操作系统　Operating System

第三章 内存管理(4)

沃天宇
woty@buaa.edu.cn
2021年3月23日

关于实验

- gxemul单步调试
  - gxemul -M 64 -E testmips -v -V gxemul/vmlinux

- 善用make
  - make test

```
…
test:
        /OSLAB/gxemul -E testmips -M 64 -v $(vmlinux_elf)

include include.mk
```

```
GXemul> unassemble
<_start>
0000000080010000: 40806000      mtc0    zr,status
0000000080010004: 00000000      nop
0000000080010008: 40809000      mtc0    zr,watchlo
000000008001000c: 00000000      nop
0000000080010010: 40809800      mtc0    zr,watchhi
0000000080010014: 00000000      nop
0000000080010018: 40088000      mfc0    t0,config
000000008001001c: 2401fff8      addiu   at,zr,-8
0000000080010020: 01014024      and     t0,t0,at
0000000080010024: 35080002      ori     t0,t0,0x0002
0000000080010028: 40888000      mtc0    t0,config
000000008001002c: 3c1d8040      lui     sp,0x8040
0000000080010030: 0c004014      jal     0x0000000080010050      <main>
0000000080010034: 27bd0000      addiu   sp,sp,0
<loop>
0000000080010038: 0800400e      j       0x0000000080010038      <loop>
…
GXemul> print sp
0xffffffffa0007f00
GXemul> step
```

对比boot/start.S

关于实验

- 善用make
  - make test

```
…
test:
        /OSLAB/gxemul -E testmips -M 64 -v $(vmlinux_elf)

include include.mk
```

- 多读代码，特别是汇编代码
- gxemul单步调试
  - gxemul -M 64 -E testmips -v -V gxemul/vmlinux

REVISIT-页面大小

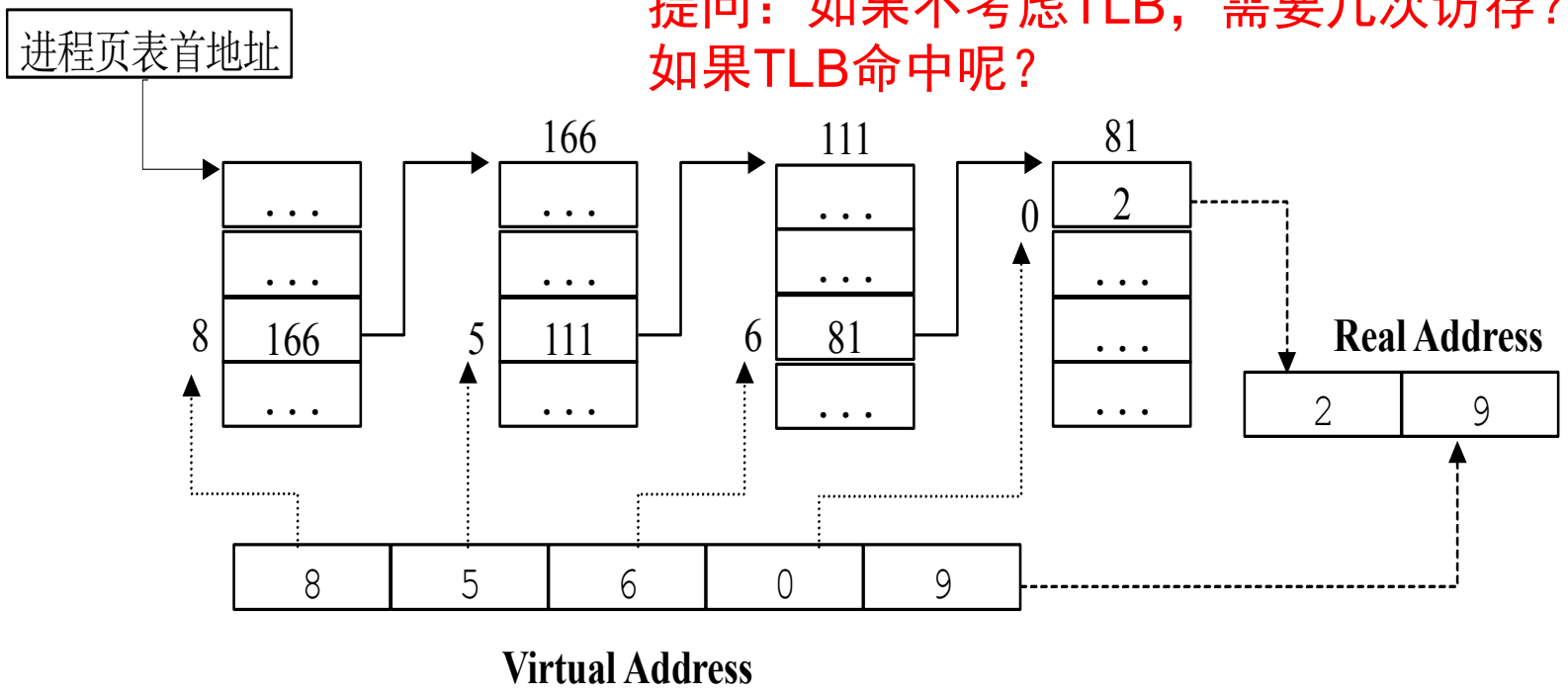
1. 假设进程的平均大小是1MB，每个页表项需要8个字节。页面大小设置为多少字节比较好？

答案：f(p) = se/p + p/2，求min f(p)
4KB。

REVISIT – 多级页表

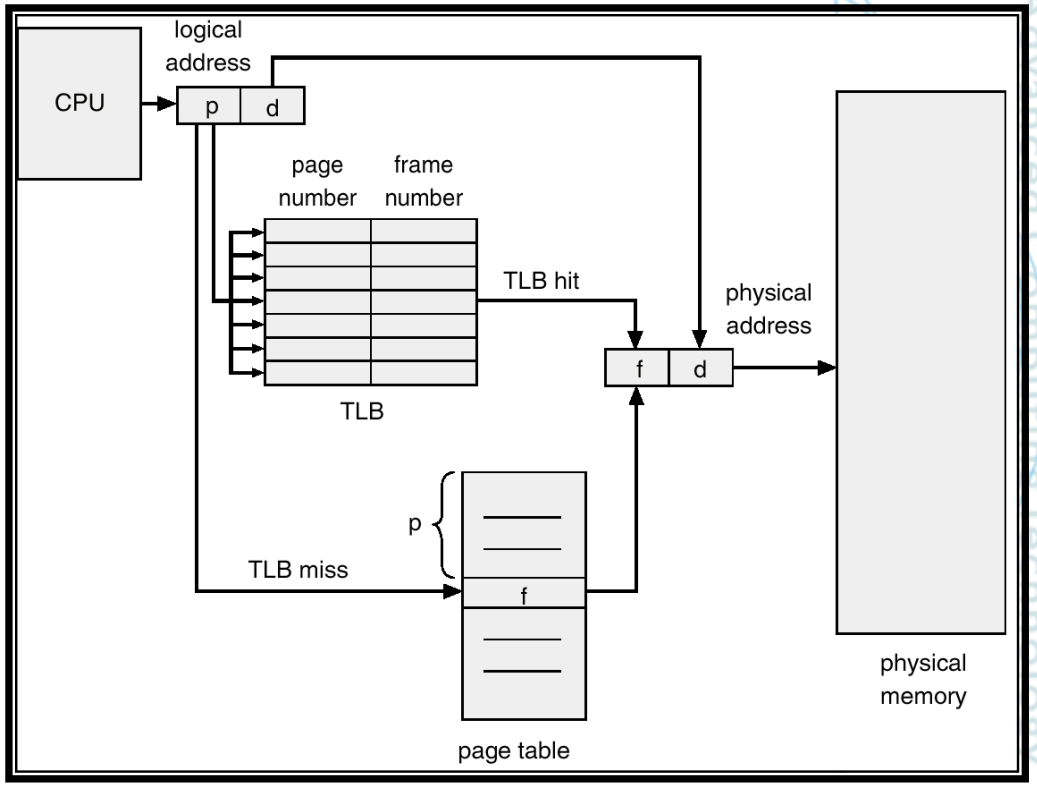
- 多级页表结构中，指令所给出的地址除偏移地址之外的各部分全是各级页表的页表号或页号，而各级页表中记录的全是物理页号，指向下级页表或真正的被访问页

提问：如果不考虑TLB，需要几次访存？
如果TLB命中呢？



快表（TLB）

- TLB（Translation Lookaside ）Buffer
- 专门针对页表项的高速缓存



快表(TLB)

- 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或全部内容。
- 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物理块；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。

快表示例

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

问题：怎么检索？怎么更新？

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）

TLB的其它特性（问题：所有TLB表项同样重要么？）

- 有的TLB在每个TLB条目中还保存地址空间标识码（address-space identifier，ASID）。ASID可用来唯一标识进程，并为进程提供地址空间保护。当TLB试图解析虚拟页号时，它确保当前运行进程的ASID与虚拟页相关的ASID相匹配。如果不匹配，那么就作为TLB失效。
- 除了提供地址空间保护外，ASID允许TLB同时包含多个进程的条目。如果TLB不支持独立的ASID，每次选择一个页表时（例如，上下文切换时），TLB就必须被冲刷（flushed）或删除，以确保下一个进程不会使用错误的地址转换。

TLB多说两句

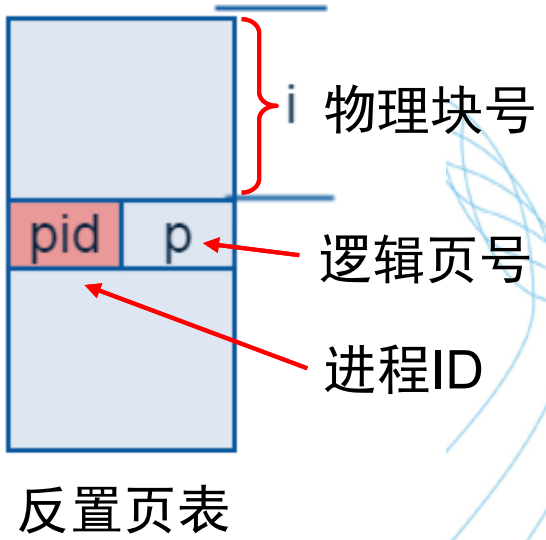
- TLB miss由谁来处理？
  - 硬件控制 vs 软件控制
  - 如何找到一个空闲的TLB表项？
  - TLB表项内容？和多级页表的关系？
  - 注意区分TLB miss和page fault （后者通常需要由OS软件处理）
- 硬件控制TLB：IA-32、IA-64
- 软件控制TLB：SPARC、MIPS、Alpah、HP-PA、PowerPC

反置页表(Inverted page table)

- 一般意义上，每个进程都有一个相关页表。该进程所使用的每个页都在页表中有一项。这种页的表示方式比较自然，这是因为进程是通过页的虚拟地址来引用页的。操作系统必须将这种引用转换成物理内存地址。
- 这种方法的缺点之一是每个页表可能有很多项。这些表可能消耗大量物理内存，却仅用来跟踪物理内存是如何使用的。如每个使用32位逻辑地址的进程其页表长度均为4MB。
- 为了解决这个问题，可以使用反向页表（inverted pagetable）。

反置页表(Inverted page table)

