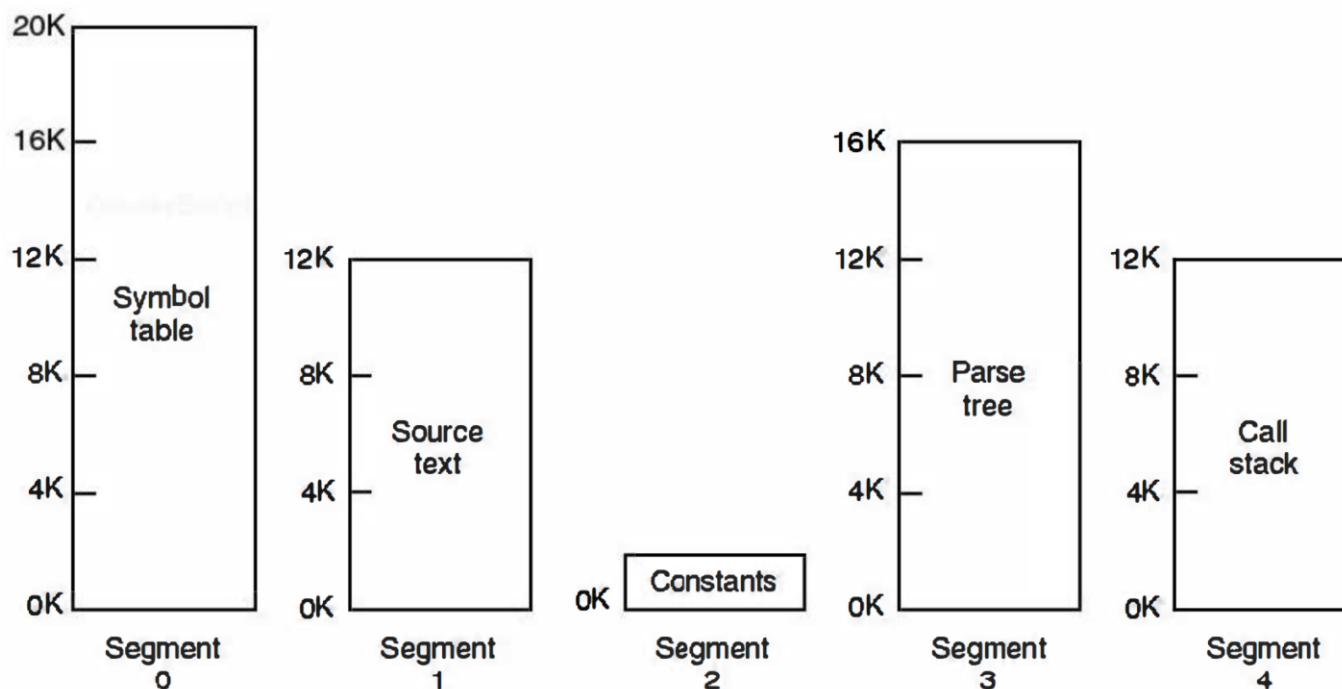


Figure 3-32. A segmented memory allows each table to grow or shrink independently of the other tables.

地址结构

段表

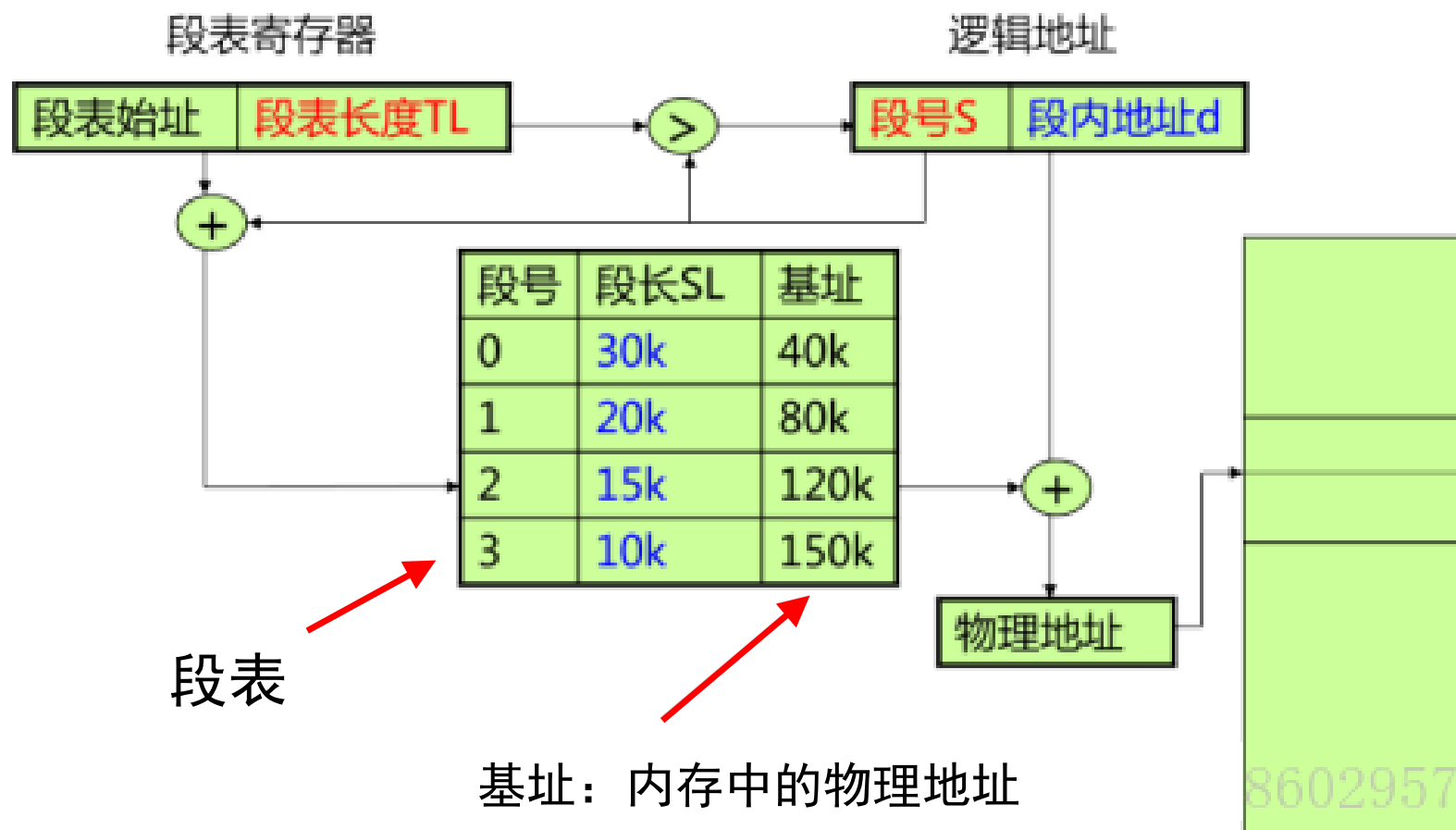
- 段表记录了段与内存位置的对应关系。
- 段表保存在内存中。
- 段表的基址及长度由段表寄存器给出。

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
----	------

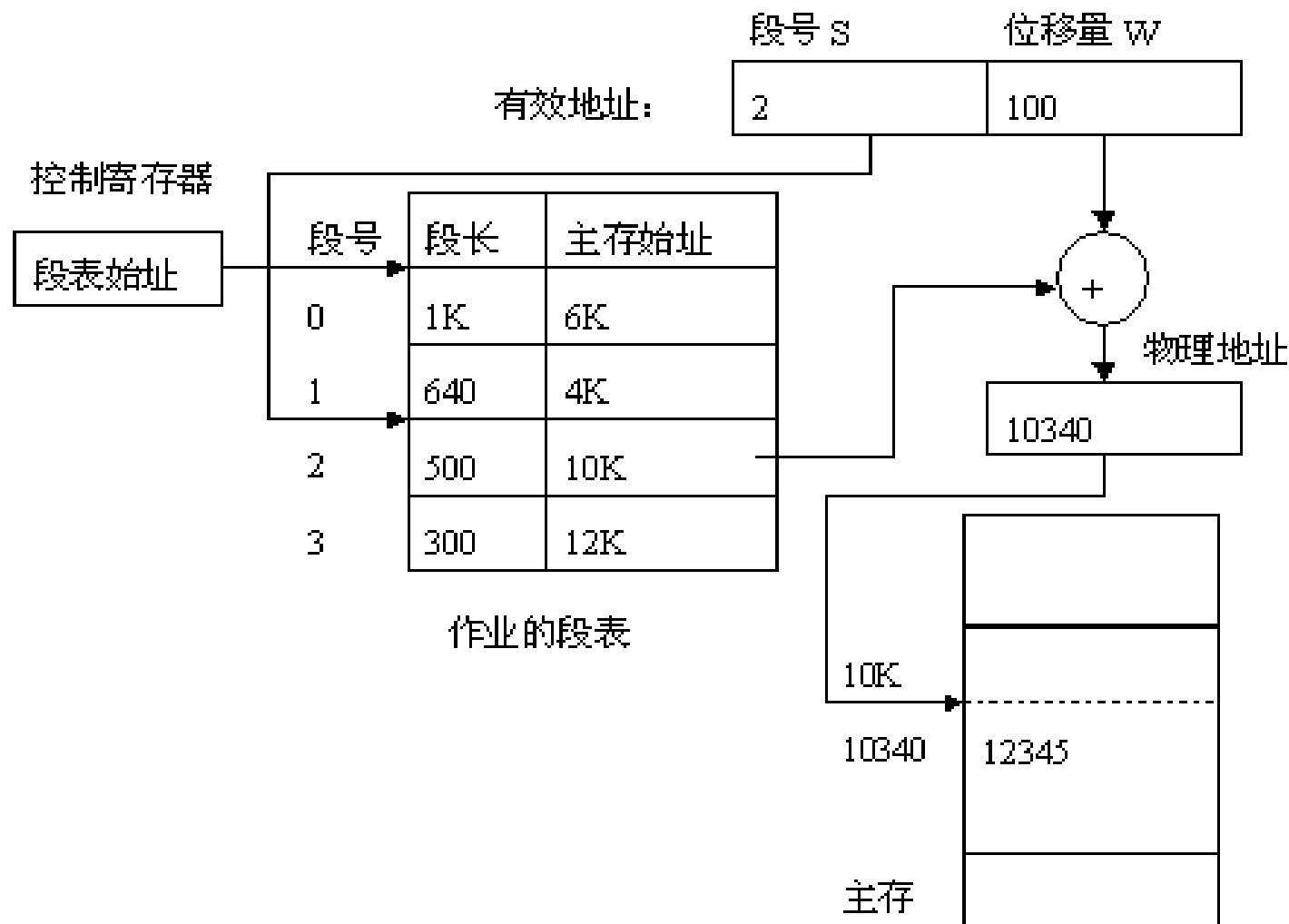
地址变换机构



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基址。
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。(如果代码可重入，通过共享能否节省空间？)

