



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY



操作系统

Operating Systems

夏文 副教授

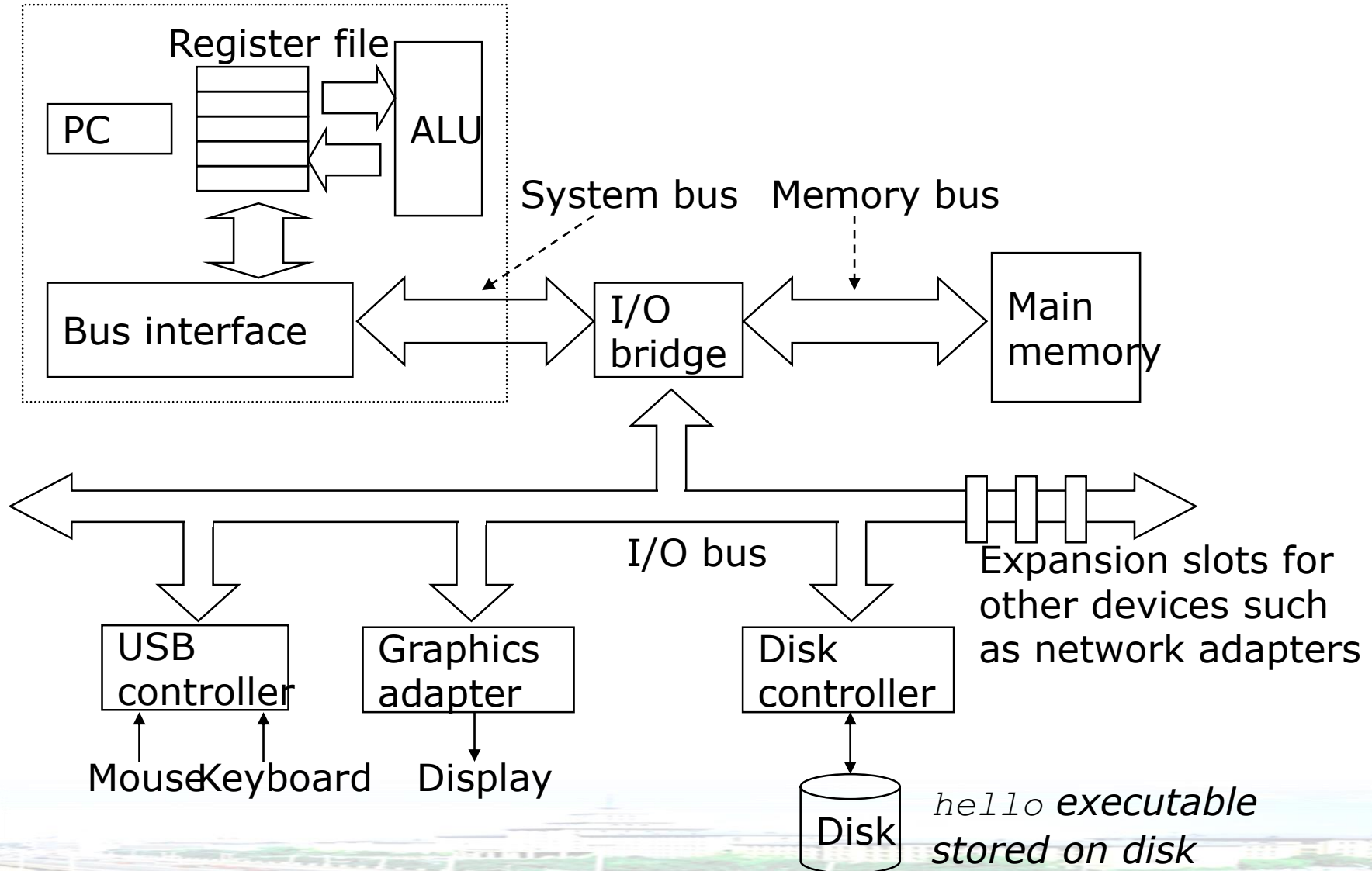
xiawen@hit.edu.cn

哈尔滨工业大学（深圳）

2019年11月

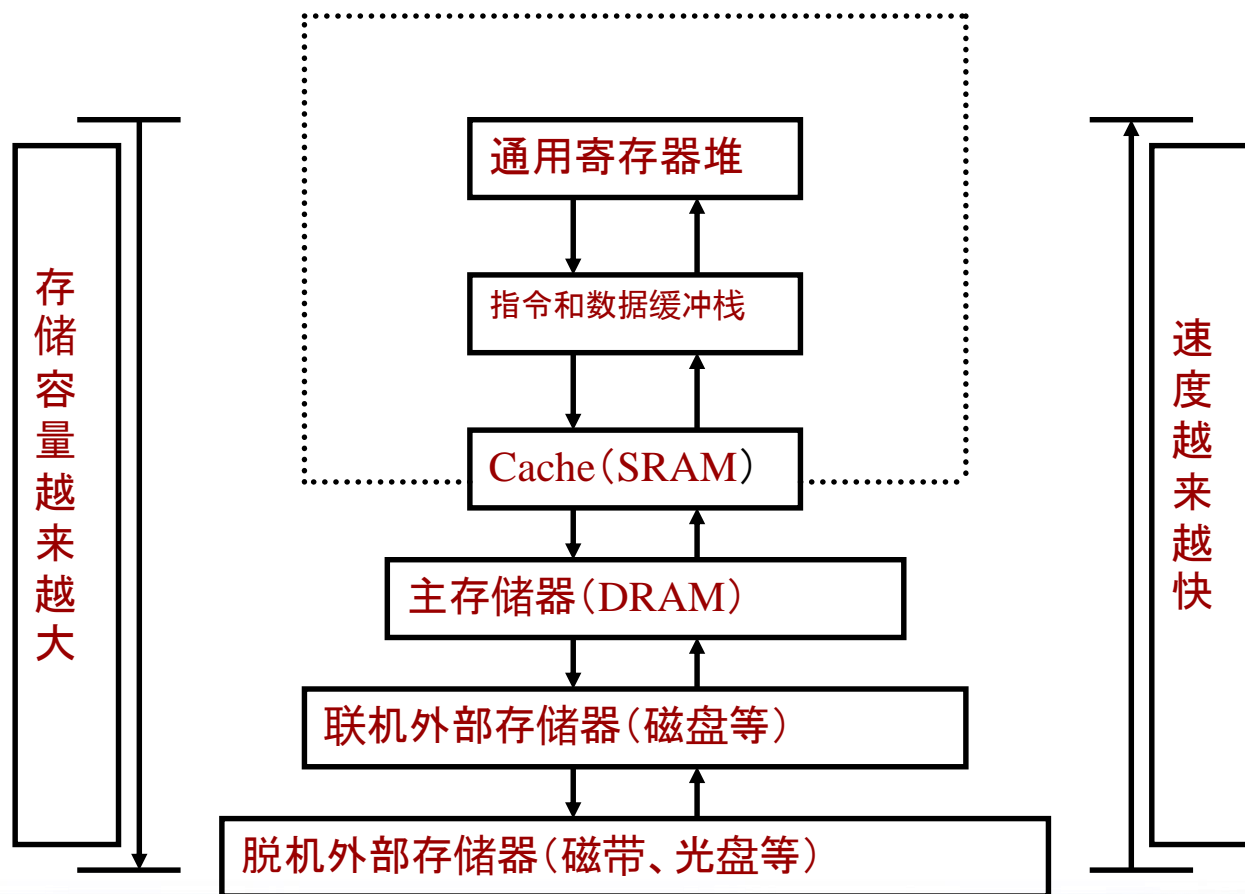
Computer Architecture

CPU

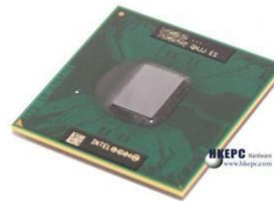


存储器的体系结构

■ 存储与计算的瓶颈问题



存储器



存储器属性

- 靠CPU侧: SRAM + DRAM

TABLE I: Characteristics of Different Types of Memory

Category	Read Latency (<i>ns</i>)	Write Latency (<i>ns</i>)	Endurance (# of writes per bit)
SRAM	2-3	2-3	∞
DRAM	15	15	10^{18}
STT-RAM	5-30	10-100	10^{15}
PCM	50-70	150-220	10^8 - 10^{12}
Flash	25,000	200,000-500,000	10^5
HDD	3,000,000	3,000,000	∞

小插曲

- A: “小雷，听说你大学的专业是计算机科学与技术？”
- B: “是的，领导。”
- A: “那好，去帮我搬下电脑。”
- B: “领导，请你尊重这门专业，计算机科学是……”
- A: “请你谈谈NAS设备卷管理模块中失效数据恢复问题的应用。”
- B: “领导，电脑放哪？”



第6章 内存管理

主要内容

6.1 背景

- (1) 程序的装入、运行和地址映射;
- (2) 逻辑地址空间和物理地址空间
- (3) 进程内外存交换

6.2 连续内存分配

- (1) 固定等长分区、固定变长分区、可变分区、分区分配算法;
- (2) 碎片问题与内存紧缩

6.3 分段内存管理

6.4 分页内存管理

6.5 段页结合内存管理



6.1 背景

- 内存是做什么用的？
- 程序是如何装入内存的？
- 数据在内存中是如何组织的？
- 程序在运行中地址是如何管理的？
- 内存不够用怎么办？

再回到那个恒久的话题

⑩ 执行程序是计算机的基本任务

```
int main(int argc, char* argv[])
{
    int i , to, sum = 0;
    to = atoi(argv[1]);
    for(i=1; i<=to; i++)
    {
        sum = sum + i;
    }
    printf("%d", sum);
}
```



```
C:\>sum 3127
4890628
C:\>sum 6656
22154496
C:\>sum 12345
76205685
C:\>sum 32178
517727931
C:\>
```

⑩ 静态链接？动态链接？



让程序执行起来就成了最重要的事!

⑩ 第一步: 编译——从C到汇编

```
int main(int argc, char* argv[])
{
    int i, to, sum = 0;
    to = atoi(argv[1]);
    for(i=1; i<=to; i++)
    {
        sum = sum + i;
    }
    printf("%d", sum);
}
```

源代码

```
.text
_entry: //入口地址
    mov ax, [8+sp]
    mov [_environ], ax
    call _main
    push ax //main返回值
    call _exit
_main:
    mov [_sum], 0
    sub sp, 4
    mov [sp+4], [_environ+4]
    call _atoi
    mov [_to], ax //atoi返回ax
    mov [_i], 1
```

汇编代码

1:

```
j> 2f, [_i], [_to]
mov ax, [_sum]
add ax, [_i]
mov [_sum], ax
add [_i], 1
jmp 1b
```

2:

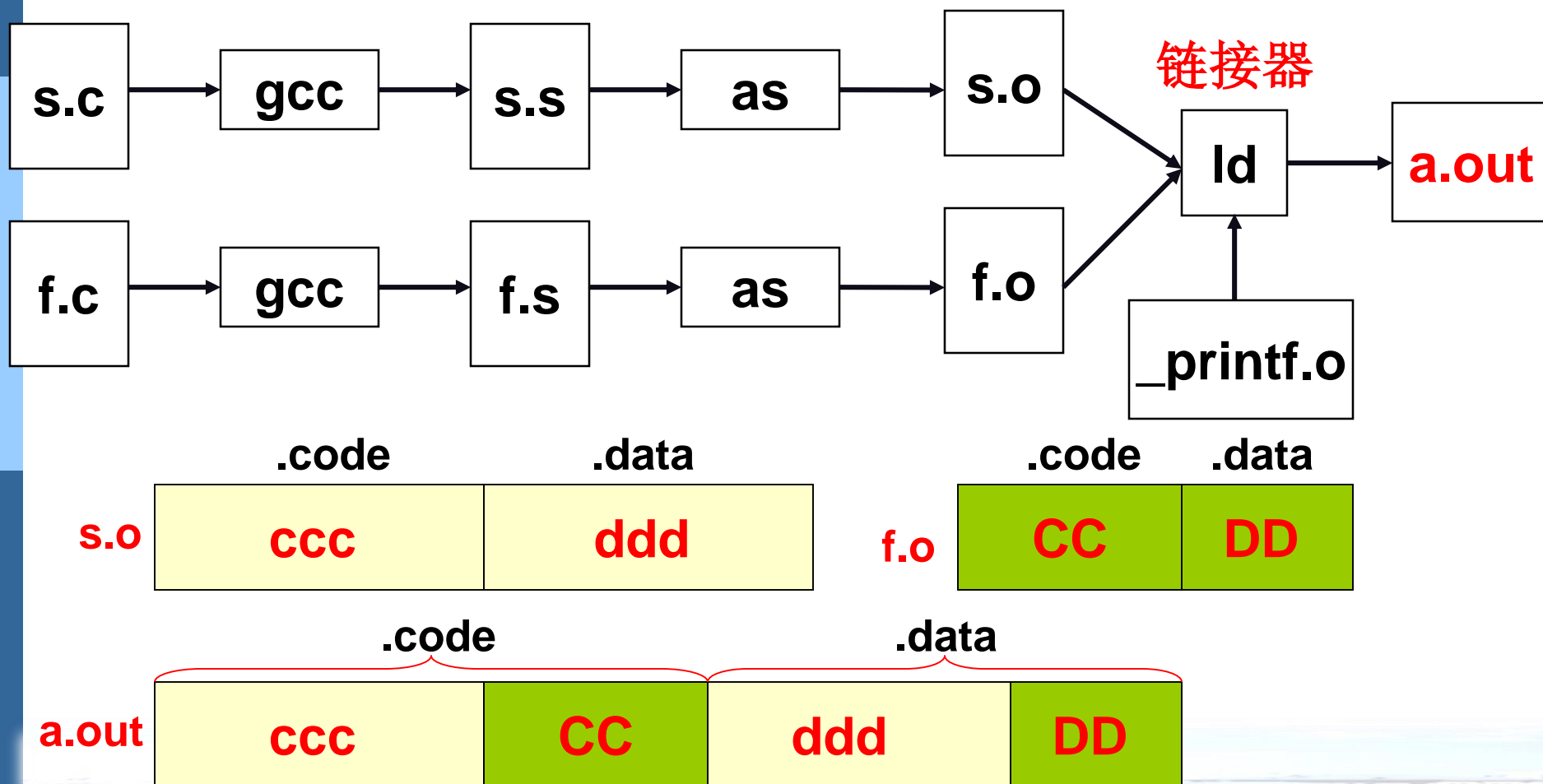
```
sub sp, 8
mov [sp+4], [_sum]
call _printf
ret
```

.data:

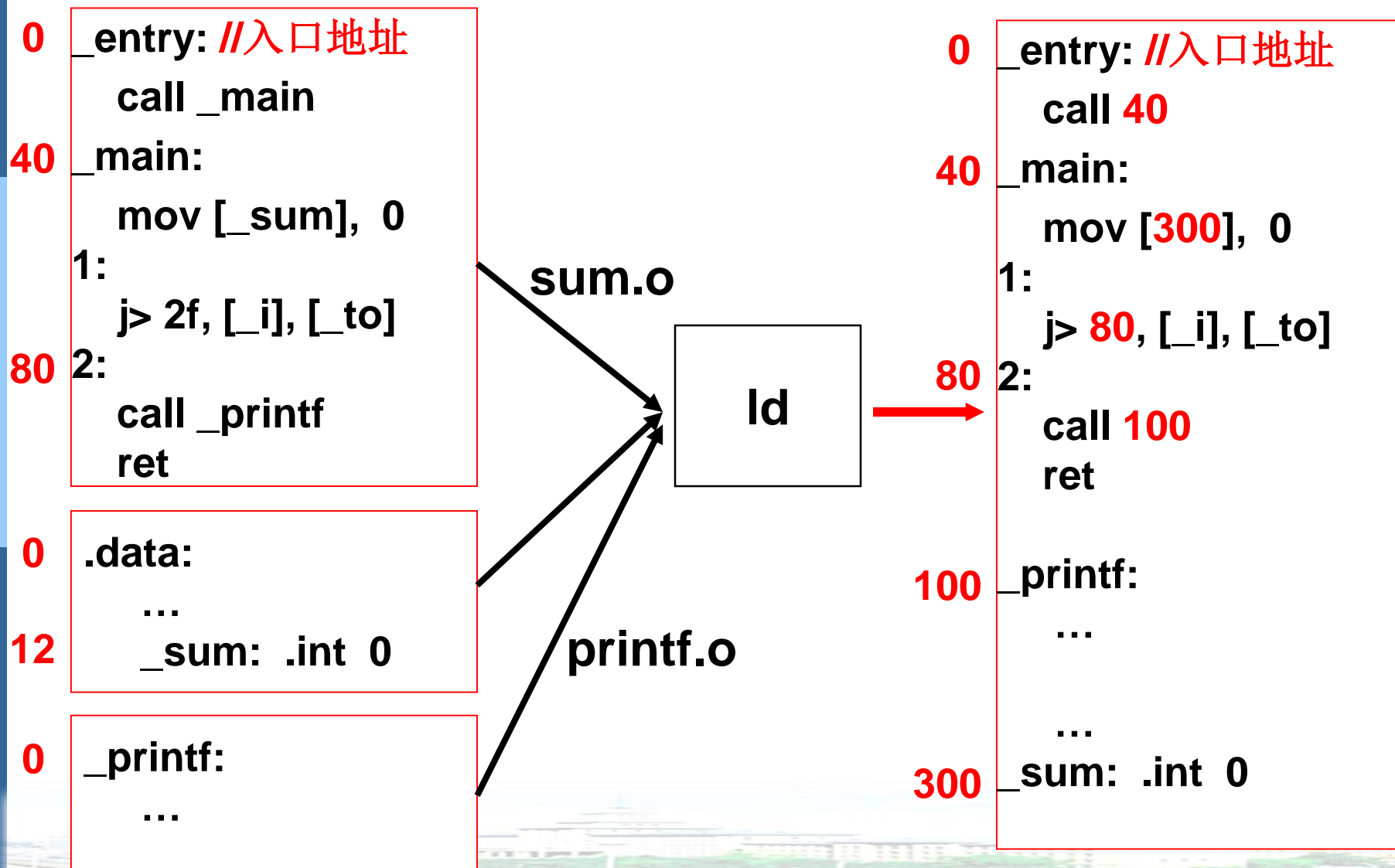
```
_environ: .long 0
_i: .int 0
_to: .int 0
_sum: .int 0
```

许多东西有待明确...

