第8章 内存管理

孙承杰

E-mail： sunchengjie@hit.edu.cn
哈工大计算学部人工智能教研室
2023年秋季学期

主要内容

8.1 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

8.2 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

8.3 分段内存管理

8.4 分页内存管理

8.5 段页结合内存管理

8.1 背景

- 多个程序是如何装入内存的？
- 程序在运行中地址是如何管理的？
- 内存有限不够用怎么办？

再回到那个恒久的话题

- 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:\>_
```

让程序执行起来就成了最重要的事!

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);
}
```

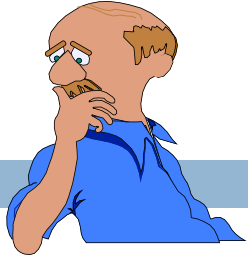
源代码

```
.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
```

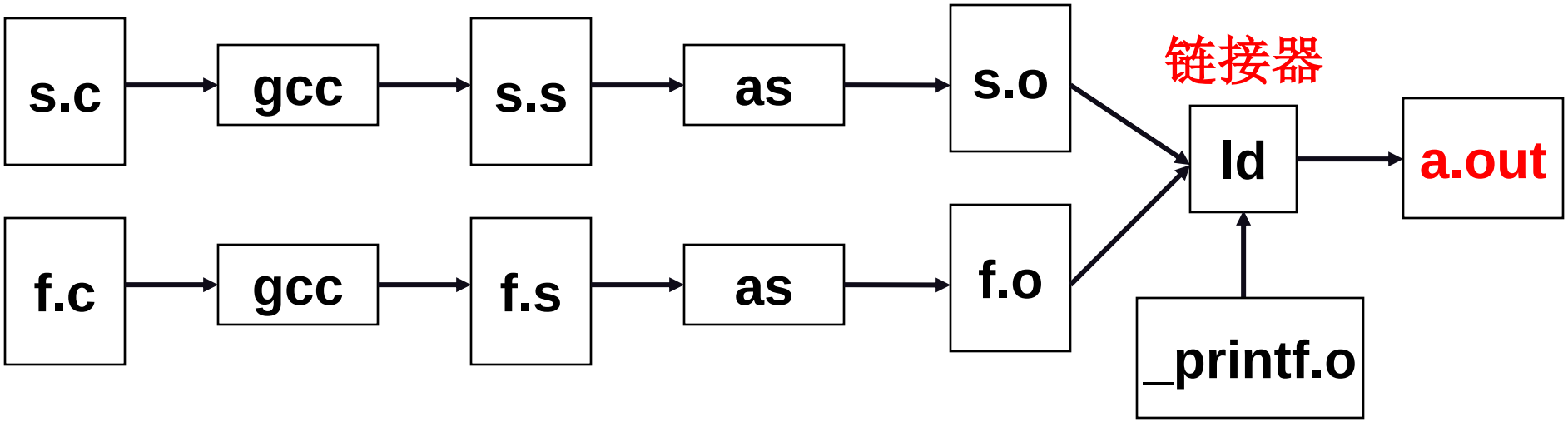
```
1:
    j> 2f, [_i], [_to]
    mov ax, [_sum]
    add ax, [_i]
    mov [_sum], ax
    add [_i], 1
    jmp   1b
2:  sub sp, 8
    mov [sp+4], [_sum]
    call _printf
    ret
.data:
     _environ:  .long 0
     _i:  .int  0
     _to:  .int  0
     _sum:  .int  0
```

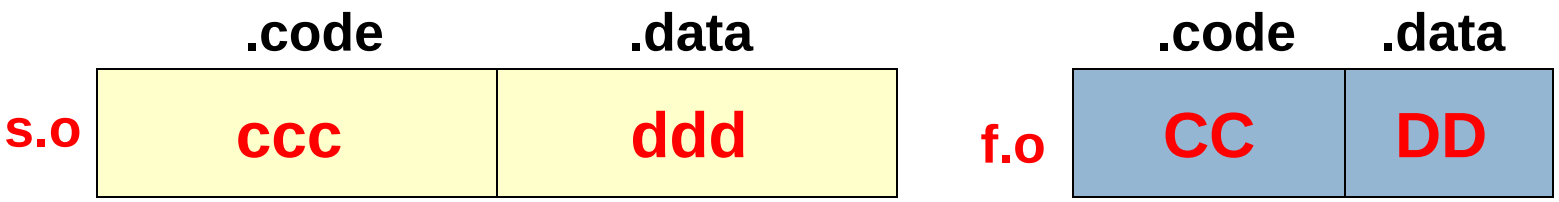
汇编代码

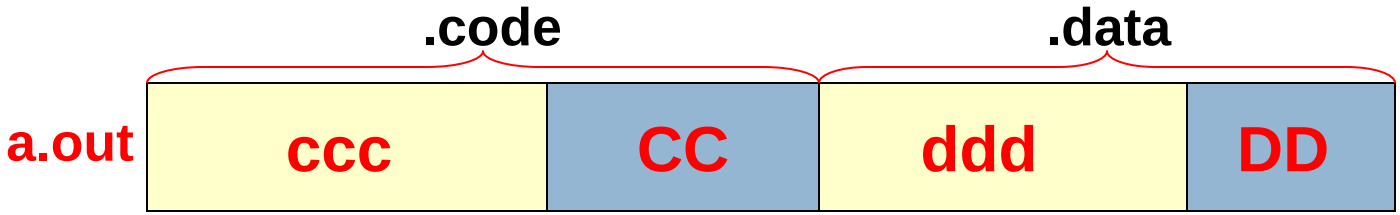
许多东西有待明确...