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。

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。

重入攻击-包含状态修改不应该可重入

1	contract BountyHunt{
2	...
3	function claimBounty() preventTheft {
4	uint balance = bountyAmount[msg.sender];
5	if (msg.sender.call.value(balance)()) {
6	totalBountyAmount -= balance;
7	bountyAmount[msg.sender] = 0;
8	}
9	}
10	}
11	contract AttackerAgent{
12	① ... ② ③
13	function AgentCall(address contract addr,bytes msg_data){
14	call contract addr = contract addr;
15	call msg_data = msg_data;
16	contract_addr.call(msg_data);
17	}
18	function () payable{
19	call contract addr.call(call msg_data);
20	}
21	}

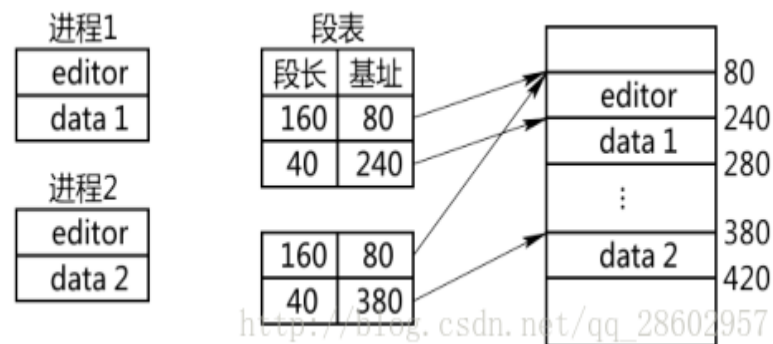
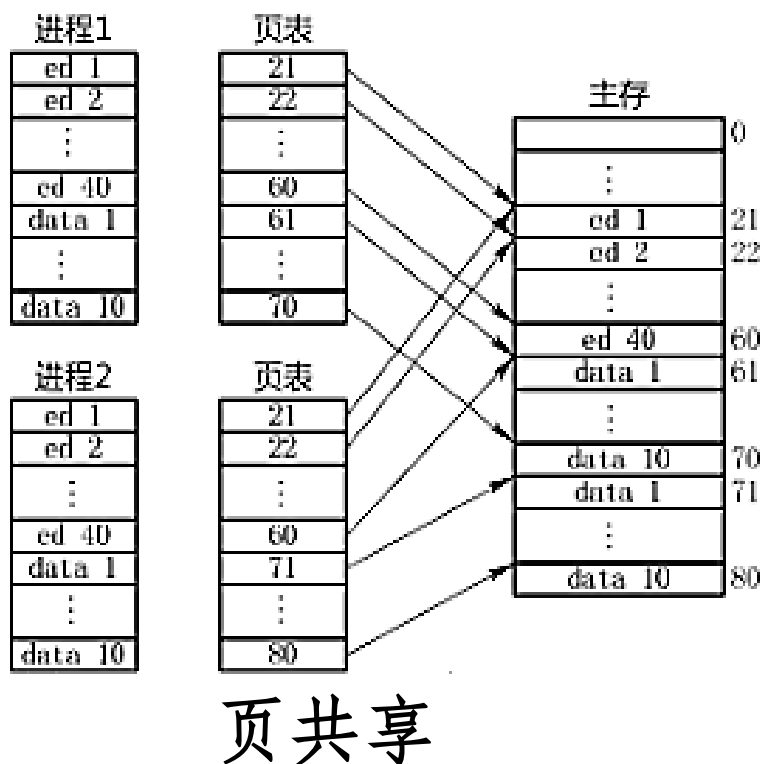
重入攻击-包含状态修改不应该可重入

1	contract BountyHunt{
2	...
3	function claimBounty() preventTheft {
4	uint balance = bountyAmount[msg.sender];
5	if (msg.sender.call.value(balance)()) {
6	totalBountyAmount -= balance;
7	bountyAmount[msg.sender] = 0;
8	}
9	}
10	}
11	contract AttackerAgent{
12	① ... ② ③
13	function AgentCall(address contract addr,bytes msg, data){
14	call contract addr.call(msg, data);
15	}
16	}
17	
18	
19	call contract addr.call(call msg, data);
20	}
21	}

The DAO bug: 6000万美元以太币损失

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：

- 地址变换花费时间；段表需要存储空间。
- 为满足分段的动态增长和减少外零头，要采用紧缩手段。
- 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，是系统对于主存的管理。
- 分段是用户可见的（可以在用户编程时确定，也可以在编译时根据数据性质确定）。

分页与分段的比较

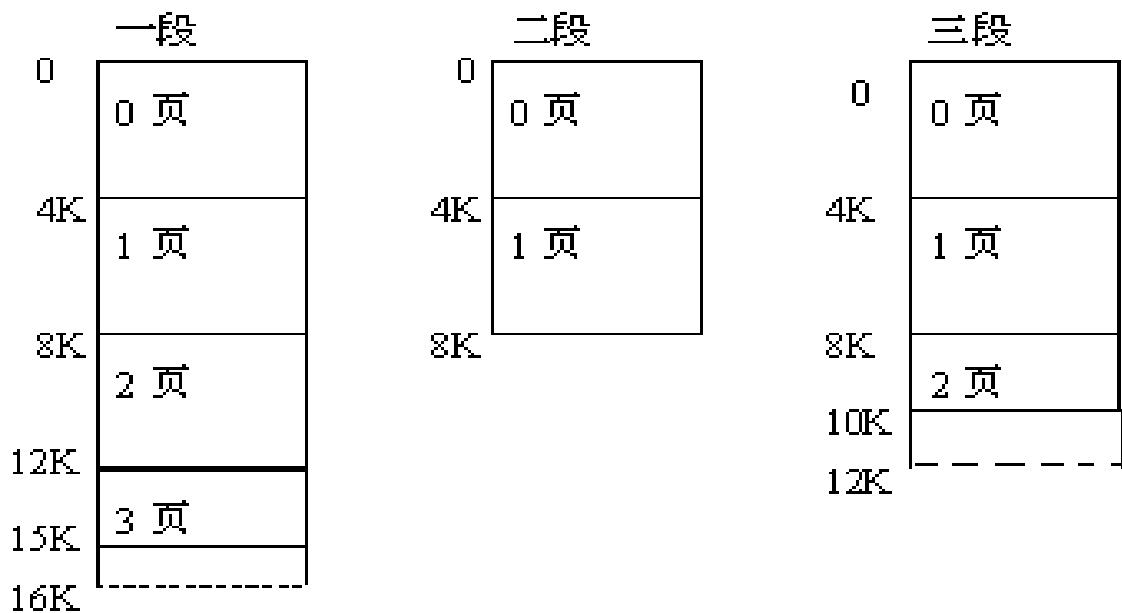
	页式存储管理	段式存储管理
目的	实现非连续分配，解决碎片问题	更好地满足用户需要
信息单位	页（物理单位）	段（逻辑单位）
大小	固定（由系统定）	不定（由用户程序定）
内存分配单位	页	段
作业地址空间	一维	二维
优点	有效解决了碎片问题（没有外碎片，每个内碎片不超过页大小）；有效提高内存的利用率；程序不必连续存放。	更好地实现数据共享与保护；段长可动态增长；便于动态链接

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）
段式存储管理，如果段规模很大，不适合完全放入内存 可以对段进行分页		
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物理存储器。



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。

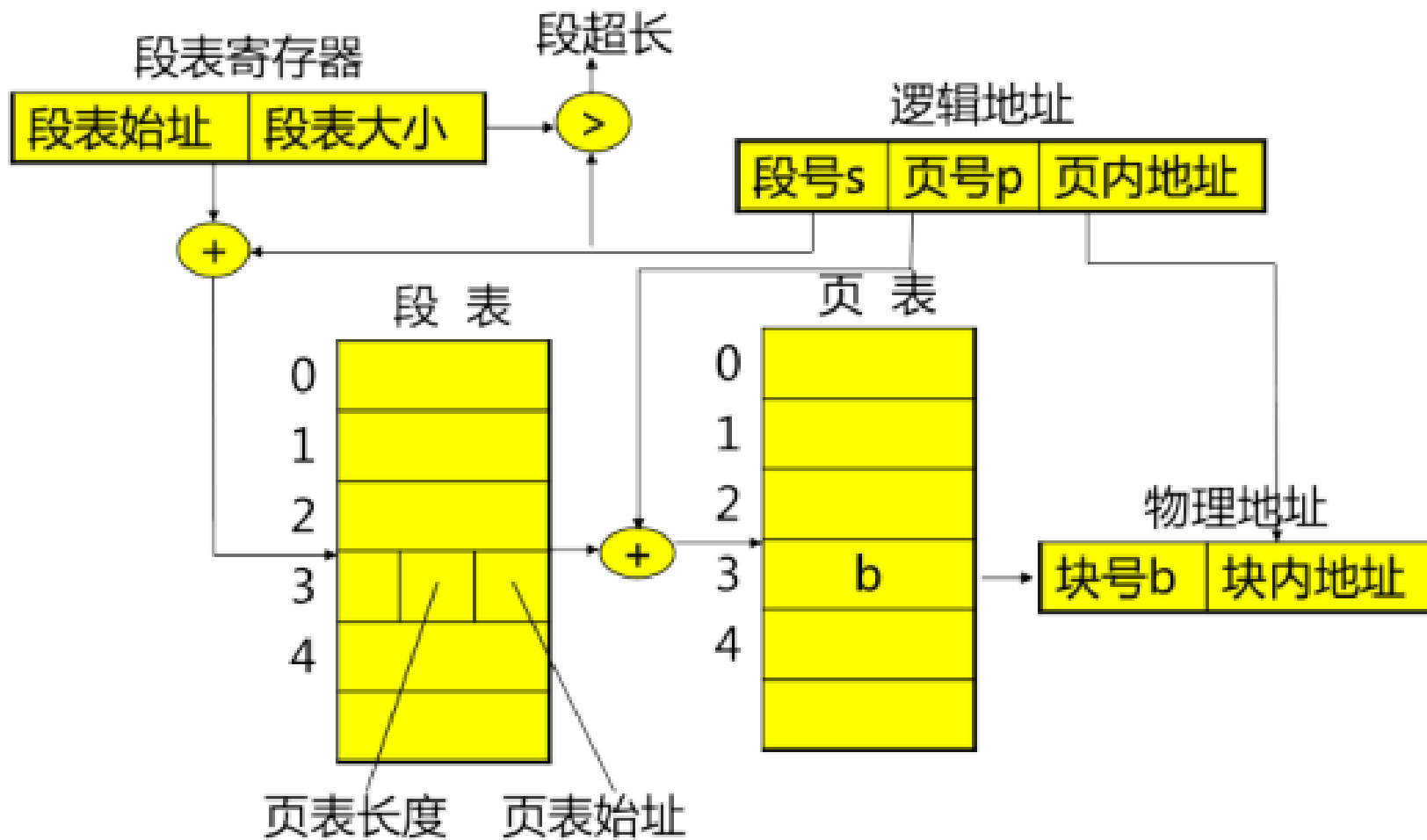
段号 (S)	段内页号 (P)	页内地址 (W)
--------	----------	----------