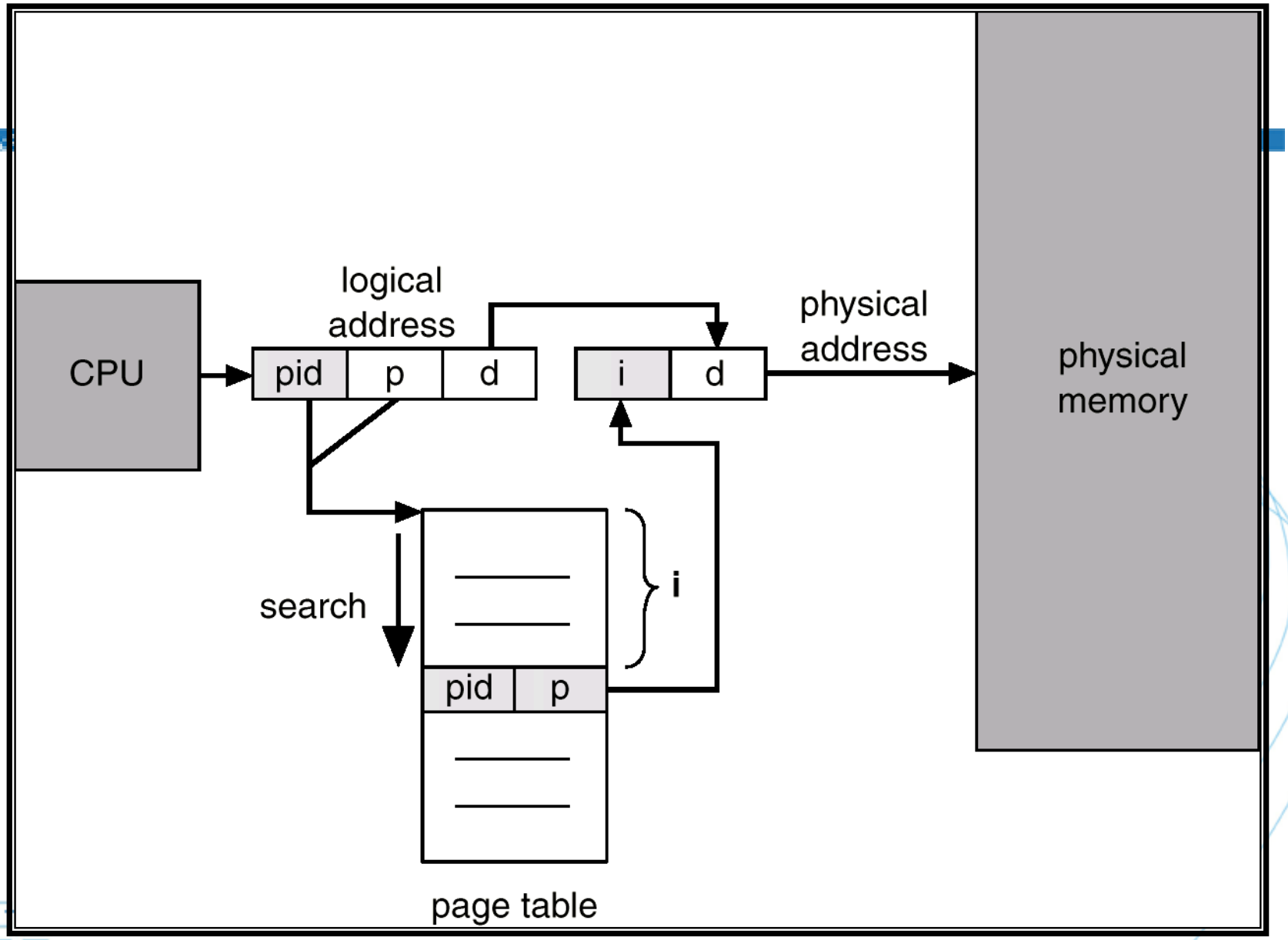
- 反置页表不是依据进程的逻辑页号来组织，而是依据该进程在内存中的物理页面号来组织（即：按物理页面号排列），其表项的内容是逻辑页号 P 及隶属进程标志符 pid 。
- 反置页表的大小只与物理内存的大小相关，与逻辑空间大小和进程数无关。如: 64M主存,若页面大小为 4K,则反向页表只需 64KB。
- 如64位的PowerPC, UltraSparc等处理器。



反置页表

利用反置页表进行地址变换：

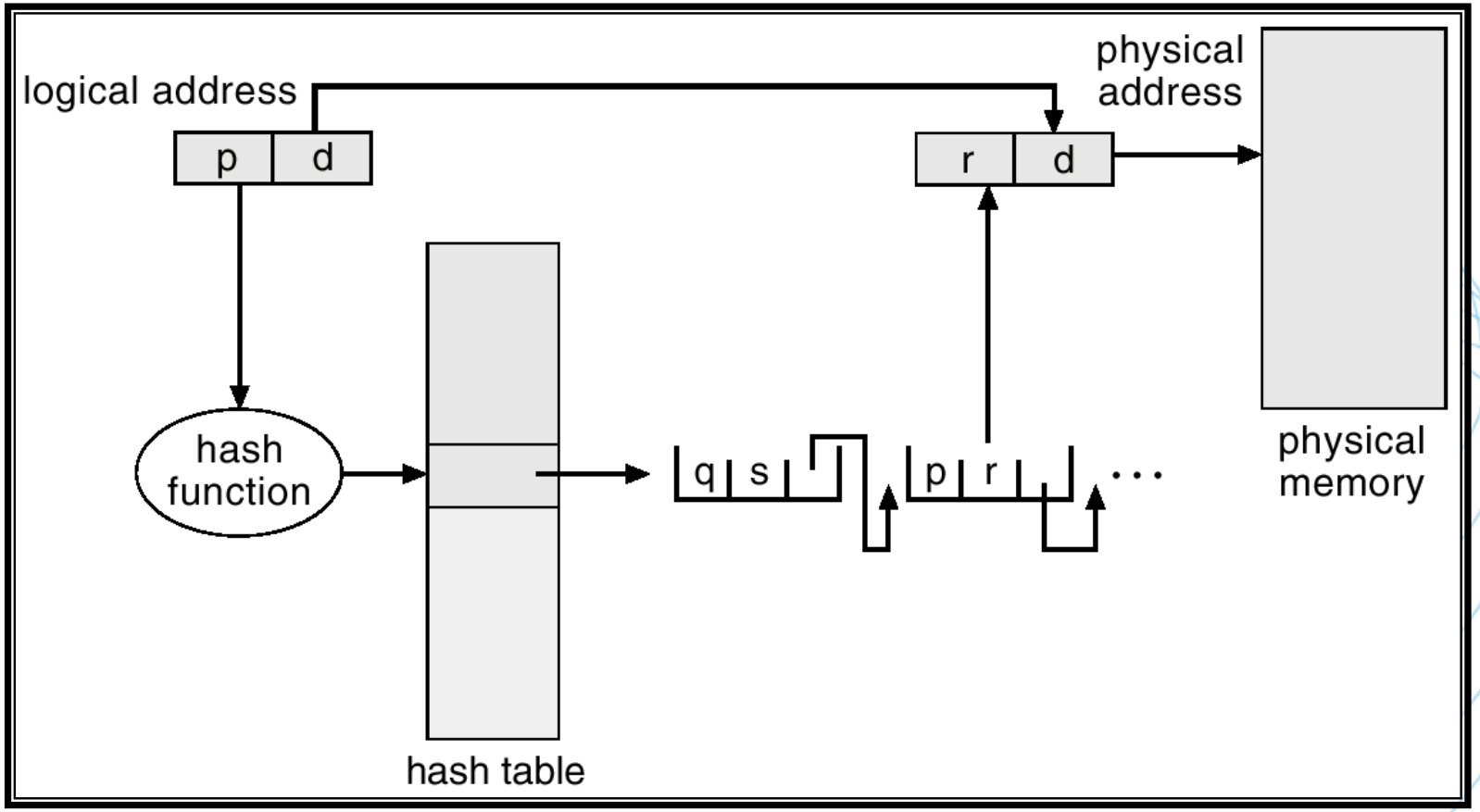
- 用进程标志符和页号去检索反置页表。
- 如果检索完整个页表未找到与之匹配的页表项，表明此页此时尚未调入内存，对于具有请求调页功能的存储器系统产生请求调页中断，若无此功能则表示地址出错。
- 如果检索到与之匹配的表项，则表项的序号 i 便是该页的物理块号，将该块号与页内地址一起构成物理地址。



反置页表(Inverted page table)

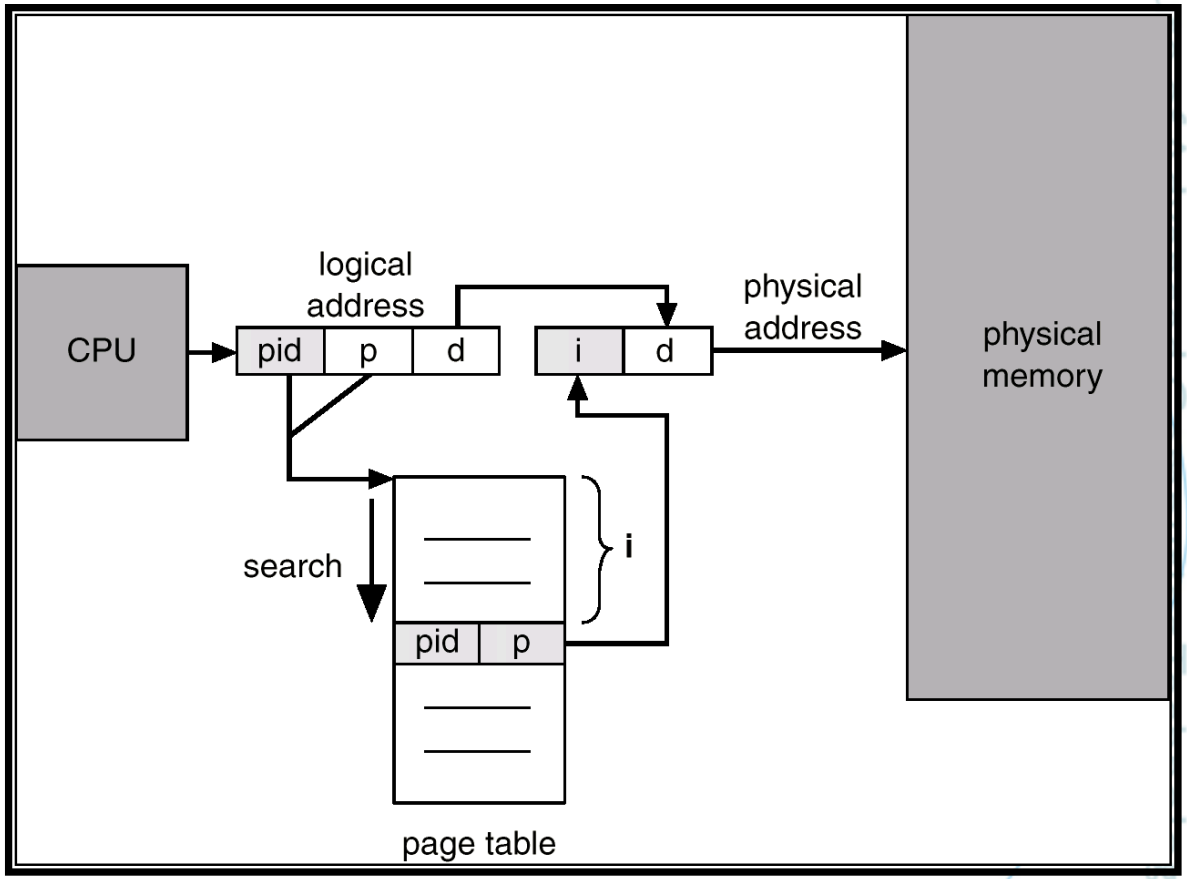
- 反向页表按照物理地址排序，而查找依据虚拟地址，所以可能需要查找整个表来寻找匹配。
- 可以使用 哈希页表 限制页表条目或加入 TLB 来改善。
- 通过哈希表(hash table)查找可由逻辑页号得到物理页面号。虚拟地址中的逻辑页号通过哈希表指向反置页表中的表项链头（因为哈希表可能指向多个表项），得到物理页面号。
- 采用反向页表的系统很难共享内存，因为每个物理帧只对应一个虚拟页条目。