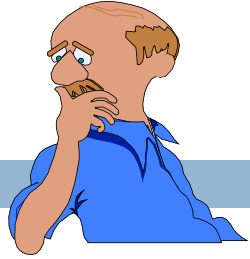
- 第二步: 汇编与链接——从汇编到可执行程序







前面的程序经过链接以后…



```
0   _entry: //入口地址
      call _main
40  _main:
      mov [_sum],  0
1:
      j> 2f, [_i], [_to]
80  2:
      call _printf
      ret
```

```
0   .data:
        …
12      _sum:  .int  0
```

```
0   _printf:
        …
```

sum.o

printf.o

ld

```
0   _entry: //入口地址
      call 40
40  _main:
      mov [300],  0
1:
      j> 80, [_i], [_to]
80  2:
      call 100
      ret

100 _printf:
        …

        …
300 _sum:  .int  0
```

可执行文件中的：
指令、数据、函数
调用都需要地址

现在还是可执行文件，不是进程

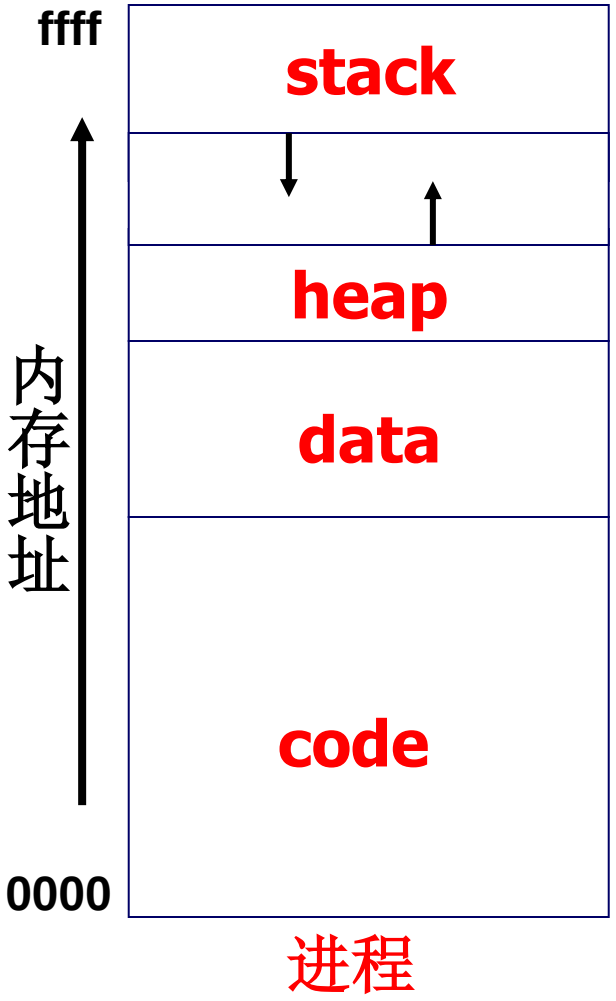
```
0   _entry: //入口地址
      call 40
40  _main:
      mov [300],  0
    1:
      j> 80, [_i], [_to]
80  2:
      call 100
      ret

100 _printf:
      …
      …
300 _sum:  .int  0
```

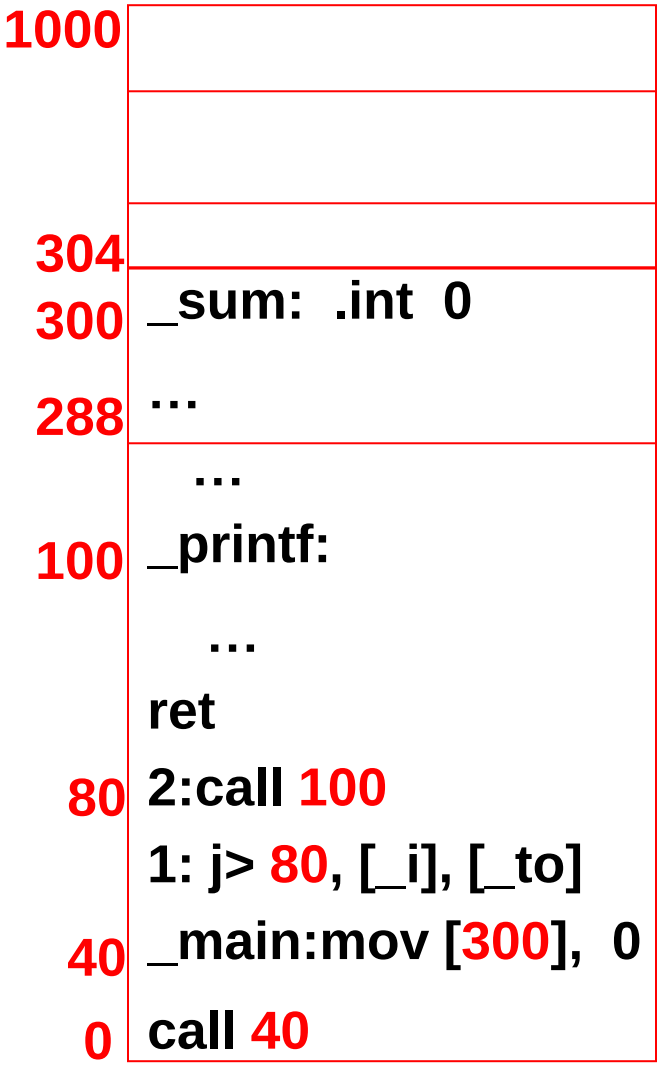
可执行文件：二进制指令和数据

loader

调用文件系统，解析可执行文件头，调用内存管理和进程管理单元



进程可以执行了吗?