- 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，每个段表项对应一张页表。
- 段表含段号、页表始址和页表长度。页表含页号和块号。

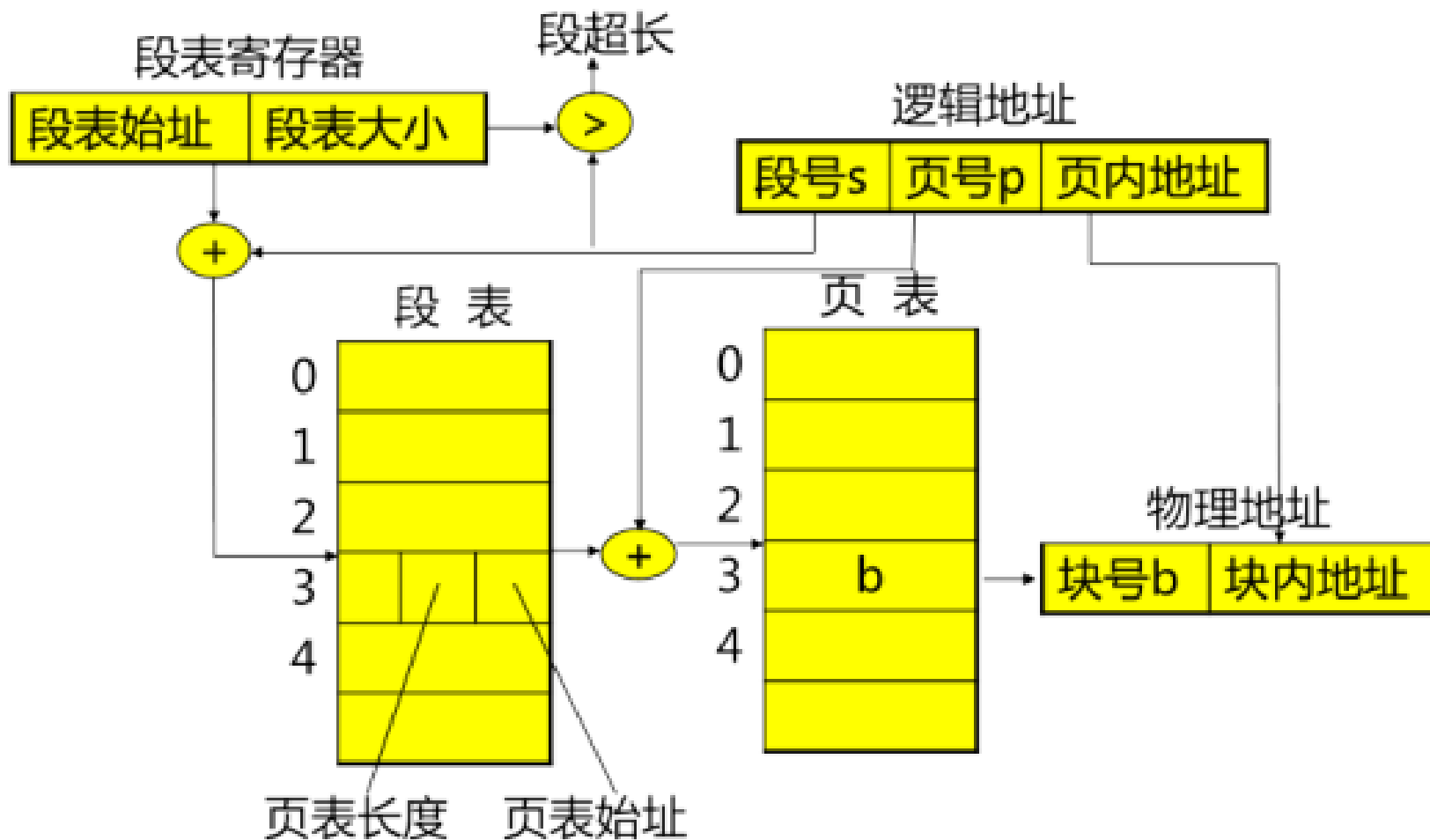
段页式存储管理的地址变换

- 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