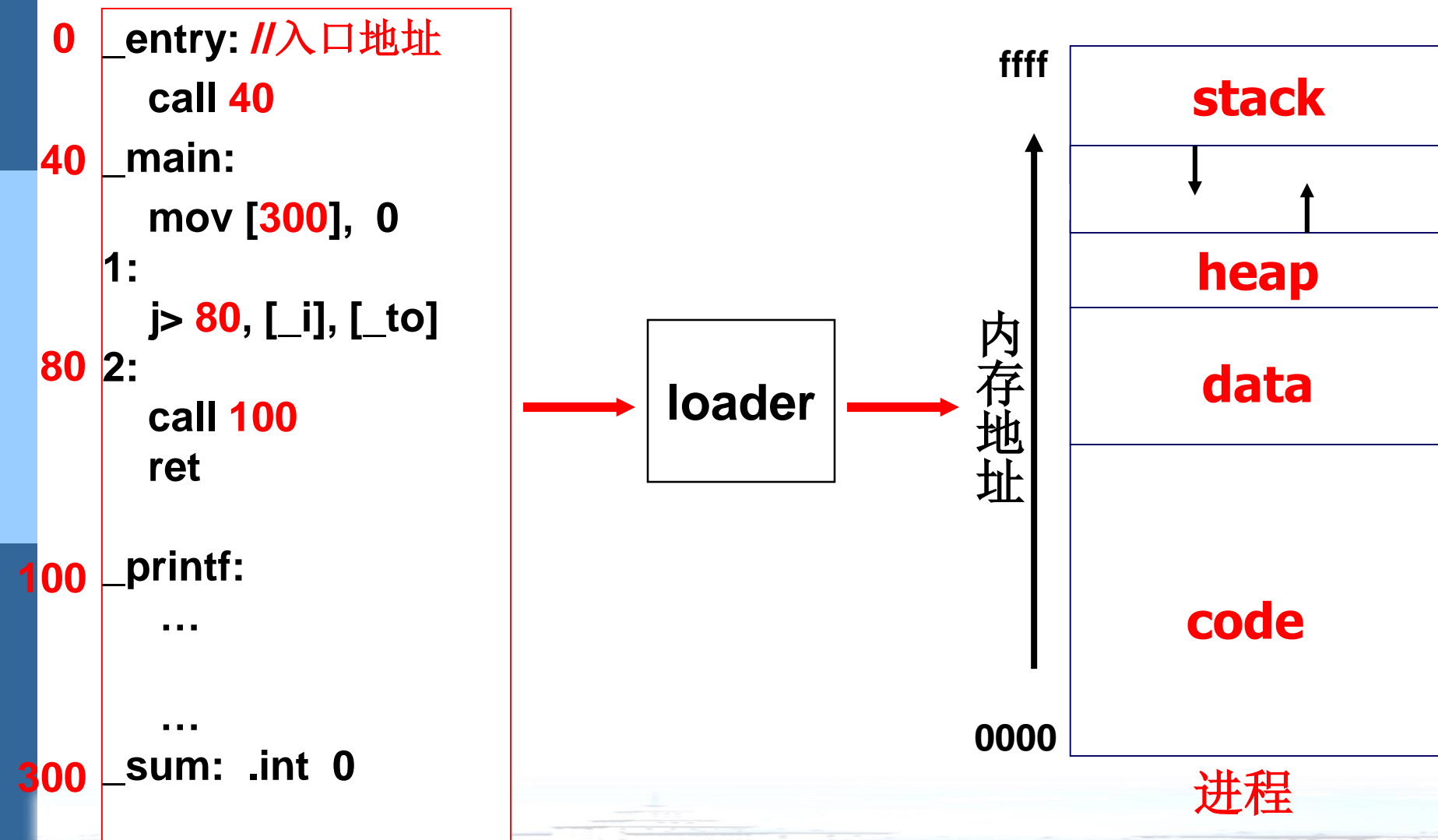
⑩ 第二步: 汇编与链接——从汇编到可执行程序



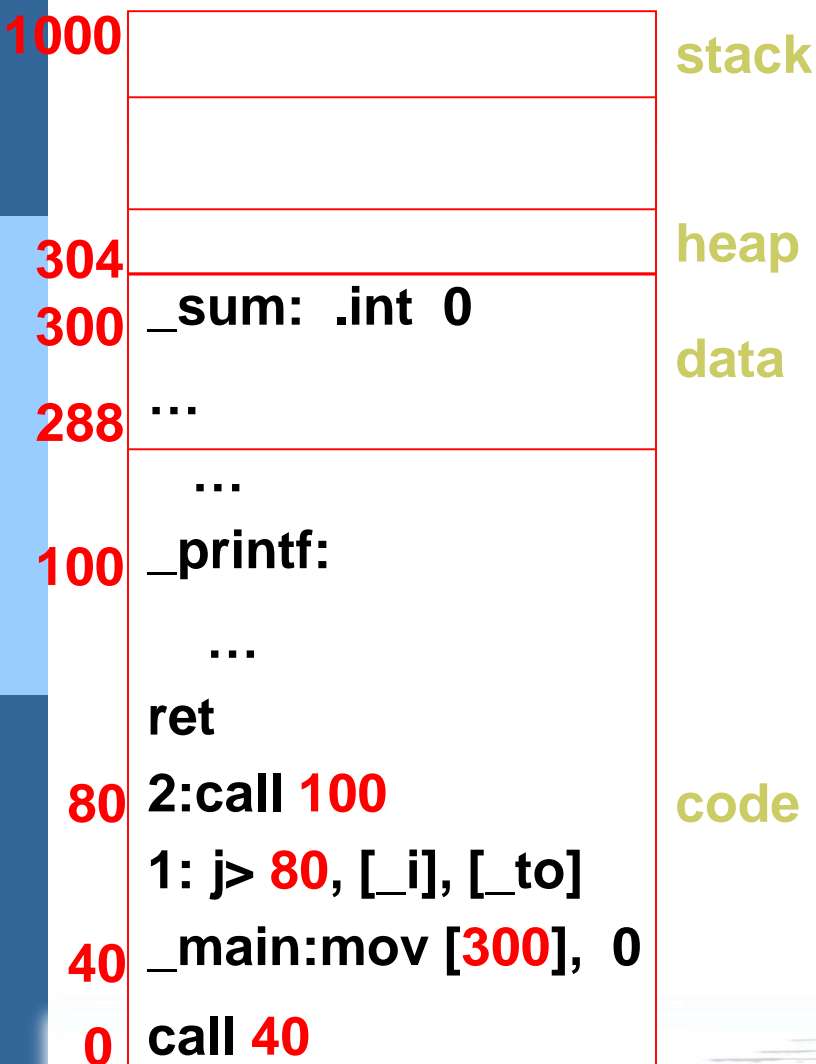
前面的程序经过链接以后...



现在还是程序，不是进程



程序可以执行了吗？



⑩ 程序怎么能正确开始？

- 将代码段放在内存中从0开始的地方
- 将数据段放在内存中从288开始的地方
- 设置PC=0
- 如果内存中从0开始的一段内存有专门的用途怎么办？
如存放中断处理程序

需要重定位!

1000是由硬件和操作系统决定的!



⑩ 假设内存从地址1000以后是可以使用的

```
1300 _sum: .int 0
1288 ...
...
1100 _printf:
...
ret
1080 2:call 100
1: j> 80, [_i], [_to]
1040 _main:mov [300], 0
1000 call 40
```

重定位

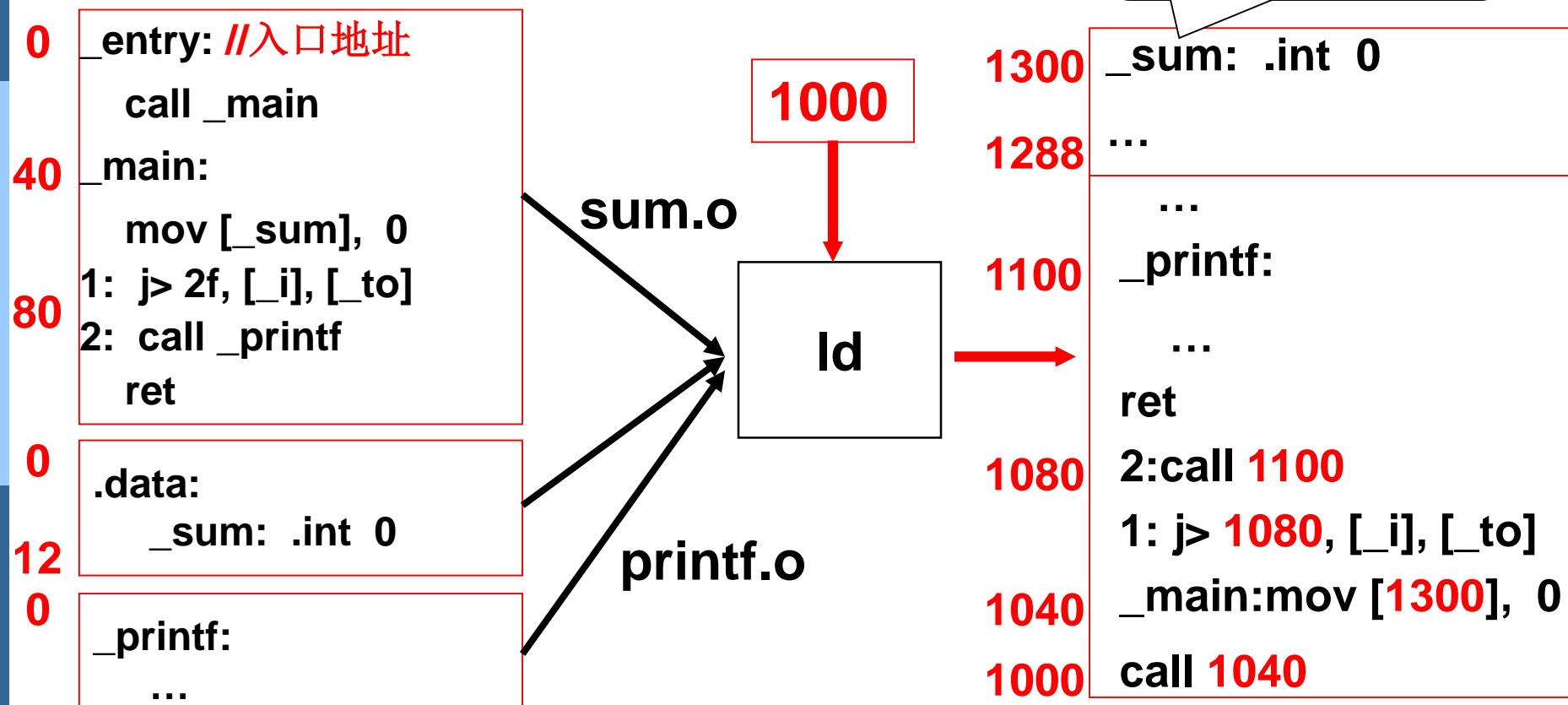


```
1300 _sum: .int 0
1288 ...
...
1100 _printf:
...
ret
1080 2:call 1100
1: j> 1080, [_i], [_to]
1040 _main:mov [1300], 0
1000 call 1040
```

■ **重定位:** 为执行程序而对其中出现的地址所做的修改

重定位可以执行的时机

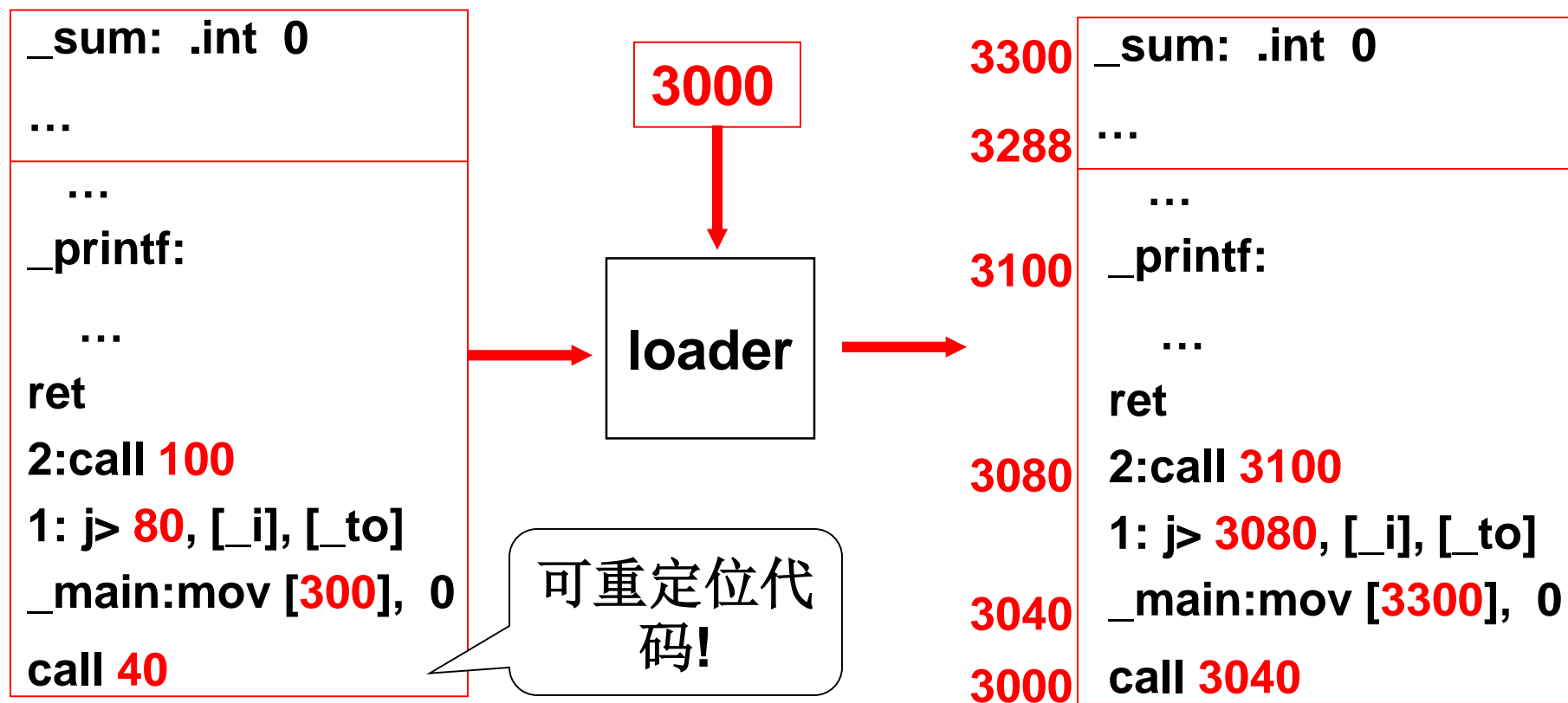
⑩ 第一种时机：在编译链接时



■ **绝对代码:** 这样的代码只能放在事先确定的位置上

并发 \Rightarrow 多个程序同时在内存中

⑩ 分别用1000, 2000, ... 吗? 第二种时机: 载入时



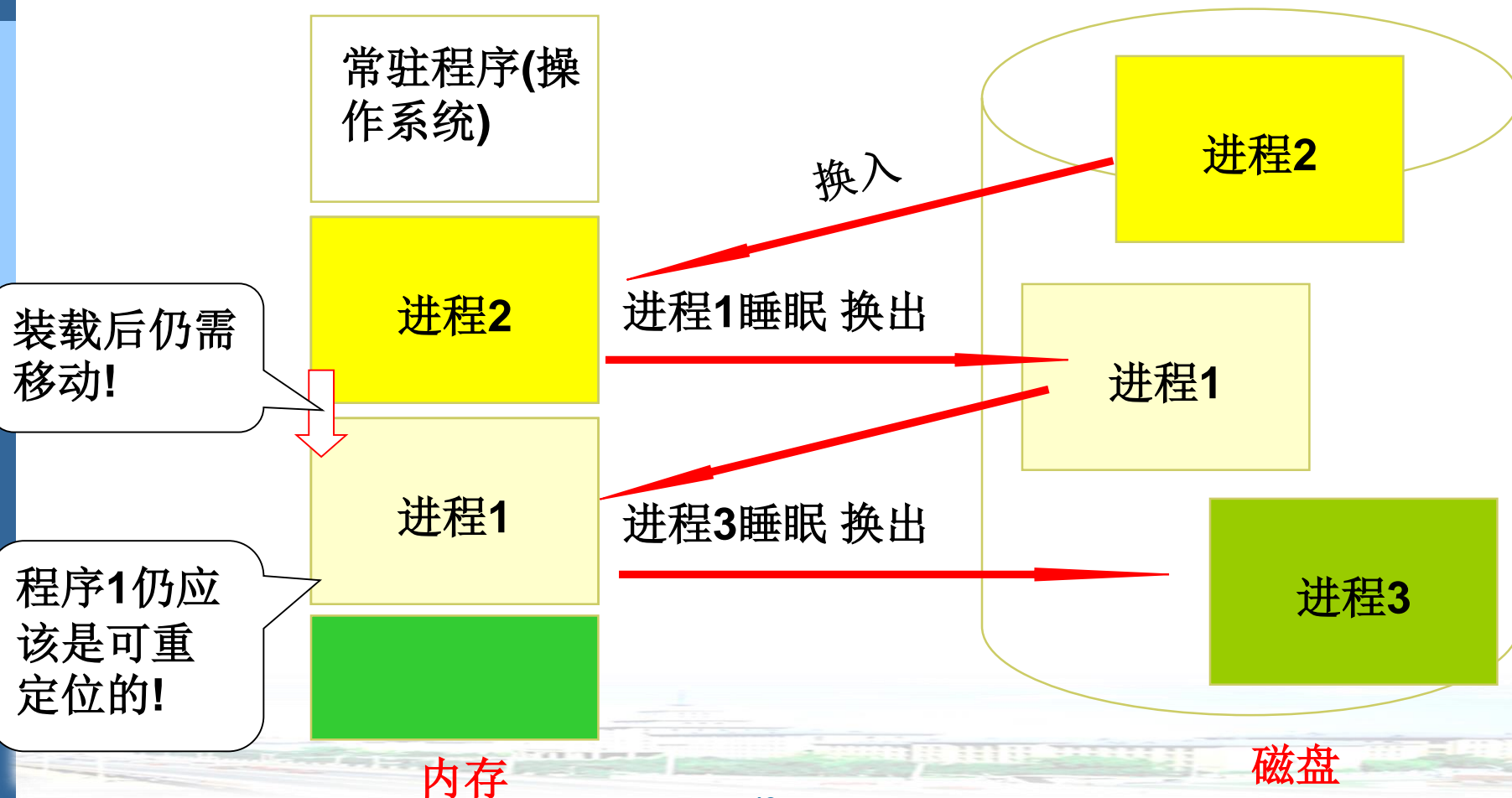
■ 装载时的重定位仍然存在缺点... 一旦载入不能移动

移动也是很有必要的!

⑩ 一个重要概念: **交换(swap)**

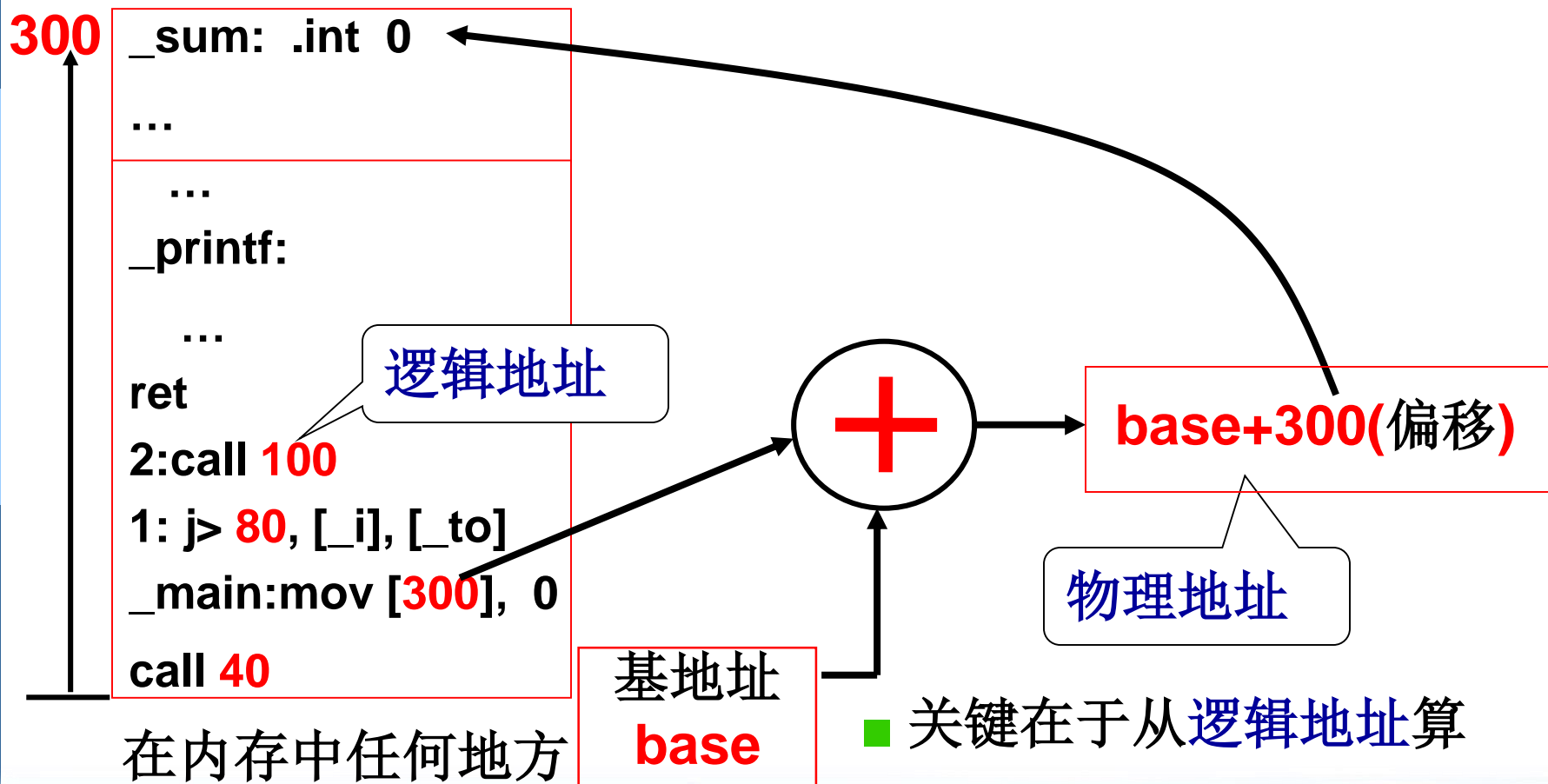
如何实现呢?

