IA32 物理地址的形成

内容

- 虚拟存储器
- 主存储器分段管理
- 主存储器物理地址的形成

  8086和x86-32实方式下物理地址的形成

  保护方式下物理地址的形成

虚拟存储器

- 程序员需要关心物理内存吗？
  - 内存条有多大？
  - 程序能全部放入内存吗？
  - 程序放在内存的什么位置？

早期：程序员自己管理主存，通过分解程序并覆盖主存的方式执行程序

- **在实时性要求较高的嵌入式微控制器中，大多需要关注内存；**
- **取指令和存储操作数所有的地址都是物理地址；**
- **执行速度快，无需进行地址转换；**
- **未采用虚拟存储机制。**

虚拟存储器

- 1961年，英国曼切斯特研究人员提出自动执行overlay的方式。
- 动机：把程序员从大量繁琐的存储管理工作中解放出来，

  使得程序员编程时不用管主存容量的大小。

基本思想

- 把地址空间和主存容量的概念区分开来;
- 程序员在地址空间里编写程序;
- 程序则在真正的内存中运行;
- 由一个专门的机制实现地址空间和实际主存之间的映射;
- CPU 中的 MMU (Memory Management Unit) 负责将逻辑地址（即虚拟地址）转换为 内存的物理地址。

虚拟存储器

逻辑地址（即虚拟地址）转换为 内存的物理地址的方式：

- **分页式**
- **分段式**
- **段页式**

分段

- C语言程序中，变量的定义和指令写在一起
- C语言程序中无分段的概念
- 机器语言层次上，是要分段的
- 在C程序编译时，将变量的空间分配和指令分开，分别放在不同段中。

思考题：

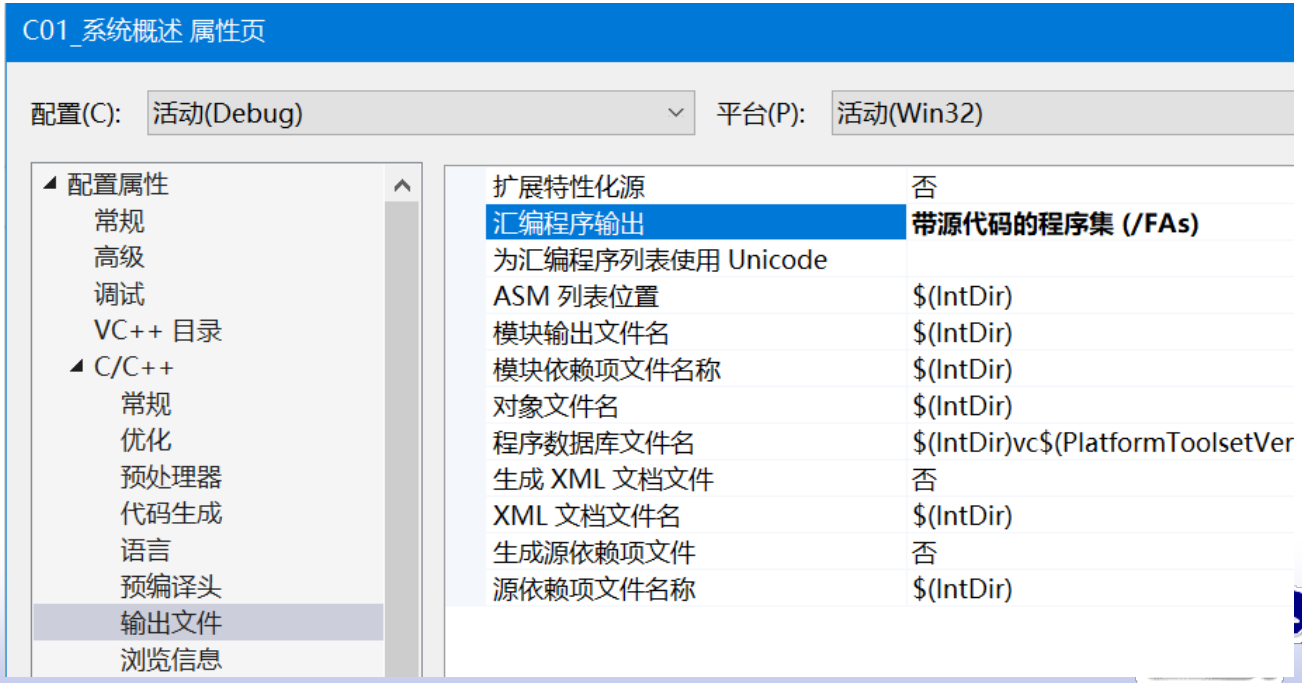
为什么机器指令和数据存放要分开呢？

分段

- 根据程序的模块化性质，按程序的逻辑结构划分成多个相对独立的部分，称为段；
- 一个程序可以有多个代码段；
- 一个程序可以有多个数据段；
- 段通常有段名、段起点、段长、段属性等信息

分段


```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    int  x, y, z;
    x = 10;
    y = 20;
    z = 3 * x + 6 * y + 4 * 8;
    printf("3*%d+6*%d+4*8=%d\n", x, y, z);
    return 0;
}
```