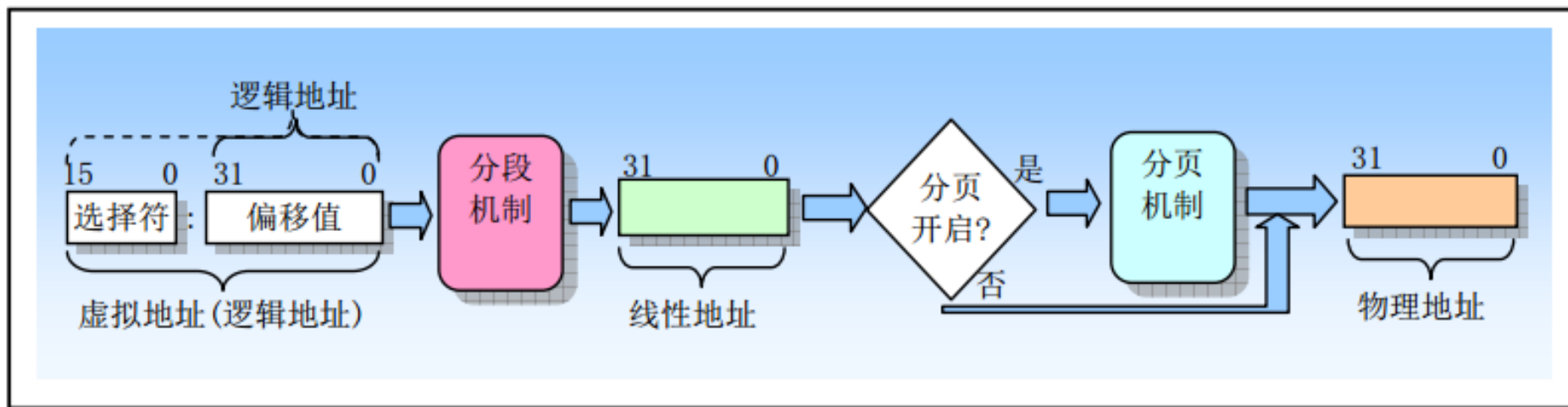
段页式存储管理的地址变换



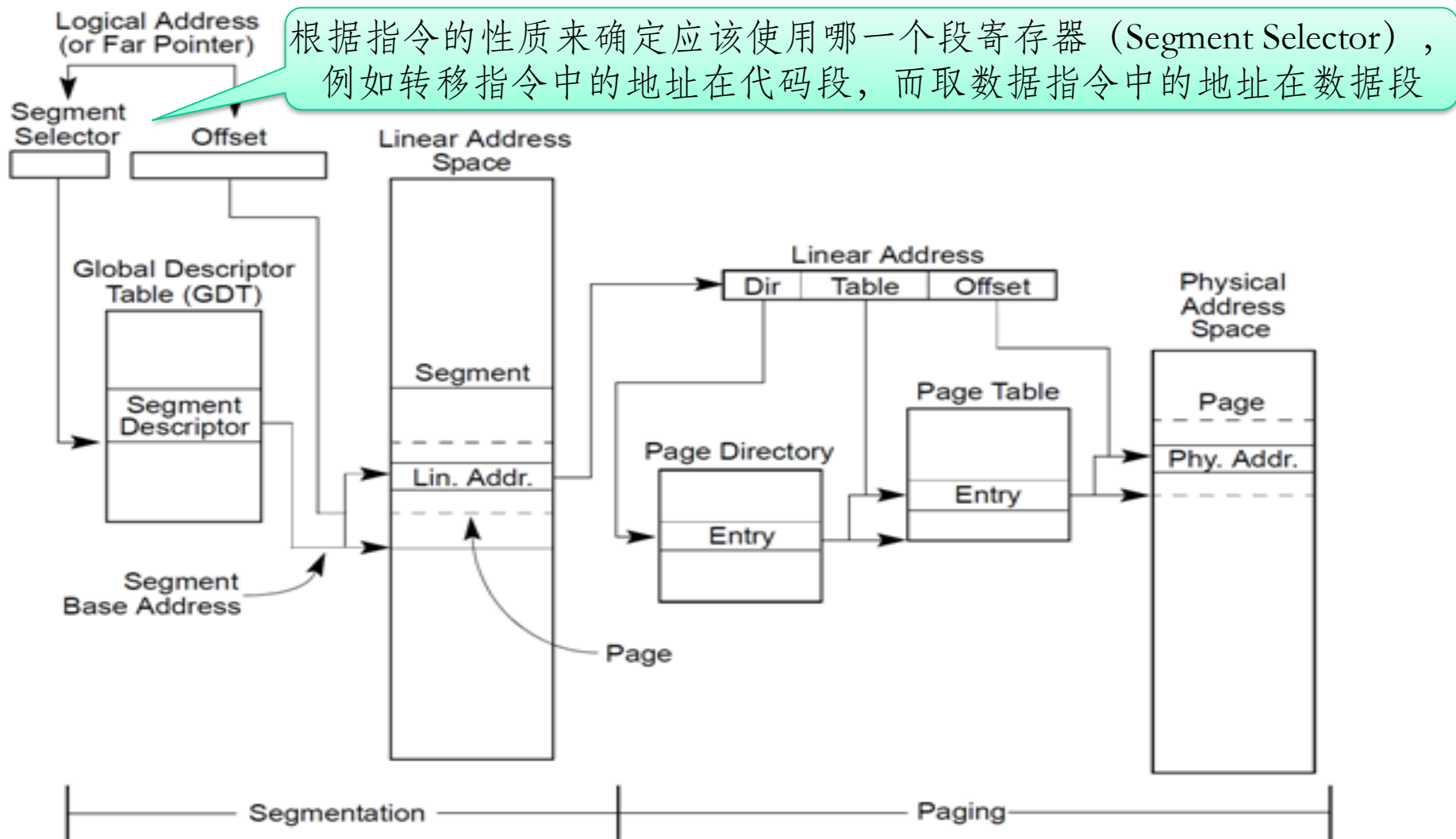
需要三次访存

实例：X86的段页式地址映射

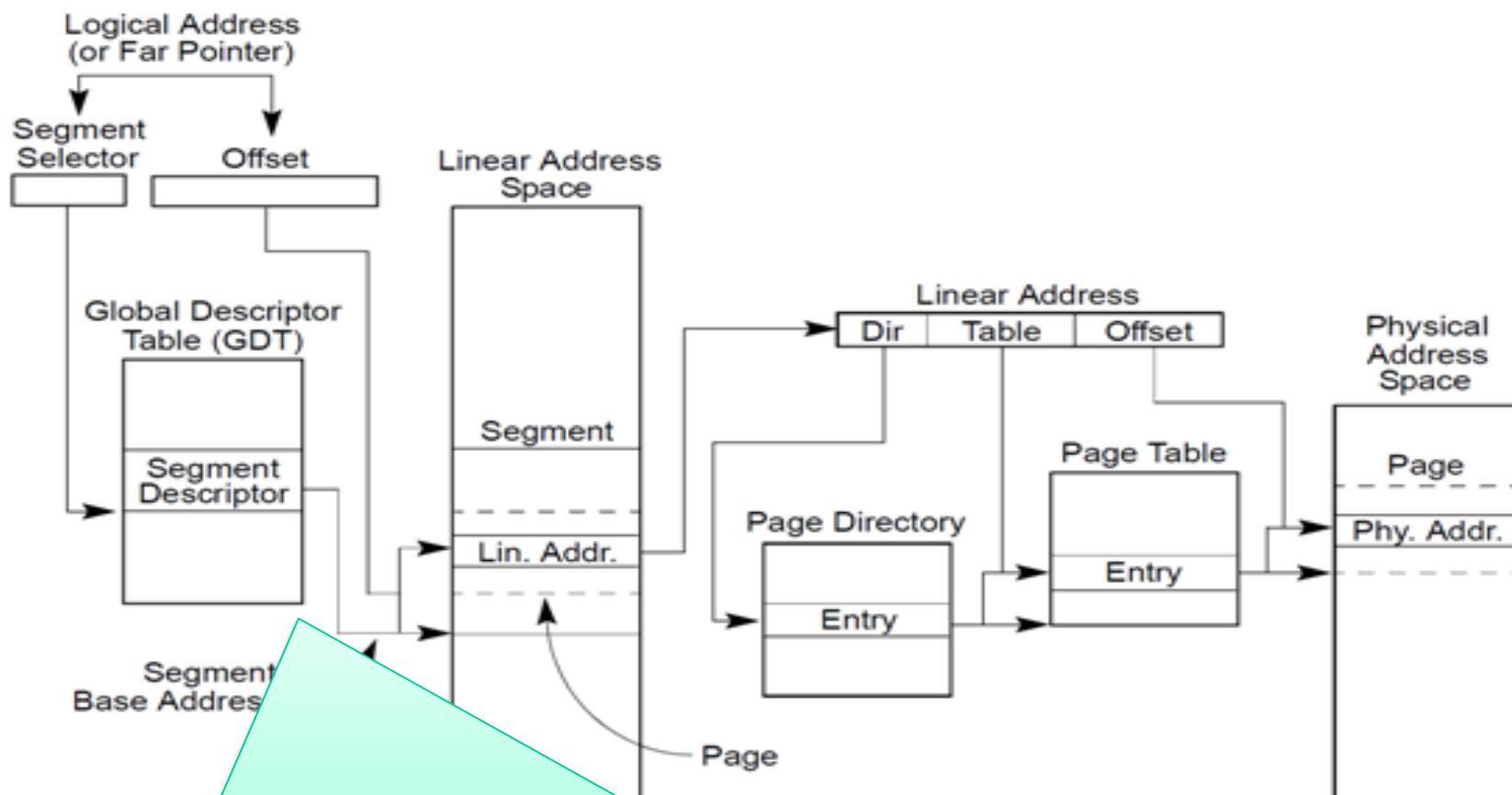
- X86的地址映射机制分为两个部分：
 - 段映射机制，将逻辑地址映射到线性地址；
 - 页映射机制，将线性地址映射到物理地址。