哈希页表（Hashed Page Table）



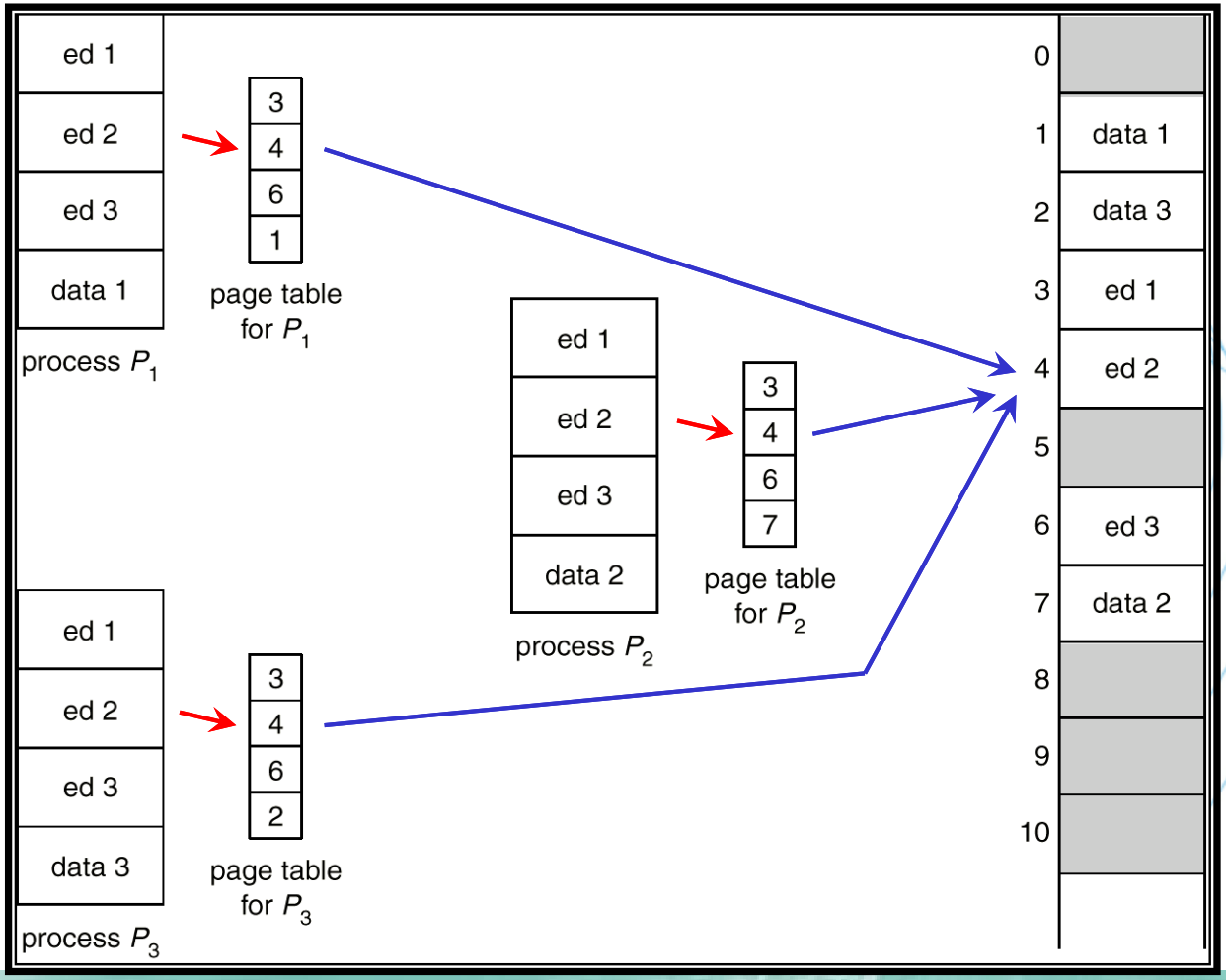
REVISIT

- 反置页表(Inverted page table)
  - 优缺点？
  - 整个OS全局有几个反置页表？



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。



页共享与保护

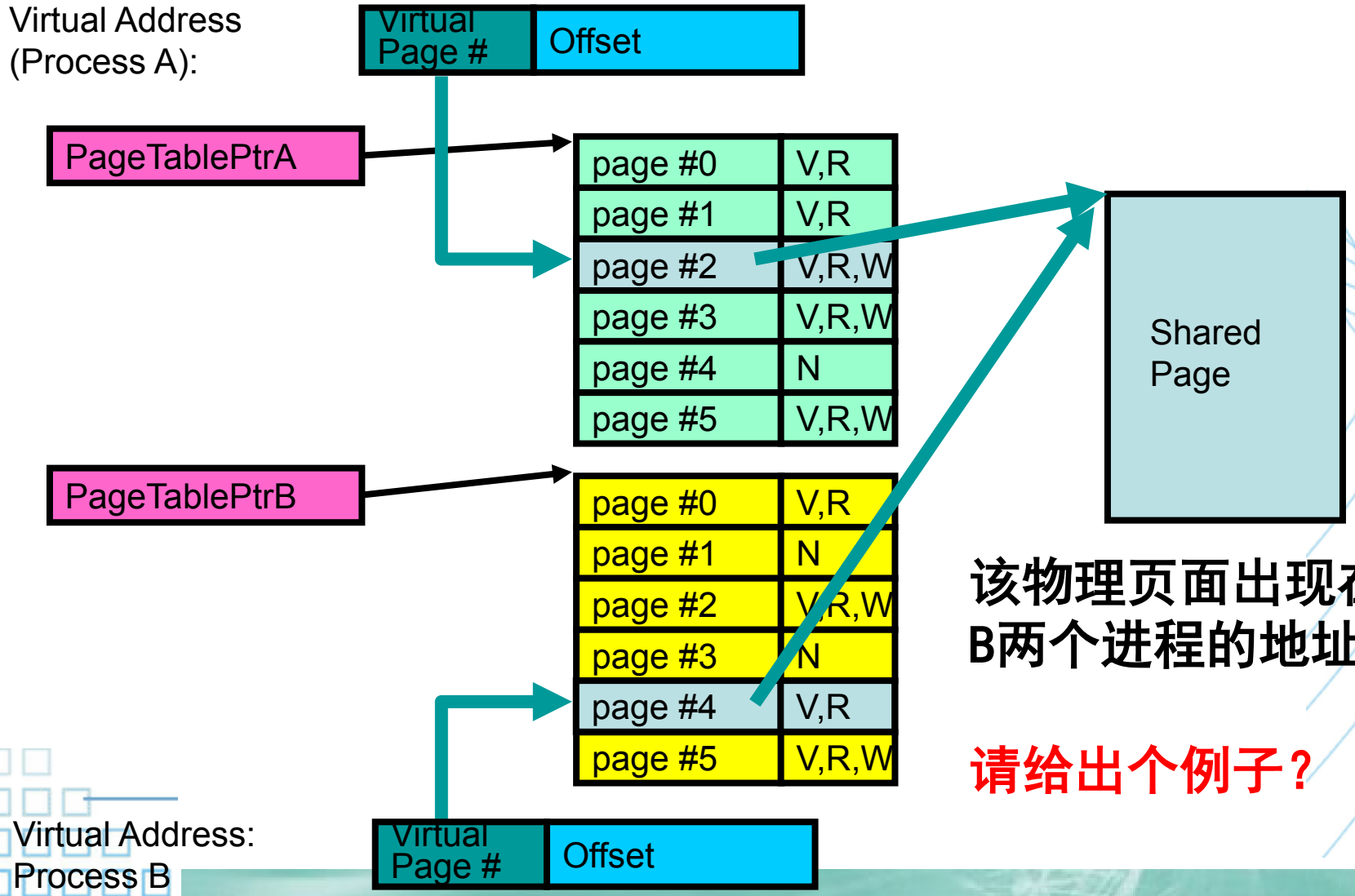
页的保护

- 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）

共享带来的问题

- 若共享数据与不共享数据划在同一块中，则：
  - 有些不共享的数据也被共享，不易保密。
- 实现数据共享的最好方法：分段存储管理。

页共享



该物理页面出现在A和B两个进程的地址空间

请给出个例子？

分段存储管理

- 问题：一个程序的各个组成部分对于内存的需求来说都是完全同质的么？
  - 指令、数据、堆栈的内存管理需求不同
- 主要讨论
  - 方便编程
  - 分段共享
  - 分段保护
  - 动态链接
  - 动态增长

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：

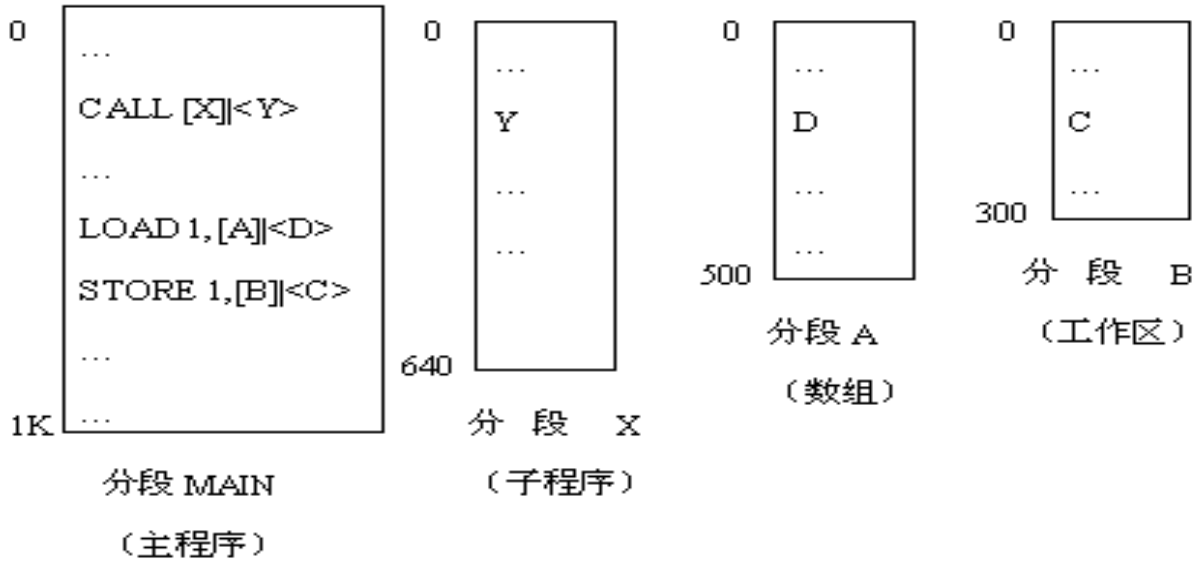
- 实际应用中，某些段（数据段）会不断增长，前面的存储管理方法均难以实现。

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。

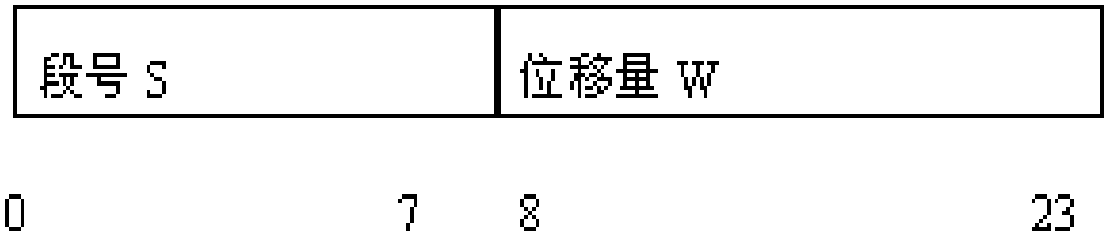
分段地址空间

- 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，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。（全局连续 vs 局部连续）