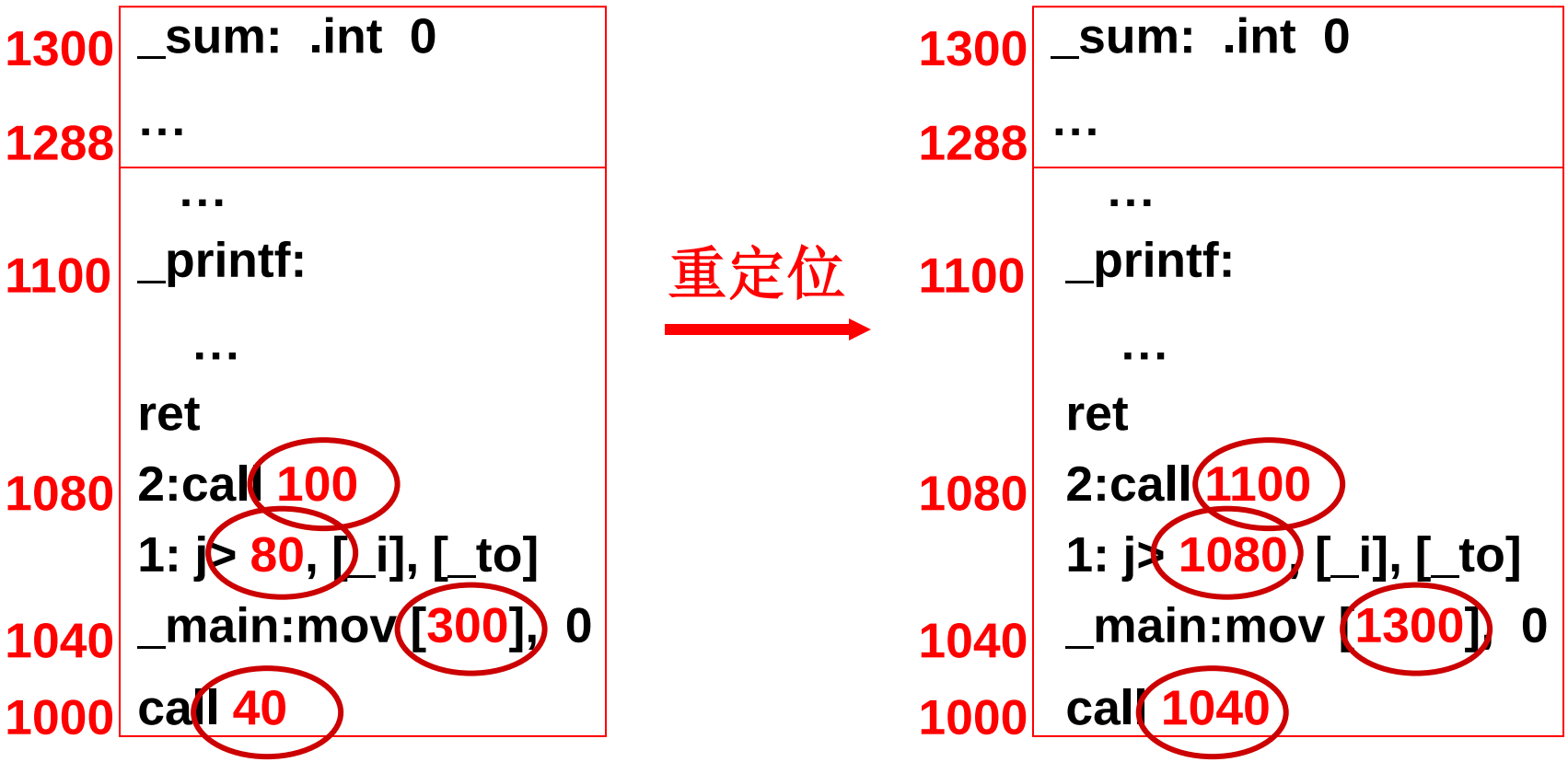
- 进程怎么能正确开始?
  - 将代码段放在内存中从0开始的地方
  - 将数据段放在内存中从288开始的地方
  - 设置PC=0
  - 如果内存中从0开始的一段内存有专门的用途怎么办?

    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!