⑩ 能让更多的进程并发, 提高内存的利用率



重定位最合适的时机 – 运行时重定位

⑩ 内存中的代码总是可重定位的!

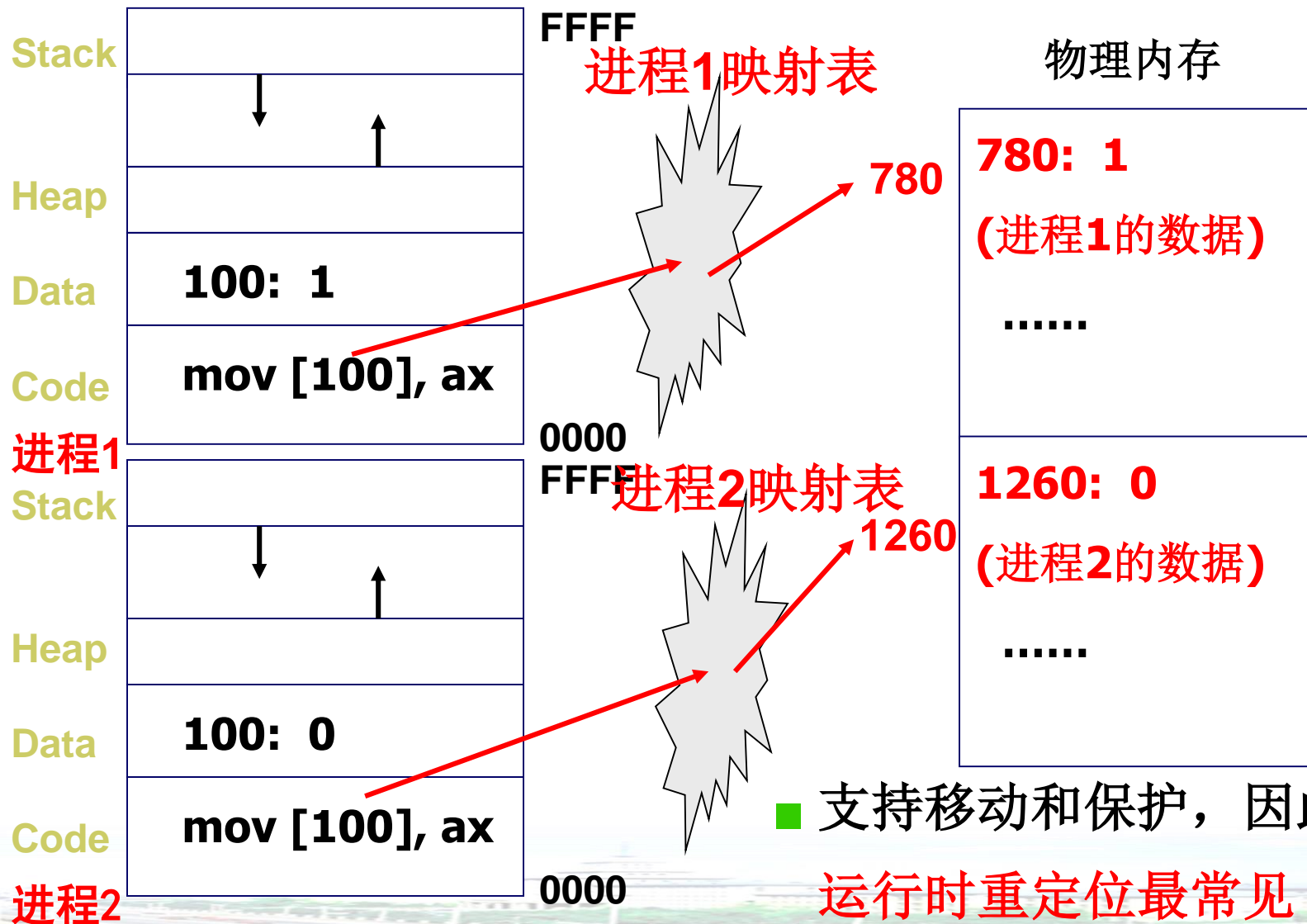


■ 关键在于从**逻辑地址**算出**物理地址**: **地址翻译**



运行时重定位还有一个好处: **进程保护**

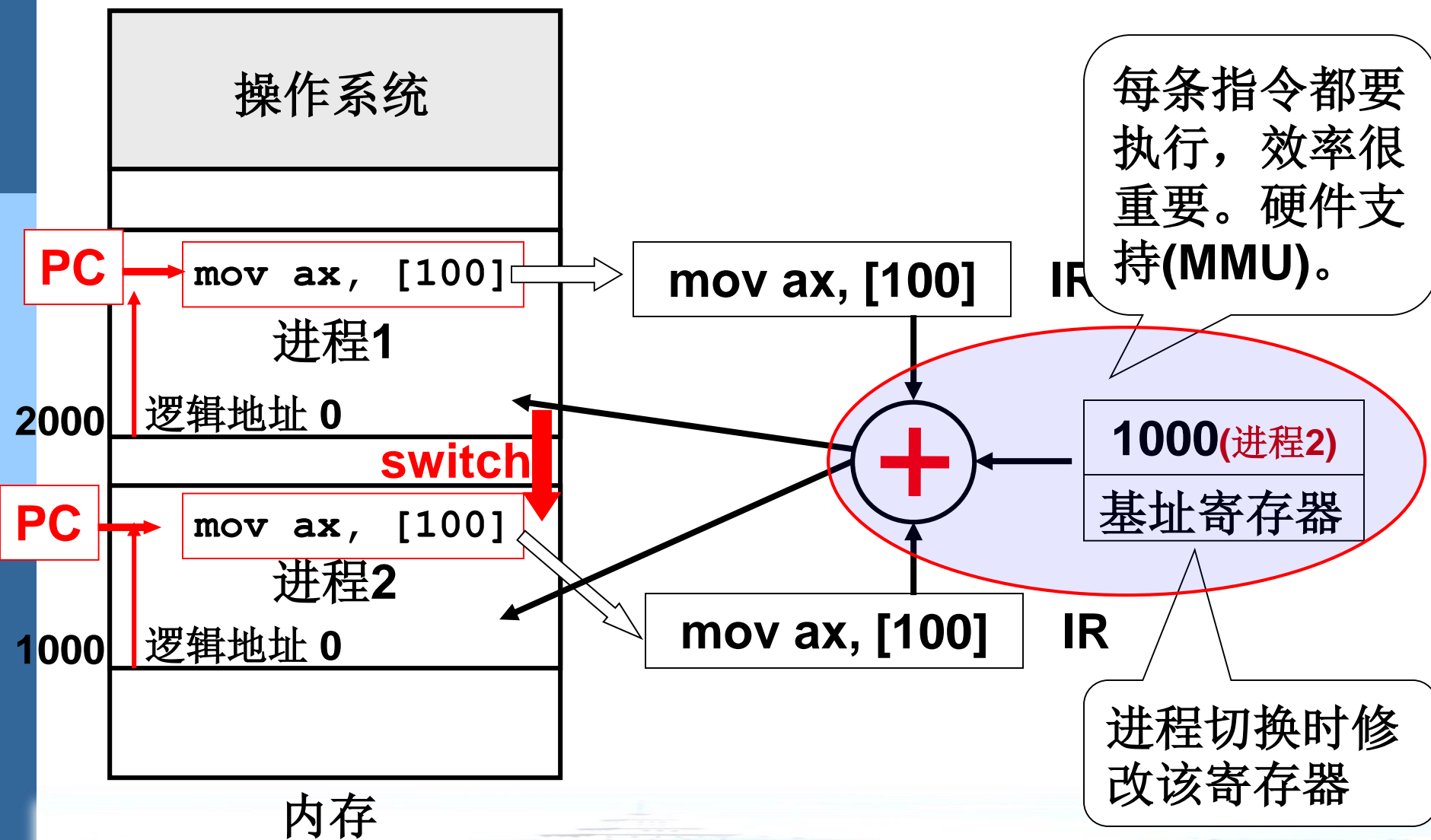
重定位寄存器和界地址寄存器





整理一下思路...

现在的问题集中在左边：
内存怎么分配？



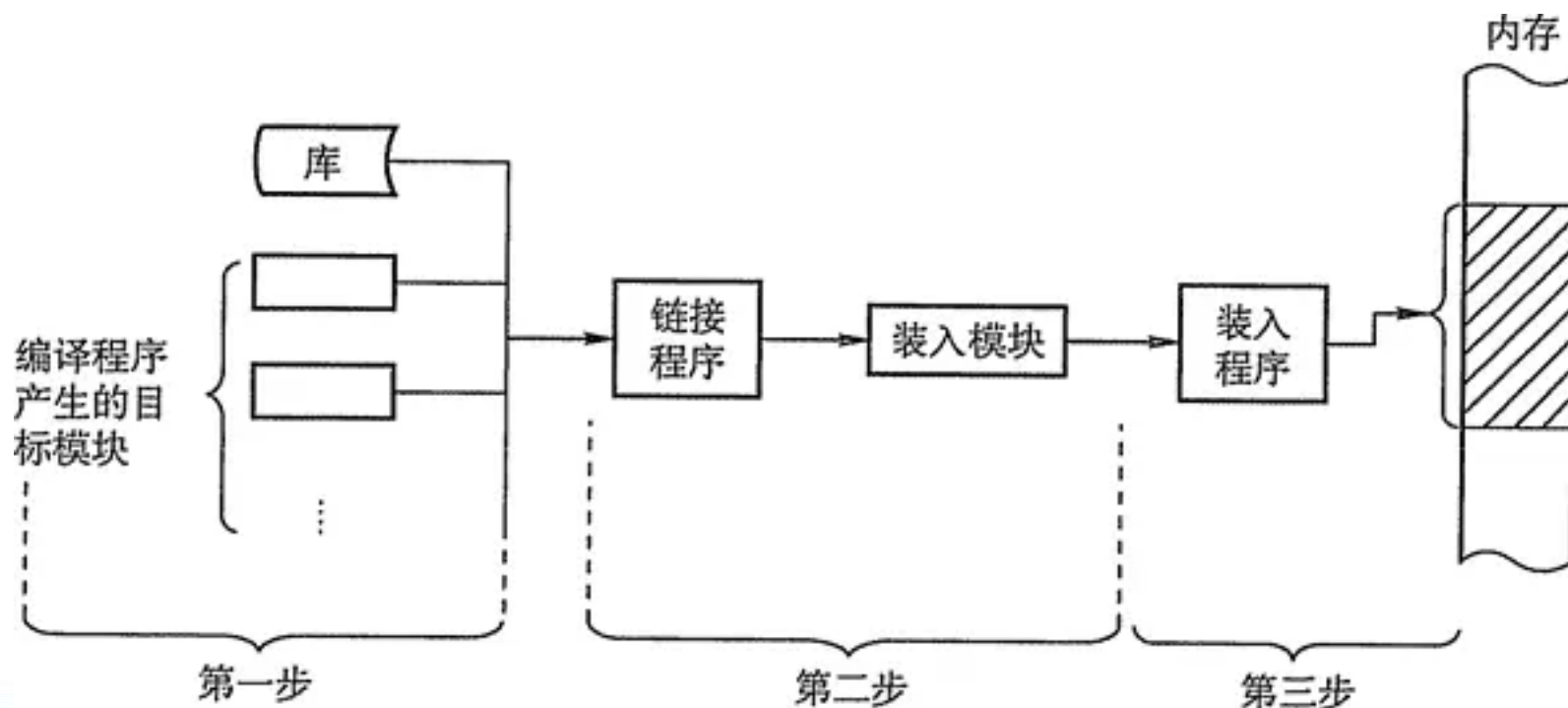


简单总结一下

- 操作系统将可执行程序加载到内存，合理**分配内存**。
- 可执行程序经过编译产生了逻辑地址，**为了提高进程的并发和内存利用率**，逻辑地址和运行物理地址间需要**地址转换**。
- 嵌入式操作系统多数内存管理非常简单，甚至没有内存管理。
- PC和服务端、分布式等现代操作系统基本都支持虚拟内存管理。

简单总结一下(续)

- 创建进程首先要将程序和数据装入内存。将用户源程序变为可在内存中执行的程序，通常需要编译、链接、装入三个步骤





第1组3个重要概念

- **虚拟地址：**用户编程时将代码（或数据）分成若干个段，每条代码或每个数据的地址由段名称 + 段内相对地址构成，这样的程序地址称为虚拟地址
- **逻辑地址：**虚拟地址中，段内相对地址部分称为逻辑地址
- **物理地址：**实际物理内存中所看到的存储地址称为物理地址
- **比较：**虚拟地址由用户编写程序时定义的全局地址；逻辑地址是用户定义的局部地址，是虚拟地址的组成部分；物理地址就是实际存在的以Byte为单位的存储单元的编号



第2组3个重要概念

- **逻辑地址空间：**

在实际应用中，将虚拟地址和逻辑地址经常不加区分，通称为逻辑地址。逻辑地址的集合称为逻辑地址空间

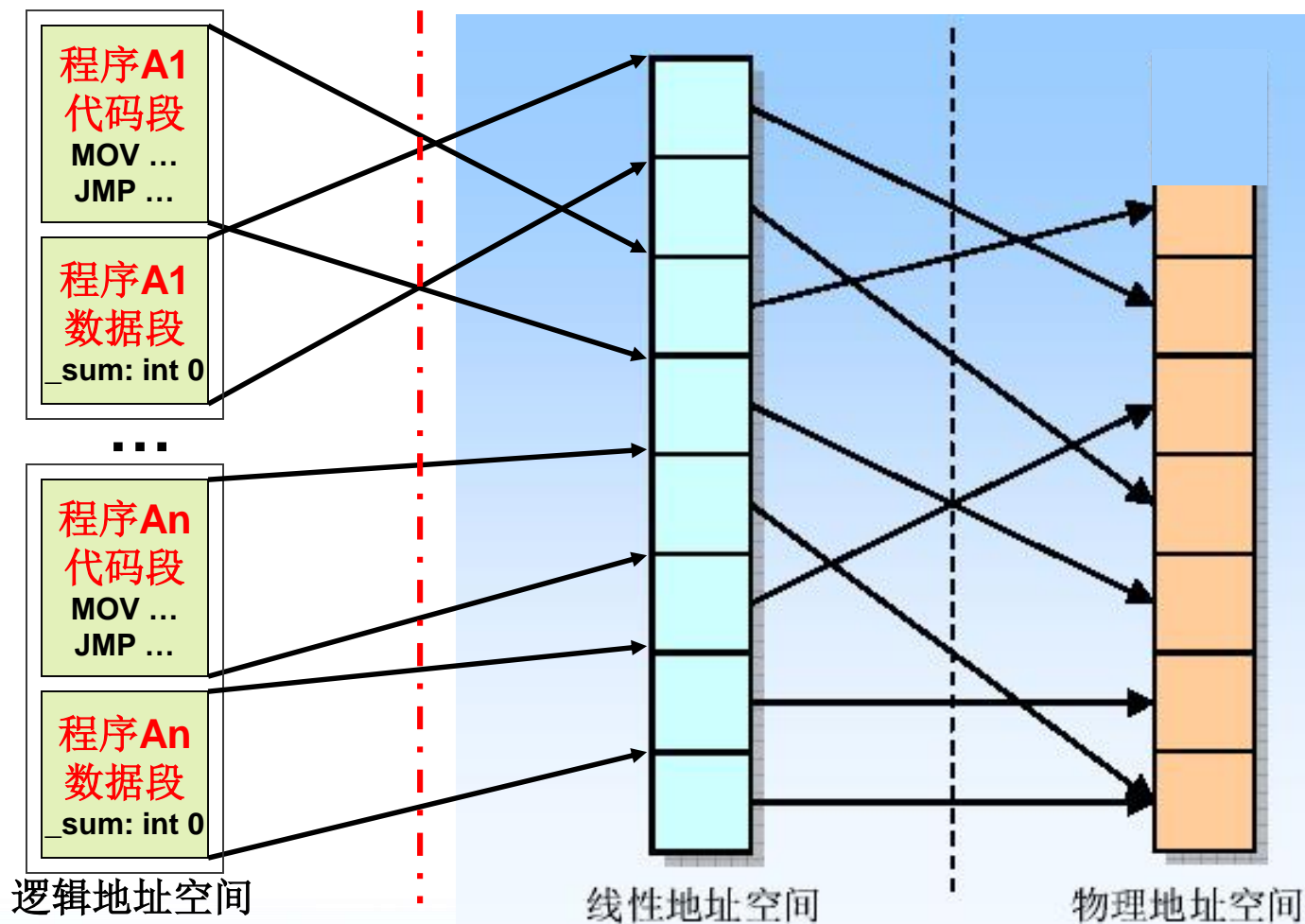
- **线性地址空间：**

CPU地址总线可以访问的所有地址集合称为线性地址空间

- **物理地址空间：**

实际存在的可访问的物理内存地址集合称为物理地址空间

逻辑地址空间-线性地址空间-物理地址空间的关系



第3组3个重要概念

- **MMU (Memory Management Unit 内存管理单元):**
实现将用户程序的虚拟地址（逻辑地址）
→ 物理地址映射的CPU中的硬件电路
- **基地址:**
在进行地址映射时，经常以段或页为单位并以其
最小地址（即起始地址）为基值来进行计算
- **偏移量:**
在以段或页为单位进行地址映射时，相对于基地址的地址值

虚拟地址与物理地址

- 嵌入式操作系统多数内存管理非常简单, 甚至没有内存管理。**虚拟地址=物理地址** → ■ 前者链接时重定位
- PC和服务端、分布式等现代操作系统基本都支持虚拟内存管理。**虚拟地址≠物理地址** → ■ 后者运行时重定位



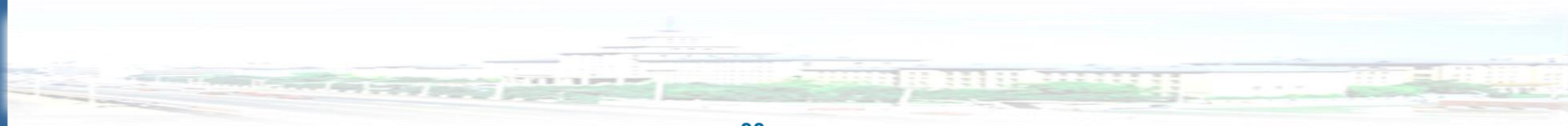
内存分配与管理方案

- 连续内存分配与管理
- 分段分配与管理
- 分页分配与管理
- 段页结合管理
- 请求式分页管理(虚拟内存)



6.2 连续内存分配与管理

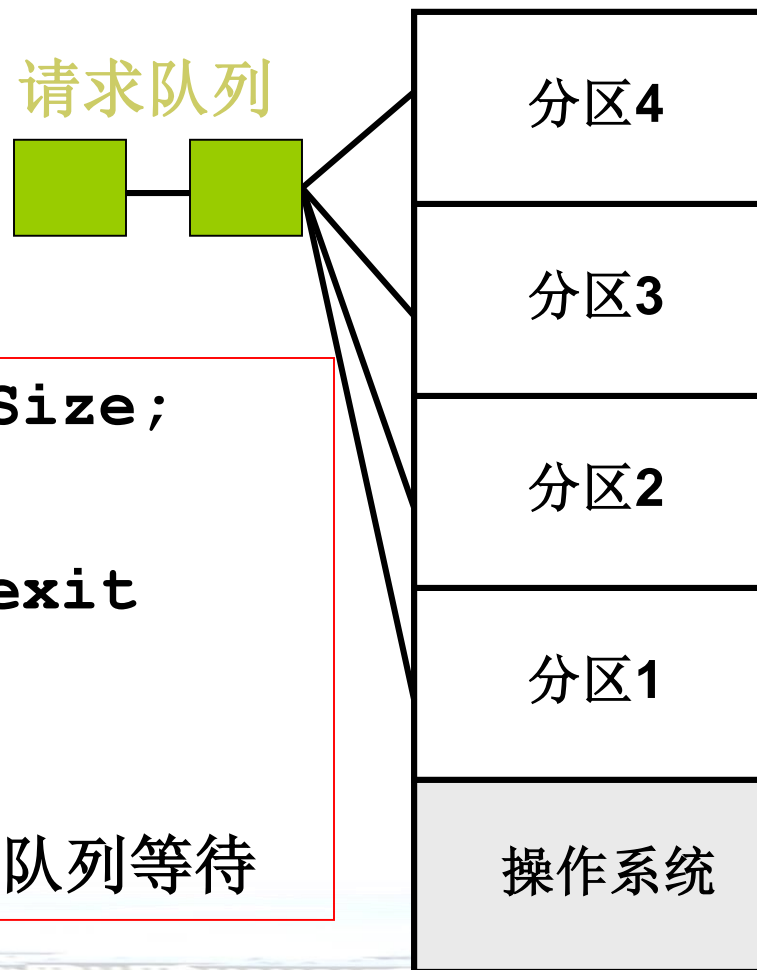
从最简单的分配方案开始!