地址结构

- 逻辑地址结构：　段号S + 位移量W



- 和分区的区别？
- 和分页的区别？

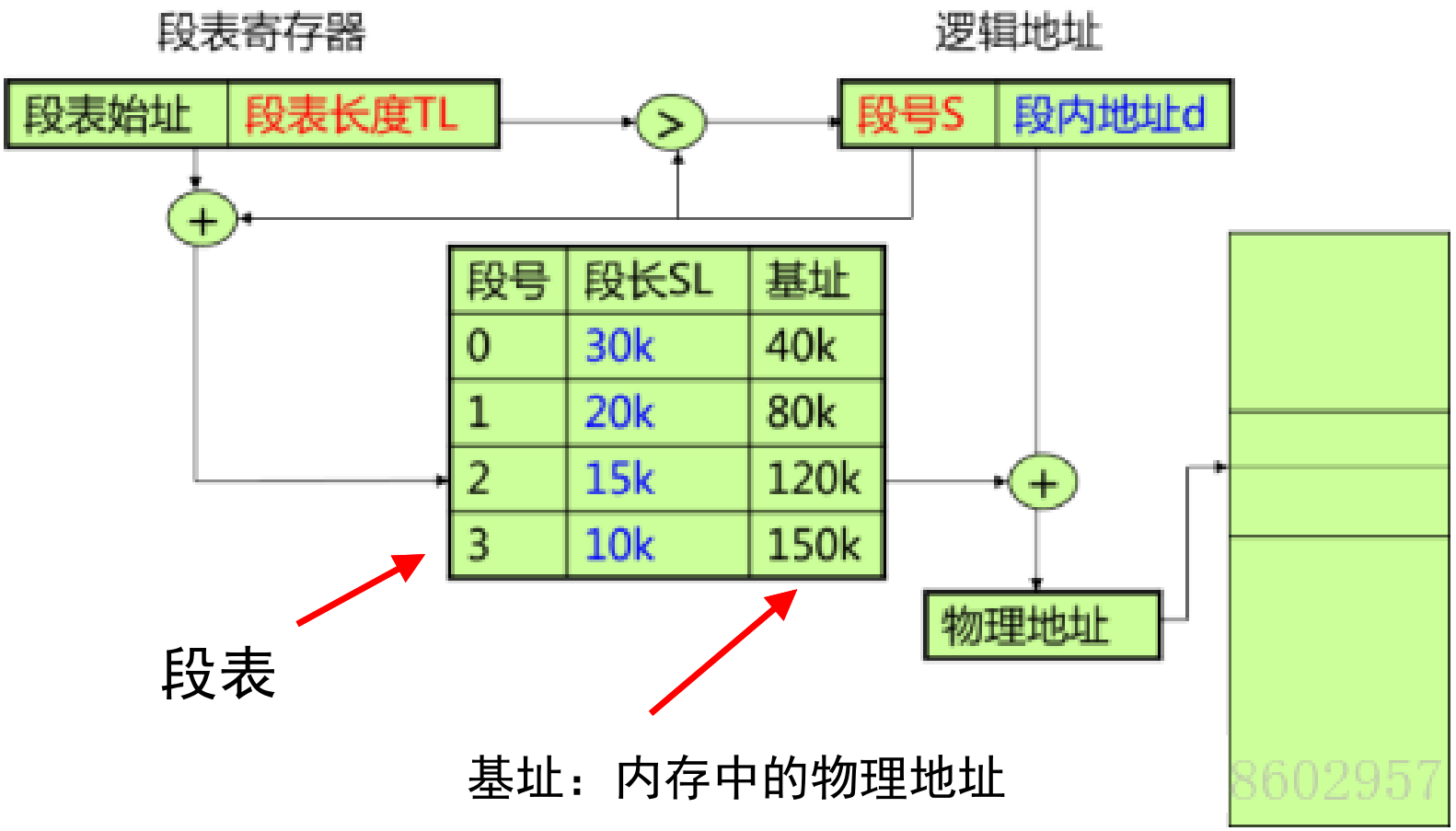
地址结构

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。

- 访问一个字节的数据/指令需访问内存两次 (段表一次，内存一次)。

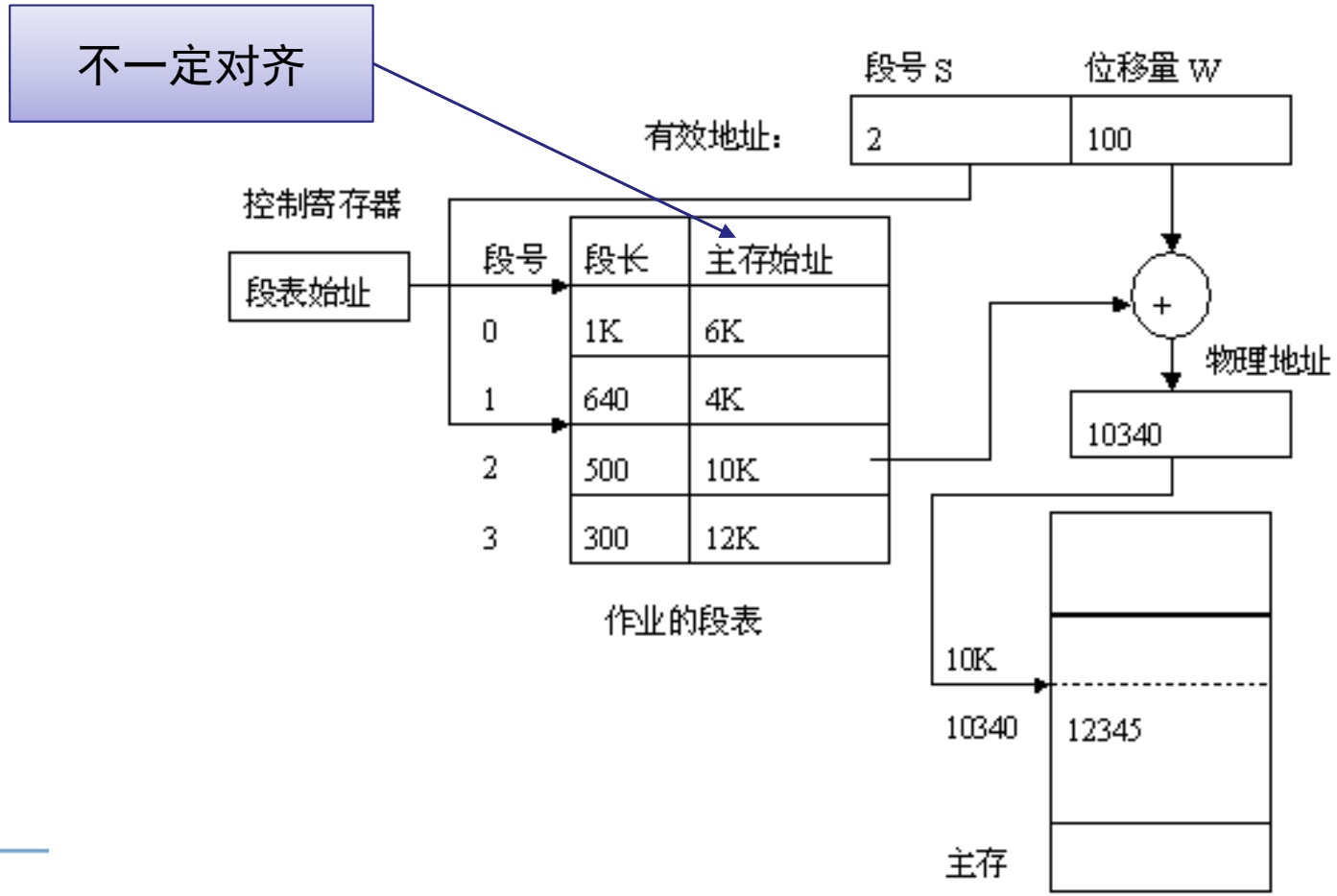
地址变换机构

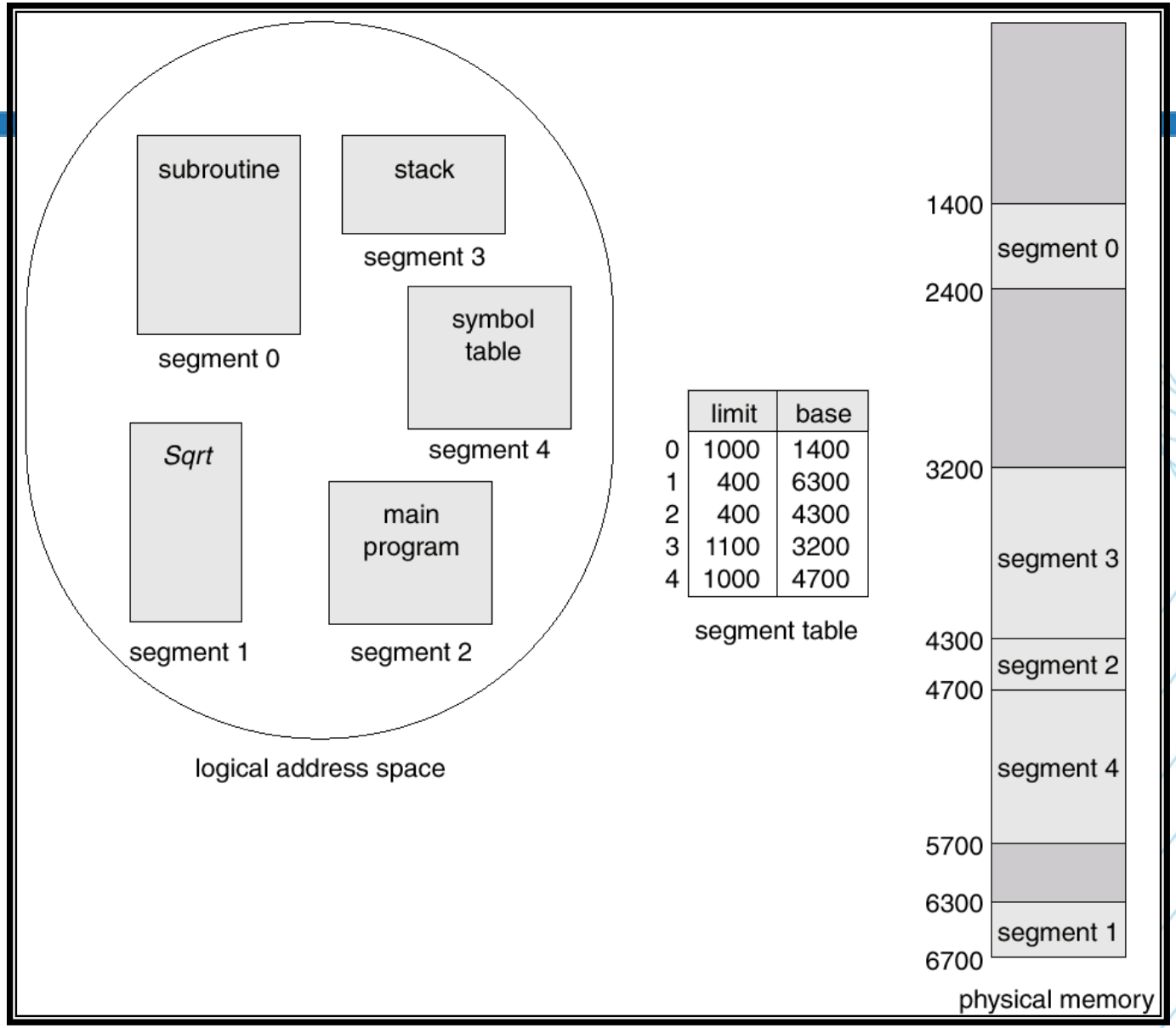


地址变换过程

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。

分段地址变换过程





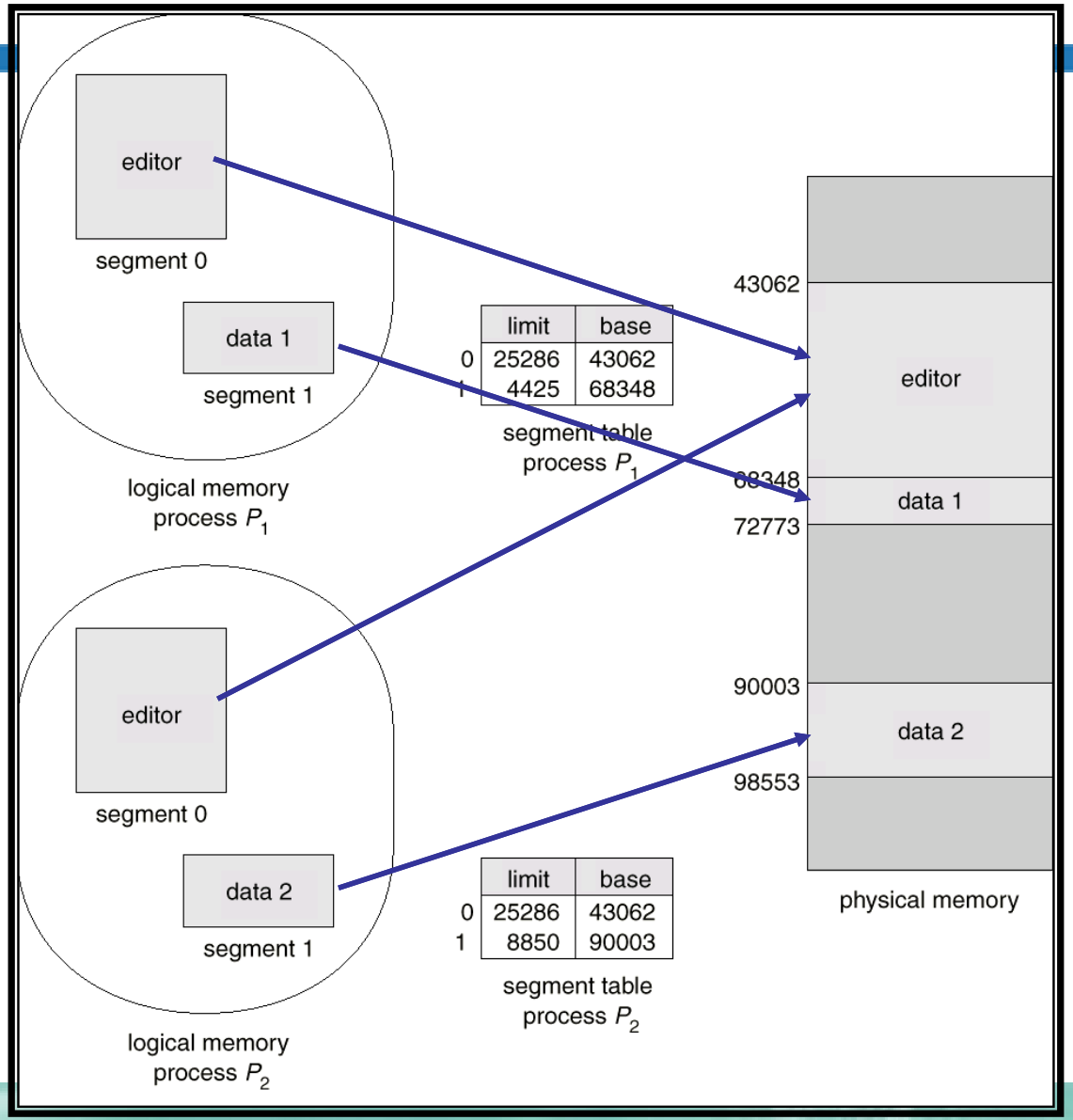