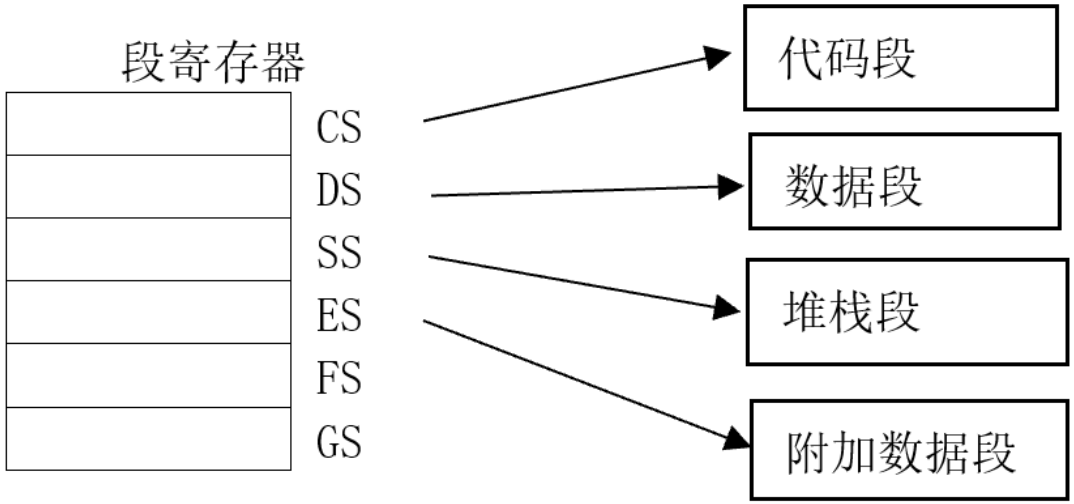


分段


```
CONST    SEGMENT
??_C@_0BC@LJIGKNNH@3?$CK?$CFd?$CL6?$CK?$CFd?$CL4?$CK8?$DN?
    DB   'd+6*%d+4*8=%d', 0aH, 00H              ; `string'
CONST    ENDS

_TEXT    SEGMENT
_z$ = -32                              ; size = 4
_y$ = -20                              ; size = 4
_x$ = -8                               ; size = 4
_argc$ = 8                             ; size = 4
_argv$ = 12                            ; size = 4
_main    PROC                              ; COMDAT

; 4     : {

    push     ebp
    mov ebp, esp
    sub esp, 228                       ; 000000e4H
    push     ebx
```