(1)固定分区--等长分区

⑩ 给你一个面包， n 个孩子来吃，怎么办？

- 等分...
- 操作系统初始化时将内存等分成 k 个分区



```
bool AvailSec[k]; int SecSize;
```

内存请求算法 //进程创建时

1. `if (reqSize > SecSize) exit`
2. 找出`AvailSec[i]`为真的 i
3. 如果有，返回分区 i 的基址
4. 否则，将`current`加入请求队列等待

(2)固定分区--变长分区

⑩ 孩子有大有小，进程也有大有小...

- 初始化时将内存分成k个大小不同的分区

请求队列



```
struct Section AvailSec[k];
```

内存请求算法 //进程创建时

1. `if (reqSize > MaxSize) exit`
2. 找出 `AvailSec[i].Size > reqSize` 且 `AvailSec[i].Size` 最小的空闲分区 `i`
3. 如果有，返回分区 `i` 的基址
4. 否则，将 `current` 加入请求队列等待



(3)可变分区

⑩ 合理的方法应该是根据孩子饥饿程度来分割

- 根据reqSize进行动态分割

请求队列



内存请求算法 //进程创建时

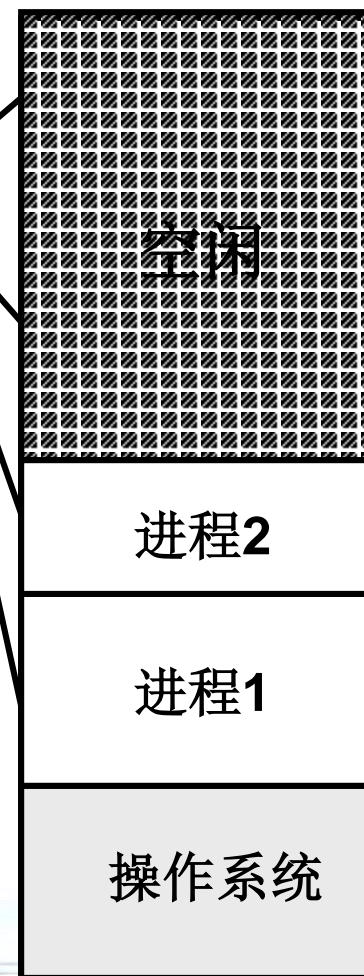
1. `if (reqSize > 内存大小) exit`

2. `if (reqSize > 空闲空间总尺寸)`

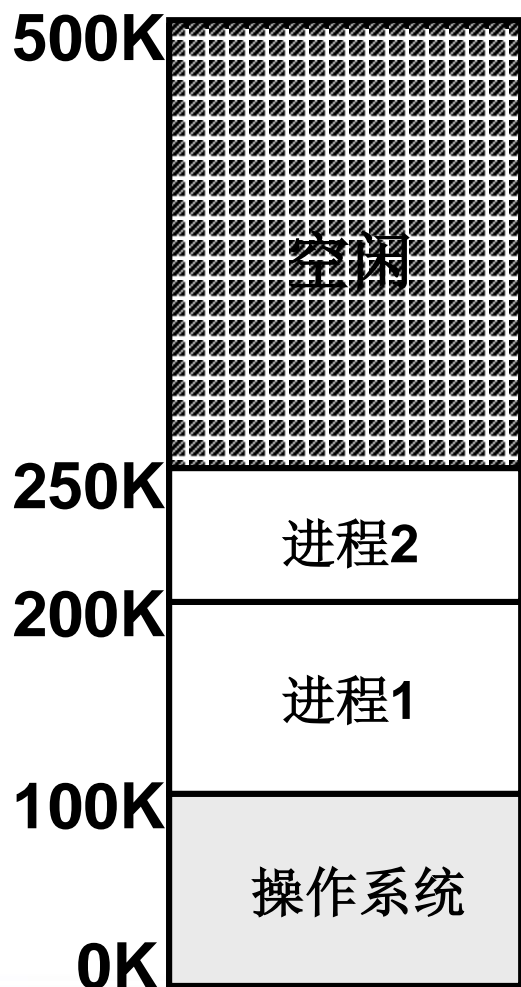
将current加入请求队列等待

3. 从空闲分区划出一个reqSize, 并返回其基地址 //那个空闲分区

4. 修改分区数据结构



可变分区的数据结构



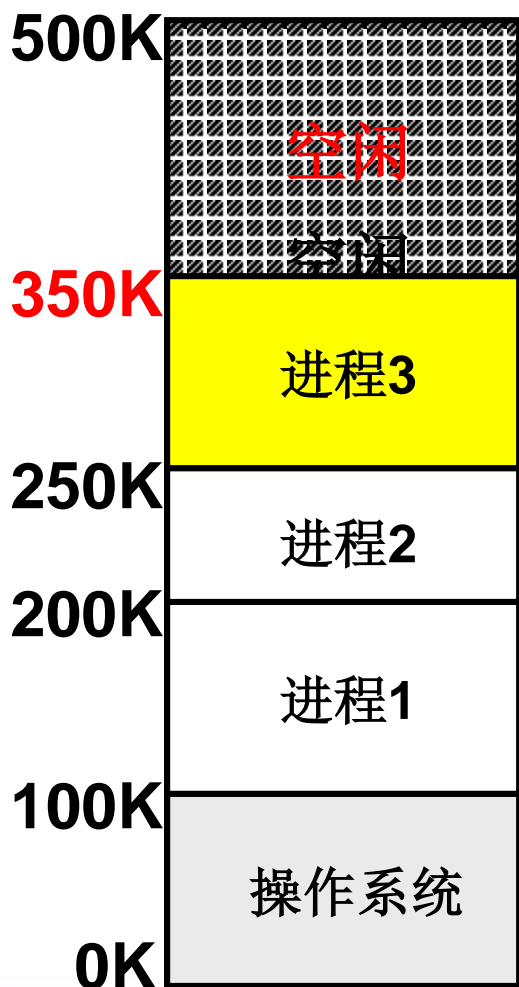
空闲分区表

始址	长度
250K	250K

已分配分区表

始址	长度	标志
0K	100K	OS
100K	100K	P1
200K	50K	P2

可变分区数据结构的变化(1)



⑩ 内存请求: reqSize = 100K

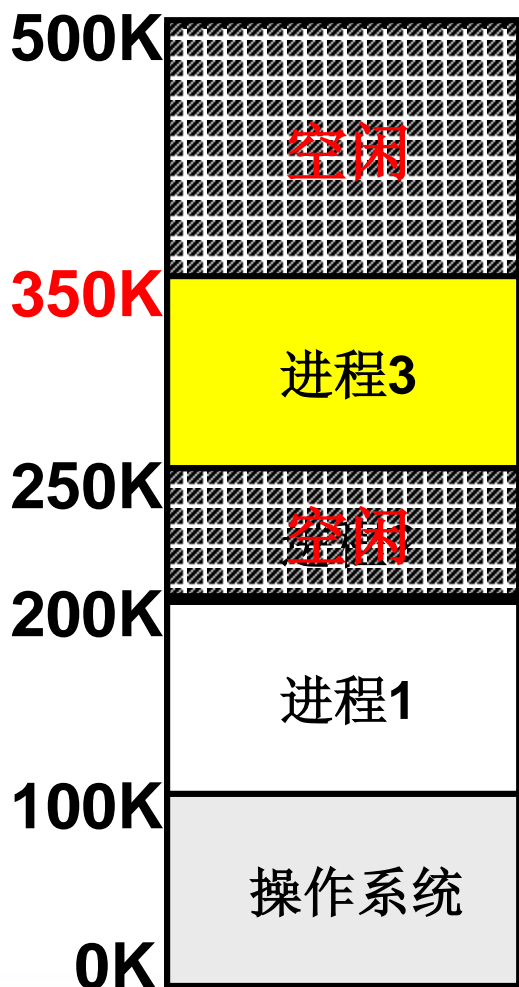
空闲分区表

始址	长度
250K	250K

已分配分区表

始址	长度	标志
0K	100K	OS
100K	100K	P1
200K	50K	P2
250K	100K	P3

可变分区数据结构的变化(2)



⑩ 进程2执行完毕，释放内存

空闲分区表

始址	长度
350K	150K
200K	50K

已分配分区表

始址	长度	标志
0K	100K	OS
100K	100K	P1
200K	100K	P2
250K	100K	P3

(4)分区分配算法 – 找到合适的空闲分区

⑩ 发起请求reqSize=40K怎么办？

⑩ 首次适配(**first-fit**):首次找到的满足要求的空闲分区 (350,150)。特点：快速！

⑩ 最佳适配(**best-fit**): 查找最小的满足要求的空闲分区(200,50)。特点:(1)搜索整个空闲分区表，慢！ (2)会产生许多小的空闲分区

⑩ 最差适配(**worst-fit**):查找最大的满足要求的空闲分区(350,150)。特点:(1)搜索整个空闲分区表，慢！ (2)新产生的空闲分区大一些

500K

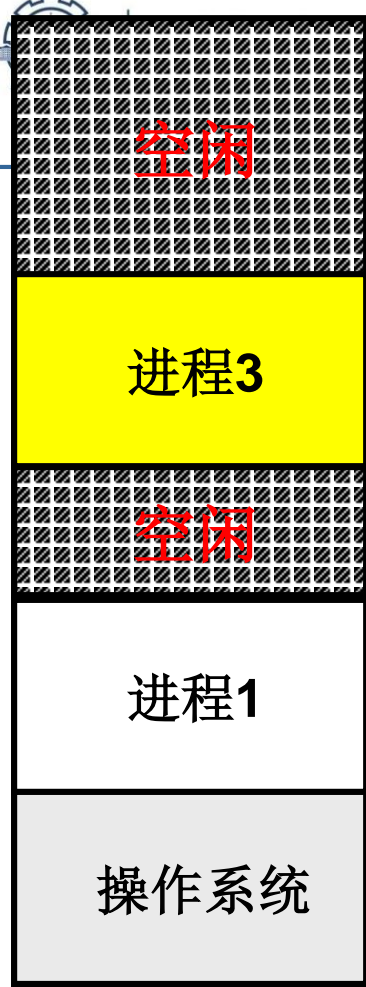
350K

250K

200K

100K

0K



空闲分区表

始址	长度
350K	150K
200K	50K



(5)碎片问题

- ⑩ **碎片**：内存中剩余的无法使用的存储空间
- ⑩ **外部碎片**：随着进程不断的装入和移出，对分区不断的分割，使得内存中产生许多**特别小的分区**，它们并不连续可用
- ⑩ **内部碎片**：对固定分区来说，只要分区被分配给某进程使用，其中并未占用的空间不能分给其他进程
- ⑩ 碎片的产生和分配“制度”有关！不易避免！
- ⑩ 碎片问题可以消除（**内存紧缩**），但时间成本太高！
- ⑩ 磁盘也存在碎片问题！

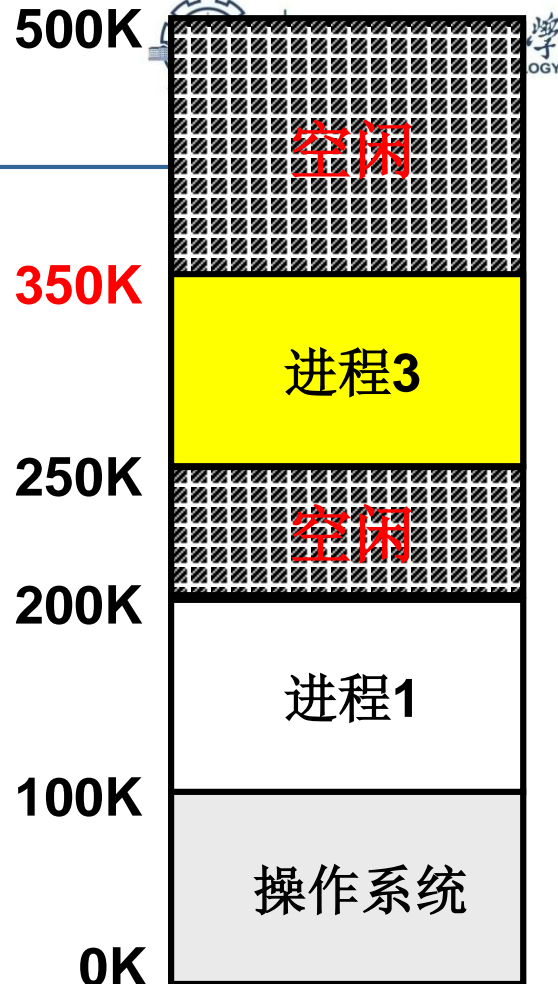
(6)外部碎片处理 – 内存紧缩

⑩ 发起请求reqSize=160K怎么办？

- 总空闲空间>160，但没有一个空闲分区>160，怎么办？
- **内存紧缩**：将空闲分区合并在一起，需要移动进程3(复制内容)
- 内存紧缩需要花费大量时间，如果一台1GB内存的计算机可以没20ns复制4个字节，它紧缩全部内存大约花费多久？？

磁盘碎片是什么？

该值表明连续分配技术不合适！



空闲分区表

始址	长度
350K	150K
200K	50K

分区方案中还有一个问题没有搞清楚

⑩ reqSize值怎么确定？

- $\text{reqSize} = \text{code} + \text{data} + \text{stack} + \text{heap}$
- **code**和**data**不难处理，**stack**和**heap**难处理：动态增长，预先不知道
- 怎么办？预留空间
- 预留空间用完了怎么办？找一个更大的空闲空间，移动该进程
- 实际上移动**code**和**data**纯粹是浪费
- 怎么办？各个段区别对待，分别分配

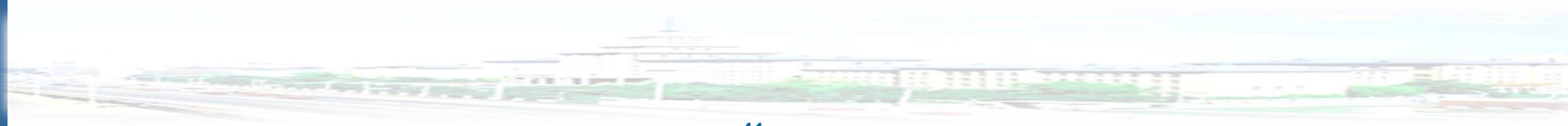
还有利于建立合适的保护策略！





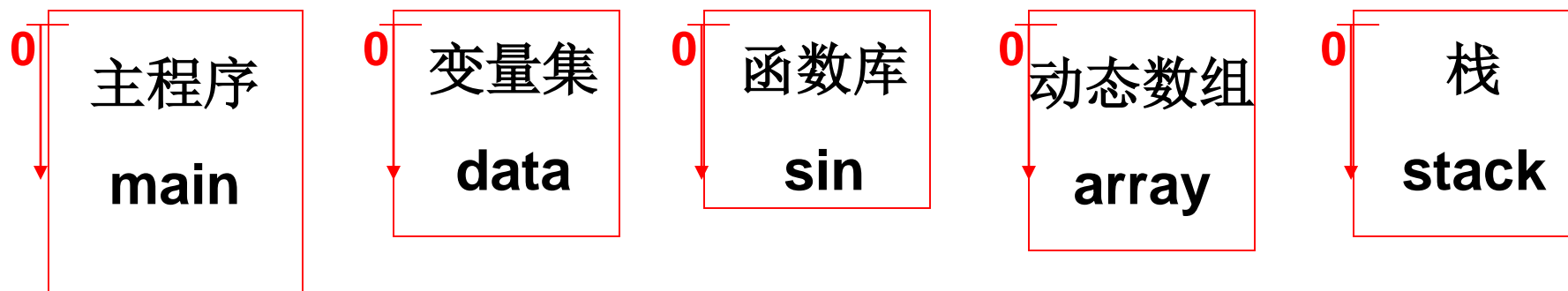
6.3 分段内存管理

分段(Segmentation)!



程序员眼中的程序(从汇编—高级语言)

⑩ 由若干部分(段)组成, 每个段有各自的特点、用途!