信息共享

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(40×40+160)，而不是(160+40)×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。

段共享

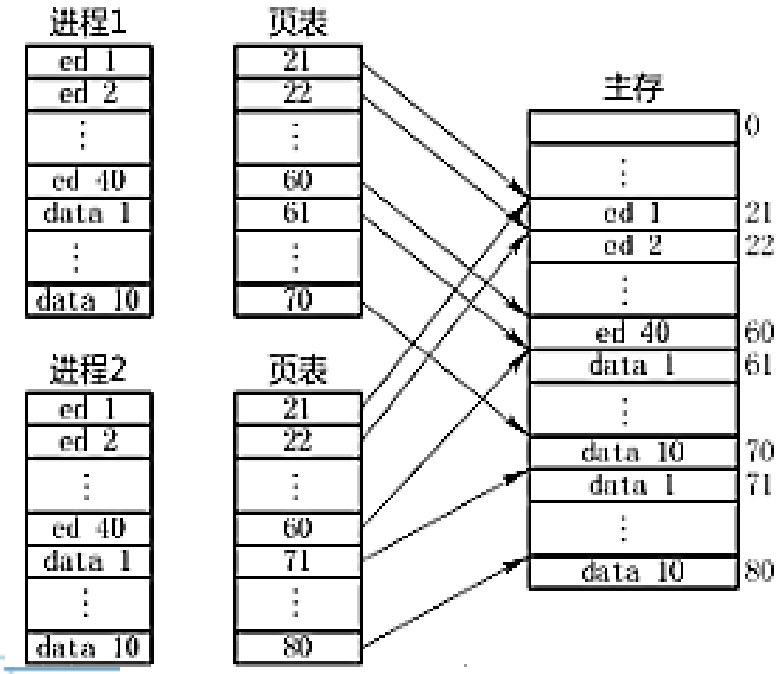


分段管理的优缺点

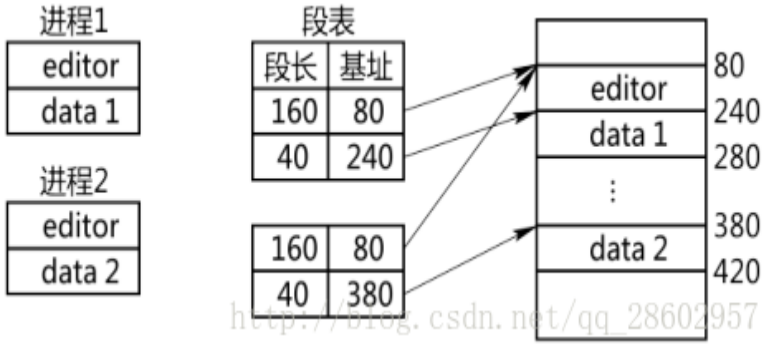
- 优点：
  - 分段系统易于实现段的共享，对段的保护也十分简单。
- 缺点：
  - 处理机要为地址变换花费时间；要为表格提供附加的存储空间。
  - 为满足分段的动态增长和减少外碎片，要采用拼接手段。
  - 在辅存中管理不定长度的分段困难较多。
  - 分段的最大尺寸受到主存可用空间的限制。