主存储器分段管理

- 在分段内存模型中，每个分段通常加载不同的分段选择器，以便每个段寄存器指向线性地址空间内的不同段。



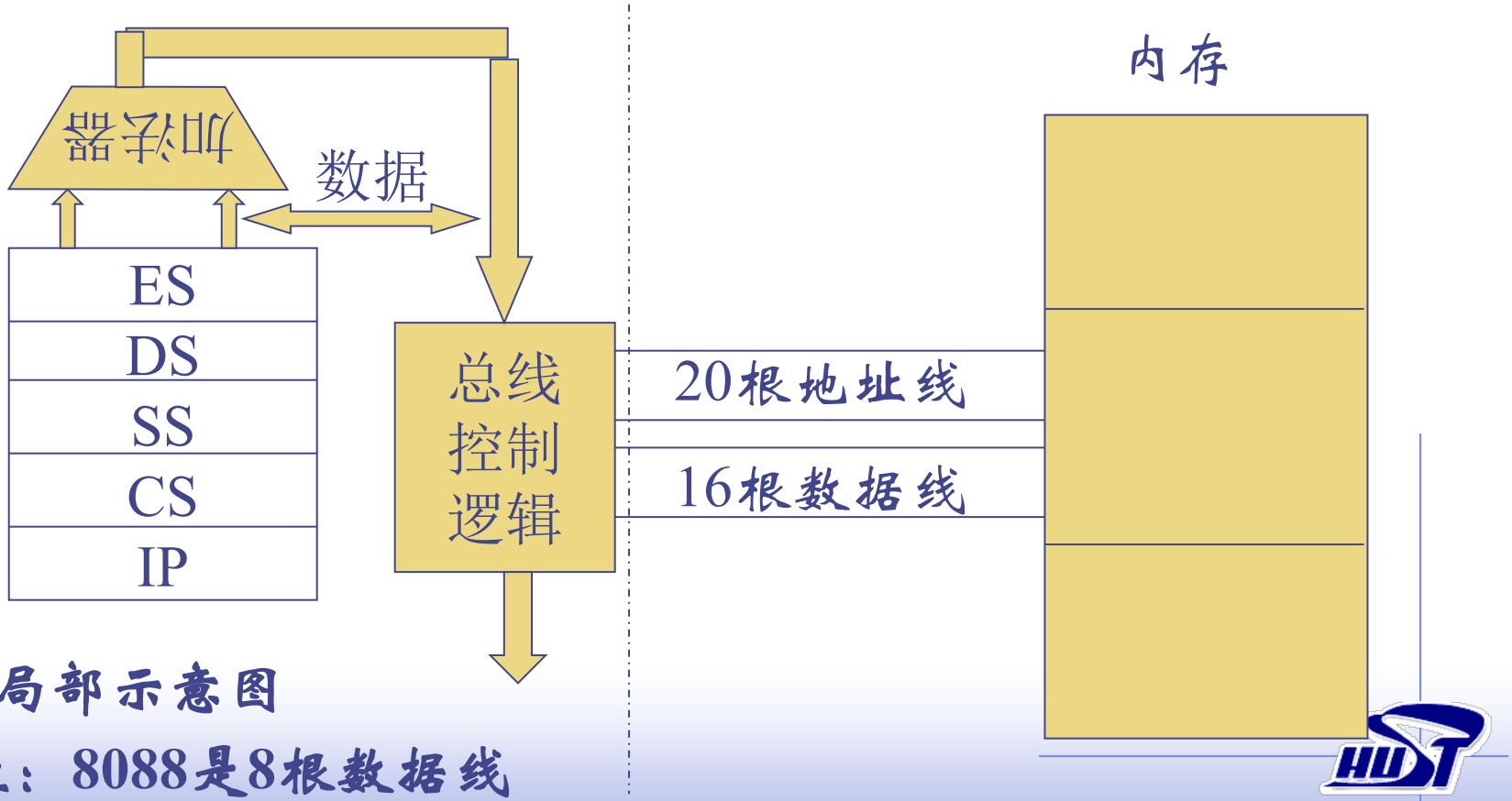
主存储器物理地址的形成

- 8086和x86-32实方式下物理地址的形成
- 保护方式下物理地址的形成

8086和x86-32实方式下物理地址的形成

内存1M, 20位物理地址，CPU中是16位的寄存器。

16位的寄存器如何与20位的物理地址建立对应关系？



CPU局部示意图

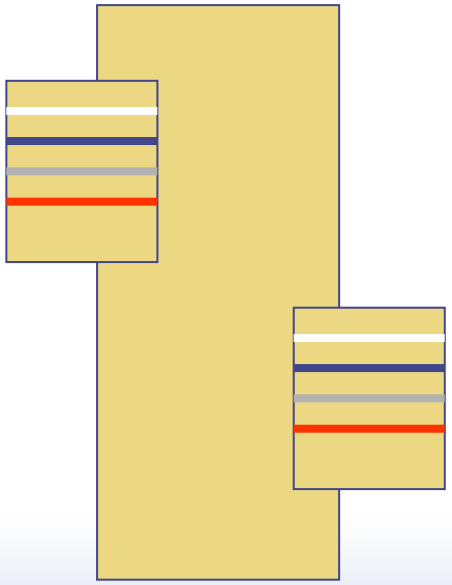
注：8088是8根数据线

8086和x86-32实方式下物理地址的形成

Q: 程序中能够直接使用物理地址吗？
有必要使用物理地址吗？

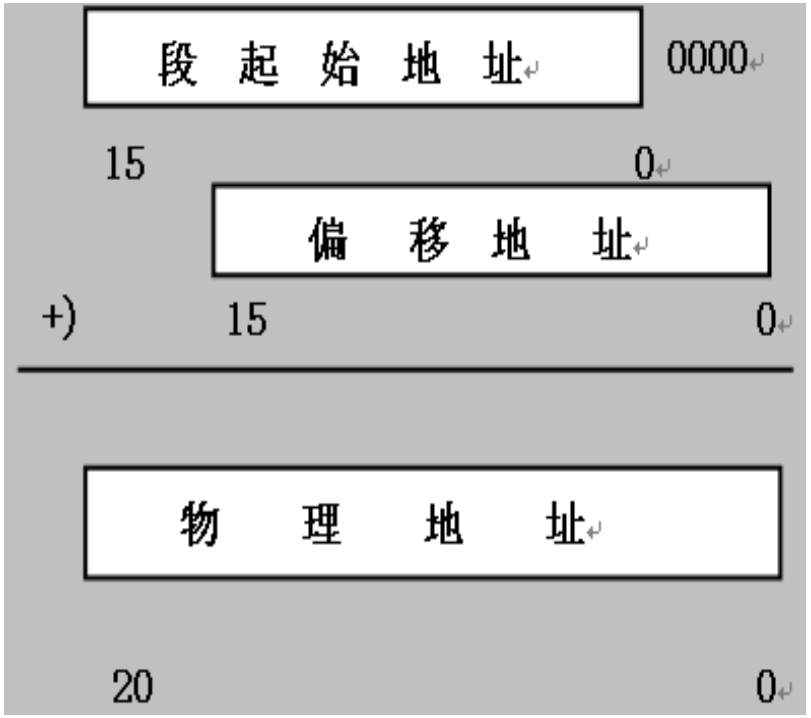
程序中单元（如变量等）的相对位置，逻辑地址

几个不同颜色的线条。
确定了白线条的位置，其它线条的位置可以由它们之间的相对位置关系计算。



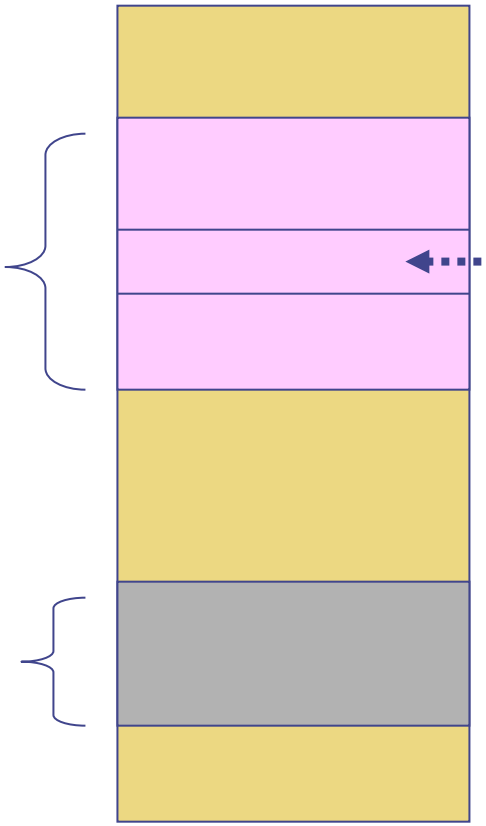
1. 实方式物理地址的形成

- ✓32位CPU与8086一样
- ✓只能寻址1M物理存储空间
- ✓可以访问6个段
- ✓ CS, DS, SS, ES, FS, GS
- ✓ 每个段至多64K