X86的段页式地址映射

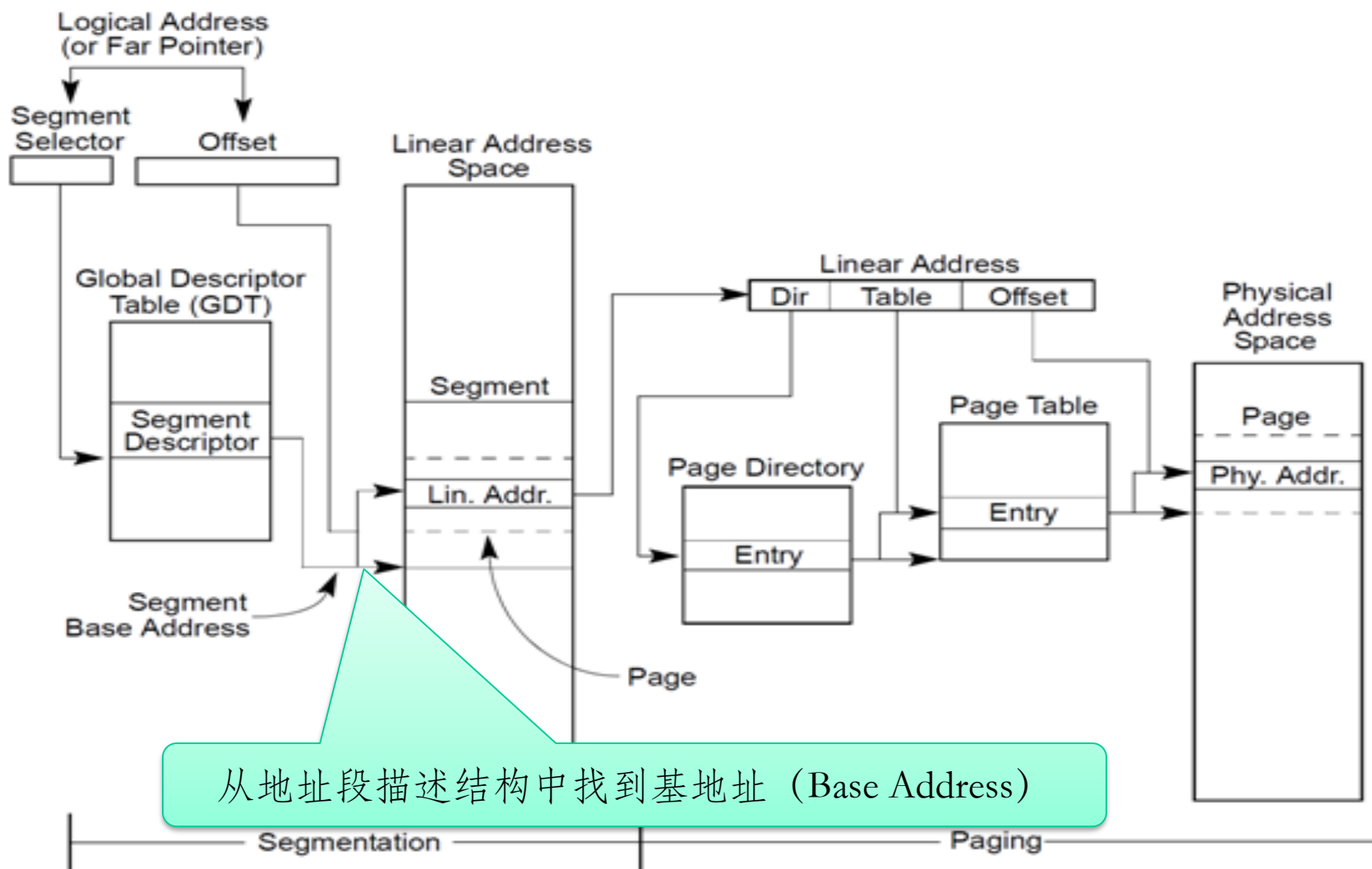


X86的段页式地址映射

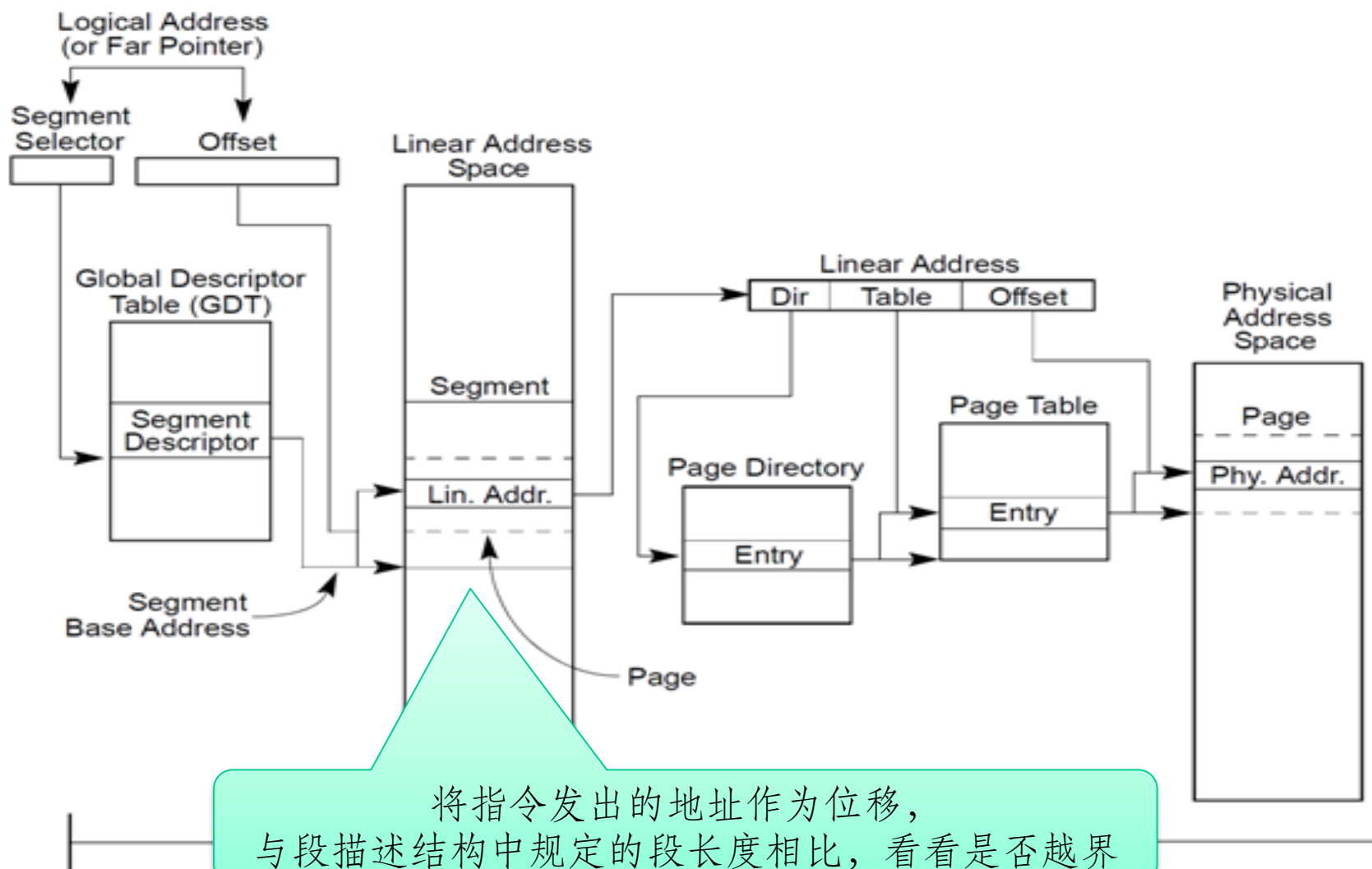


根据段寄存器的内容，找到相应的“地址段描述结构”（Segment Descriptor），段描述结构都放在一个表（Descriptor Table）中（GDT或LDT等），而表的起始地址保存在GDTR、LDTR等寄存器中

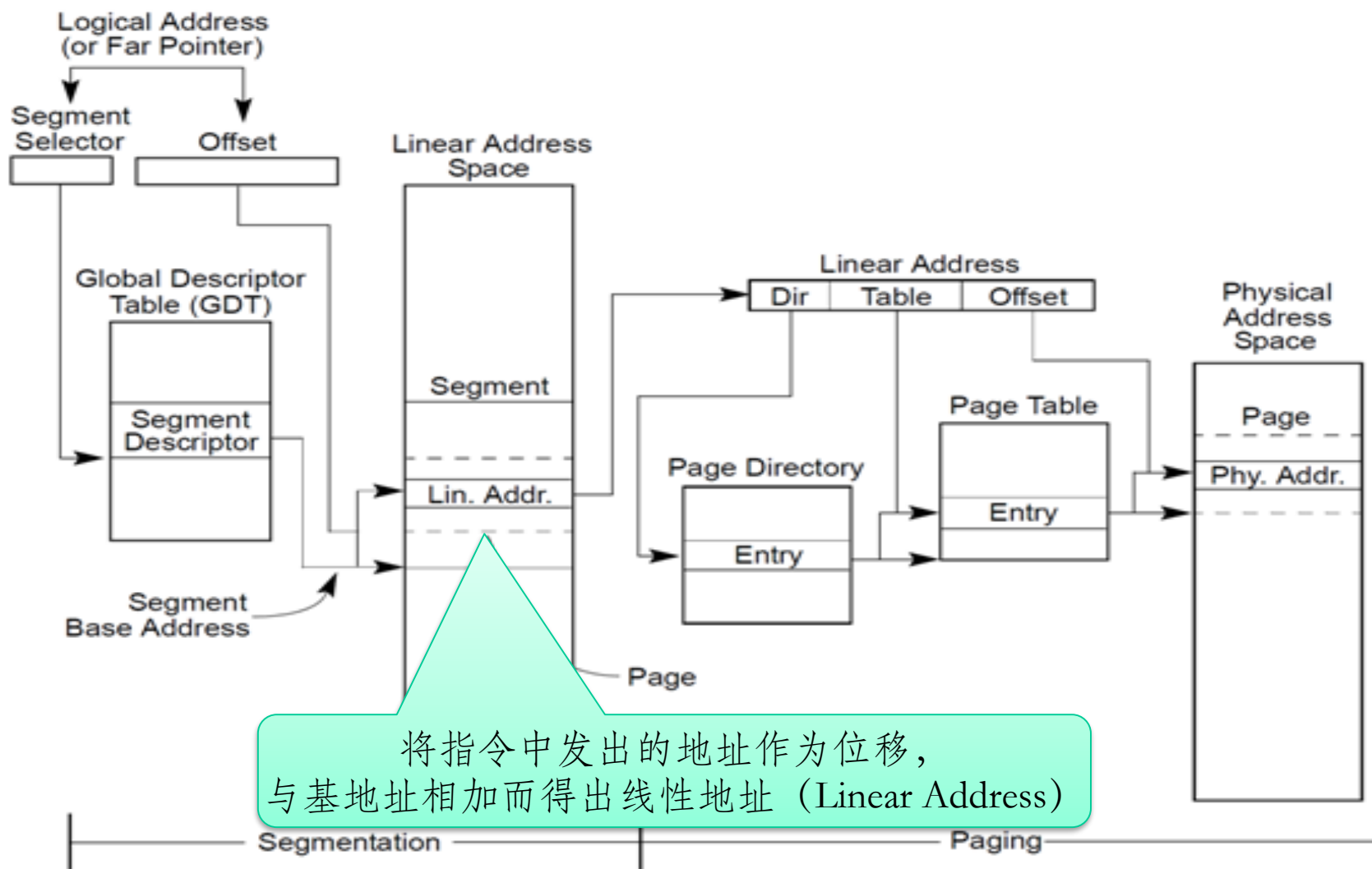
X86的段页式地址映射



X86的段页式地址映射



X86的段页式地址映射

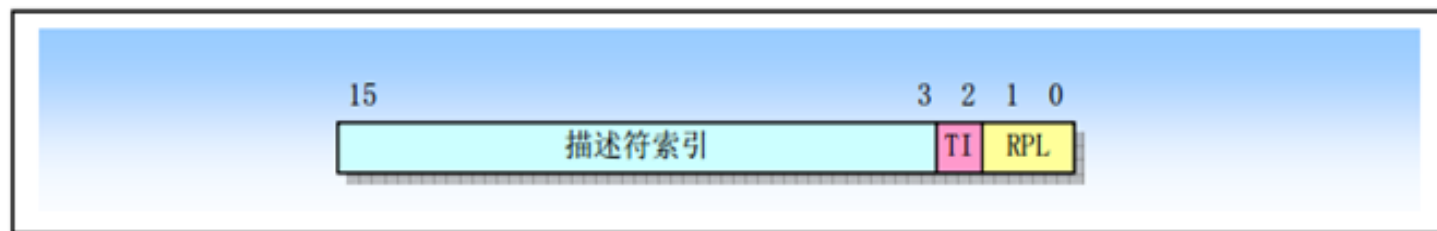


段式地址映射过程

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（Selector），
 - 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。