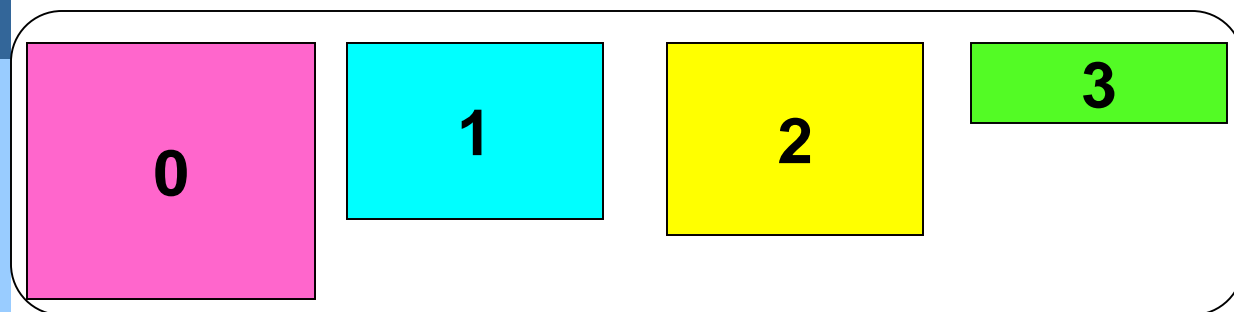
程序员眼中的一个程序

- 程序员怎么定位具体指令(数据): **<段号, 段内偏移>**
- 分段符合用户观点: 用户可独立考虑每个段(分治)

将段放入内存

引入段表

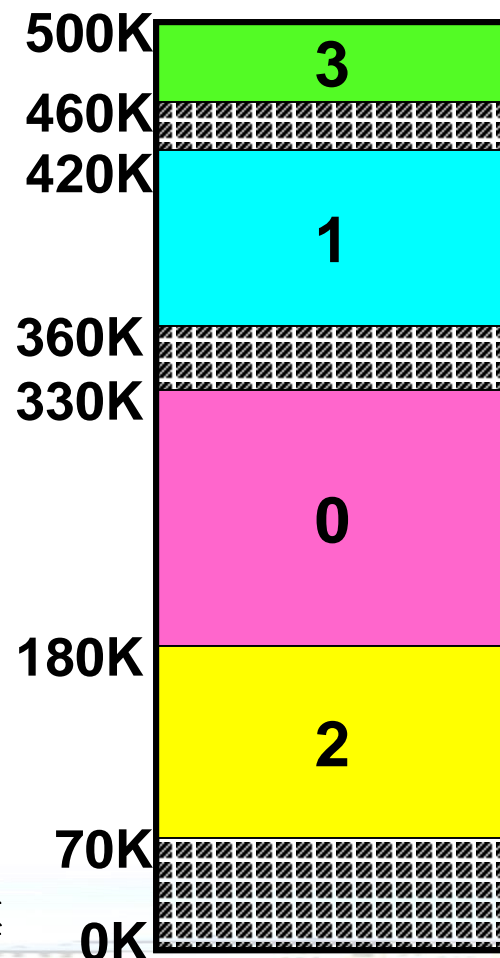
⑩ 分段制造了二维空间，而内存是一维的 各个段可分散



■ 一个进程需要记录多个基址...

进程段表

段号	基址	长度	保护
0	180K	150K	R
1	360K	60K	R/W
2	70K	110K	R/W
3	460K	40K	R



■ 仍有内存分区表，内存分配算法等等

分段的地址翻译

⑩ 来看一个例子!

Seg#	Offset
------	--------

15 14 13 0

逻辑地址格式

可以有多种格式, 如
es:bx, cs:ip等

PC = 0x240

逻辑地址

0x4240 mov ax, _var 0x240
...
_var dw 0x314159 0x4050

取出指令

段号	基址	长度	保护
0	0x4000	0x0800	R
1	0x4800	0x1400	R/W
2	0xF000	0x1000	R/W
3	0x0000	0x3000	R

段号:0, 偏移240

异常

800>240

0x4000+240

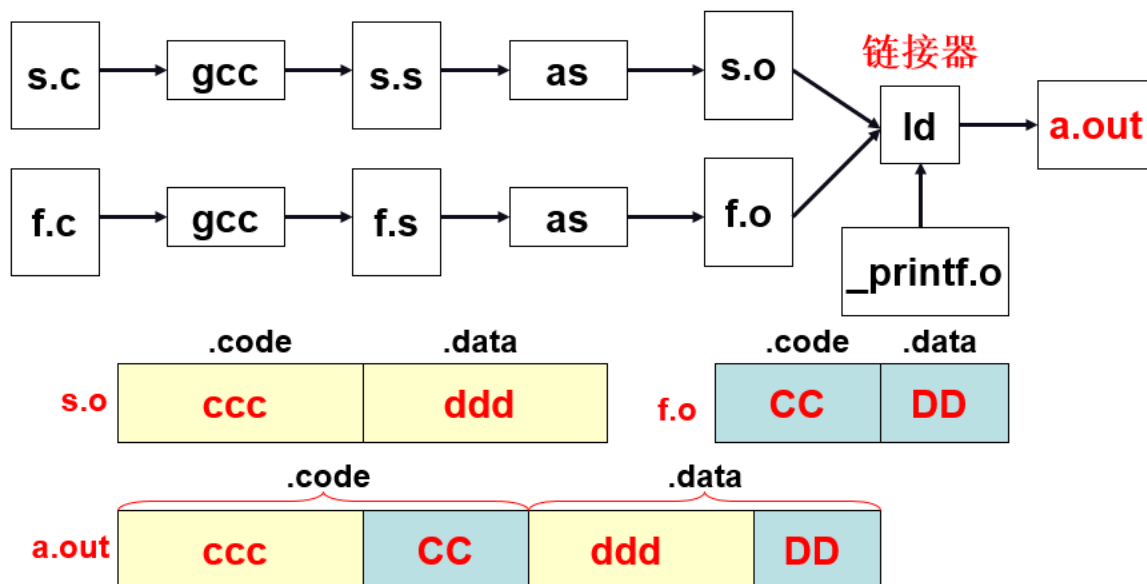
物理地址

分段技术总结

⑩ 实现原理

■ 第二步：汇编与链接——从汇编到可执行程序

- 将程序按含义分成若干部
- ld从0开始编址每个段(链)
- 创建进程(分别载入各个段)
- 内存仍用可变分区进行管
- PC及数据地址要通过段表
- 进程切换时，进程段表也跟着切换



进程、内存、编译环境、编程思想被扭结在一起了，这正是操作系统的复杂之处！



分段技术优缺点分析

⑩ 优点: 符合人的习惯, 程序员感觉舒服

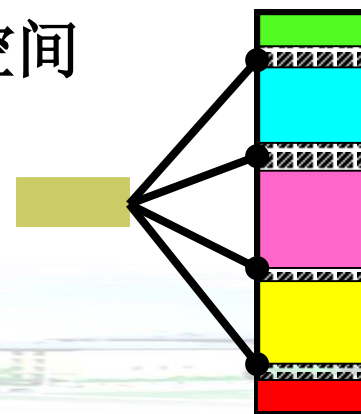
- 不同的段有不同的含义, 可区别对待
- 每个段独立编址, 编程容易(如果是一个大的一维地址空间, 程序员一会儿就糊涂了!) (分治)

⑩ 缺点: 靠近了我们, 必然会远离... 空间低效

- 空间预留; 空闲空间很大却不能分配; 内存紧缩
- 著名的碎片概念: 空闲的却用不上的空间

内部碎片

外部碎片

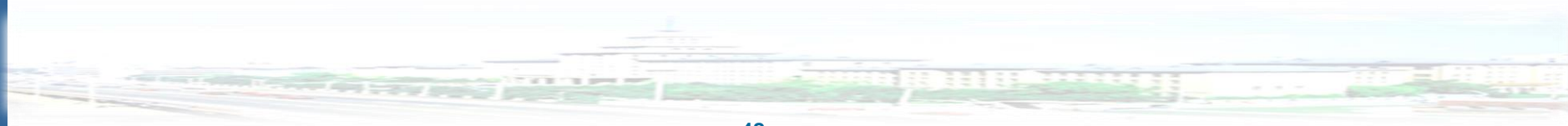




6.4 分页内存管理

分页(Paging)!

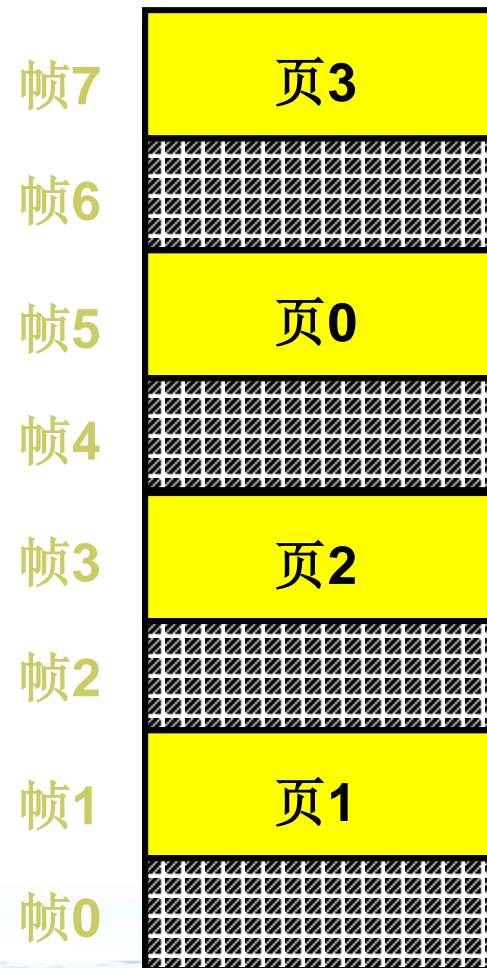
思考：按需供给内存



从连续到离散

⑩ 只有吃到最后才能知道到底有多饿!

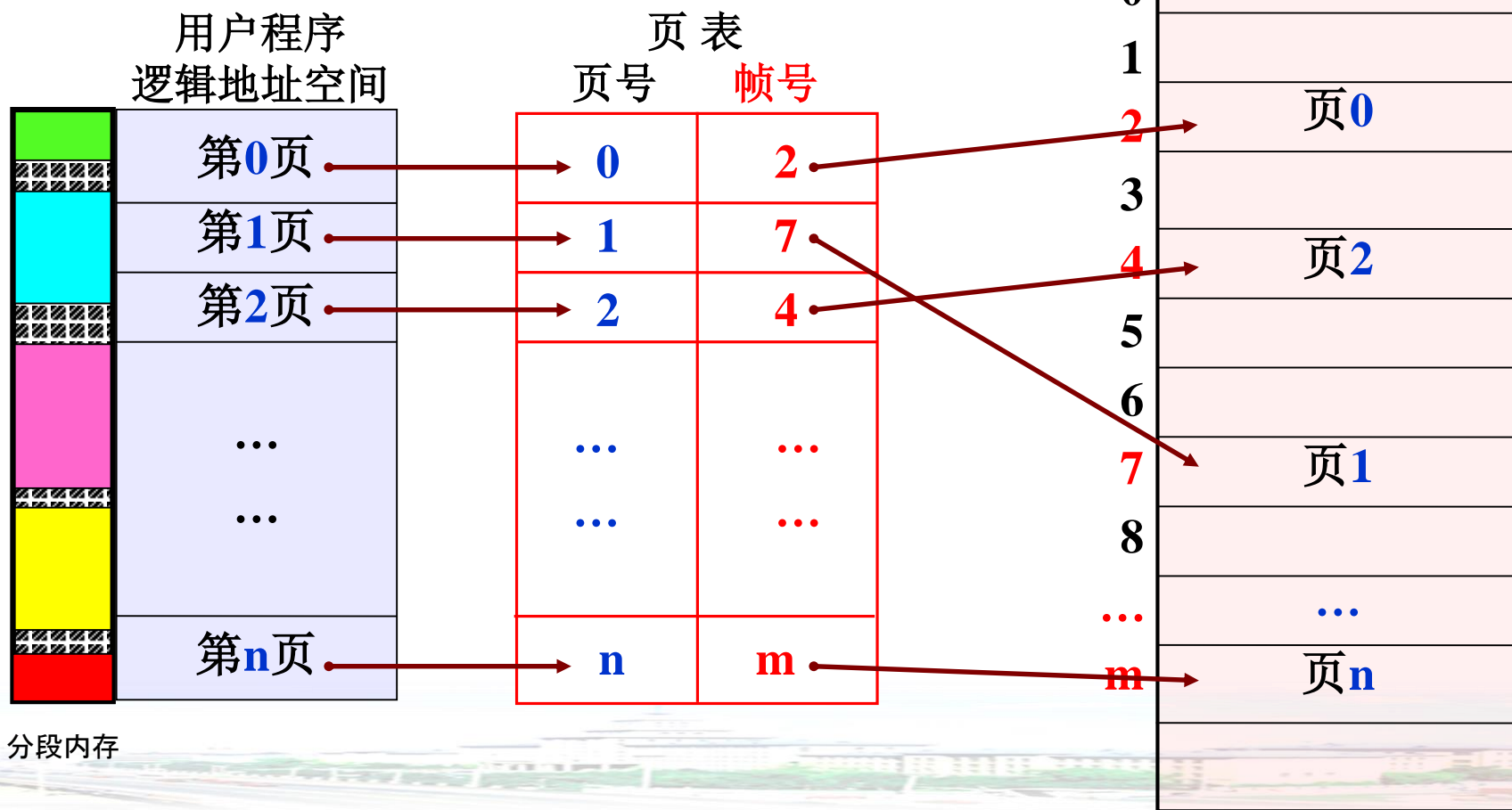
- 对于进程的堆栈段，只有运行完才知道嵌套深度
- 一次分配给一点(没有外部碎片，内部碎片有上界)
- 将面包切成薄片，将内存分成页



分页内存管理思想

如此映射优势在哪？

1) 程序逻辑地址中的页是不是一直被访问？2) 某些未被访问的页是否映射到物理内存？3) 长时间未访问的页如何处理？



分页机制中的页表

⑩ 和分段类似，分页依靠页表结构

逻辑地址格式

Page#	Offset
-------	--------

? 12 11 0

多少页

页面尺寸(4K)

进程页表

页号	页框号	保护
0	5	R
1	1	R/W
2	3	R
3	7	R

页框7

页3

页框6

页框5

页0

页框4

页框3

页2

页框2

页框1

页1

页框0

分页的地址翻译

⑩ 一个实例!

```
mov ax, _var 0x240
...
_var dw 0x314159 0x4050
```

逻辑地址格式

Page#	Offset
?	12110

页号

偏移

逻辑地址

0x00

0x240

页表指针

PCB中应有
此值

页号	页框号	保护
0	5	R
1	1	R/W
2	3	R/W
3	7	R

5	240
---	-----

物理地址: 0x5240

权限检查

访问错误

为什么?

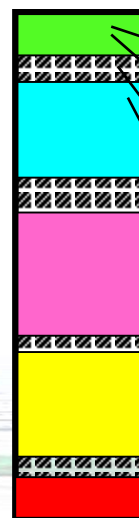
来考虑一些细节问题

⑩ 和段表不一样，页表可能会很大！

- 页是用来解决碎片问题的 \Rightarrow 页面尺寸应尽量小
- 页面尺寸通常为4K，32位机器，有 2^{20} 个页面
- 每条指令都需要查几次页表，因此查表效率很重要
- 如果页号不连续，需要查找，折半 $\log(2^{20})=20$

页号	页框号	保护
0	5	R
1	1	R/W
3	3	R

应该是连续的！



页号	页框号	保护	有效
0	5	R	1
1	1	R/W	1
2			0
3	7	R	1

可以删掉！

页表项数据结构

可以删掉!

⑩ 页基本数据结构Page Table Entry(PTE)

Intel x86结构的PTE

页号	页框号	保护	有效
0	5	R	1
1	1	R/W	1
2			0
3	7	R	1

页框号(物理页号)ppn	保留	0	L	D	A	PCD	PWT	U	W	P
31-12 ?	11-9	8	7	6	5	4	3	2	1	0

由逻辑地址va找到物理地址，只有一级页表，每页4k

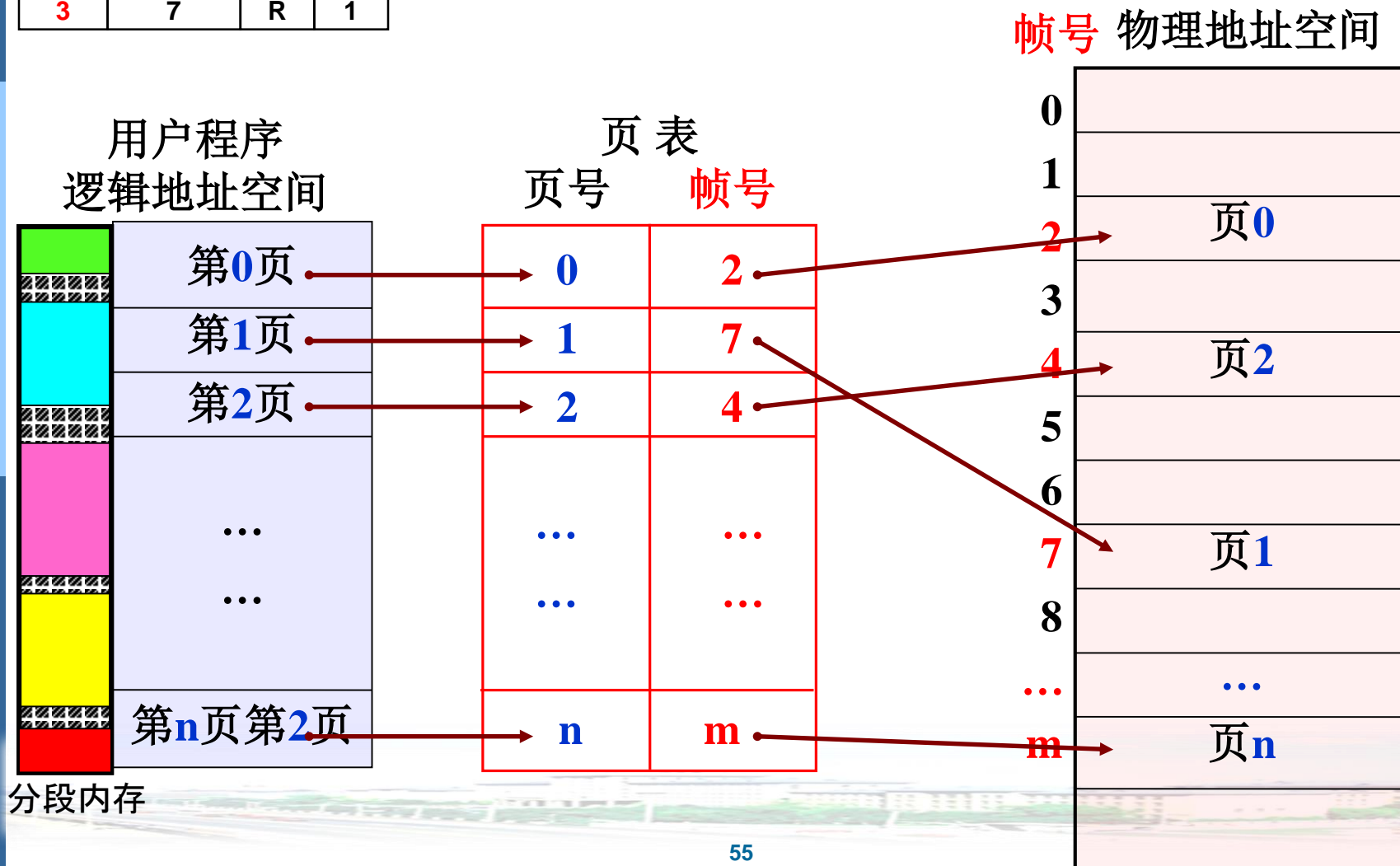
```
unsigned translate(unsigned va, int wr) {
    struct pte *pte = &page_table[va >> 12]; // 页号
    if(!pte->valid || (wr && !pte->writeable))
        throw address_fault;
    return (pte->ppn << 12) | (va & 0xfff); }
```

分页内存管理，逻辑地址空间范围

页号	页框号	保护	有效
0	5	R	1
1	1	R/W	1
2			0
3	7	R	1

逻辑地址和物理地址分开，其范围？

逻辑地址空间的页可以按需对应物理地址中的帧。由虚到实



多级页表

⑩ **32位地址空间+ 4K页面+页号连续 $\Rightarrow 2^{20}$ 个页表项**

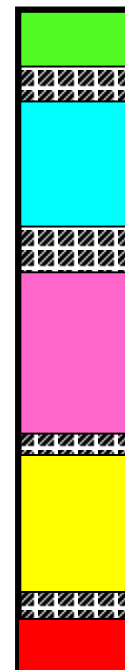
- 2^{20} 个页表项，每个4字节，都放在内存，要4M内存
- 系统中并发10个进程，需要40M内存
- 实际上大部分逻辑地址根本不会用到

为什么每个进程需要独立一个页表？

32位：总空间[0, 4G-1]!