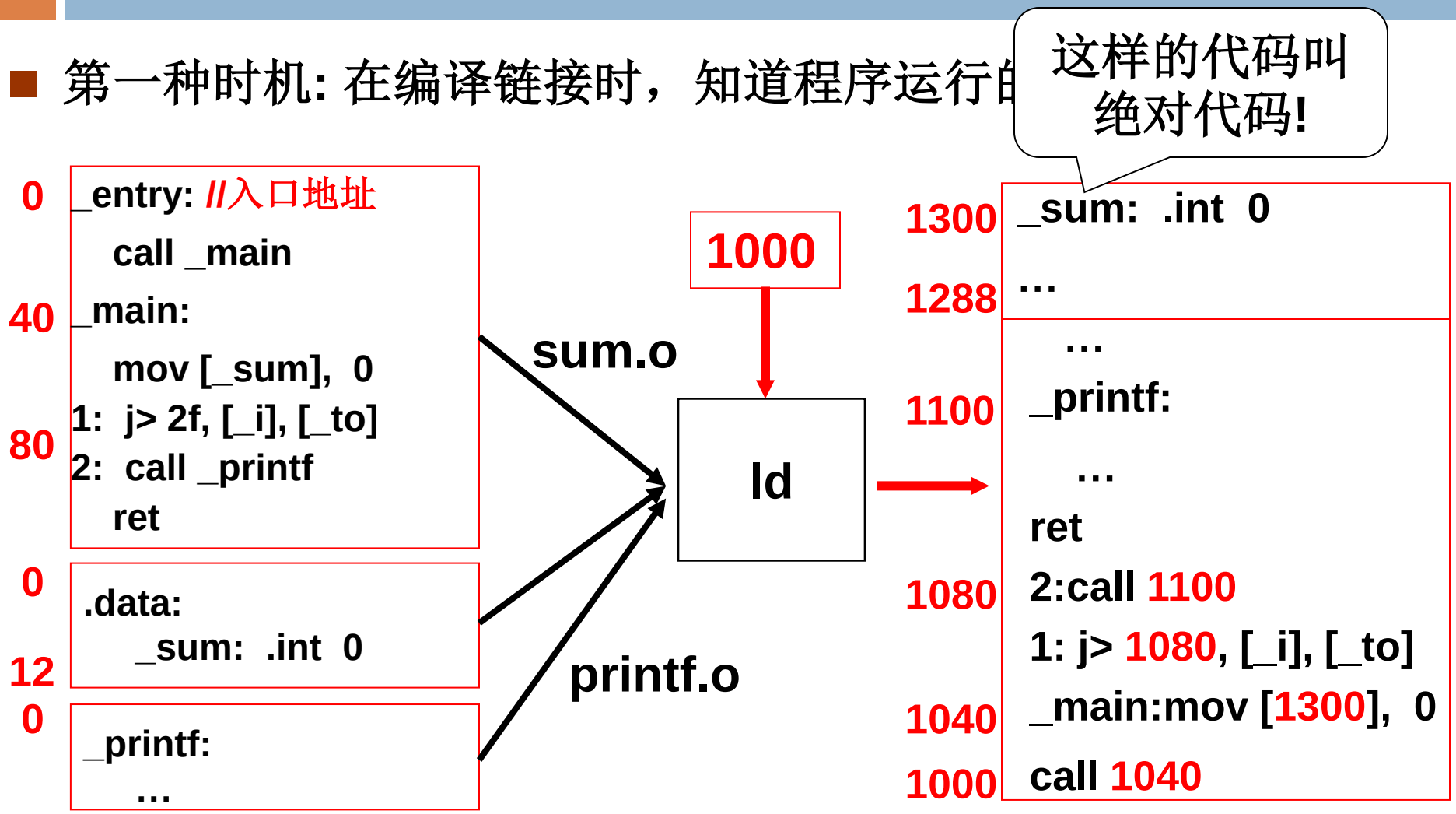
- 假设内存从地址1000以后是可以使用的



- 重定位: 为执行程序而对其中出现的地址所做的修改

重定位操作的时机

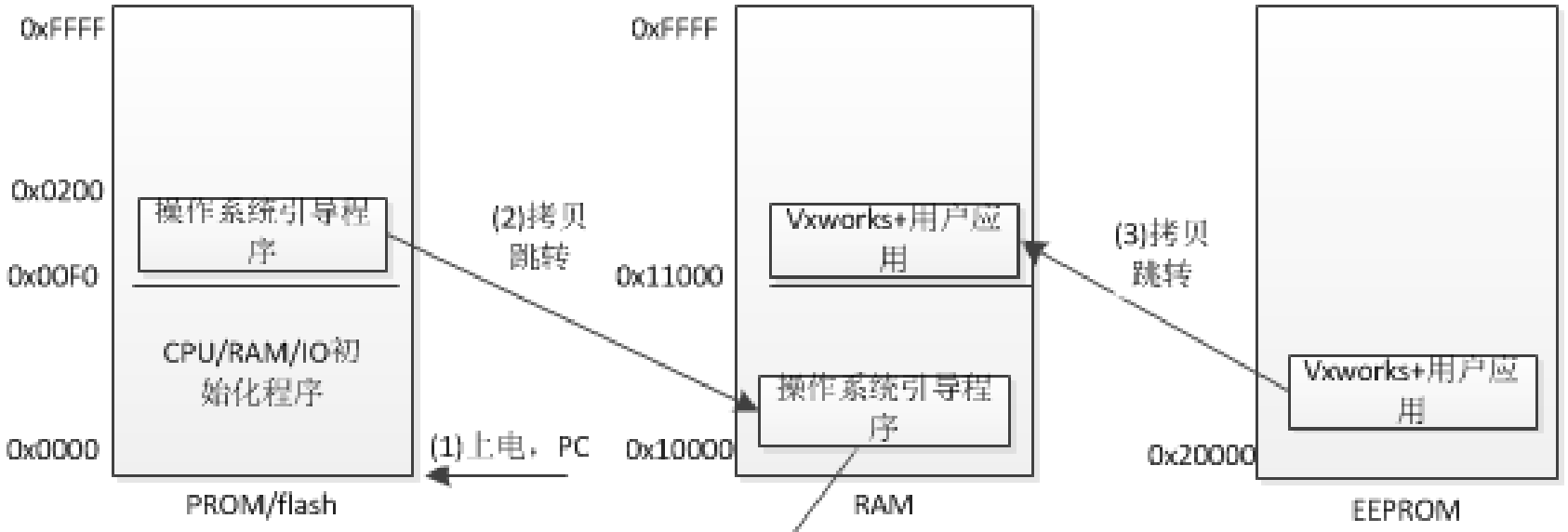
- 第一种时机: 在编译链接时，知道程序运行的



- 绝对代码: 代码只能放在事先确定的内存位置上

重定位操作的时机

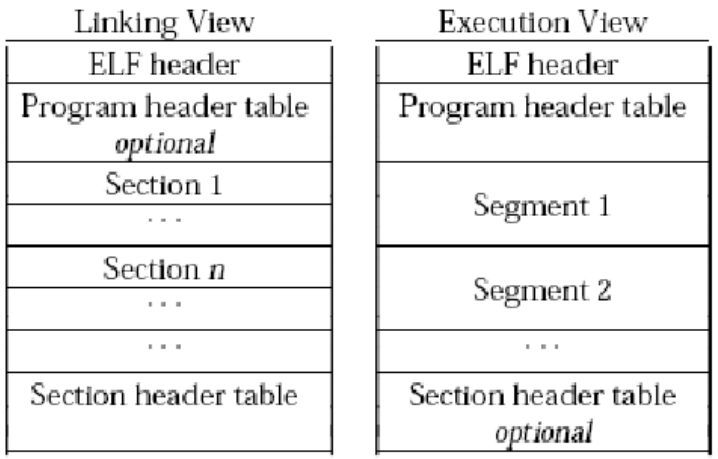
- 第一种时机：在编译链接时——嵌入式OS多数为静态链接，链接时重定位。
  - 如何控制链接器按特定地址链接？
  - 链接脚本：ld -verbose查看默认的lds




```
(Unsigned int *)  APP_Addr,RAM_Addr;
APP_Addr=(Unsigned int *)0x20000;
RAM_Addr=(Unsigned int *)0x11000;
FOR(...)
{    *RAM_Addr++= *APP_Addr++ ;  …}
```

重定位操作的时机

- 第一种时机: 在编译链接时——嵌入式OS多数为静态链接，链接时重定位
- 编译器要了解体系结构、遵从可执行文件格式




```
① GNU ld version 2.19.51.0.14-34.fc12 20090722
②       Supported emulations:
③       elf_i386
④       i386linux
⑤       elf_x86_64
⑥       using internal linker script:
⑦       ==================================================
⑧       /* Script for -z combreloc: combine and sort reloc sections */
⑨       OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
⑩       OUTPUT_ARCH(i386)
⑪       ENTRY(_start)
⑫       SEARCH_DIR("/usr/i686-redhat-linux/lib"); SEARCH_DIR("/usr/local/lib");
SEARCH_DIR("/lib"); SEARCH_DIR("/usr/lib");
⑬       SECTIONS
⑭       { /* Read-only sections, merged into text segment: */
⑮       PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x08048000)); . =
SEGMENT_START("text-segment", 0x08048000) + SIZEOF_HEADERS;
```