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。



页共享



段共享

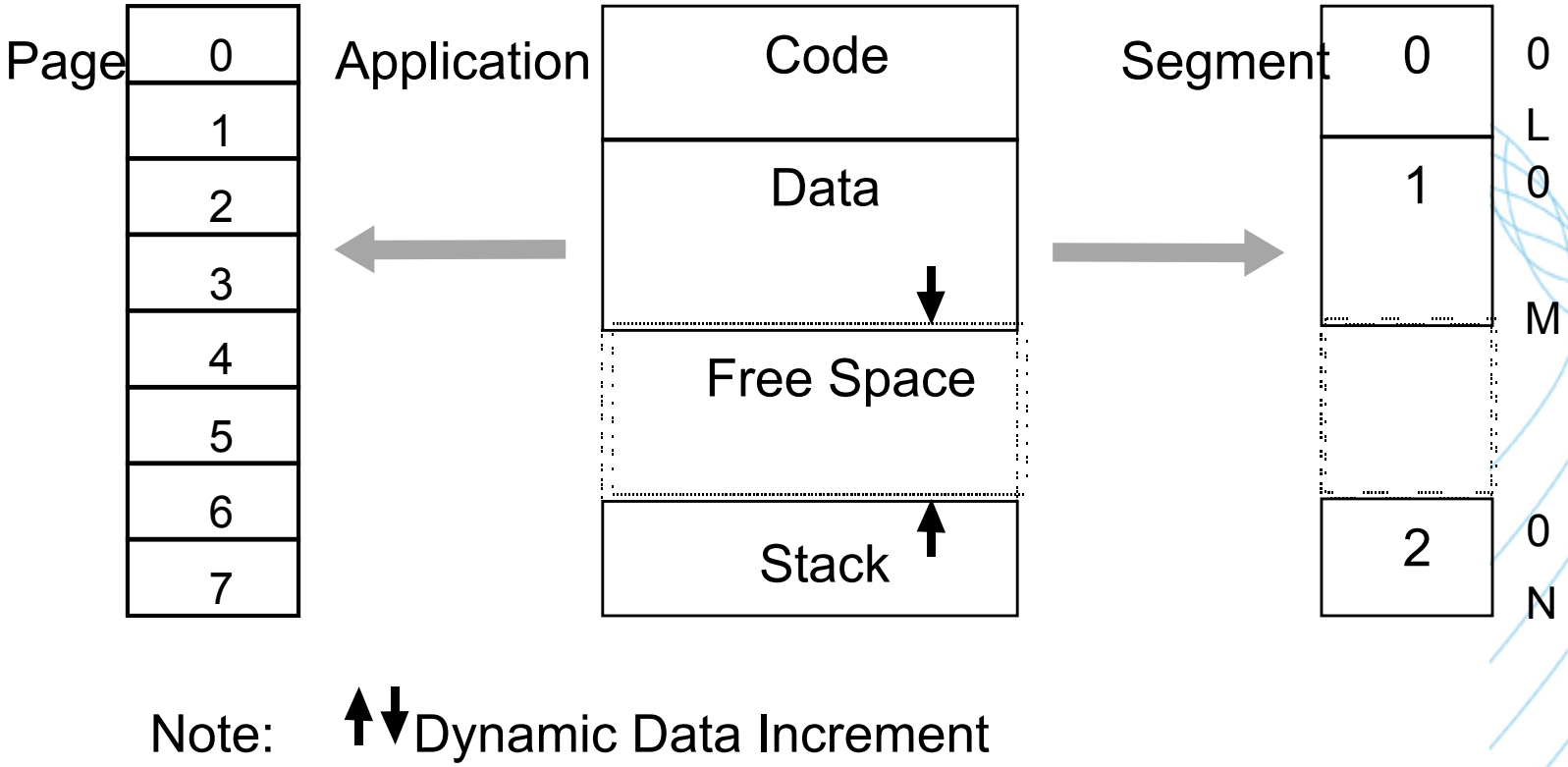
分页与分段的比较：

- 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

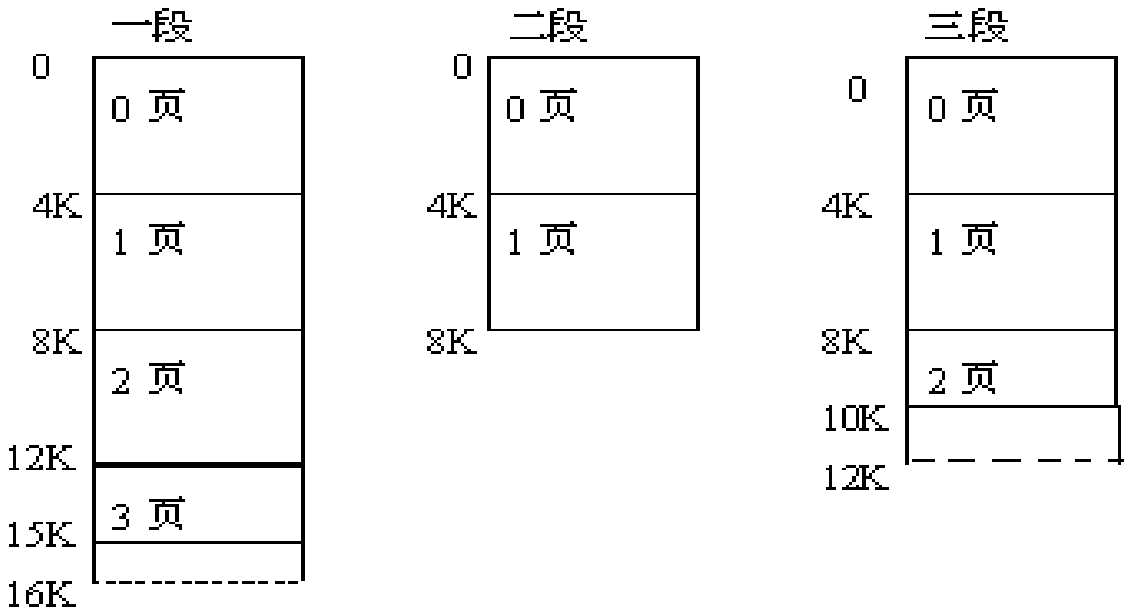
程序内存动态增长



Note: Dynamic Data Increment

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。



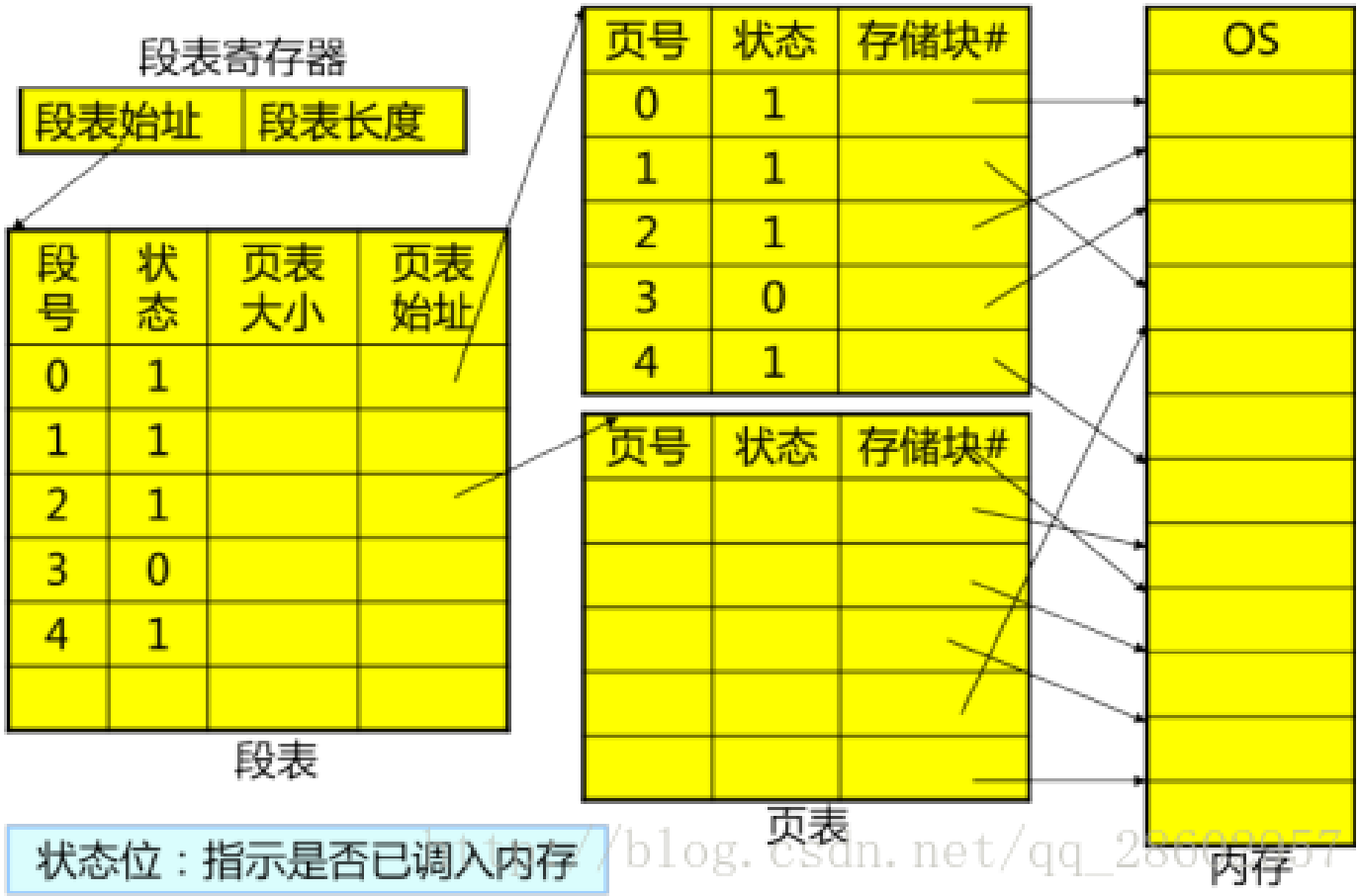
实现原理

- 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。

段号（S）	段内页号（P）	页内地址（W）

- 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。

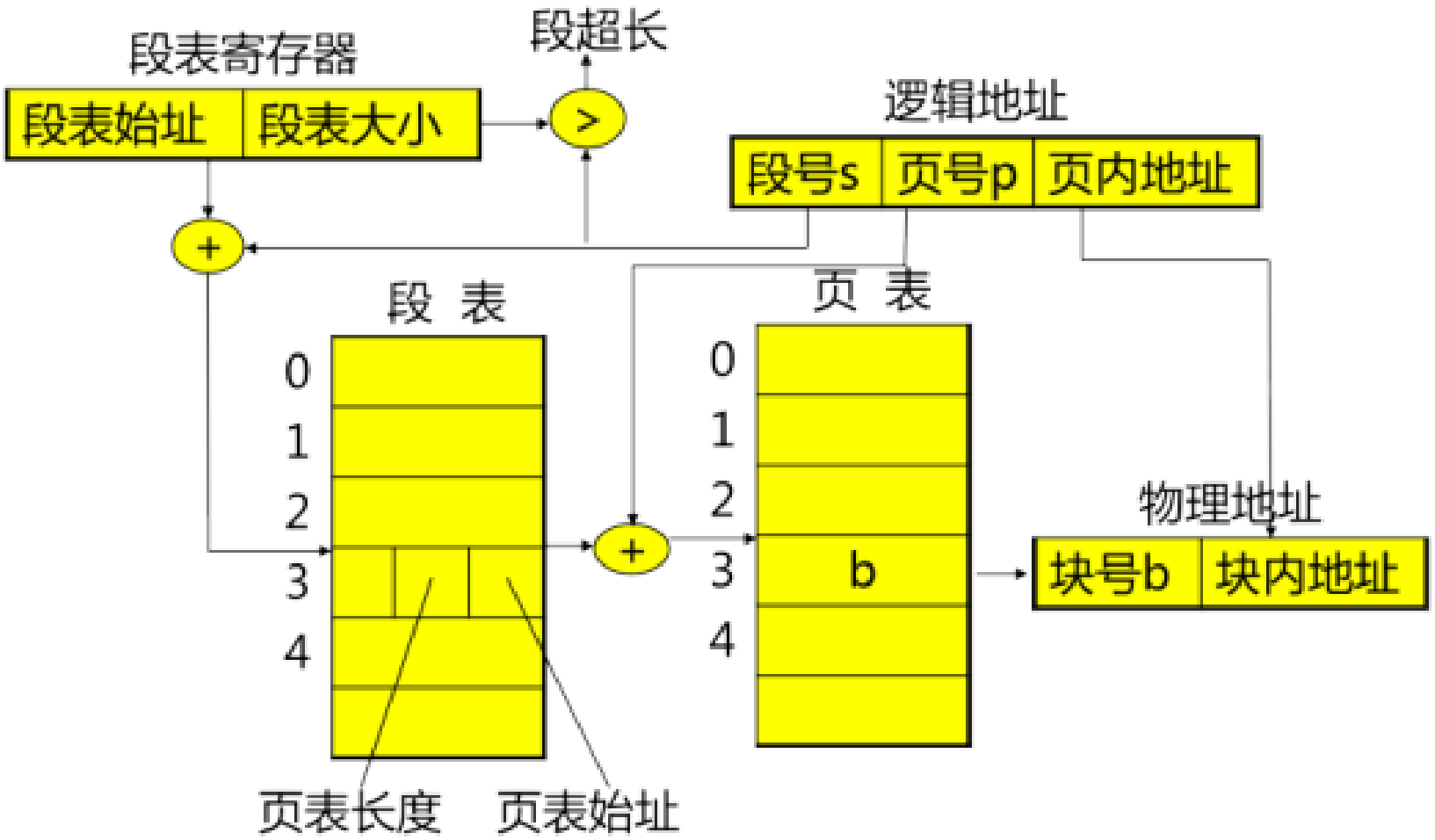
利用段表和页表实现地址映射



段页式存储管理的地址变换

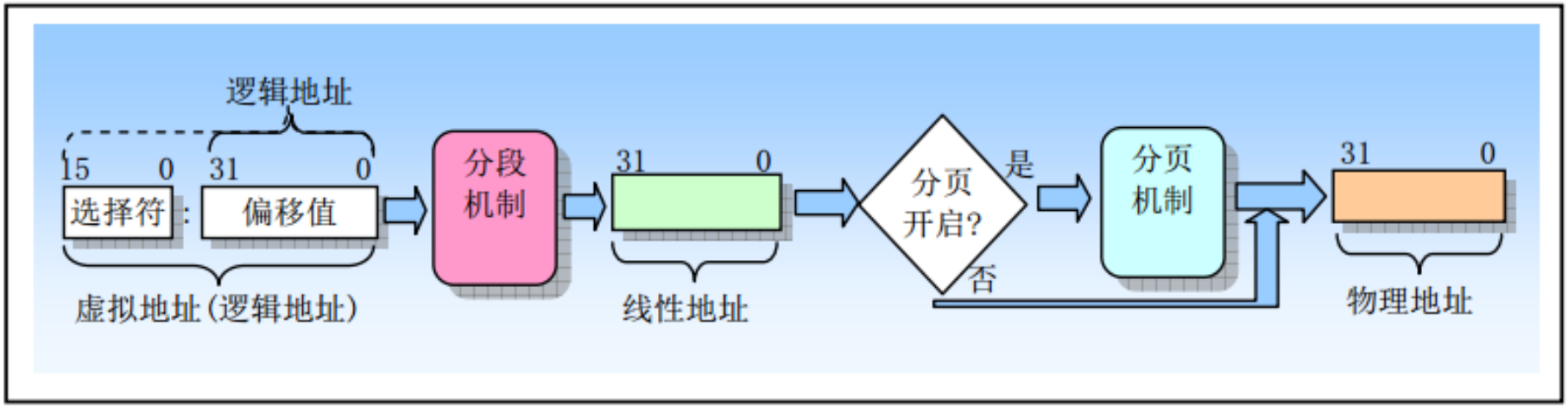
- 从 PCB 中取出段表始址和段表长度，装入段表寄存器。
- 将段号与段表长度进行比较，若段号大于或等于段表长度，产生越界中断。
- 利用段表始址与段号得到该段表项在段表中的位置。取出该段的页表始址和页表长度。
- 将页号与页表长度进行比较，若页号大于或等于页表长度，产生越界中断。
- 利用页表始址与页号得到该页表项在页表中的位置。
- 取出该页的物理块号，与页内地址拼接得到实际的物理地址。