- 引入多级页表，顶层页表常驻内存，不需要映射的逻辑地址不需要建立页表项

32位逻辑地址格式(多级页表)





多级页表

32位地址, 4G内存: 先按照4M分割, 每个4M内再按照4K分割。

32位逻辑地址格式(多级页表)

页目录号	1	页号	1	Offset	1
31	22	21	12	11	0

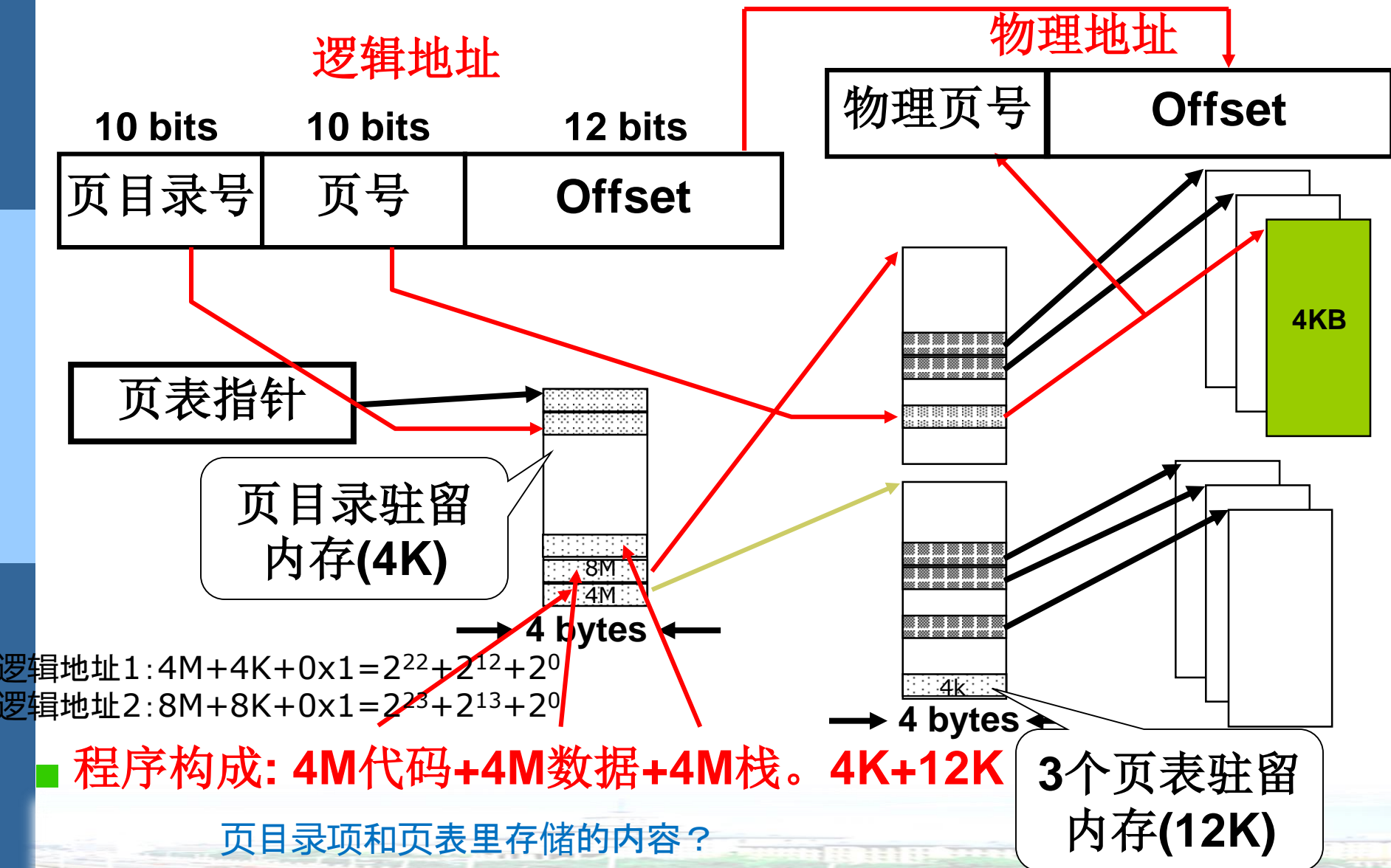
逻辑地址1:

$$4M + 4K + 0 \times 1 = 2^{22} + 2^{12} + 2^0$$

逻辑地址2:

$$8M + 8K + 0 \times 1 = 2^{23} + 2^{13} + 2^0$$

多级页表时的地址翻译



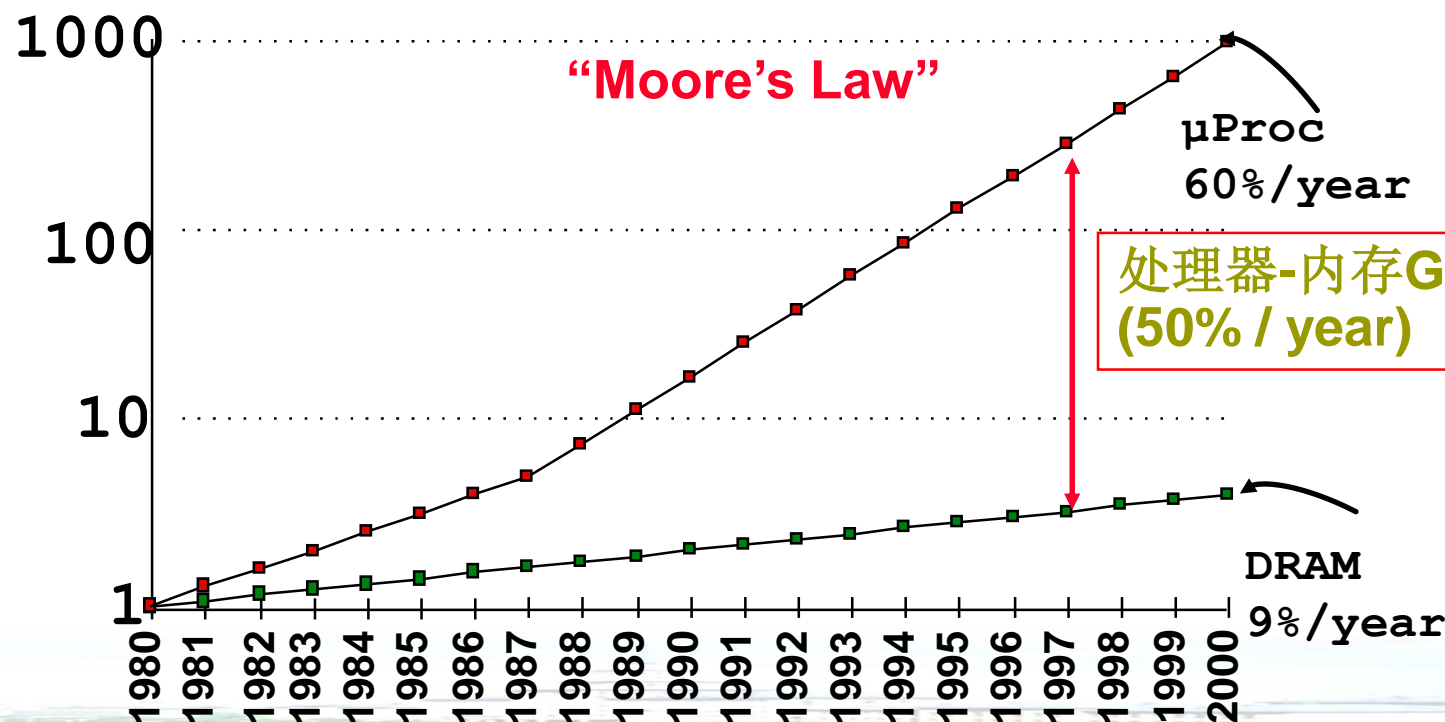
多级页表使得地址翻译效率更低

访问页表计算物理地址，
从物理地址取内容

一次
地址
访问

- 1级页表访存2次，速度下降50%
- 2级页表访存3次，速度下降到33%
- 3级页表访存4次，速度下降到25%

需要注意的事实：
内存相比CPU本
来就很慢！



提高地址翻译的效率

⑩ 多级页表的地址翻译效率很低，要提高效率

- 提高效率的基本想法：硬件支持
- 要很快：这个硬件放在哪里？寄存器
- 页表小 \Rightarrow 寄存器可行，但如果页表很大呢？
- TLB(Translation Look-aside Buffer)是一组关联快速寄存器组

有效	页号	修改	保护	页框号
1	140	0	R	56
1	20	1	R/W	23
0	19	0	R/X	29
1	21	0	R	43

采用TLB后的地址翻译

逻辑地址

页号	Offset
----	--------

有效	页号	修改	保护	页框号
1	140	0	R	56
1	20	1	R/W	23
0	19	0	R/X	29
1	21	0	R	43

关联查找-同
时进行!

TLB

TLB未命中(失效)

页表

还要查页表,
似乎更慢了!

物理地址

物理页号	Offset
------	--------

TLB命中

TLB得以发挥作用分析

- ⑩ TLB命中时效率会很高，未命中效率会降低，平均后仍表现良好。 用数字来说明：

$$\text{有效访问时间} = \text{HitR} \times (\text{TLB} + \text{MA}) + (1 - \text{HitR}) \times (\text{TLB} + 2\text{MA})$$

命中率!

内存访问时间!
假设100ns

TLB时间!
假设20ns

$$\text{有效访问时间} = 80\% \times (20\text{ns} + 100\text{ns}) + 20\% \times (20\text{ns} + 200\text{ns}) = 140\text{ns}$$

$$\text{有效访问时间} = 98\% \times (20\text{ns} + 100\text{ns}) + 2\% \times (20\text{ns} + 200\text{ns}) = 122\text{ns}$$

- TLB要想发挥作用，命中率应尽量高

平均慢了22%!

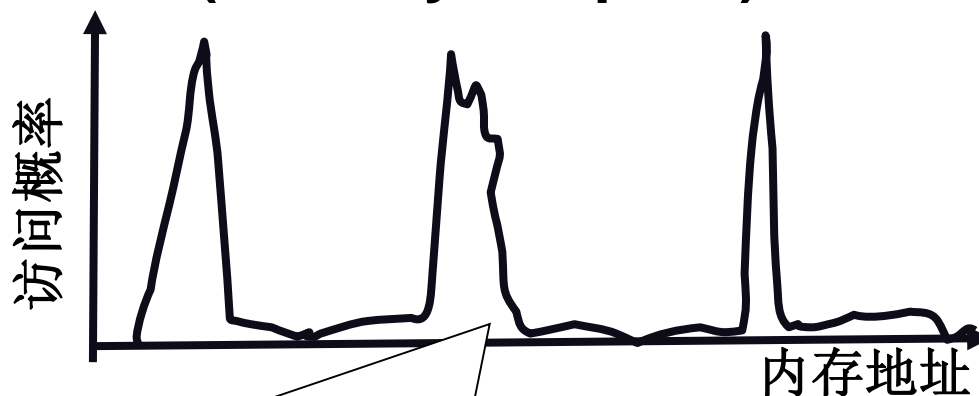
- TLB越大越好，但TLB价格昂贵，通常[64, 1024]

为什么TLB条目数在64-1024之间？

⑩ 相比 2^{20} 个页，64很小，为什么TLB就能起作用？

■ 程序的地址访问存在局部性

■ 空间局部性(Locality in Space)



程序多体现为循环、顺序结构

局部性又是计算机的一个基本特征



TLB条目少, 页表项多 \Rightarrow TLB动态变化

- 如果**TLB**未命中, 可将查到的页表项载入**TLB**
- 如果**TLB**已经满了, 需要选择一个条目来替换
- 有些时候希望某些条目固定下来(如内核代码), 某些**TLB**的设计有这样的功能, 不被选择替换
- 进程切换后, 所有的**TLB**表项都变为无效(**flush**)
- 如果进程马上又切换回来, 则这种策略就很低效。
有的**TLB**设计中条目项保存**ASID(Address-space identifier)**, 此时不需要**flush**。



分页技术总结

⑩ 实现机理

- 逻辑地址空间和内存都分割大小相等的片(页和页框)
- 每个进程用页表(多级、反向等)建立页和页框的映射
- 进程创建时申请页，可用表、位图等结构管理空闲页
- 逻辑地址通过页表算出物理地址，到达内存
- 进程切换时，页表跟着切换

分页更适合于自动化(硬件实现)!

⑩ 优点：靠近硬件，结构严格，高效使用内存

⑩ 缺点：不符合程序员思考习惯



6.5 段页结合内存管理

段、页结合!



操作系统的可执行文件格式

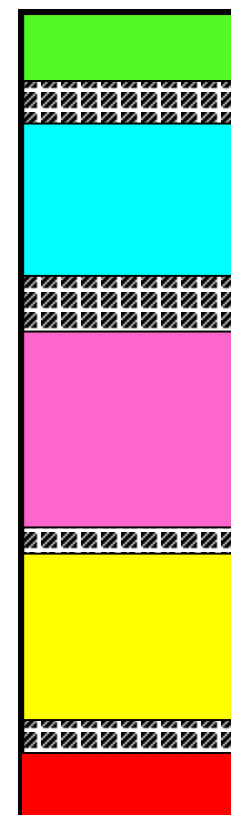
- 每种操作系统都规定了可以运行文件格式(可执行文件);
- ELF\COFF\OUT等

Linking View

ELF header
Program header table <i>optional</i>
Section 1
...
Section <i>n</i>
...
...
Section header table

Execution View

ELF header
Program header table
Segment 1
Segment 2
...
Section header table <i>optional</i>



让段面向用户、让页面向硬件



→ **段号+偏移(cs:ip)**

段号	基址	长度	保护
0	0x4000	0x0800	R
1	0x4800	0x1400	R/W
2	0xF000	0x1000	R/W
3	0x0000	0x3000	R

逻辑地址



物理地址

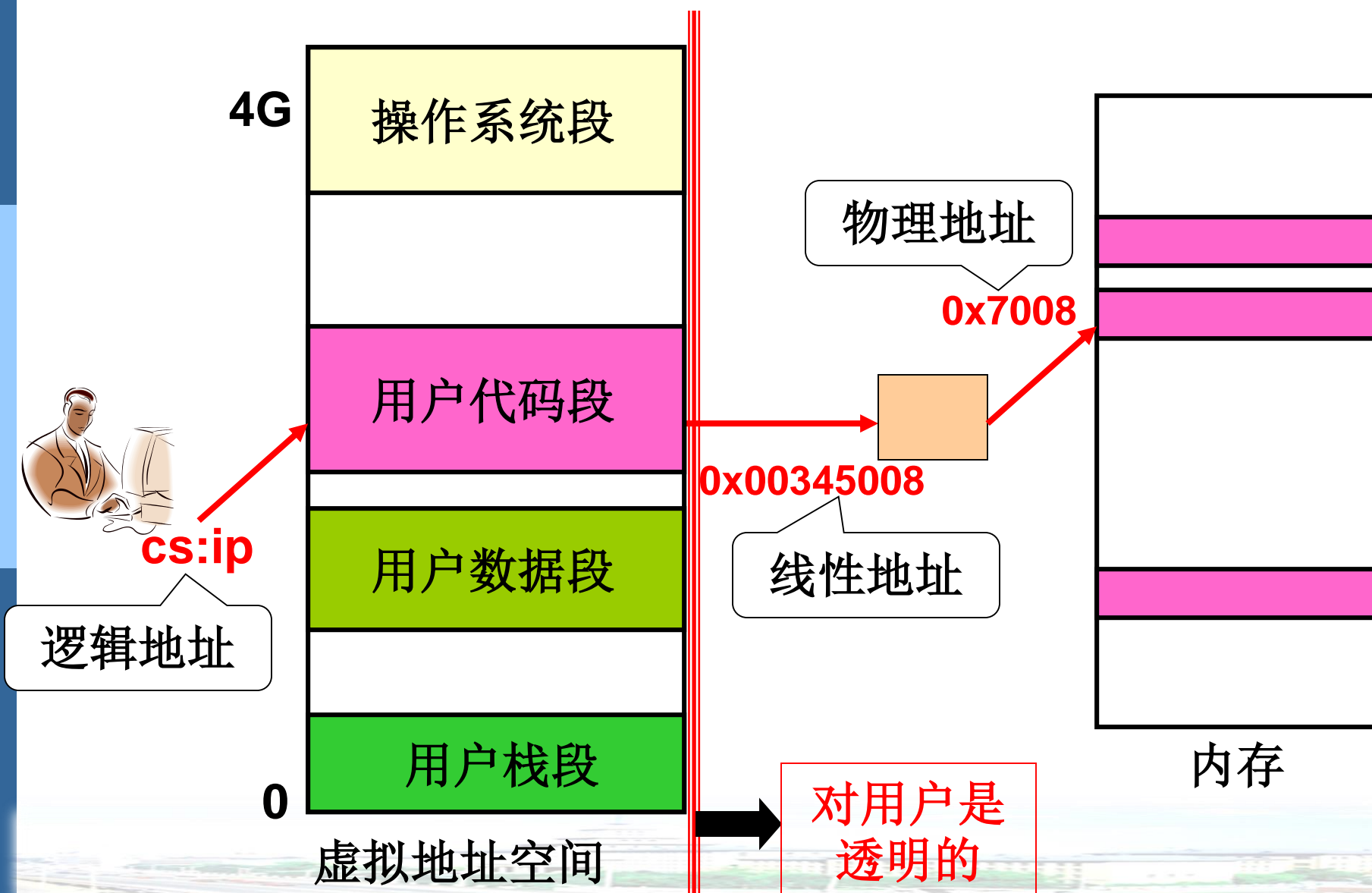
页号 **偏移**

物理页号	偏移
------	----

页框号	保护
5	R
1	R/W
3	R/W
7	R

常称为线性地址，以示区别

段页式内存管理的基本视图





段页结合技术总结

⑩ 实现机理

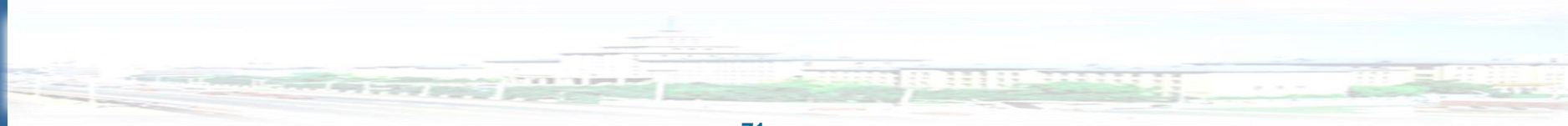
- 程序的段划分的是**线性地址空间(如0-4G)**
- 线性地址空间和内存被分割大小相等的片**(页和页框)**
- 进程用页表建立页和页框的映射
- 进程创建申请段(线性地址空间)，段申请页(物理内存)
- 逻辑地址**通过段表加页表**算出物理地址，到达内存
- 进程切换时，段表和页表都跟着切换

⑩ 优点：符合程序员习惯，并可高效利用内存

⑩ 缺点：复杂，访问一次地址需要查表好多次...