重定位操作的时机

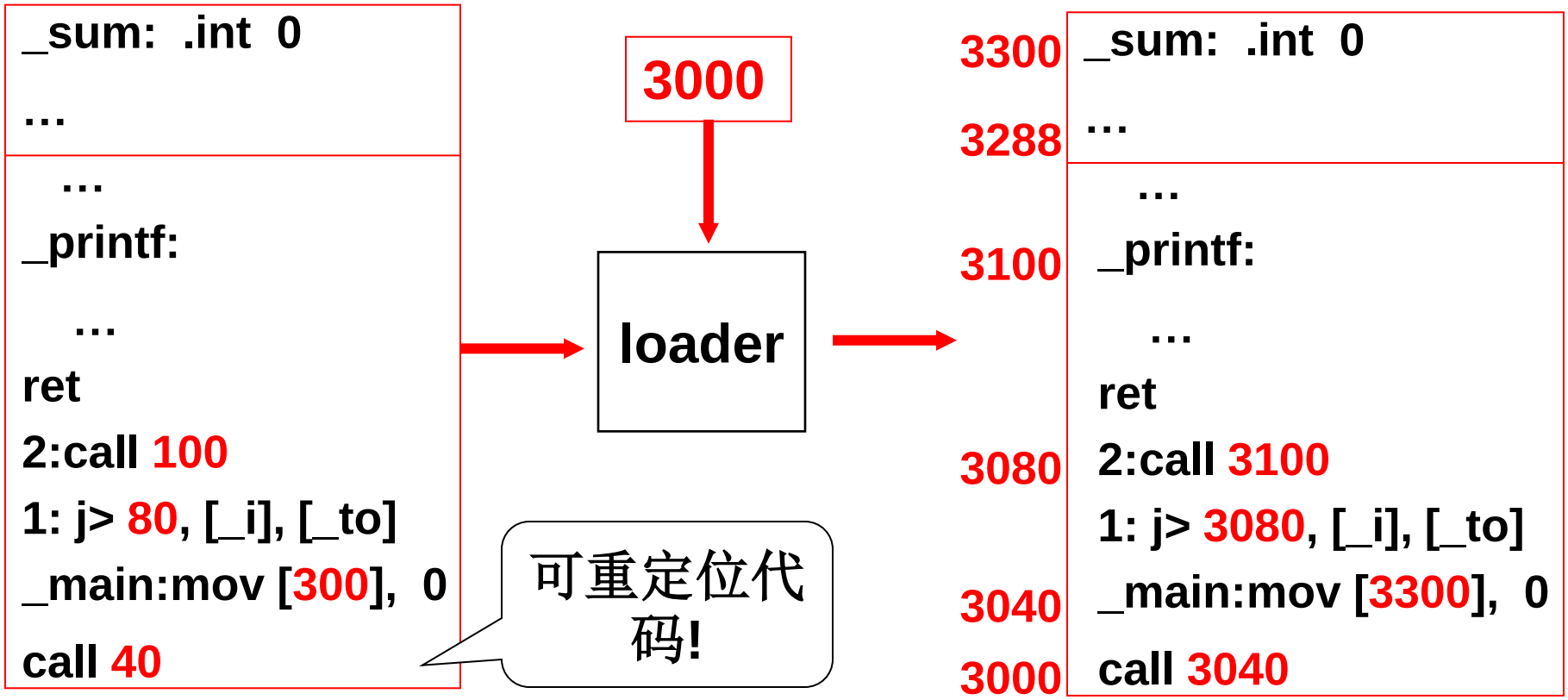
- 如何控制链接器按特定地址链接?
- 链接脚本：ld -verbose查看默认的lds
- 可以手动编写链接脚本

```
.text 0x300000 :
{
 *(.text)
 *(.rodata)
}
   .data 0x33ffff :
{
 *(.data)
}
```

- 如果只有一个进程，链接重定位没问题，多进程?