物理地址 ＝ （段寄存器）左移4位 ＋偏移地址

8086和x86-32实方式下物理地址的形成



段首址

距段首址的距离，亦称段内偏移

分段

段的开始地址要能被16整除。

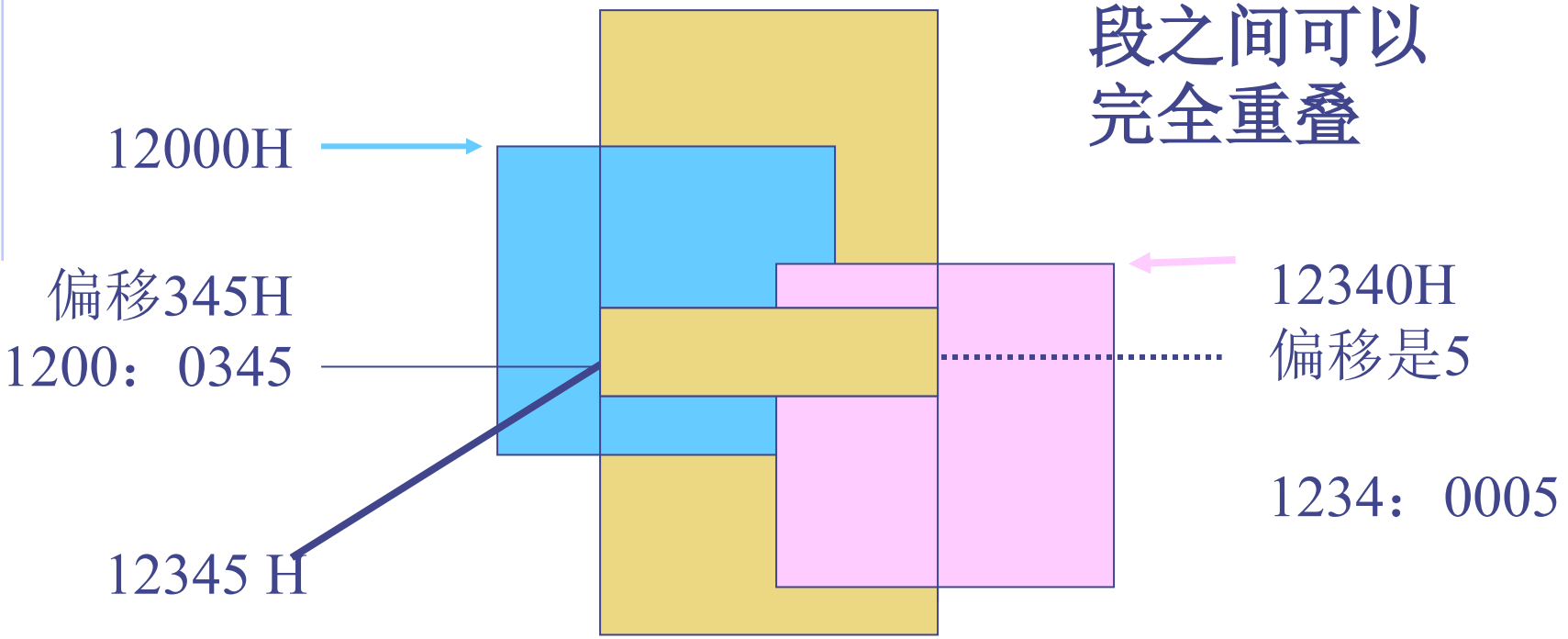
16 = 10H
= 10000B

段开始单元的物理地址 /16
→ 段址

段址 * 16 + 偏移地址 = 物理地址

8086和x86-32实方式下物理地址的形成

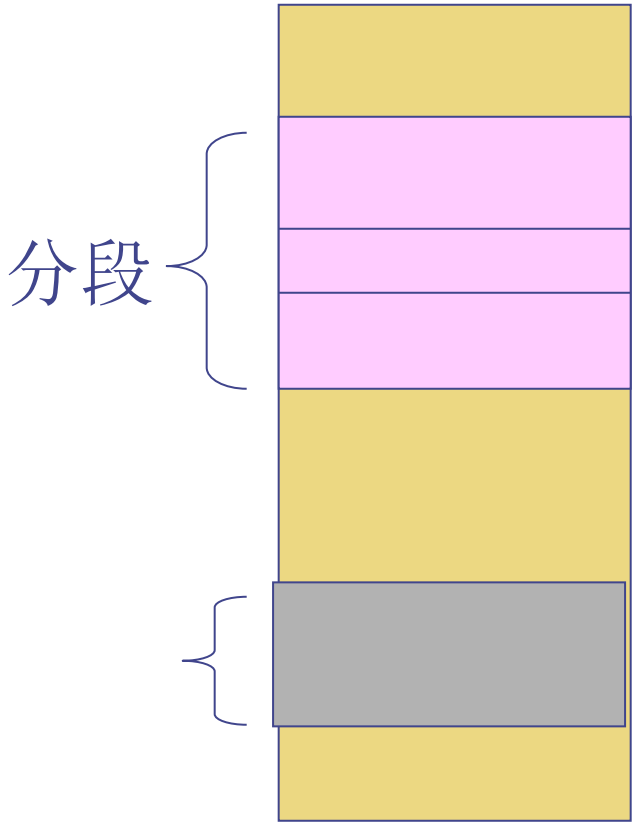
段址 * 16 + 偏移地址 = 物理地址



段中某一存储单元的地址是用两部分来表示的，“段首地址：偏移地址”，称它为二维的逻辑地址。

8086和x86-32实方式下物理地址的形成

一个段最大为64KB. (2^{16})



1M内存最少有16个段

同时访问4个段, 段寄存器CS,DS,ES,SS

8086和x86-32实方式下物理地址的形成

8086中，只有4个段寄存器CS, DS, ES, SS

在代码段中取指令时：

指令物理地址 PA= (CS) 左移四位 + (IP)

注意，使用的是IP, 而不是EIP

在数据段中读/写数据时：

数据的物理地址 PA = (DS或ES)左移四位 + 16位偏移地址 (偏移地址由寻址方式确定)

在堆栈操作时：

栈顶的物理地址 PA = (SS) 左移四位 + (SP)

2. 保护方式下物理地址的形成

80386中寄存器32位，地址线32根。

- 在多任务环境下，系统中有多个程序在运行。
- 程序之间要隔离！
- 分段是存储管理的一种方式，为保护提供基础；
- 不同程序在不同段中；
- 一个程序可以包含多个段；
- 段用于封闭具有共同属性的存储区域；

保护方式下物理地址的形成

Q： 如何实现程序之间的隔离？

- 分段是存储管理的一种方式，为保护提供基础；
- 不同程序在不同段中；
- 一个程序可以包含多个段；
- 段用于封闭具有共同属性的存储区域；

保护方式下物理地址的形成

Q：要保护一个段，应该提供哪些信息？
这些信息又存放在何处？

描述符 (description)

段的起始位置（段基地址）
段的大小（段界限）
段的特权级
段的属性（是代码段，数据段，还是堆栈段？）
（数据段是否可写？代码段是否可读出？）
段的位置（在内存还是在磁盘？）
段的类型（在系统段还是用户段？）
段的使用（段被访问过，还是没有？）