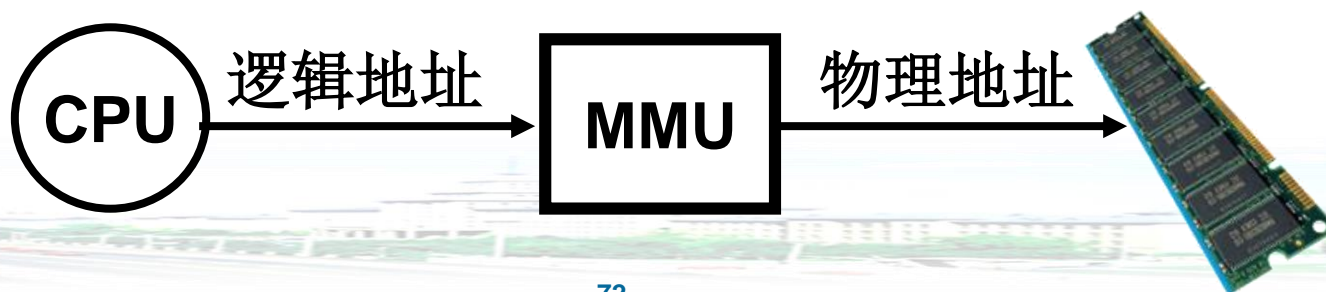
Intel x86的内存



操作系统课讲硬件，大家可能觉得奇怪

- ⑩ 段页结合时，进行一次地址翻译需要：(1)找到段表；(2)查段表；(3)找TLB；(4)找到页目录表；(5)查找页目录项；(6)找到页表；(7)查找页表；(8)形成物理地址；(9)需要段越界检查；(10)需要进行段保护权限检查；(11)需要进行页保护权限检查...

- 如此多的事情都用软件实现，其效率会很低



X86地址变换

- 保护模式开启分页机制；
- 实模式和保护模式分段管理有区别；
- 实模式**段寄存器**中存储段基址；
- 保护模式**段选择器**存储段描述符表中的索引；

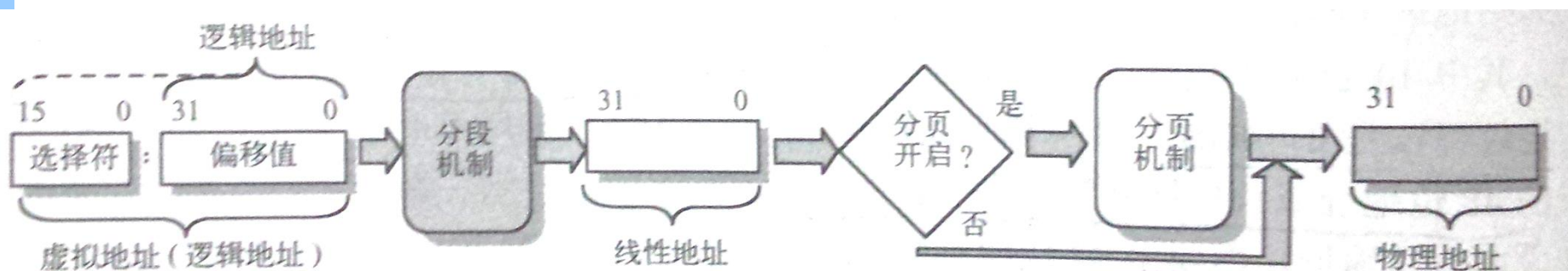


图 4-4 虚拟地址（逻辑地址）到物理地址的变换过程

嵌入式软件多数运行在没有(未开启)分页机制的处理器上
线程和任务进行直接全局数据共享有什么区别。



Intel x86的内存管理硬件!

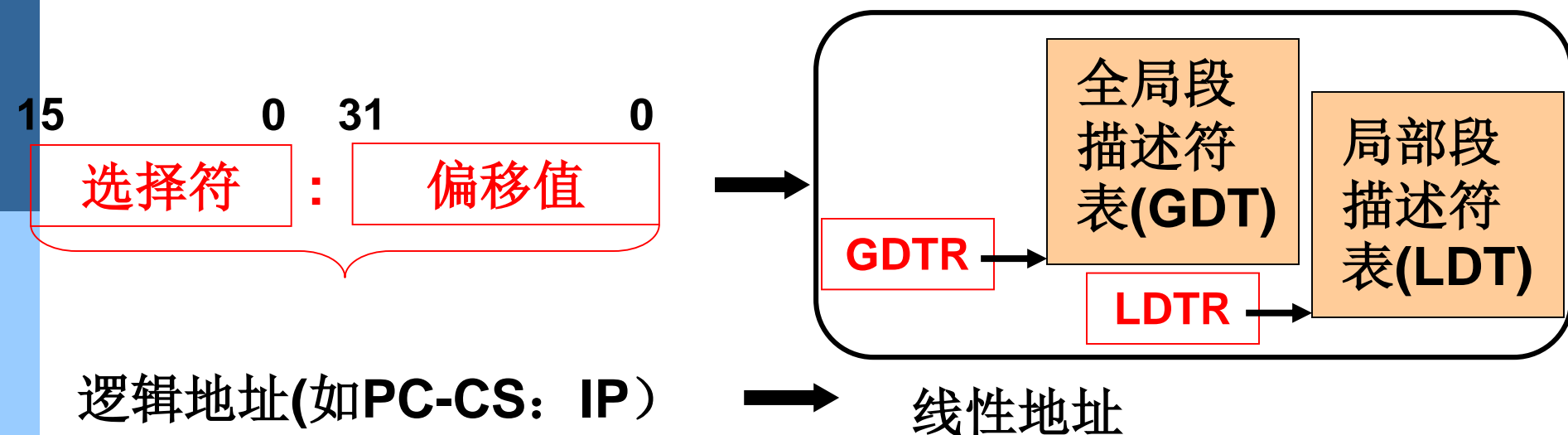
- 实模式下 : (CS,DS,ES, SS)
- `mov cx,0x2000`
- `mov ds,cx`
- `Mov [0xc0],ax`
- 基地址0x20000(左移4位, 4字节对齐)
- 实际写入物理地址0x200c0
- 逻辑地址与物理地址一致

Intel x86的分段硬件

- 选择符: **CS, SS, DS, ES, FS, GS**(16位寄存器)
- 偏移值: **EIP, ESP, ESI, EDI**(32位寄存器)
- **GDT: 全局段表**(进程共享,OS各段,进程LDT/TSS入口)
- **LDT: 局部段表**(各进程独有, 描述进程各段)
- **GDTR和LDTR: 32位段表基址(线性地址)+16位段表长度**。需用特权指令LLDT和SLDT等。
- **TSS: 任务状态段**, 保存任务各寄存器值。通过TR寄存器访问

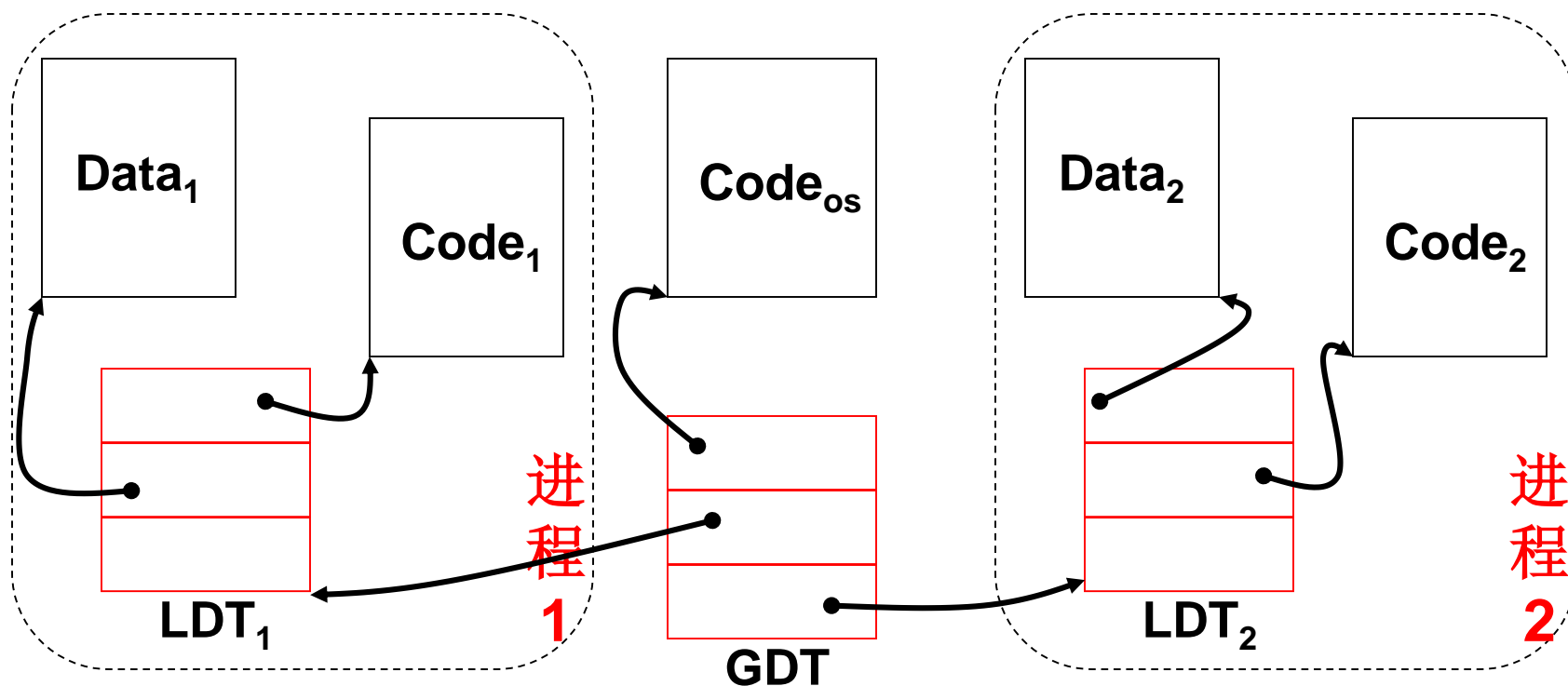
段号	基址	长度	保护
0	0x4000	0x0800	R
1	0x4800	0x1400	R/W
2	0xF000	0x1000	R/W
3	0x0000	0x3000	R

Intel x86的分段硬件



- 每个进程需要有自己的段表
- 操作系统有自己的段表, 同时还要记录访问每个进程的段表入口

Intel x86的分段硬件 – LDT、GDT



■ 段描述符: LDT(GDT)中的表项

4	段基址31..24	G	段限长19..16	P	DPL	段基址23..16
0	段基址15..0			段限长15..0		

Intel x86的分段硬件 – LDT、GDT

■ 段描述符: LDT(GDT)中的表项



十进制值	TYPE				说明
		E	W	A	
0	0	0	0	0	只读
1	0	0	0	1	只读、已访问
2	0	0	1	0	读写
3	0	0	1	1	读写、已访问
4	0	1	0	0	只读、向下扩展
5	0	1	0	1	只读、向下扩展、已访问
6	0	1	1	0	读写、向下扩展
7	0	1	1	1	读写、向下扩展、已访问

十进制值	TYPE				说明
		C	R	A	
8	1	0	0	0	只执行
9	1	0	0	1	只执行、已访问
10	1	0	1	0	执行、可读
11	1	0	1	1	执行、可读、已访问
12	1	1	0	0	只执行、一致
13	1	1	0	1	只执行、一致、已访问
14	1	1	1	0	执行、可读、一致
15	1	1	1	1	执行、可读、一致、已访问



Intel x86的分段硬件

- 全局描述符表(Global Descriptor Table)
- GDT指向LDT\TSS段描述符表入口地址
- GDT记录OS使用的内存区域的段信息
- GDT对所有进程均可用
- 整个系统只定义一个GDT, 其中存放有允许多个进程共同访问的那些段的描述符。
- GDT位置由CPU的GDTR寄存器指出。



Intel x86的分段硬件

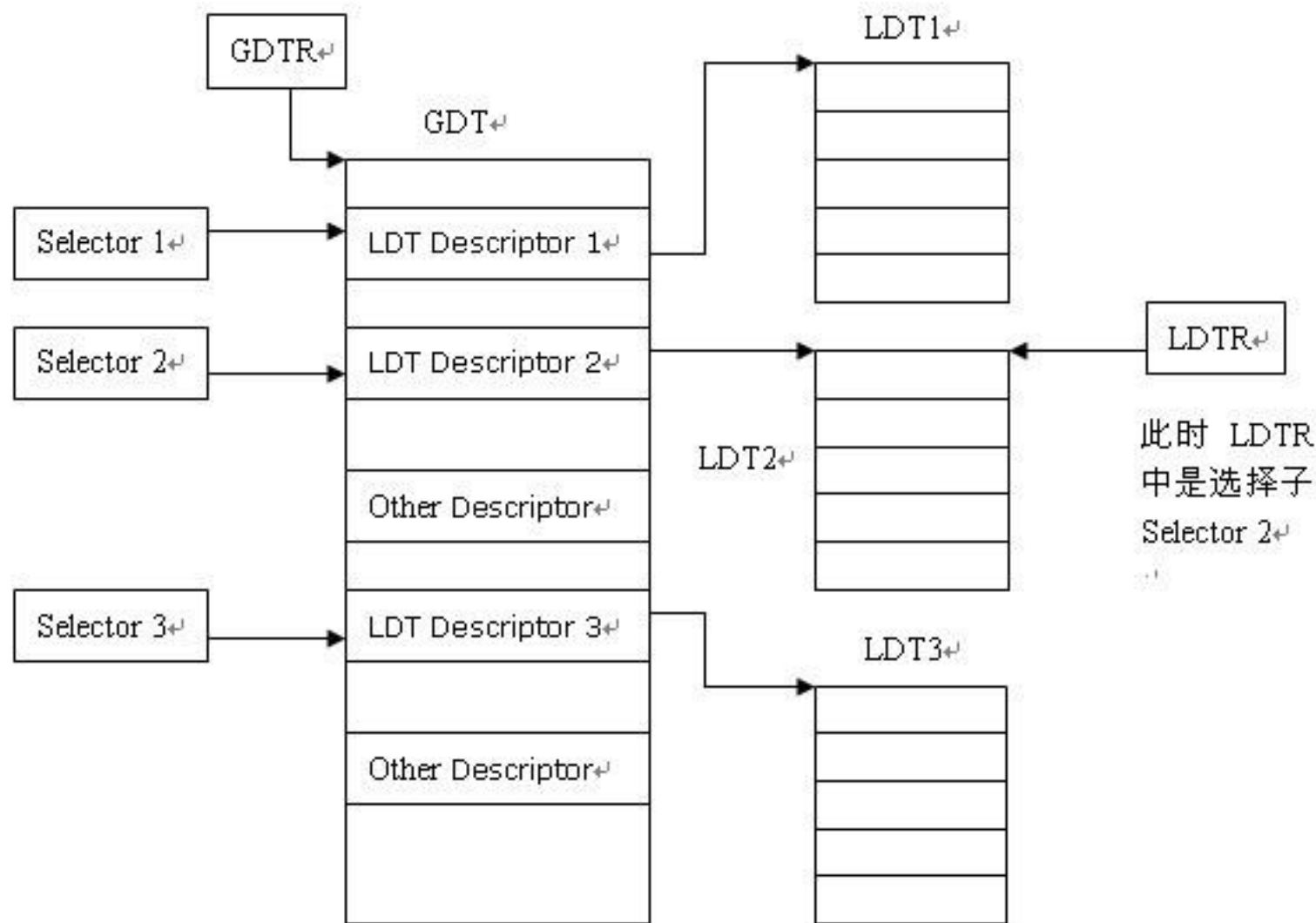
- 局部描述符表(Local Descriptor Table):
- 每个进程的局部空间的所有段描述符集中存放在一张表中, 这张表为该进程的局部描述符表(LDT)。
- 每个进程的LDT也将它看成是一个段,因此也需要一个段描述符来给出其在内存的基址、长度等信息, 称为LDT描述符, 存放在GDT中。
- 当前正在执行的进程, 其在GDT中的LDT描述符位置, 由CPU的LDTR寄存器指出。



Intel x86的分段硬件

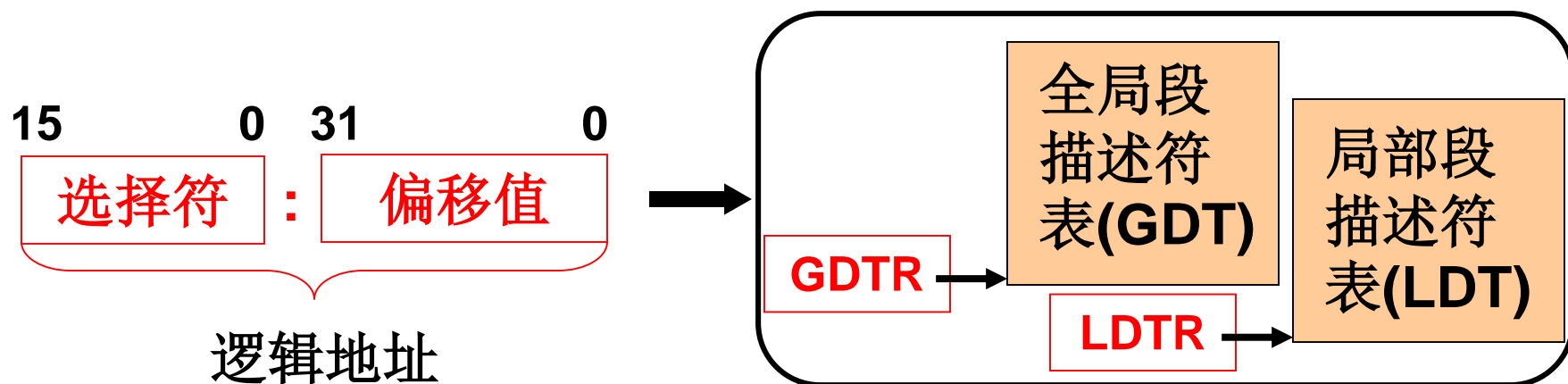
- 4.任务状态段(Task State Segment) :
- 每个任务都有一个任务状态段TSS, 描述任务状态段的描述符称为TSS描述符
- 所有任务的TSS描述符均被存放在GDT中。
- 系统可通过TR寄存器在GDT中找到正在执行的任务的TSS描述符, 从而找到相应任务的TSS。
- 任务状态段TSS中包含:
 - 1.任务的CPU现场(通用寄存器、段选择寄存器、指令指针、标志寄存器等);
 - 2.特权级分别为0、1、2时的堆栈段选择符和栈顶指针;
 - 3.该任务被调用时, 前一个任务TSS的返回连接选择符;
 - 4.I/O允许位图等。