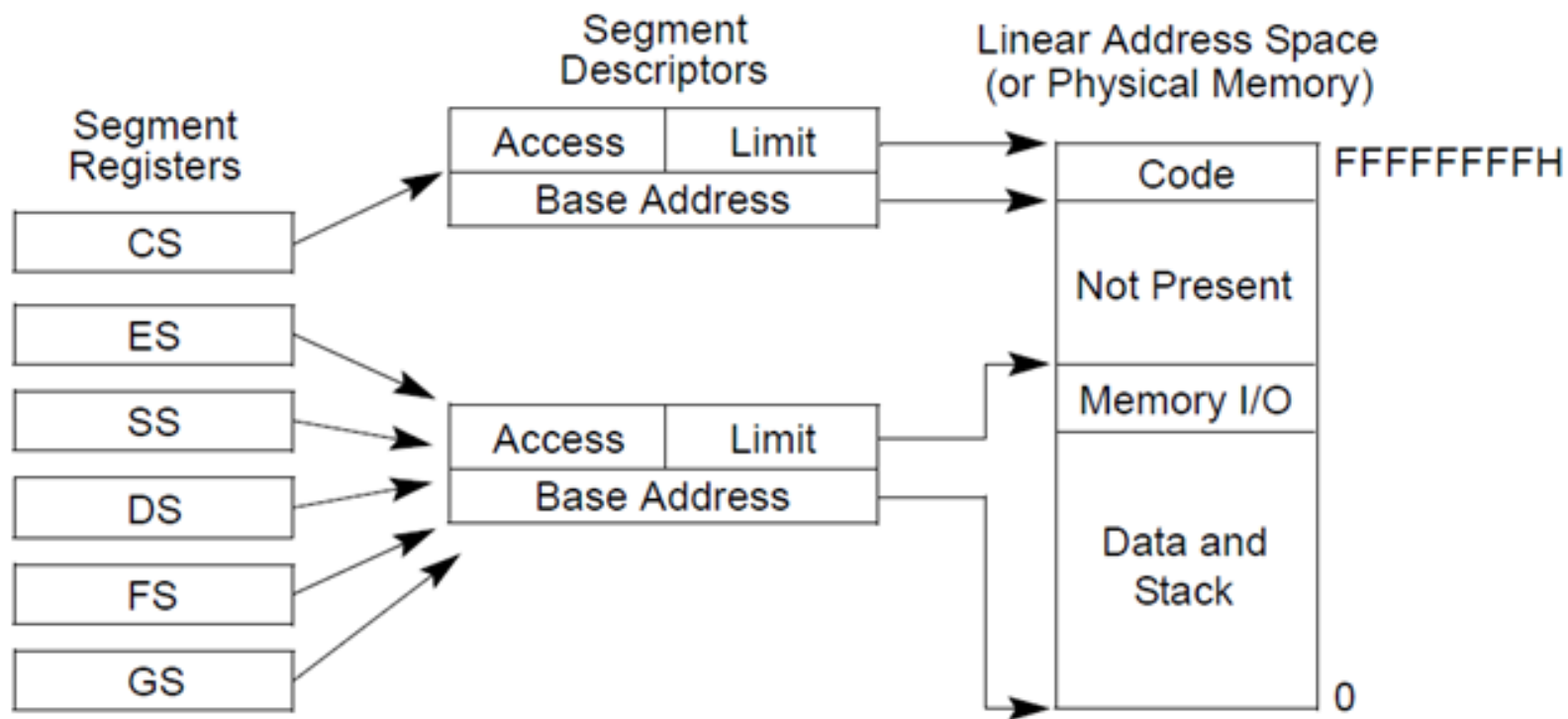


段选择符结构

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。

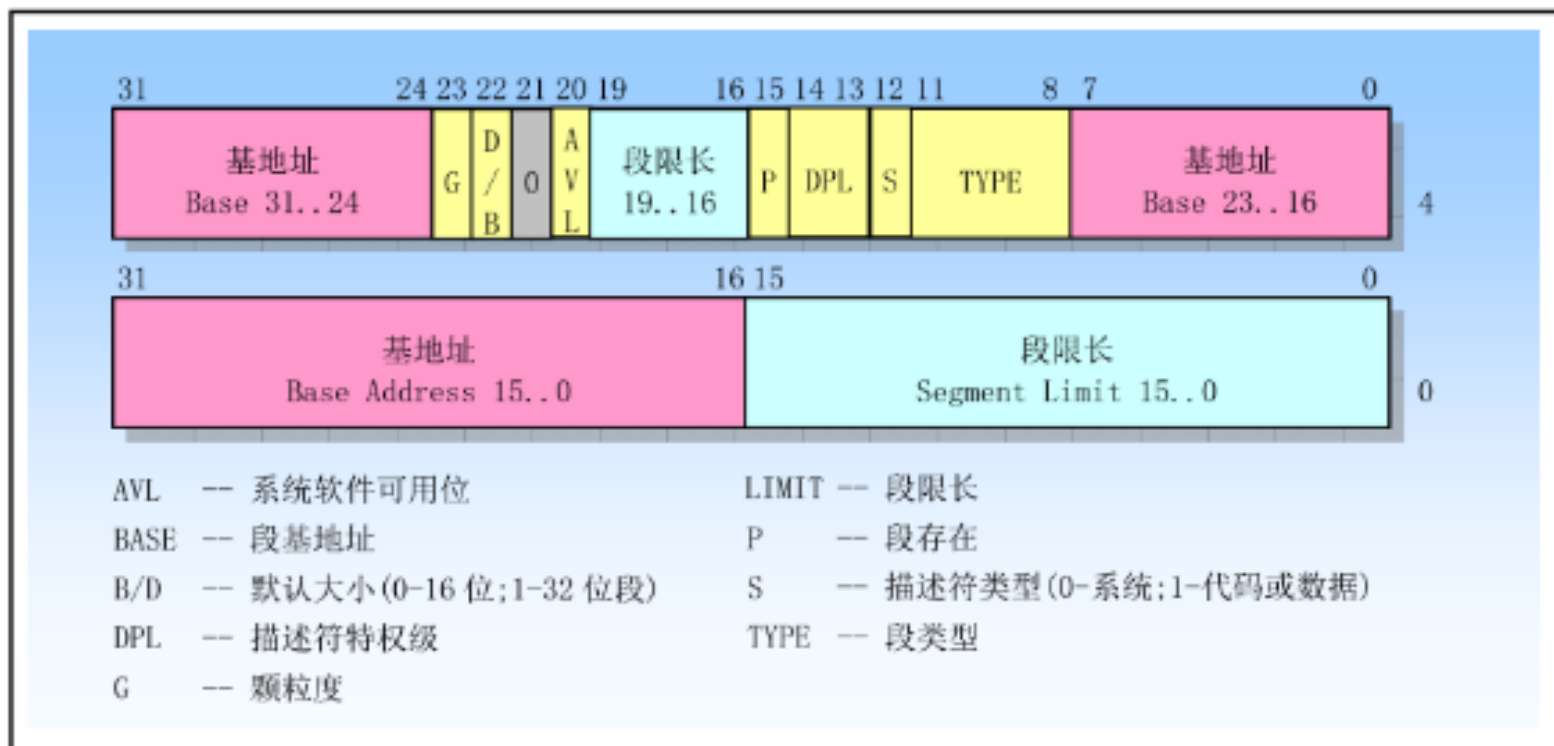
段寄存器



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



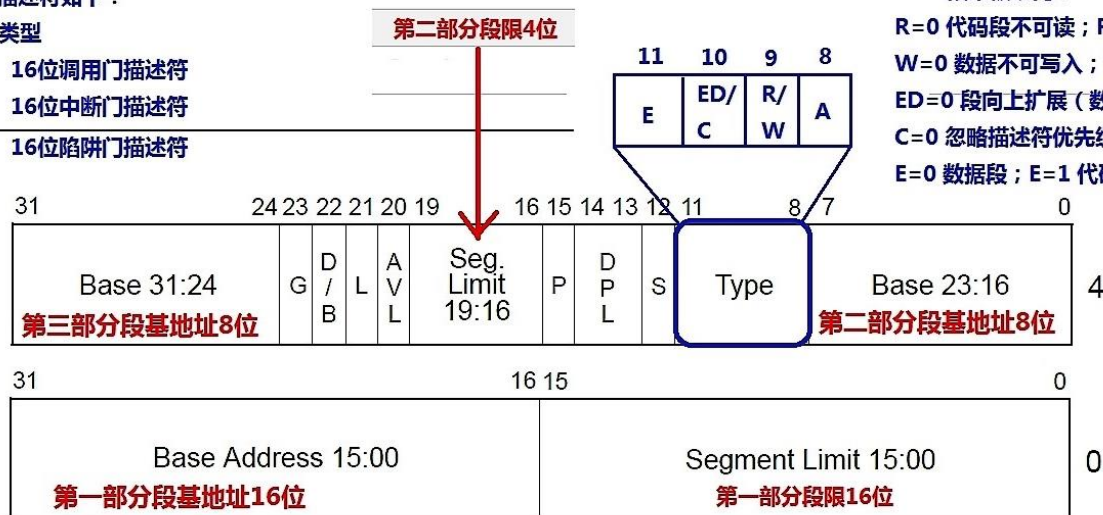
段描述符通用格式

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

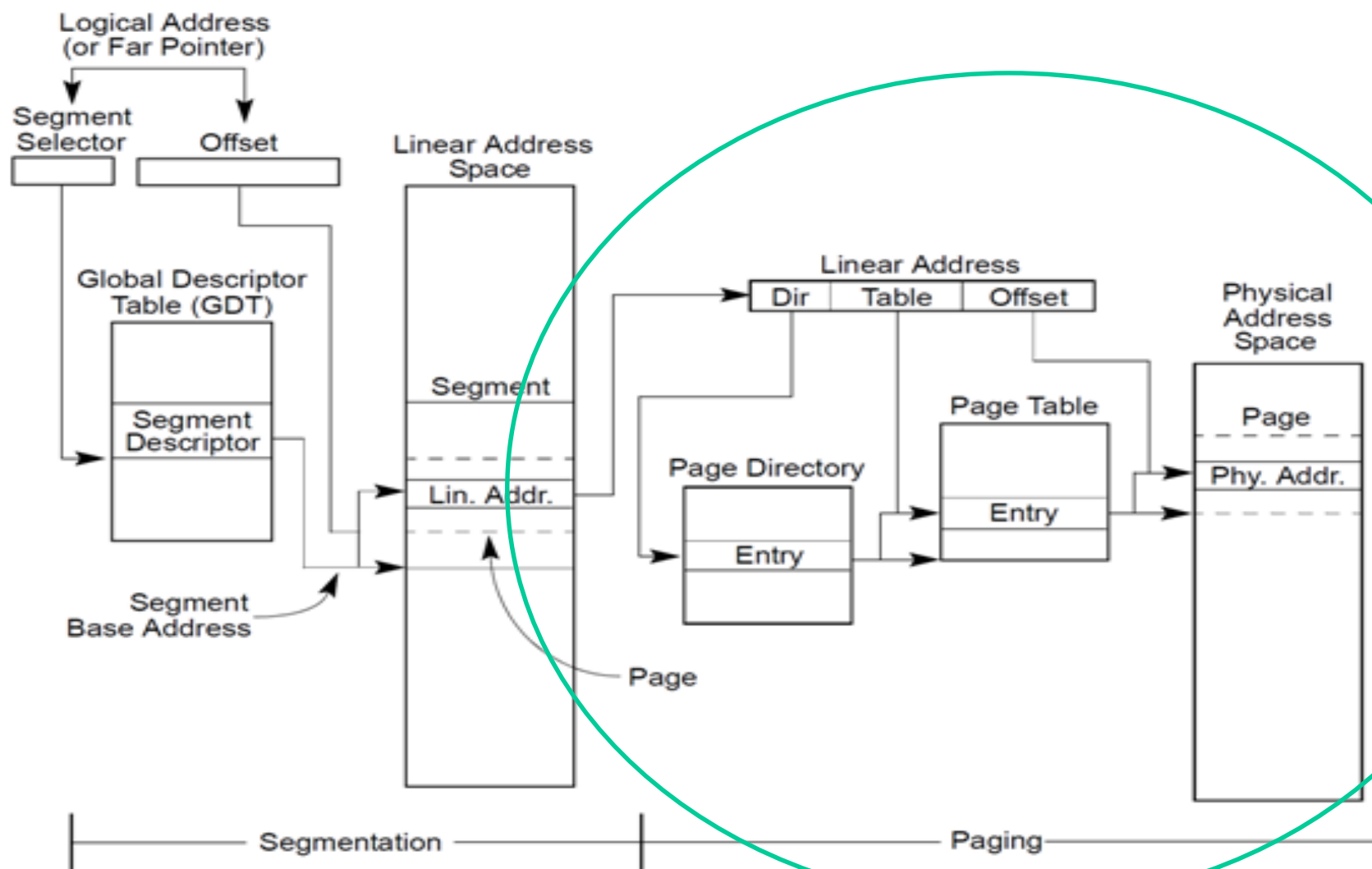
S — Descriptor type (0 = system; 1 = code or data) 段描述符类型，S=1为代码和数据段描述符，