保护方式下物理地址的形成

描述符

段的有关信息的描述

段基地址：B31 ~ B0, 共32位；

段界限：L19 ~ L0, 共20位

保护方式下物理地址的形成

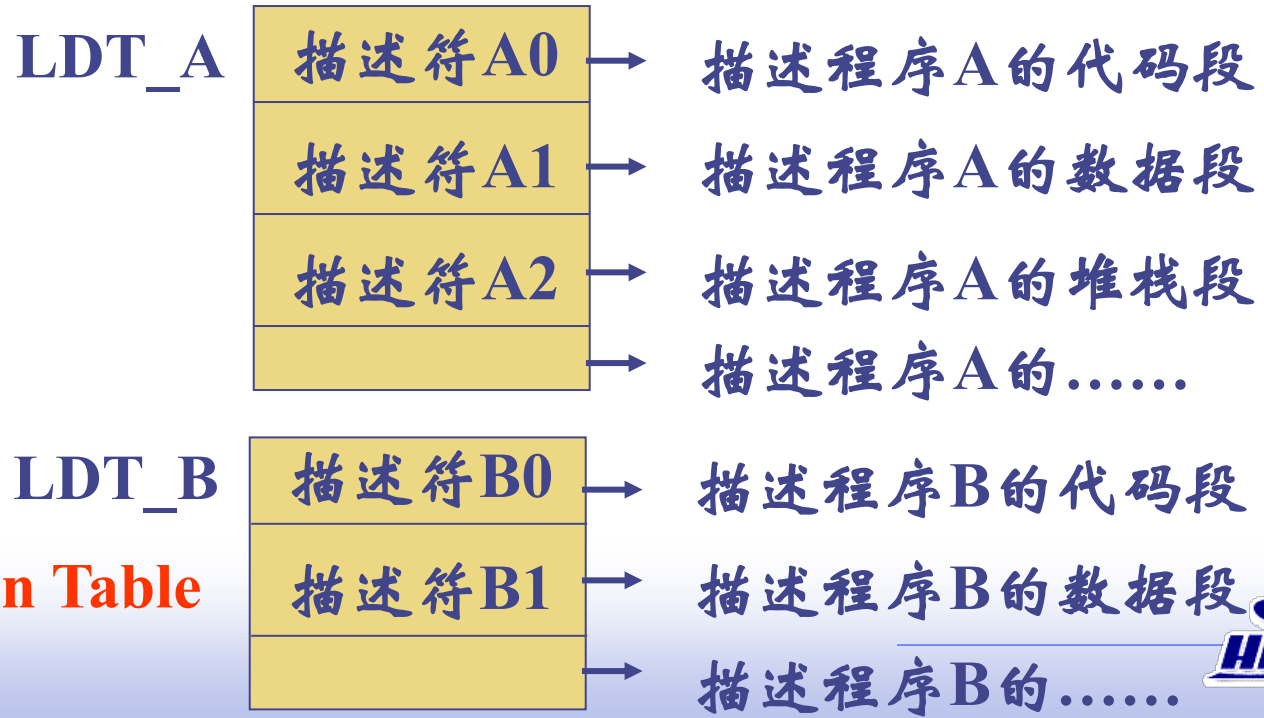
描述符表

描述符的集合 – 描述符表

局部描述符表：

一个LDT，是一个系统段，最大可为64KB,最多可存放8192个描述符。(64K Bytes / 8 Bytes per Descriptor)