段页式存储管理的地址变换

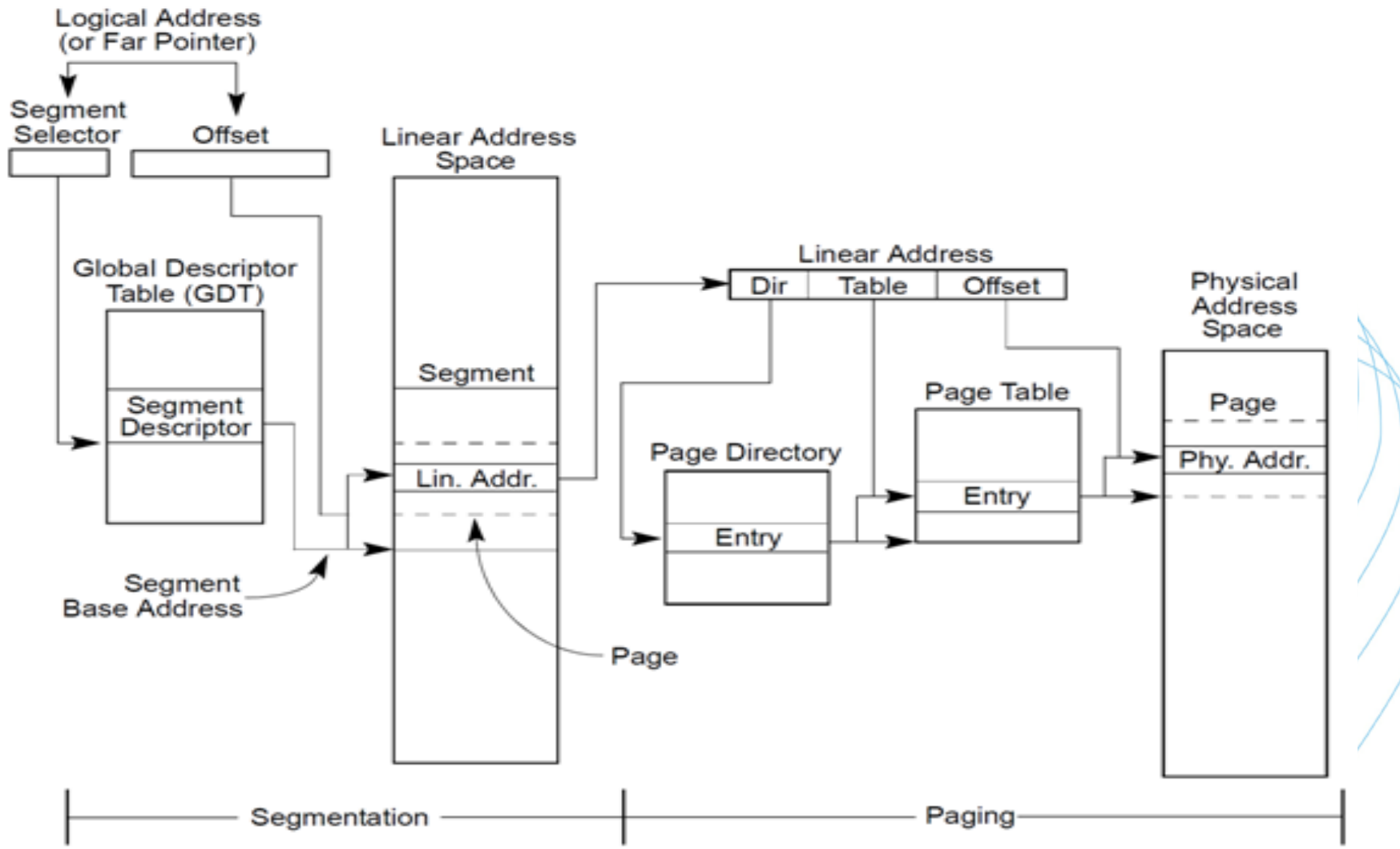


实例：X86的段页式地址映射

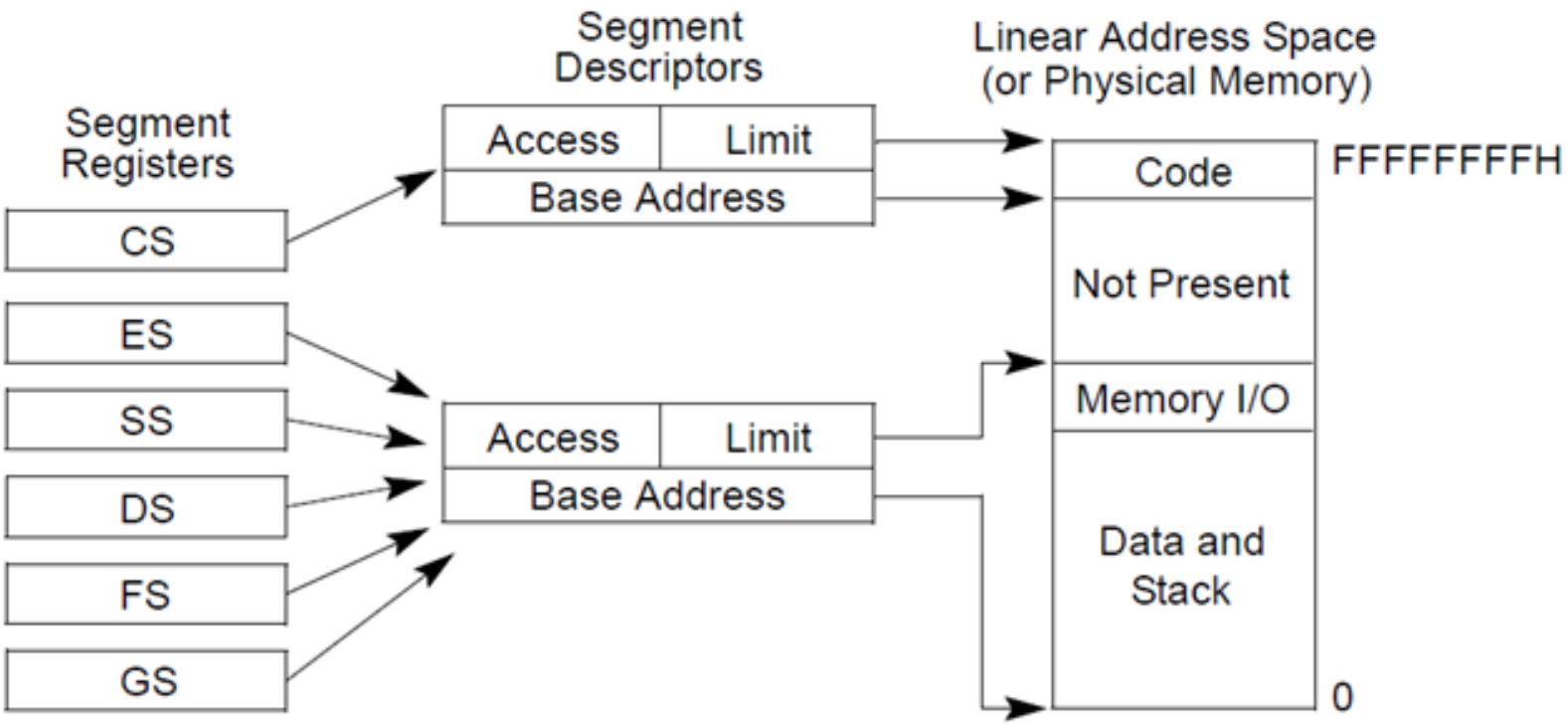
- X86的地址映射机制分为两个部分：
  - 段映射机制，将逻辑地址映射到线性地址；
  - 页映射机制，将线性地址映射到物理地址。



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

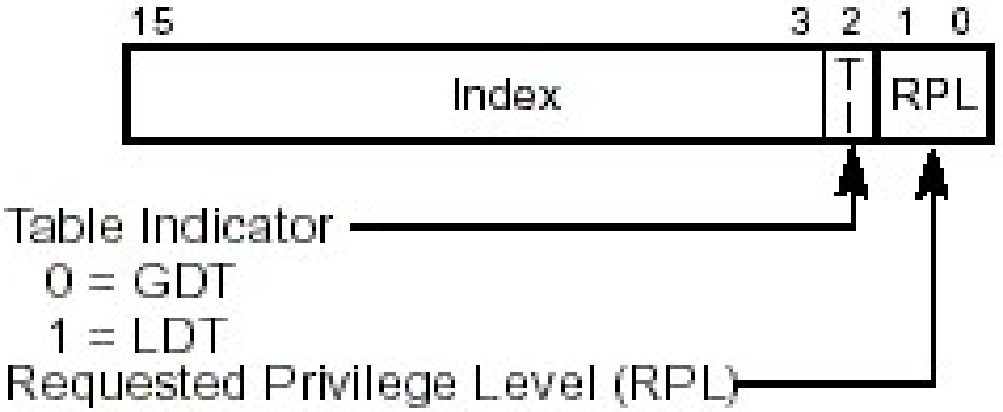
- GDT（Globle Descriptor Table)：全局描述符表，是全局性的，为所有的任务服务，不管是内核程序还是用户程序，我们都是把段描述符放在GDT中。Intel的设计者们提供了一个寄存器GDTR用来存放GDT的入口地址

32 位基地址	16 位的表界限

GDTR 结构

GDT及LDT

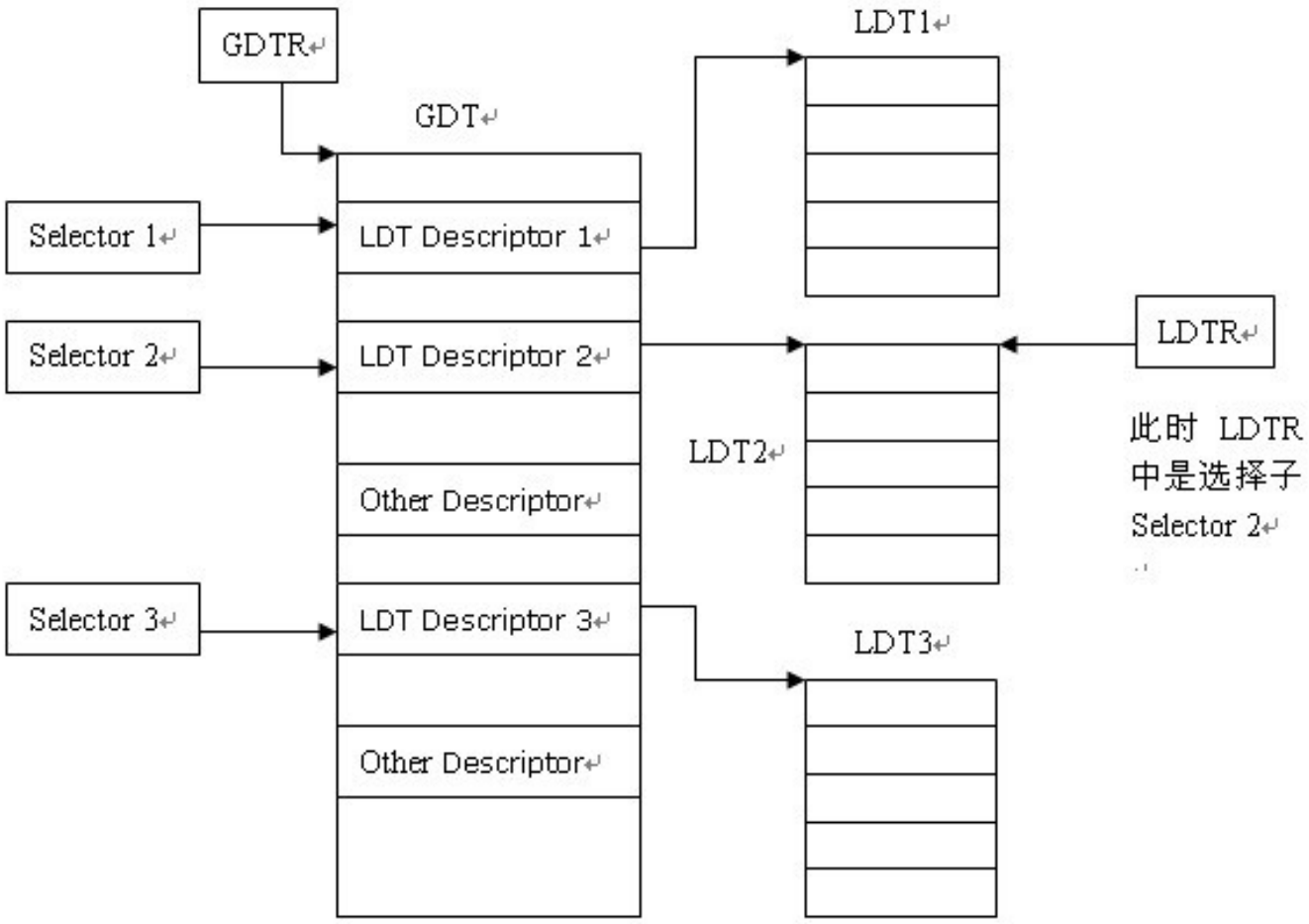
- 段选择子（Selector）由GDTR访问全局描述符表是通过“段选择子”（实模式下的段寄存器）来完成的。段选择子是一个16位的寄存器（同实模式下的段寄存器相同）。段选择子包括三部分：描述符索引（index）、TI、请求特权级（RPL）。他的index（描述符索引）部分表示所需要的段的描述符在描述符表的位置，由这个位置再根据在GDTR中存储的描述符表基址就可以找到相应的描述符