TYPE — Segment type

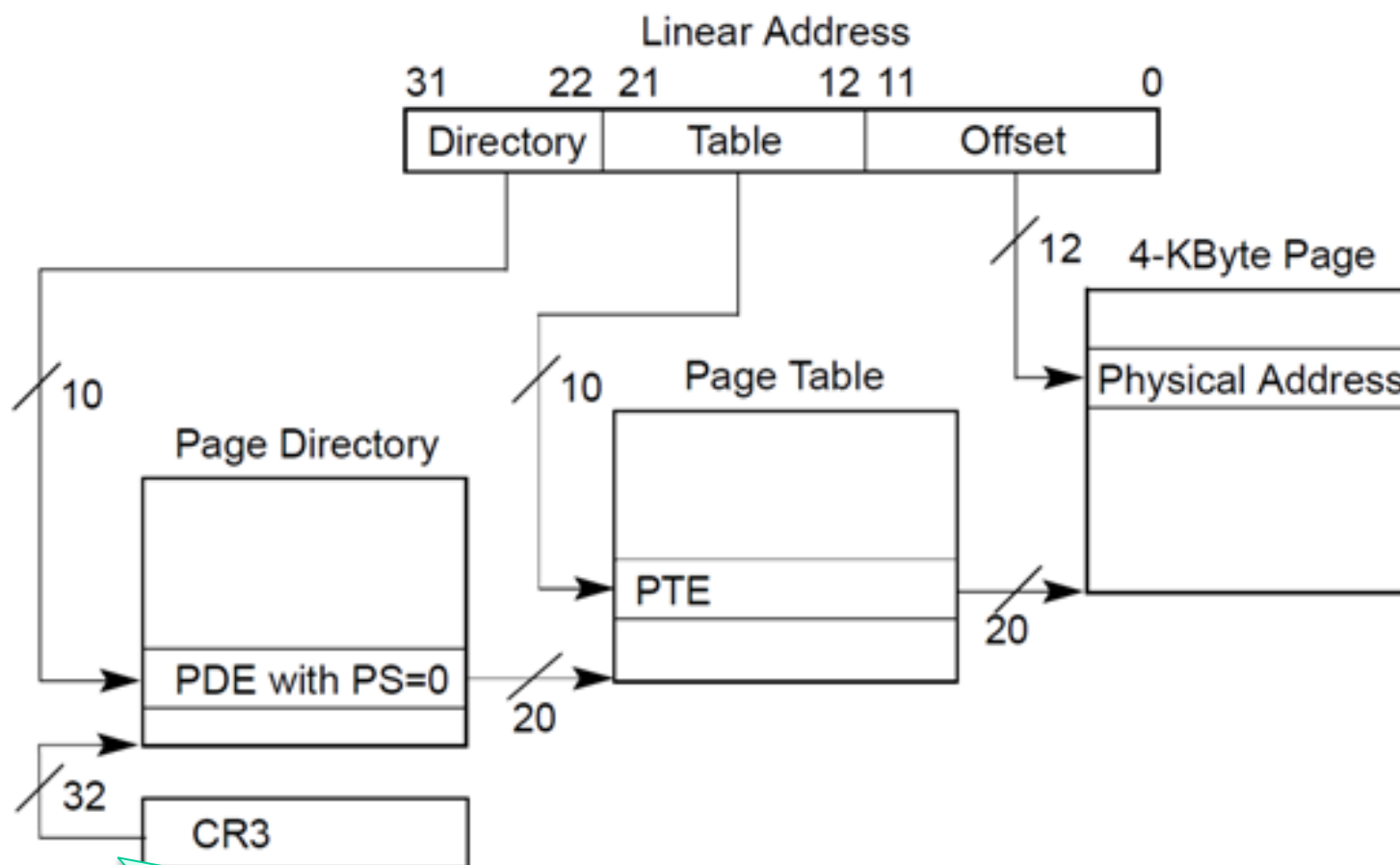
S=0为系统段描述符

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

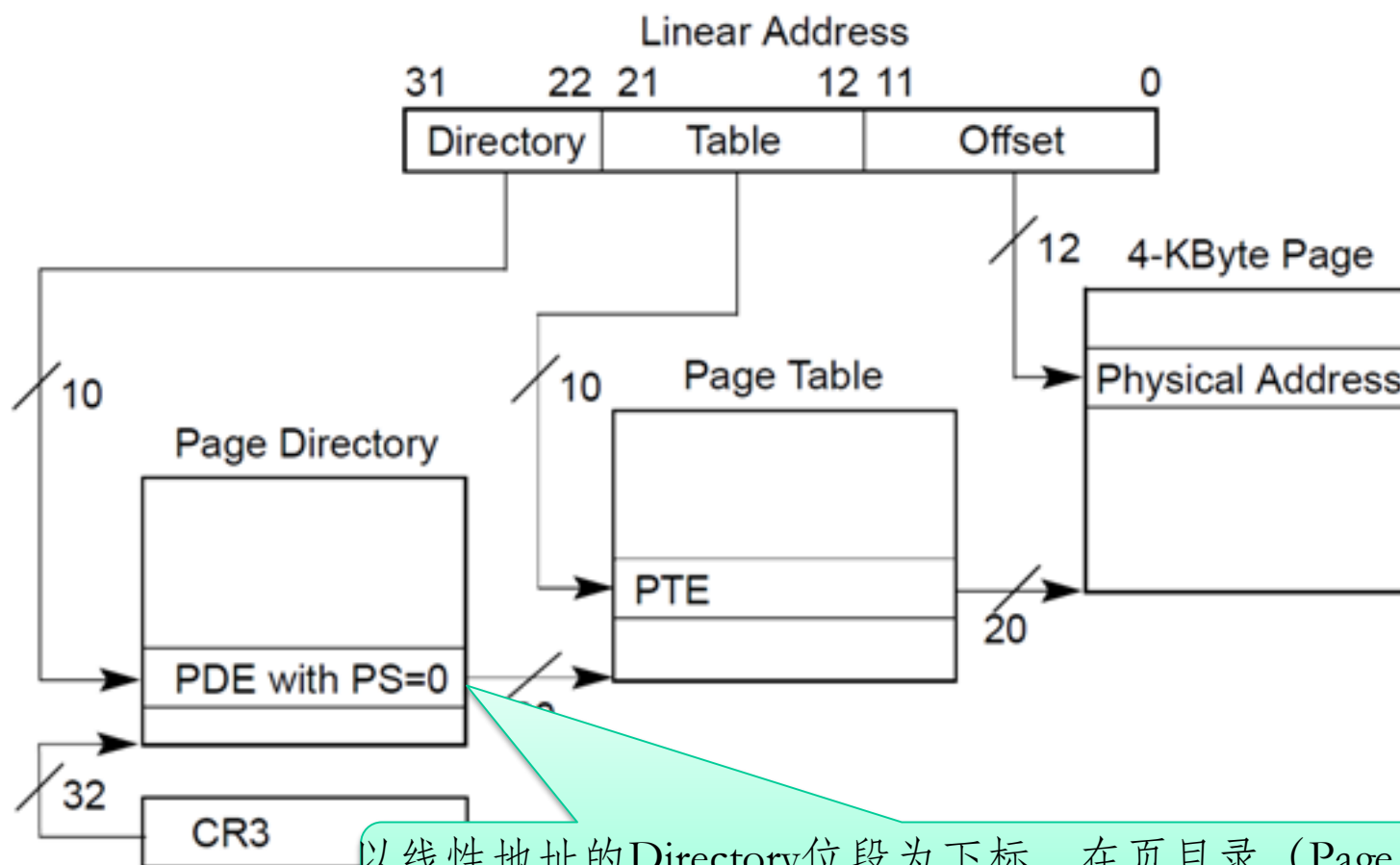


第二阶段：页式地址映射



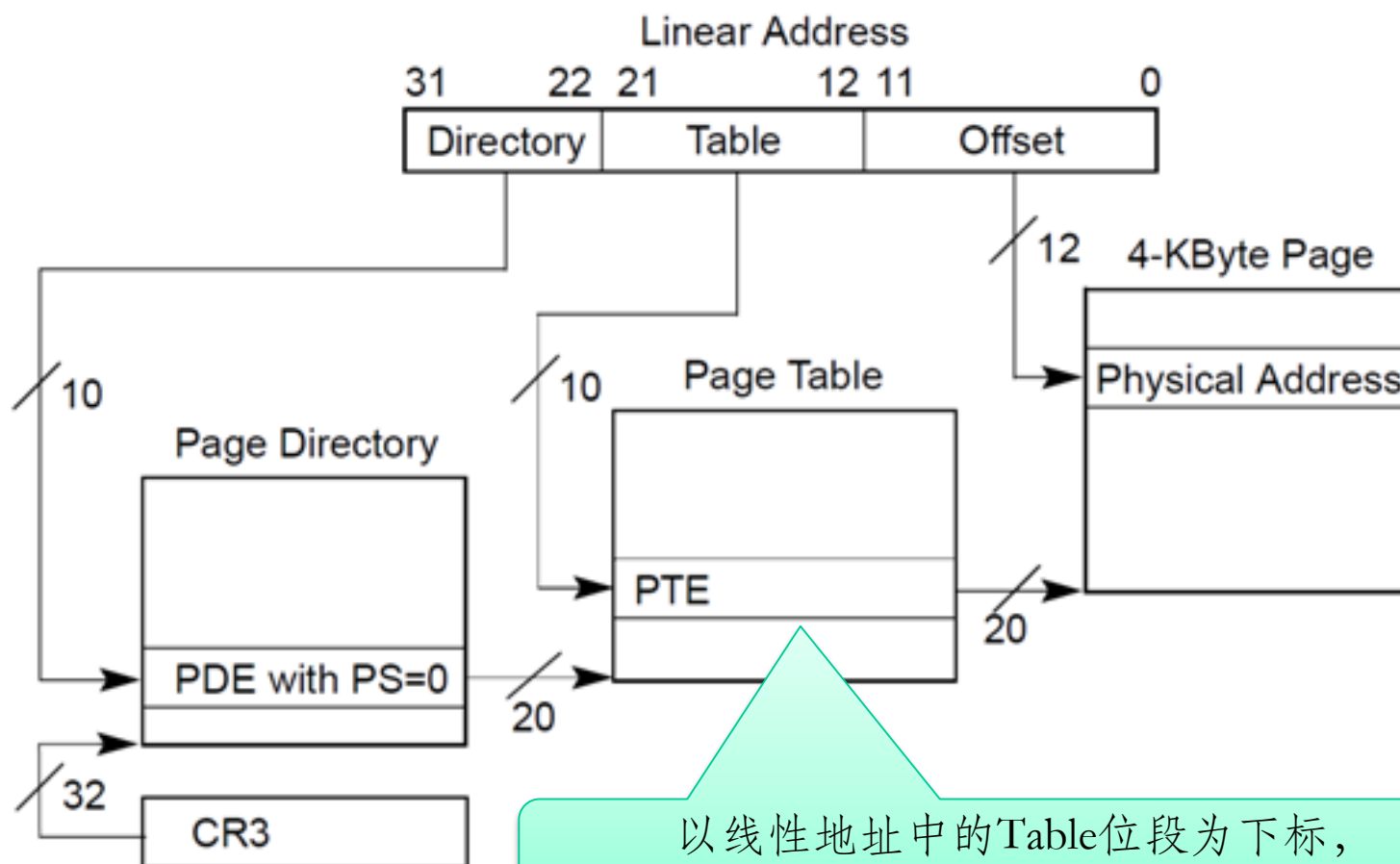
从CR3寄存器中获取页目录（Page Directory）的基地址；

第二阶段：页式地址映射

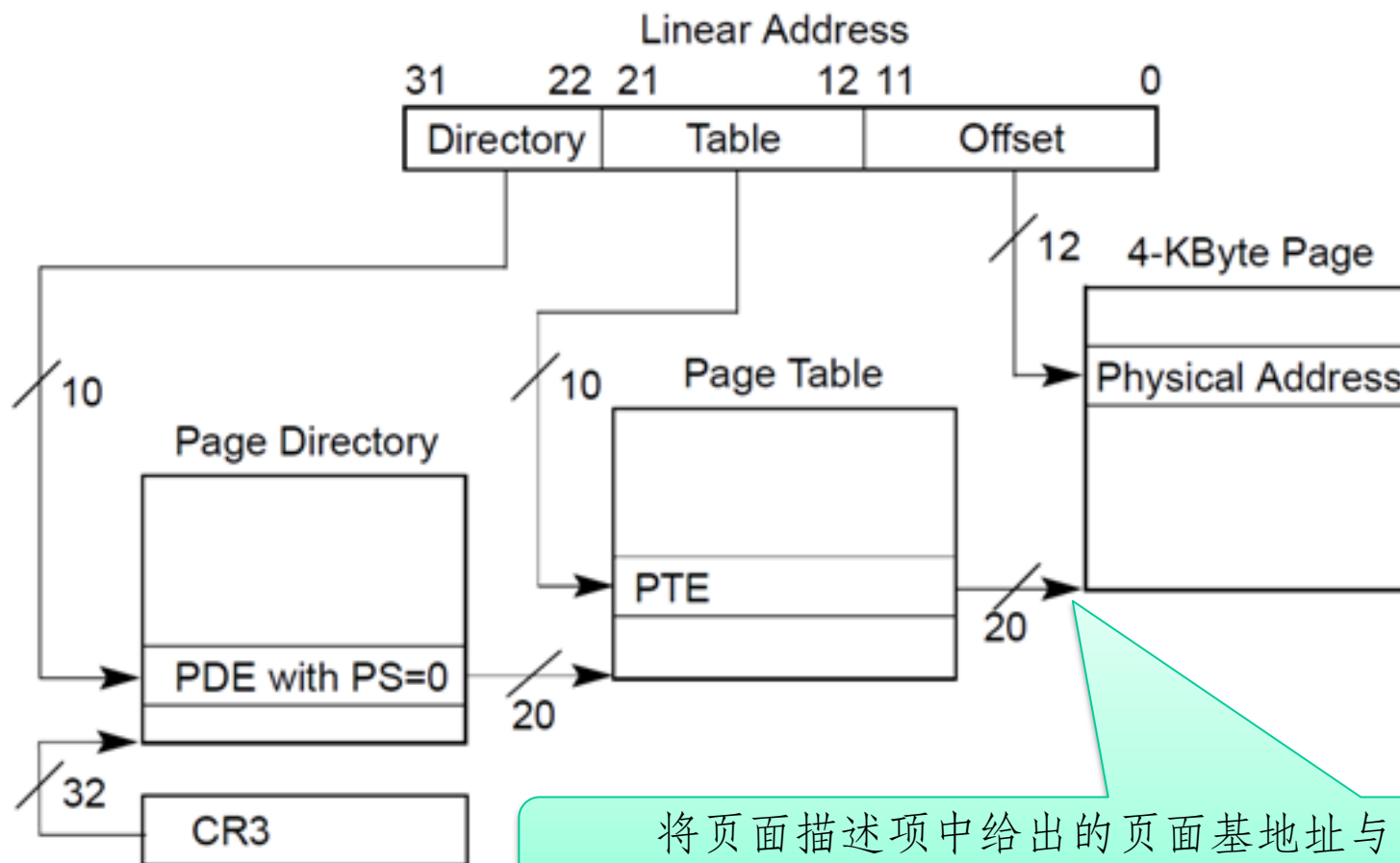


以线性地址的Directory位段为下标，在页目录（Page Directory）中取得相应页表（Page Table）的基地址

第二阶段：页式地址映射



第二阶段：页式地址映射



将页面描述项中给出的页面基地址与线性地址中的offset位段组合得到物理地址

页式地址映射过程

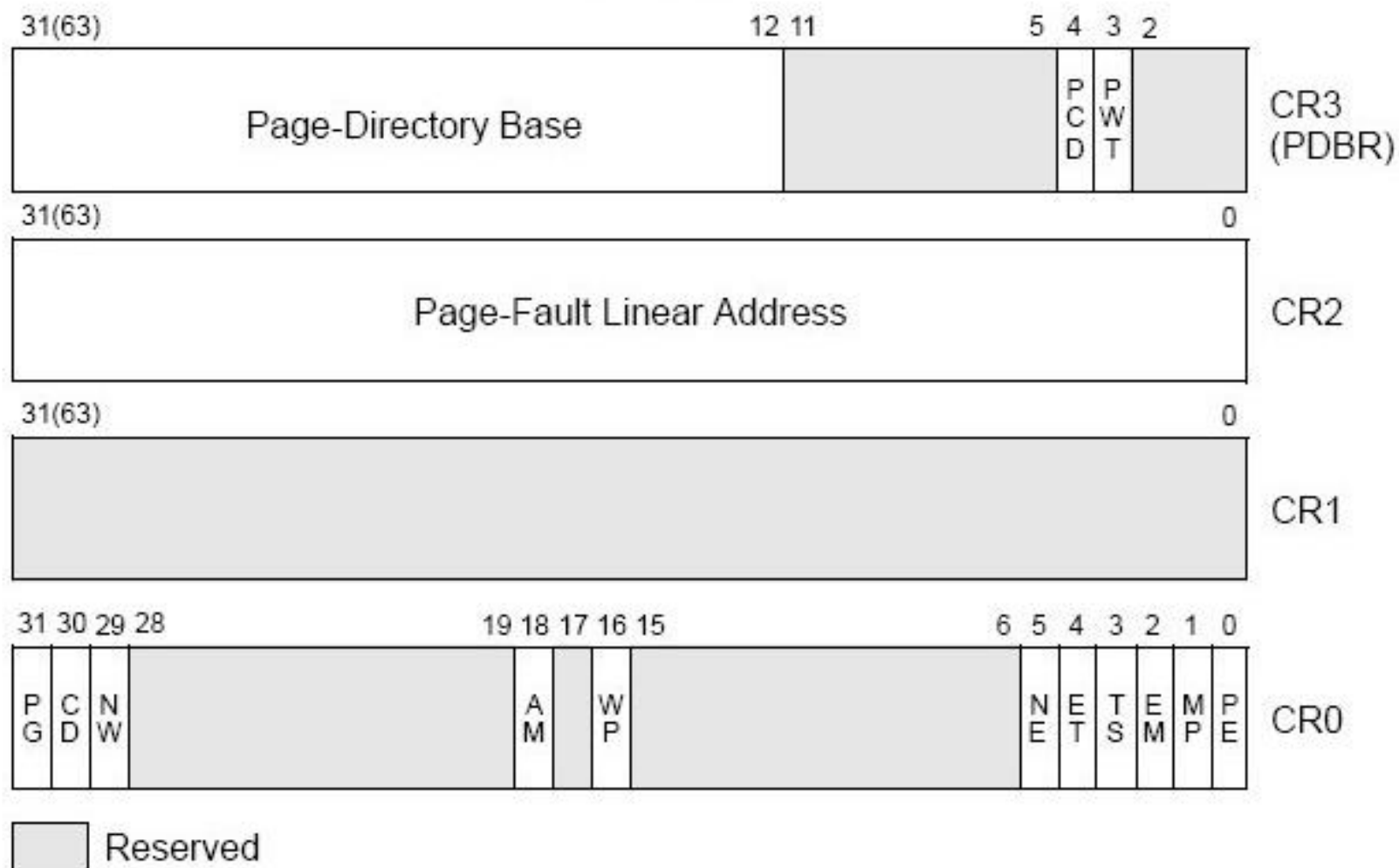
1. 从CR3寄存器中获取页目录（Page Directory）的基地址；
2. 以线性地址的Directory位段为下标，在目录（Page Directory）中取得相应页面表（Page Table）的基地址；
3. 以线性地址中的Table位段为下标，在所得到的页面表中获得相应的页面描述项；
4. 将页面描述项中给出的页面基地址与线性地址中的offset位段相加得到物理地址。

X86的控制寄存器

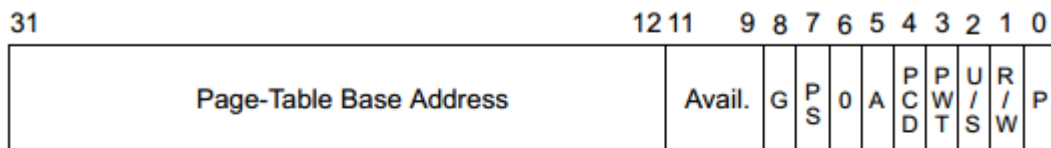
控制寄存器（CR0 ~ CR3）用于控制和确定处理器的操作模式以及当前执行任务的特性：

- 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。对于我们的实验，此位清零。

【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。对于我们的实验，此位清零。

【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。

内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
 - 局部性原理
 - 请求式分页
 - 页面置换
 - 内存保护
- 存储管理实例

内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
 - 局部性原理
 - 请求式分页
 - 页面置换
 - 内存保护
- 存储管理实例