对每一个程序，都建立一个局部描述符表（LDT）。

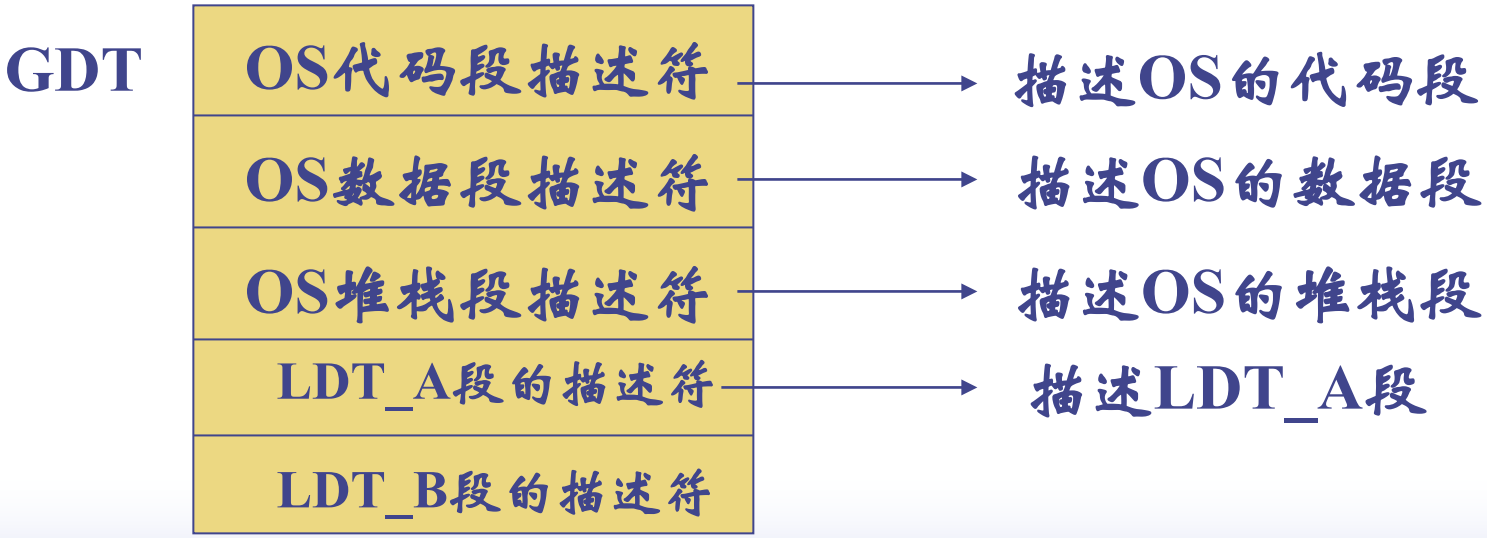


Local Description Table

保护方式下物理地址的形成

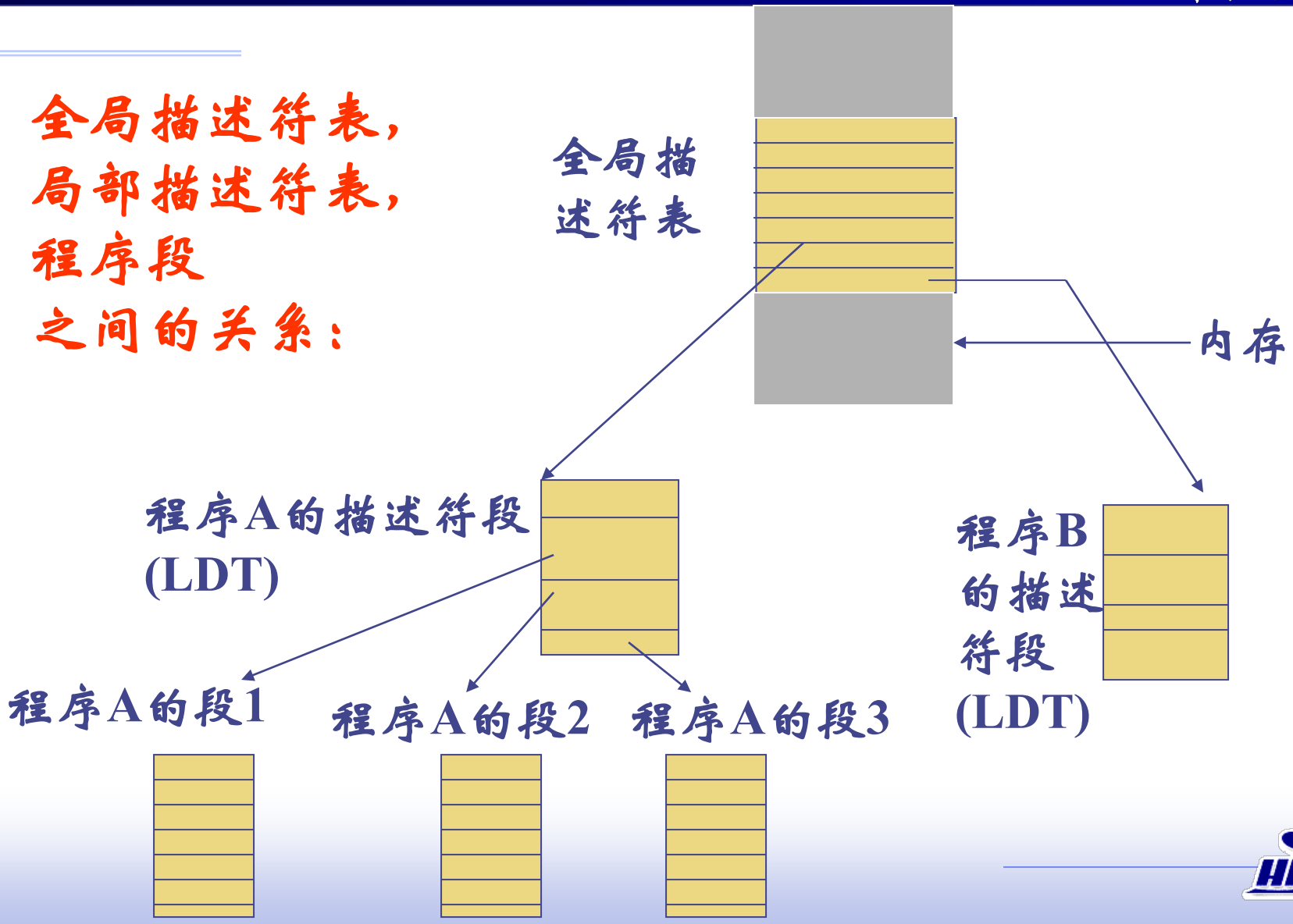
全局描述符表：只有一个。GDT最大可为64KB，存放8192个描述符。包括：

- 操作系统所使用的段的描述符；
- 各个LDT段的描述



保护方式下物理地址的形成

全局描述符表，
局部描述符表，
程序段
之间的关系：



保护方式下物理地址的形成

13位的索引值，可表示8192个索引项

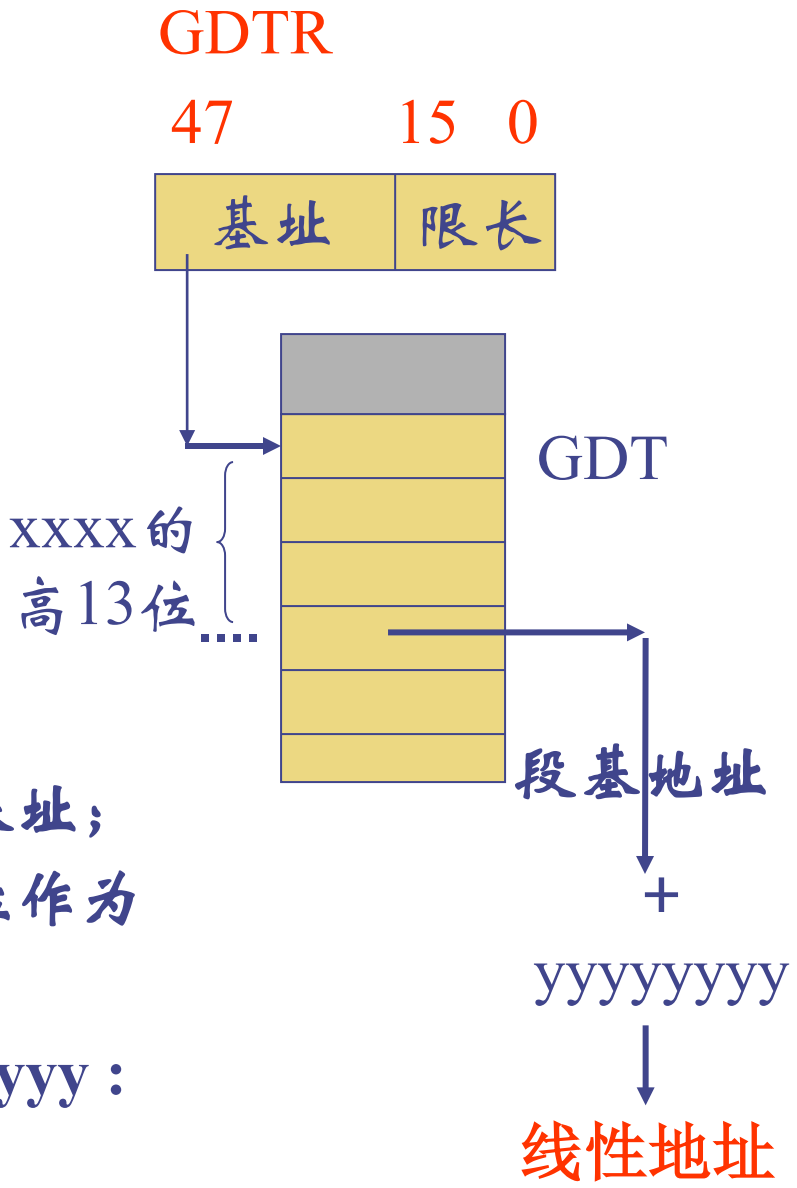
内存示意图

32位的基址，使得GDT可以定位在线性地址空间的任何位置

xxxx： cs、ds、ss、es、fs、gs

xxxx：yyyyyyyy 到线性地址的映射

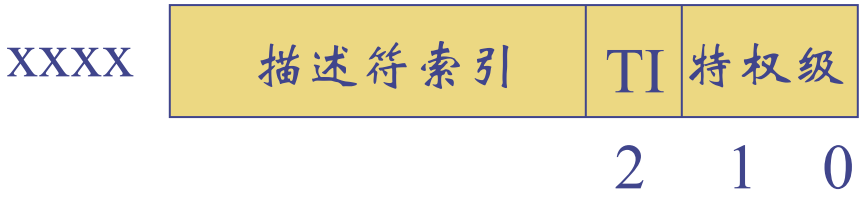
XXXX不是段的开始地址，
而是指出找相应段描述符
的方式。称为**段选择符**。