并发 ⇒ 多个程序同时在内存中

- 分别用1000，2000，… 吗？ 第二种时机: 载入时

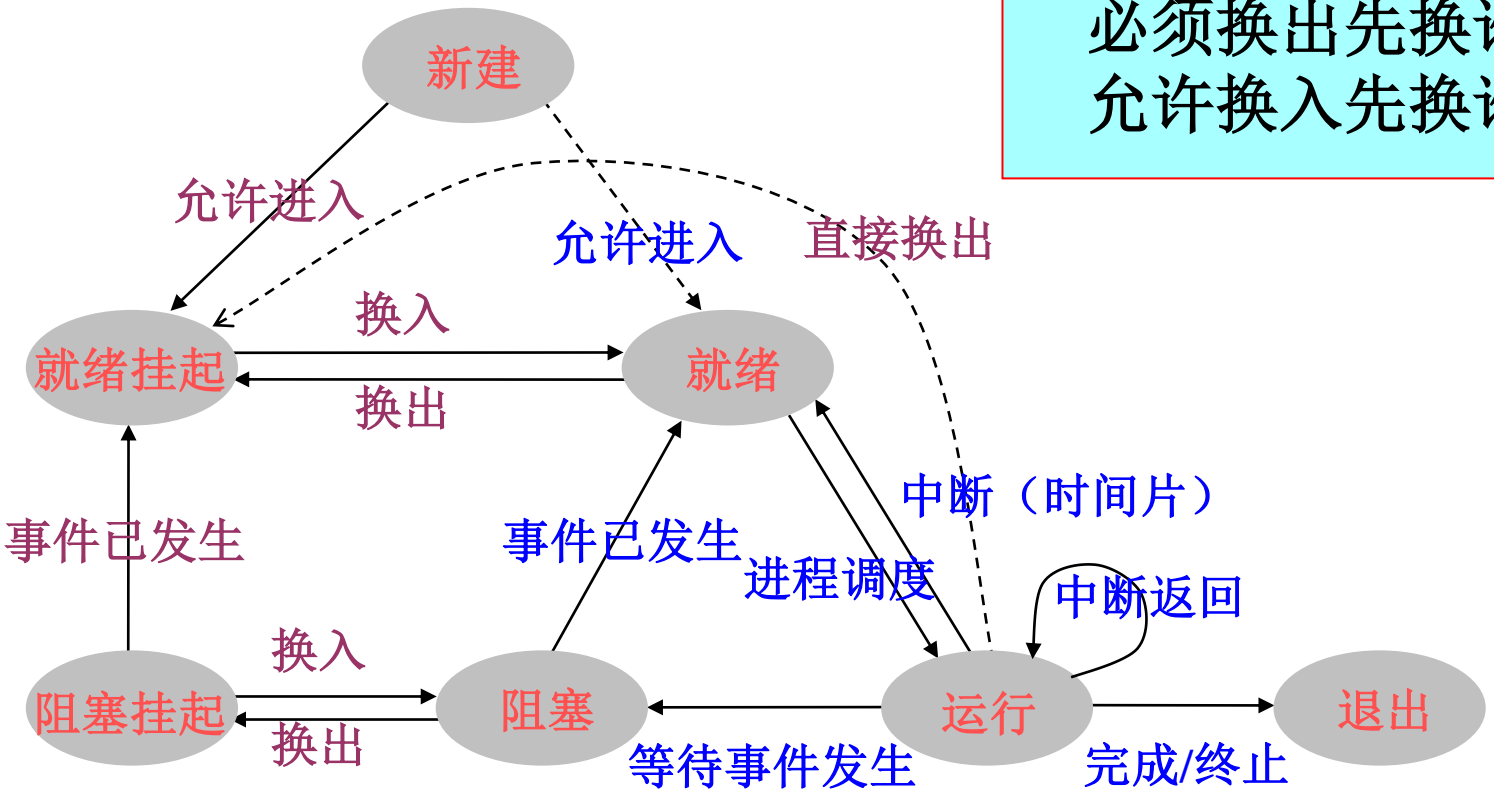


- 装载时的重定位仍然存在缺点… 一旦载入不能移动

回顾：3.3 进程的描述与表达

进程七状态变迁图

思考：
必须换出先换谁？
允许换入先换谁？



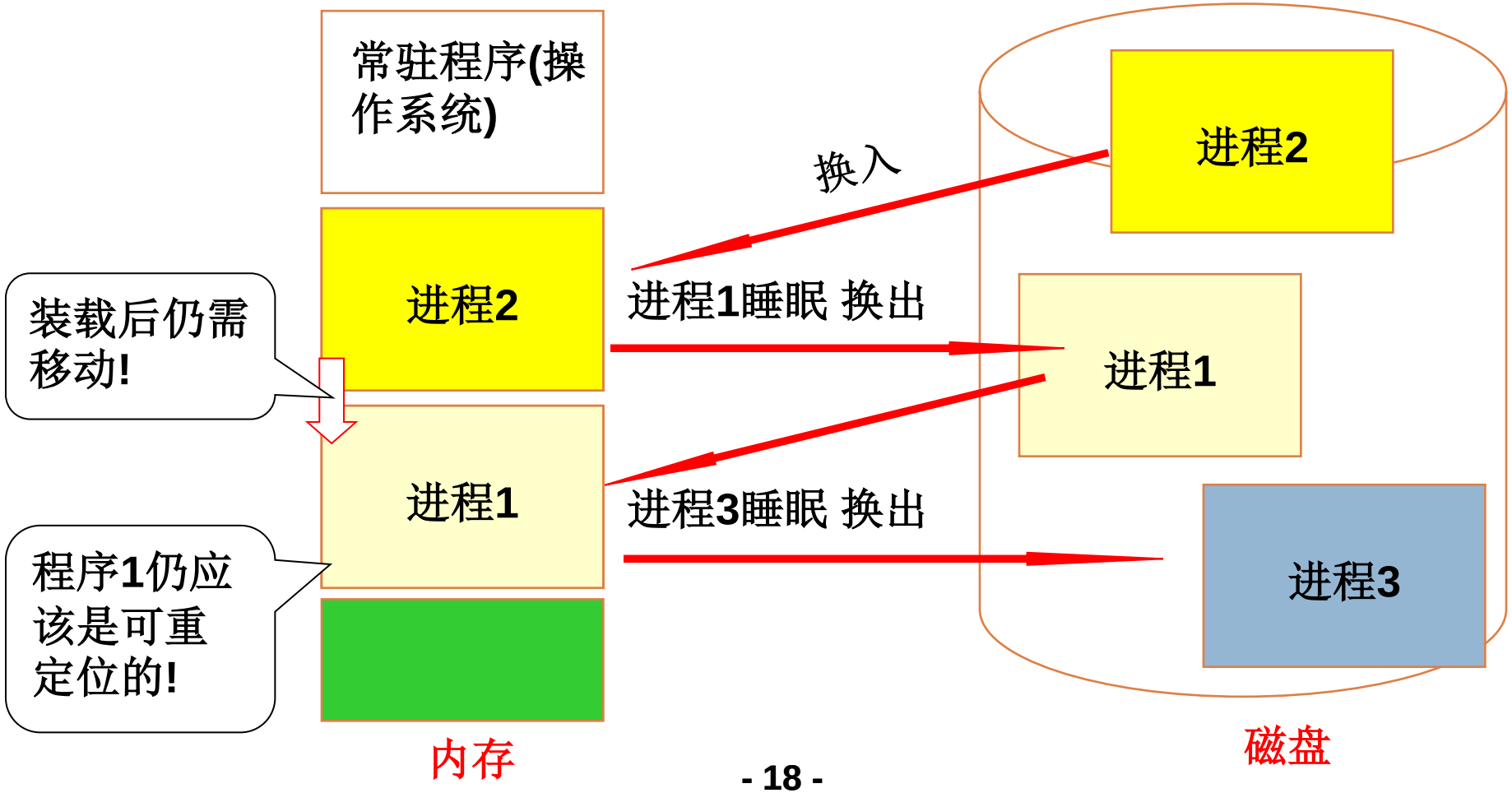
进程状态变化图（七状态）

挂起状态 – 引入主存←→外存的交换机制，虚拟存储管理的基础

移动也是很有必要的!