Intel x86的分段硬件 – LDT、GDT



图五

Intel x86的分段硬件



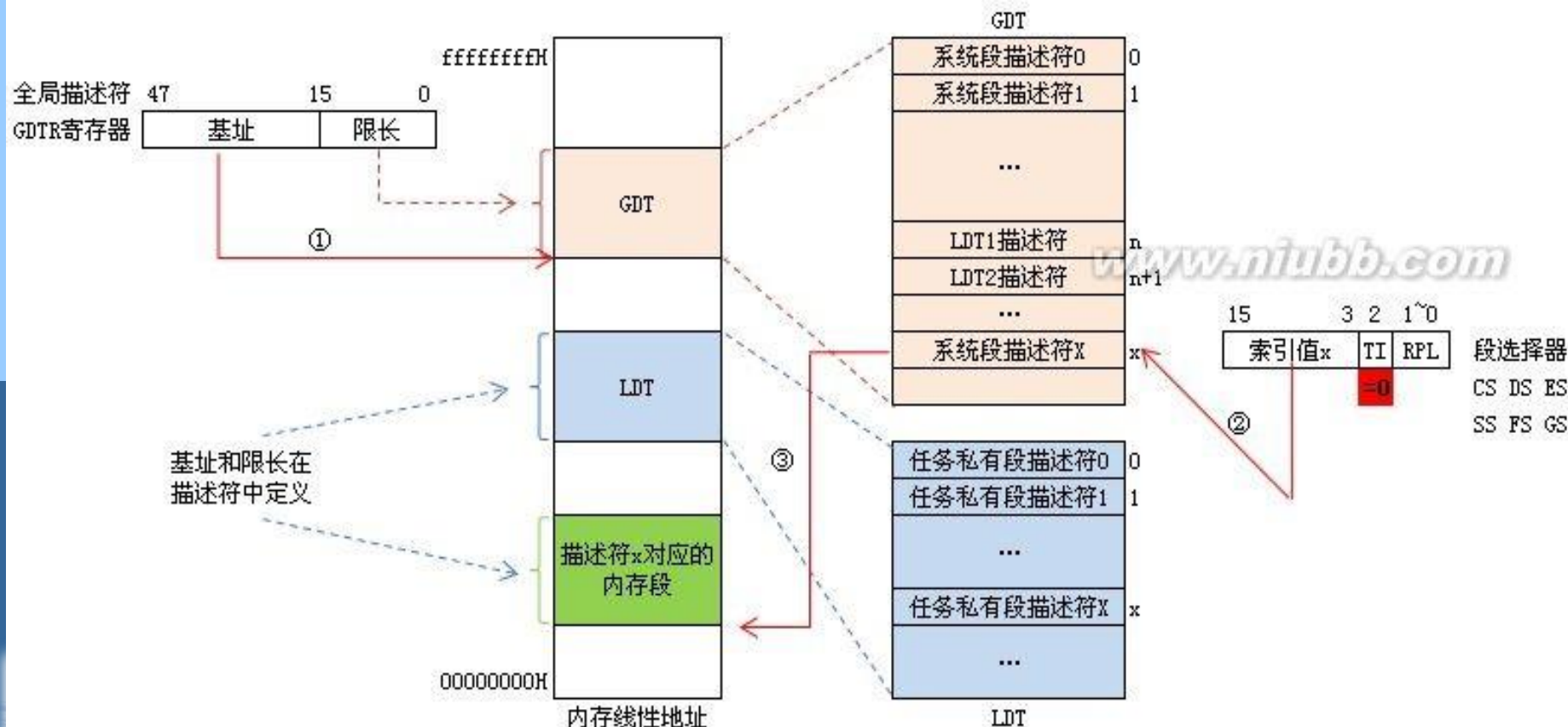
MOV、POP、LDS、LES、LSS、LGS以及LFS，这些指令显式地直接引用段寄存器。

隐式加载指令，例如使用长指针的**CALL、JMP和RET**指令、**IRET、INTn、INTO和INT3**等指令。这些指令在操作过程中会附带改变**CS**寄存器（和某些其他段寄存器）的内容。

Intel x86的分段硬件 – LDT、GDT

当TI=0时表示段描述符在GDT中

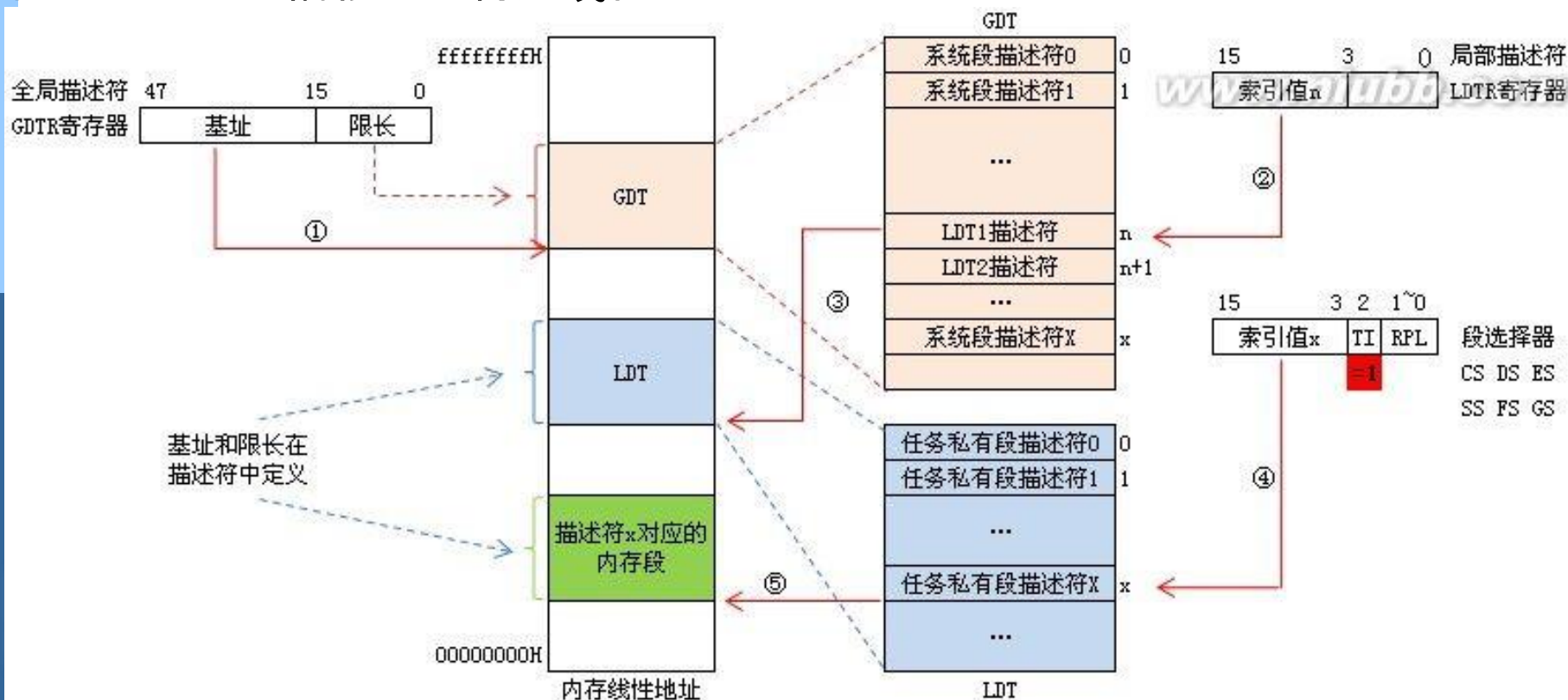
① 先从GDTR寄存器中获得GDT基址。② 然后在GDT中以段选择器高13位位置索引值得到段描述符。③ 段描述符符包含段的基址、限长、优先级等各种属性，这就得到了段的起始地址(基址)，再以基址加上偏移地址得到线性地址。



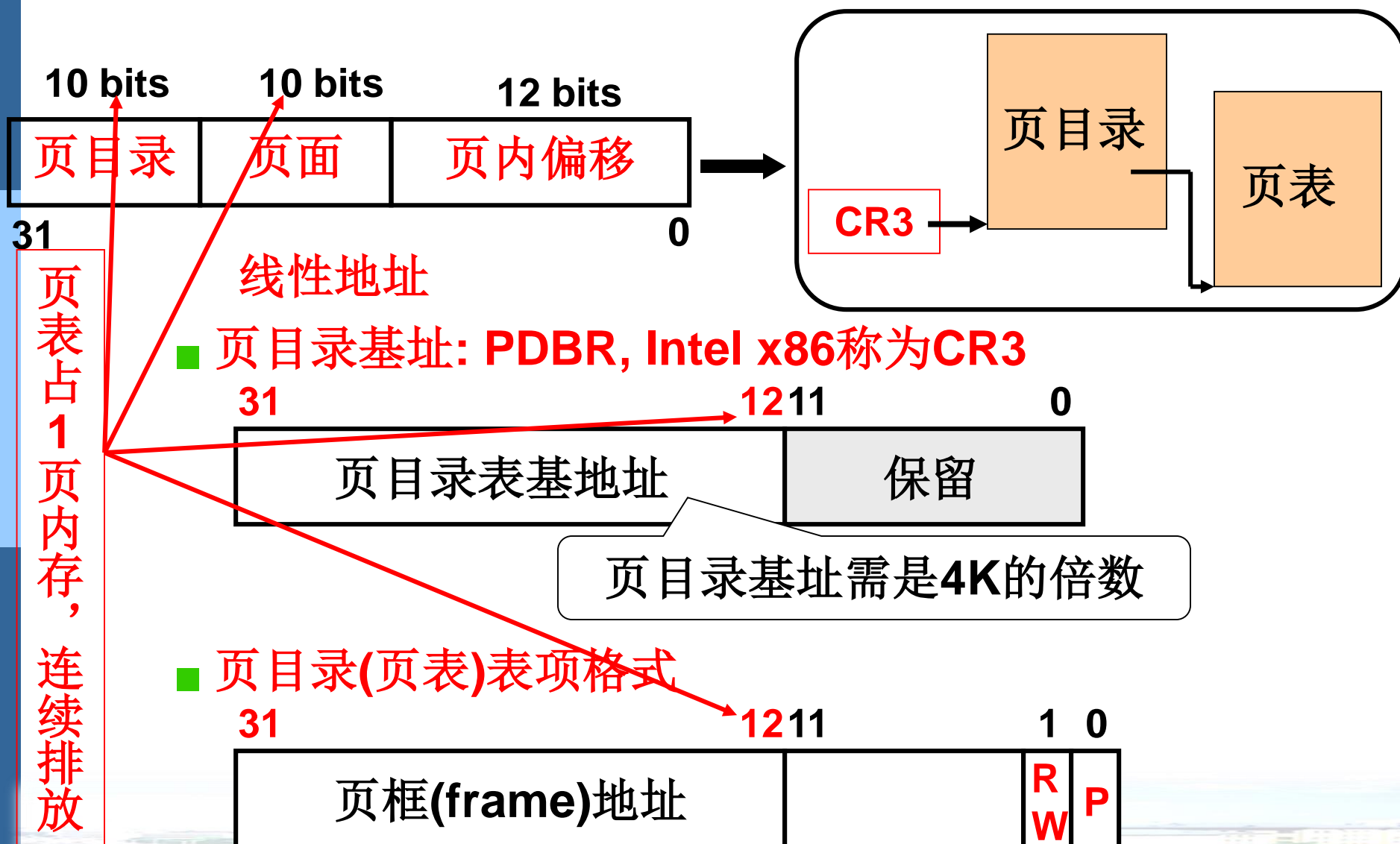
Intel x86的分段硬件 – LDT、GDT

当TI=1时表示段描述符在LDT中

① 还是先从GDTR寄存器中获得GDT基址 ② 从LDTR寄存器中获取LDT所在段的位置索引(LDTR高13位) ③ 以这个位置索引在GDT中得到LDT段描述符从而得到LDT段基址 ④ 用段选择器高13位位置索引值从LDT段中得到段描述符 ⑤ 段描述符包含段的基址、限长、优先级等各种属性, 这就得到了段的起始地址(基址), 再以基址加上偏移地址得到线性地址



Intel x86的分页硬件



Intel x86的内存管理硬件!

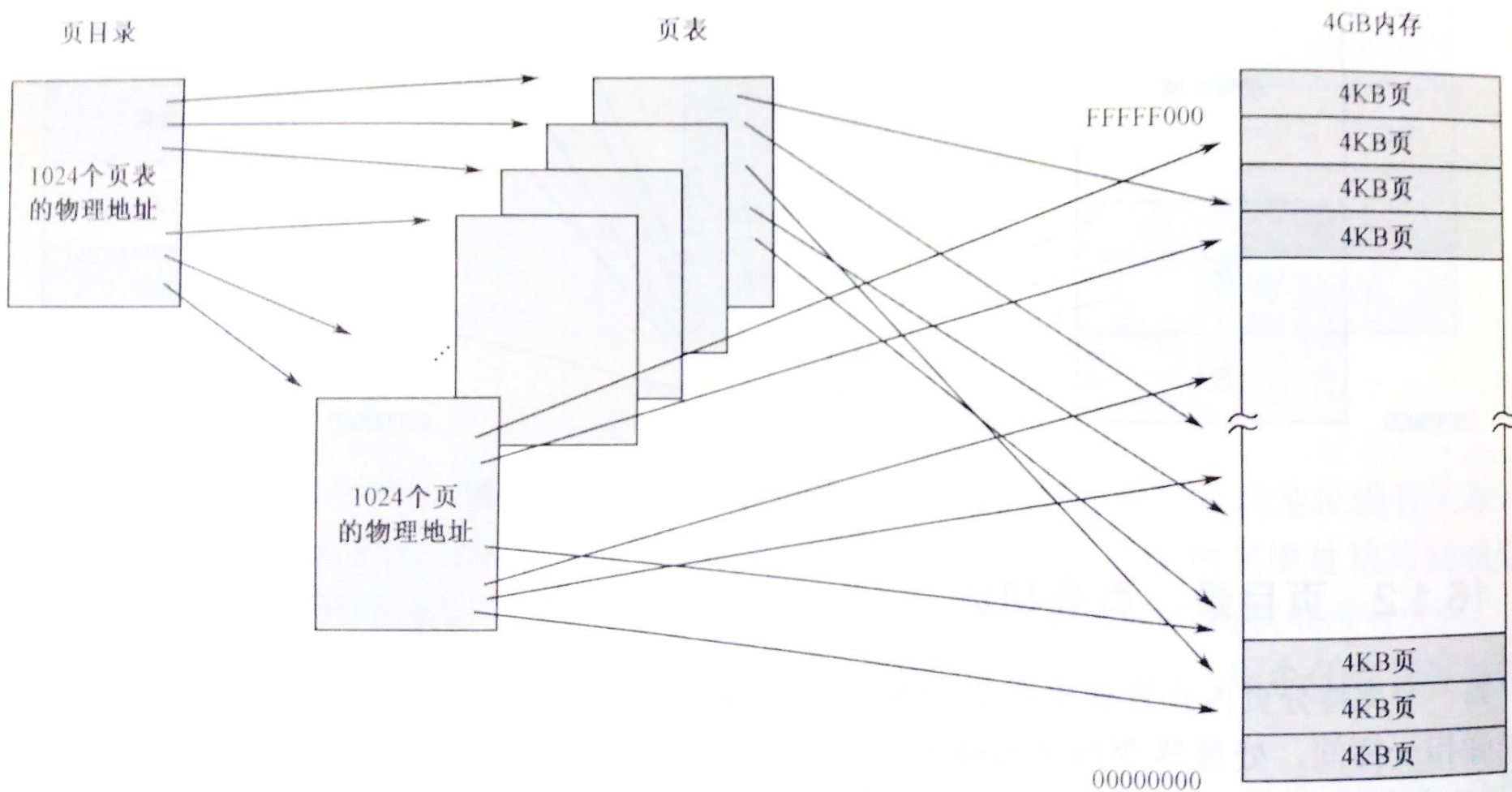


图 16-6 页目录、页表和页的对应关系



内存管理总结

- ⑩ 内存的根本目的 \Rightarrow 把程序放在内存并让其执行
- ⑩ 程序执行需要重定位 \Rightarrow 编译、载入和运行三种定位时刻
- ⑩ 运行时重定位最成熟 \Rightarrow 从逻辑地址到物理地址的翻译
- ⑩ 内存如何管理 \Rightarrow 连续内存分配(分区)最直观
- ⑩ 程序由若干段组成 \Rightarrow 以段为单位的内存分区策略 \Rightarrow 分段
- ⑩ 分段对程序员自然，但会造成内存碎片 \Rightarrow 分页 \Rightarrow 段页结合
- ⑩ 翻译、保护、内存分配是内存管理的三个核心词!

随堂测试: 1)描述一下段页结合的内存地址翻译过程



→ **段号+偏移(cs:ip)**

段号	基址	长度	保护
0	0x4000	0x0800	R
1	0x4800	0x1400	R/W
2	0xF000	0x1000	R/W
3	0x0000	0x3000	R

逻辑地址



页号 **偏移**

物理地址

物理页号	偏移
------	----

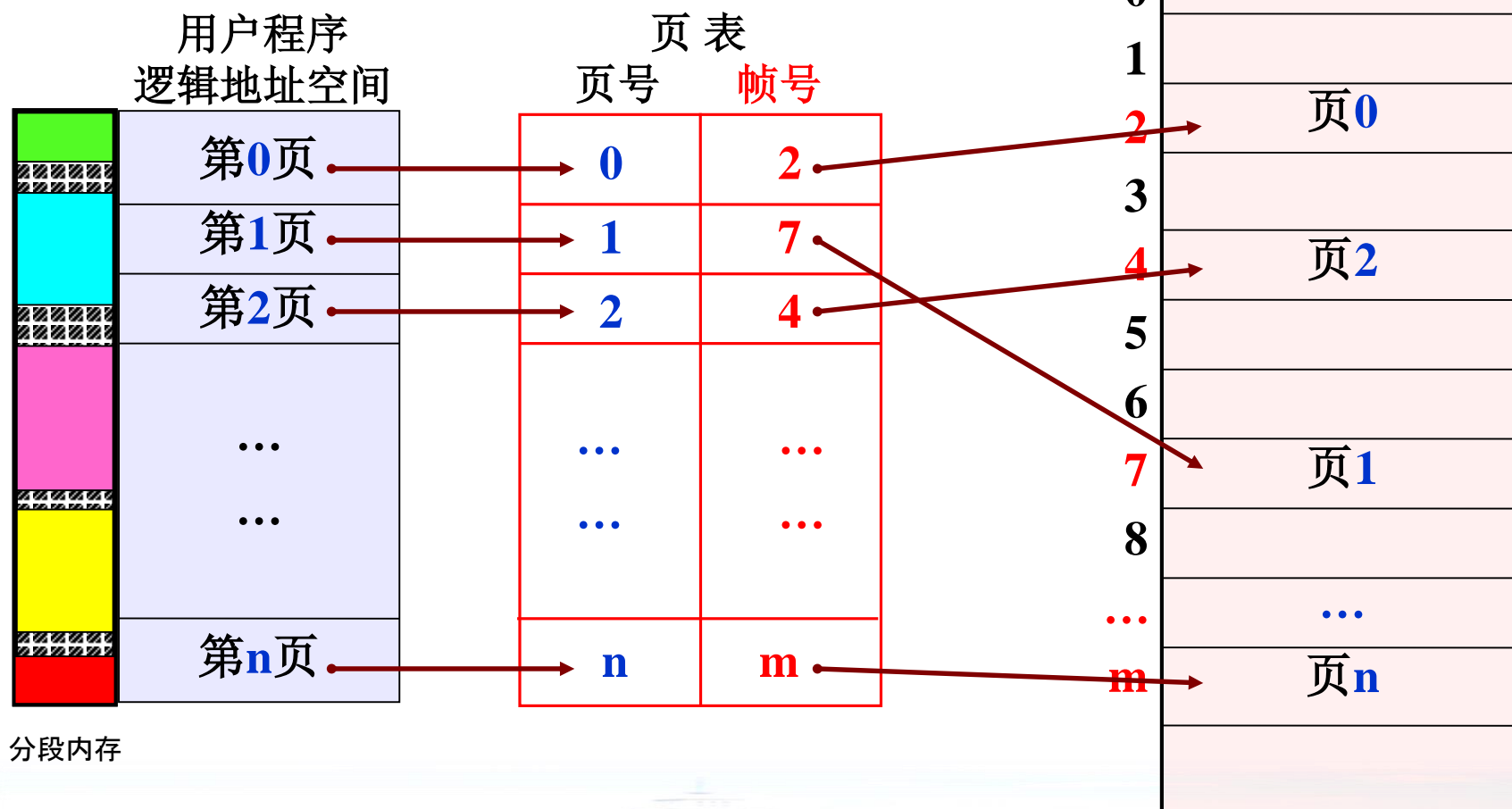
页框号	保护
5	R
1	R/W
3	R/W
7	R

常称为线性地址，以示区别



随堂测试： 2)描述一下段页结合优势

如此映射优势在哪？



段页式管理总结