(1) TI 位为 0 (操作系统段)

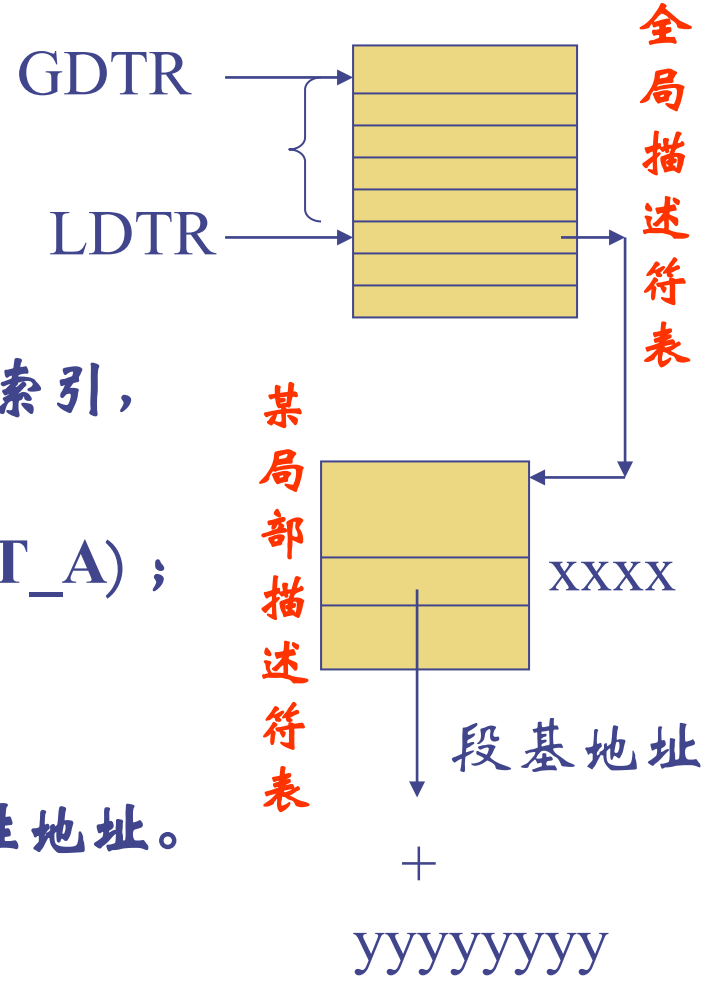
- 从GDTR寄存器中获取GDT的基址；
- 在GDT表中，以 **XXXX** 的高13位作为索引，取出一个描述符A；
- 描述符A中的段基地址 + yyyyyyyy：为要访问单元的线性地址。