GDT及LDT

- 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。我们可以这样理解GDT和LDT：GDT为一级描述符表，LDT为二级描述符表。
- LDT和GDT从本质上说是相同的，只是LDT嵌套在GDT之中。LDTR记录局部描述符表的起始位置，与GDTR不同，LDTR的内容是一个段选择子。
- 由于LDT本身同样是一段内存，也是一个段，所以它也有个描述符描述它，这个描述符就存储在GDT中，对应这个表述符也会有一个选择子，LDTR装载的就是这样一个选择子。

GDT及LDT



Segment Descriptor

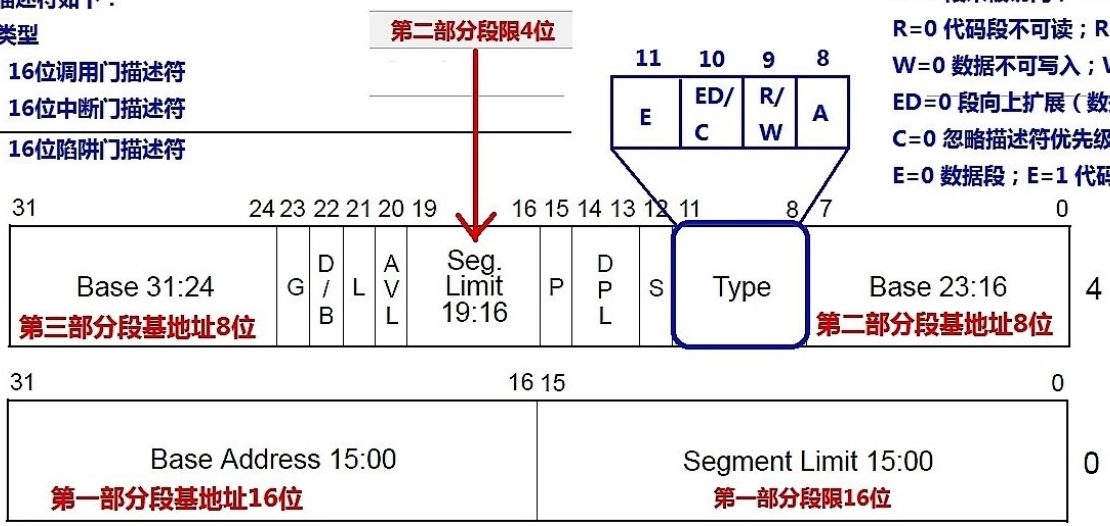
Type类型域4个bits的特定组合组合表示的门描述符如下：

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~FFFFF，即1B~1MB；G=1时，段限乘以4KB，即4KB~4MB，即00000FFF~FFFFFFFF 段限为32位
LIMIT — Segment Limit 20位段限，由2部分组成
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，S=0为系统段描述符
TYPE — Segment type

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

参考CPU手册：http://intel80386.com/386htm/s05_01.htm

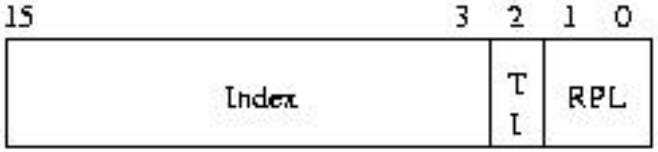
页式地址映射过程

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

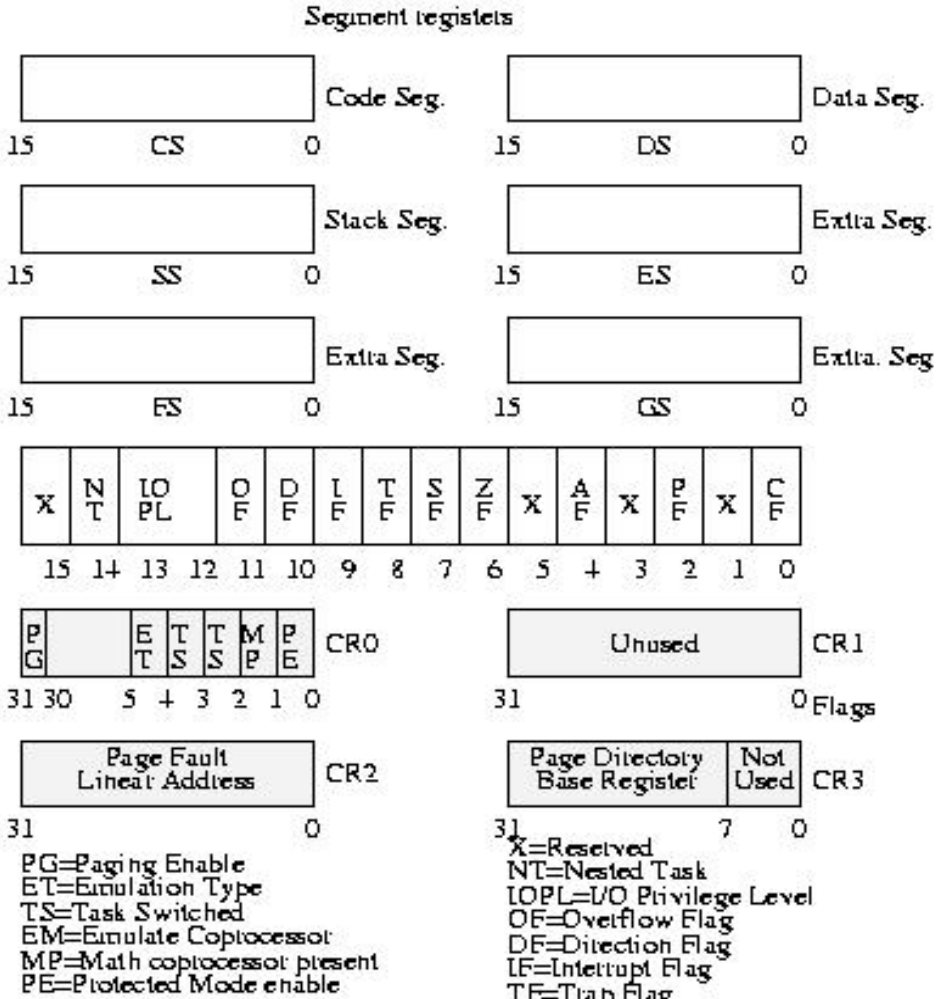
控制寄存器（CR0～CR3）用于控制和确定处理器的操作模式以及当前执行任务的特性：

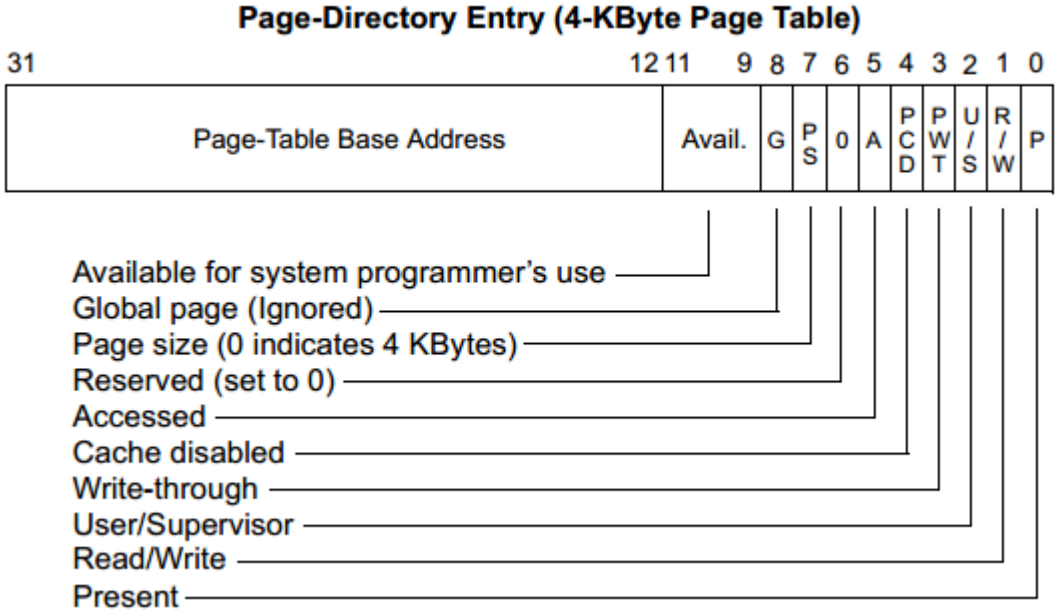
- CR0中含有控制处理器操作模式和状态的系统控制标志；
- CR1保留不用；
- CR2含有导致页错误的线性地址；
- CR3中含有页目录表物理内存基地址，因此该寄存器也被称为页目录基地址寄存器PDBR（Page-Directory Base address Register）。



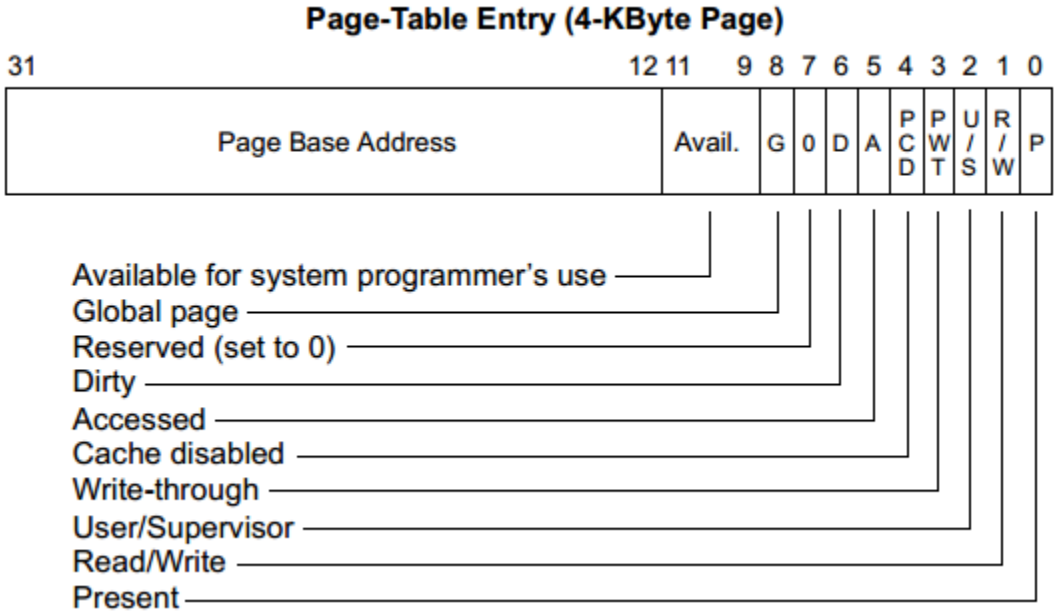
RPL = Requestor Privilege Level
TI = Table Indicator
(0 = GDT, 1 = LDT)
Index = Index into table

Protected Mode segment selector





页目录项PDE



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。

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”（可移植性考虑）
  - 虚拟地址与线性地址总是一致的

Name	Description	Base	Limit	DPL
__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

阅读：https://medium.com/hungys-blog/linux-kernel-memory-addressing-a0d304283af3