xxxx：yyyyyyyy 到线性地址的映射

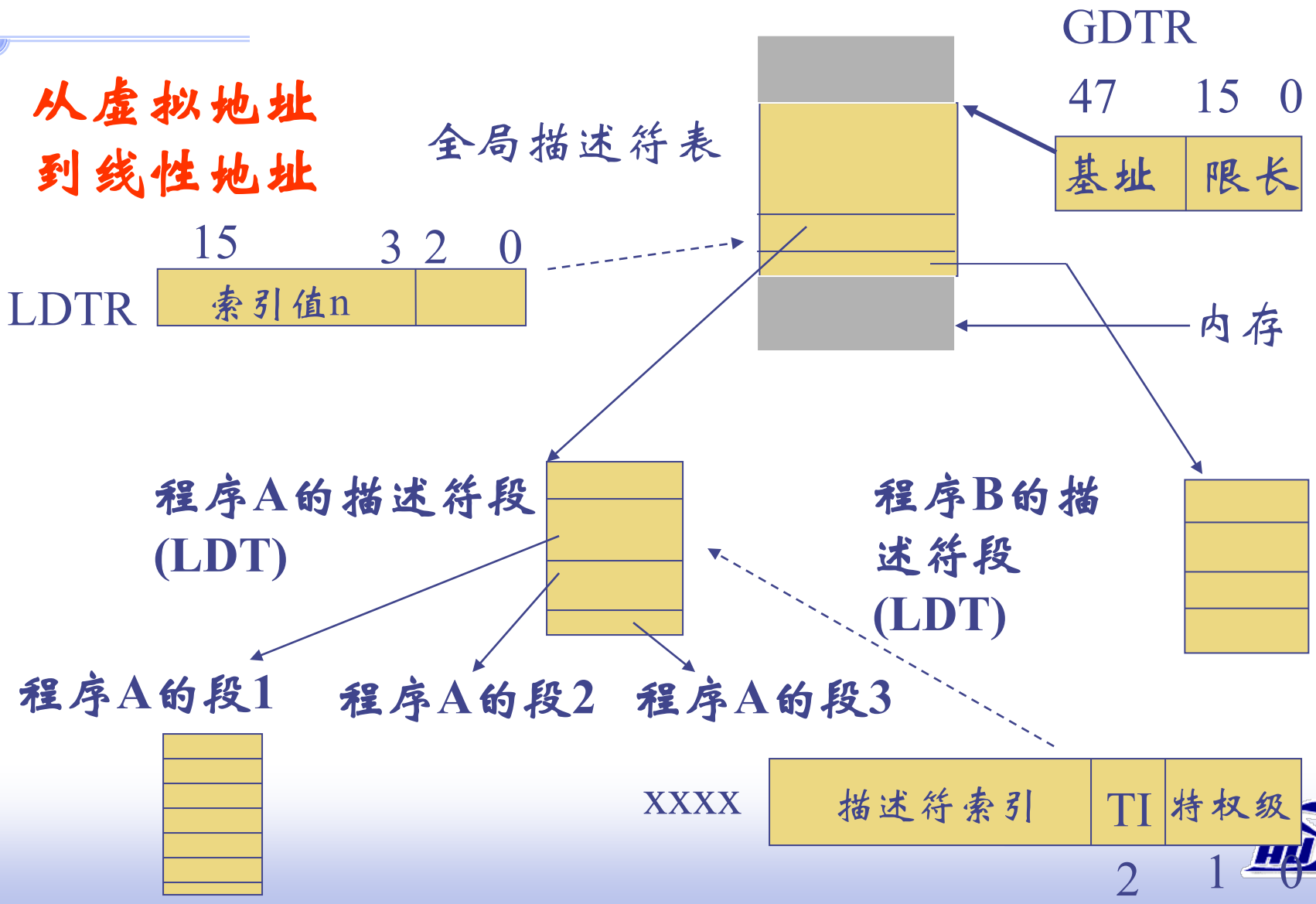


(2) 若TI 位为 1 (用户程序段)

- 从GDTR寄存器中获取GDT的基址；
- 在GDT表中，以**LDTR**的高13位作为索引，取出一个描述符A；
- 描述符A描述的段为一个LDT段(LDT_A)；
- 用 XXXX的高13位，作为索引，在LDT_A段中找到描述符P_A。
- P_A描述段的基址＋yyyyyyyy 为线性地址。

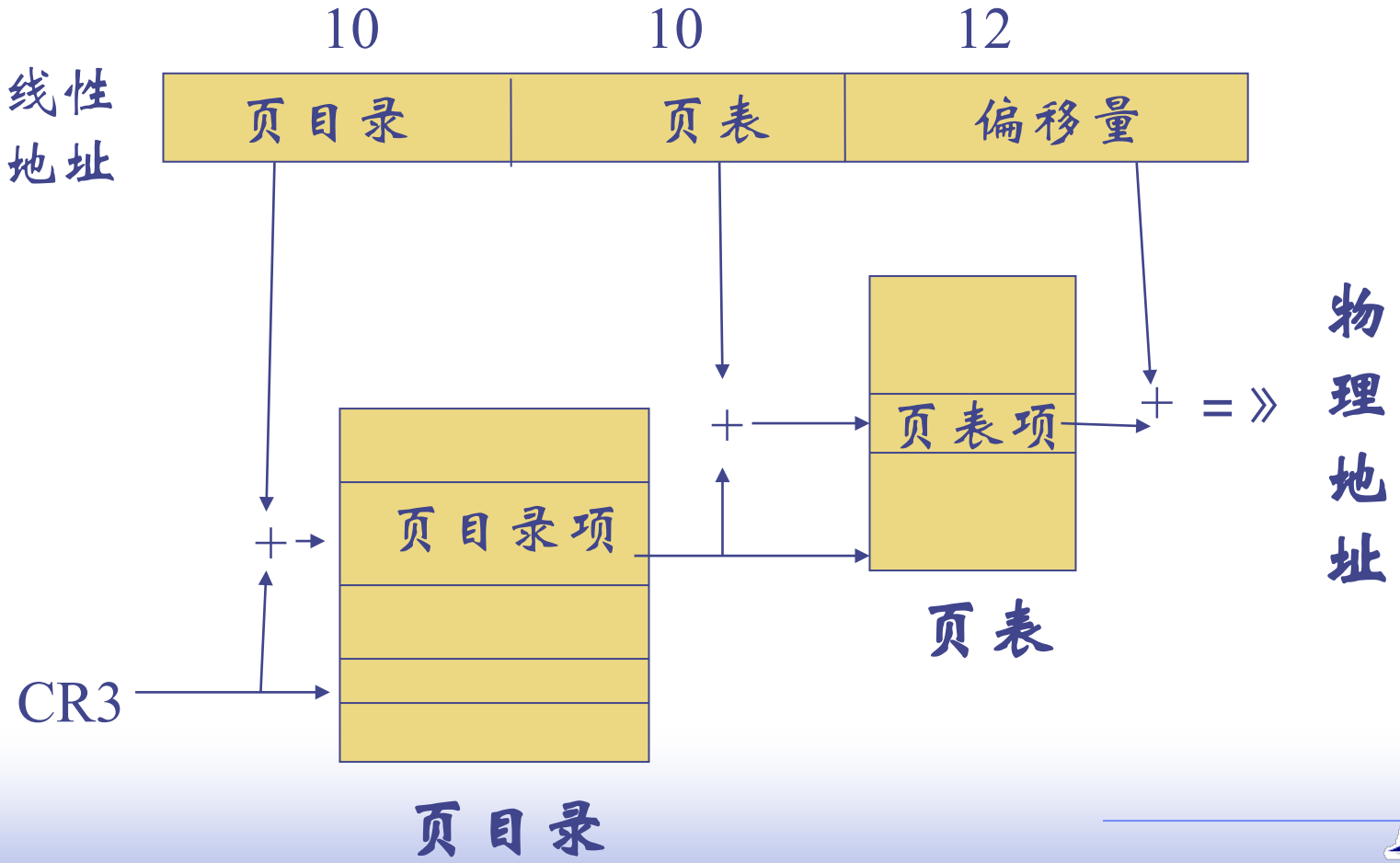


保护方式下物理地址的形成



保护方式下物理地址的形成

从线性地址到物理地址



保护方式下物理地址的形成

華中科技大學

Intel® 64 and IA-32 Architectures Software Developer's Manual

Q：GDTR 是如何设置的？

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /2	LGDT *m16&32*	M	N.E.	Valid	Load *m* into GDTR.
0F 01 /3	LIDT *m16&32*	M	N.E.	Valid	Load *m* into IDTR.
0F 01 /2	LGDT *m16&64*	M	Valid	N.E.	Load *m* into GDTR.
0F 01 /3	LIDT *m16&64*	M	Valid	N.E.	Load *m* into IDTR.

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction's operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

保护方式下物理地址的形成

Q：LDTR 是如何设置的？

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /2	LLDT *r/m*16	M	Valid	Valid	Load segment selector *r/m*16 into LDTR.

Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If bits 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

保护方式下物理地址的形成

- 描述符
- 描述符表
  - 局部描述符表
  - 全局描述符表
  - 中断描述符表
- 段选择符
- 线性地址的形成
- 物理地址的形成