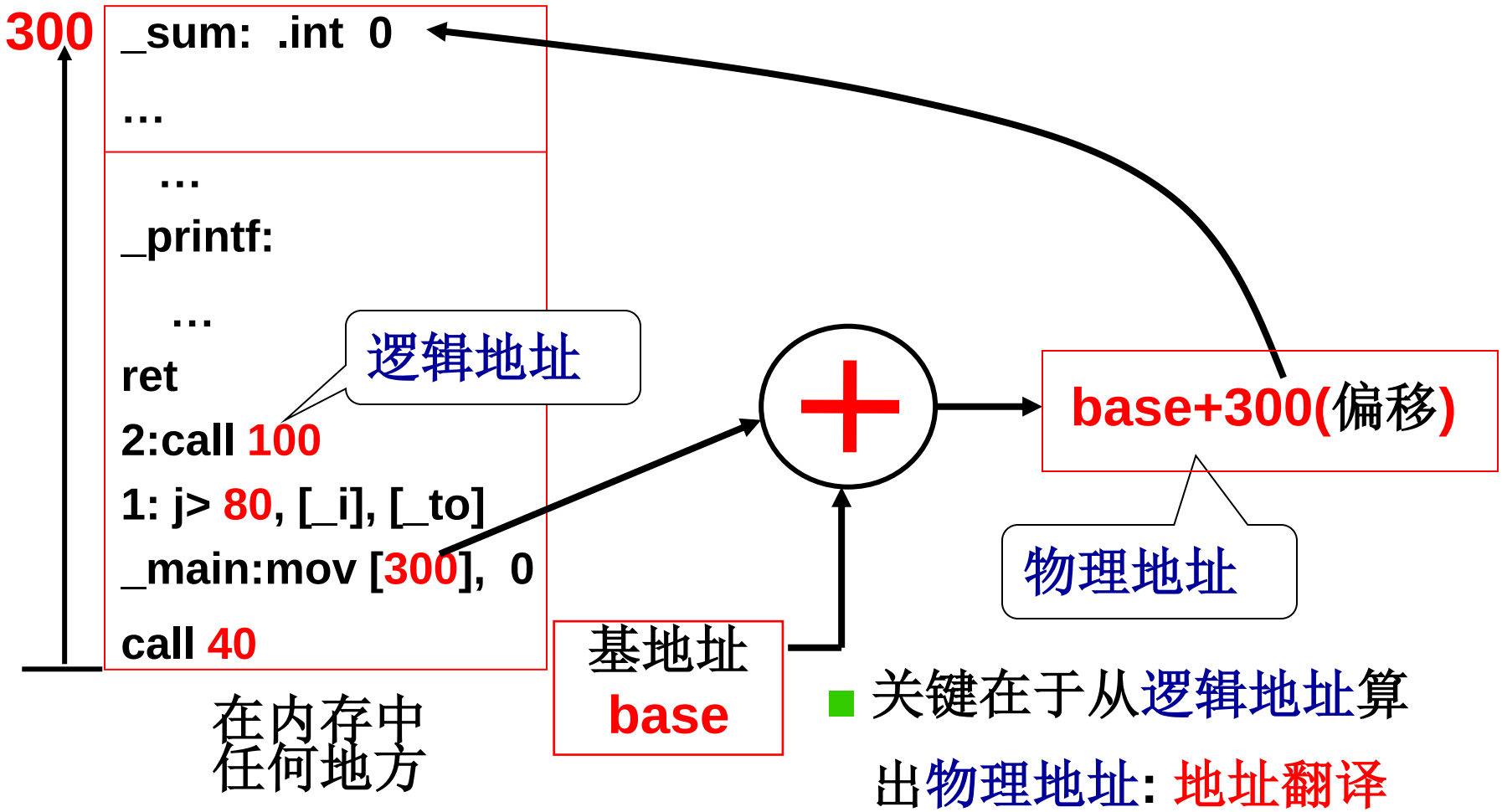
- 一个重要概念: 交换(swap)
- 能让更多的进程并发，提高内存的利用率

如何来实现呢？



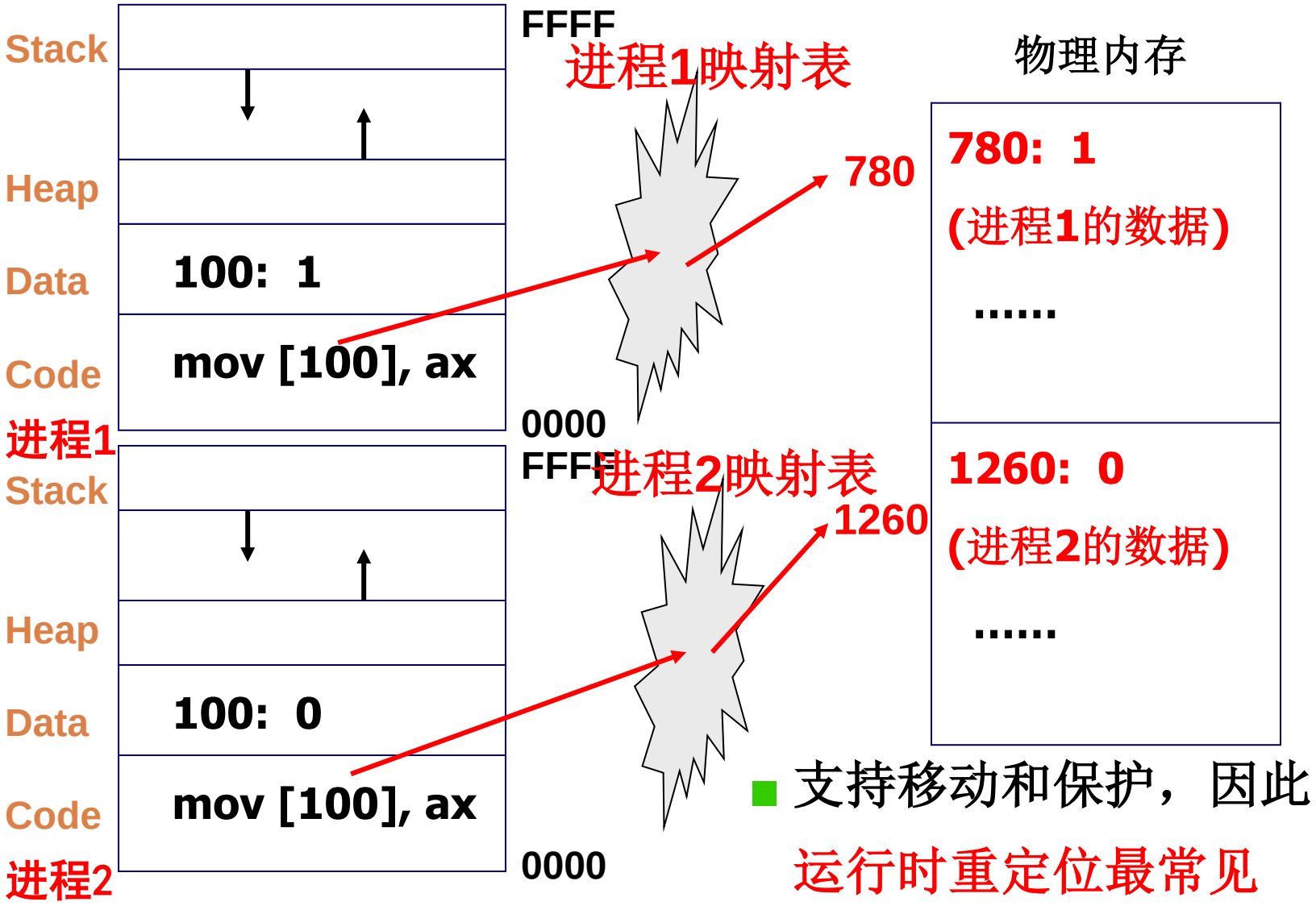
重定位最合适的时机 – 运行时重定位

- 运行时记录分配的基地址，根据这个基地址进行动态的重定位，而不是加载时直接算出并修改程序中的所有地址。



- 关键在于从逻辑地址算出物理地址: 地址翻译

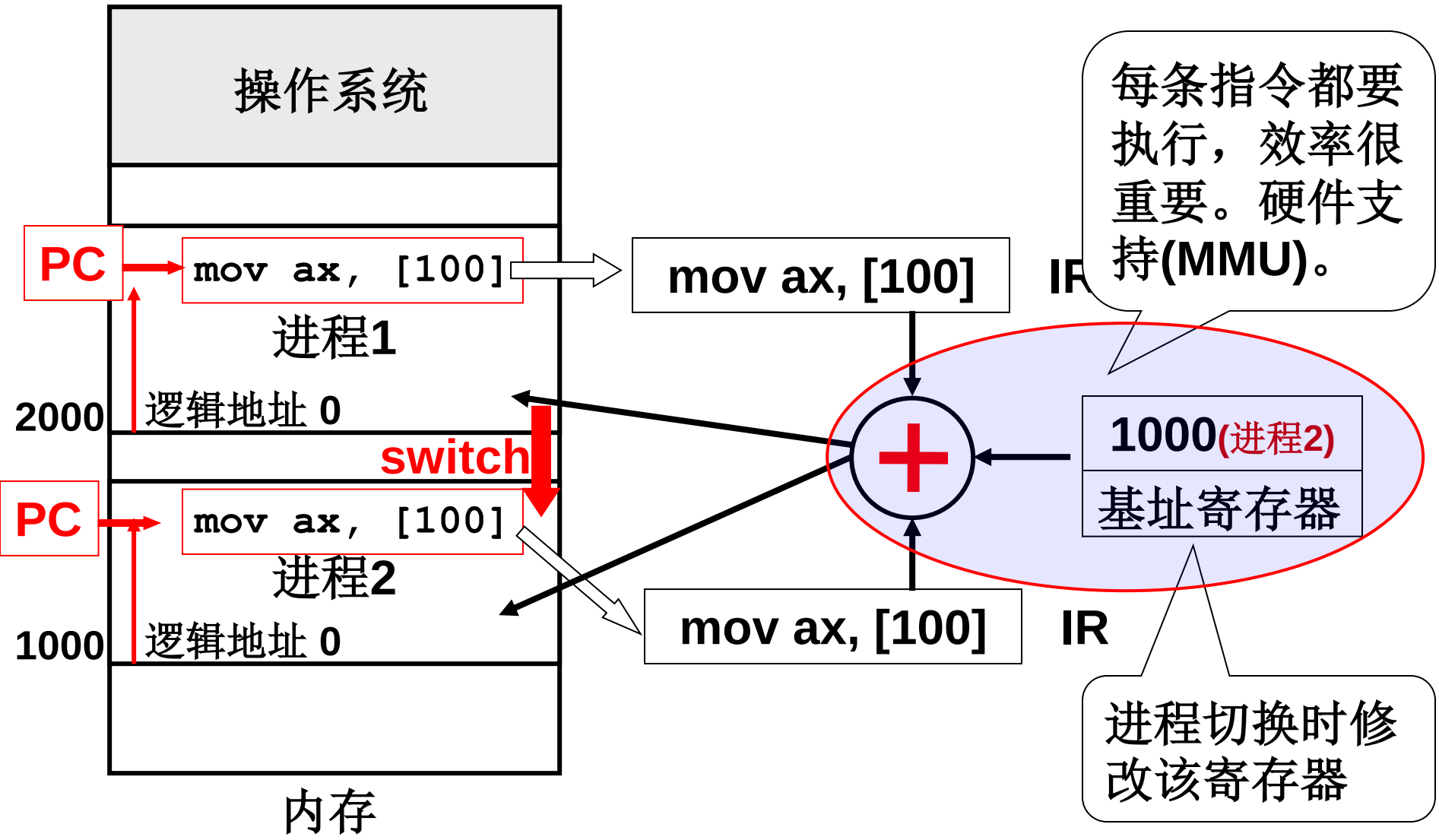
运行时重定位还有一个好处: 进程保护（隔离）



- 支持移动和保护，因此运行时重定位最常见

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和服务器、分布式等现代操作系统基本都支持虚拟内存管理。

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：**虚拟地址是由用户编写程序链接时定义的全局地址；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，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（链接脚本中可以在寻址空间中任意定义的段基址以及编译器计算出的段内偏移）

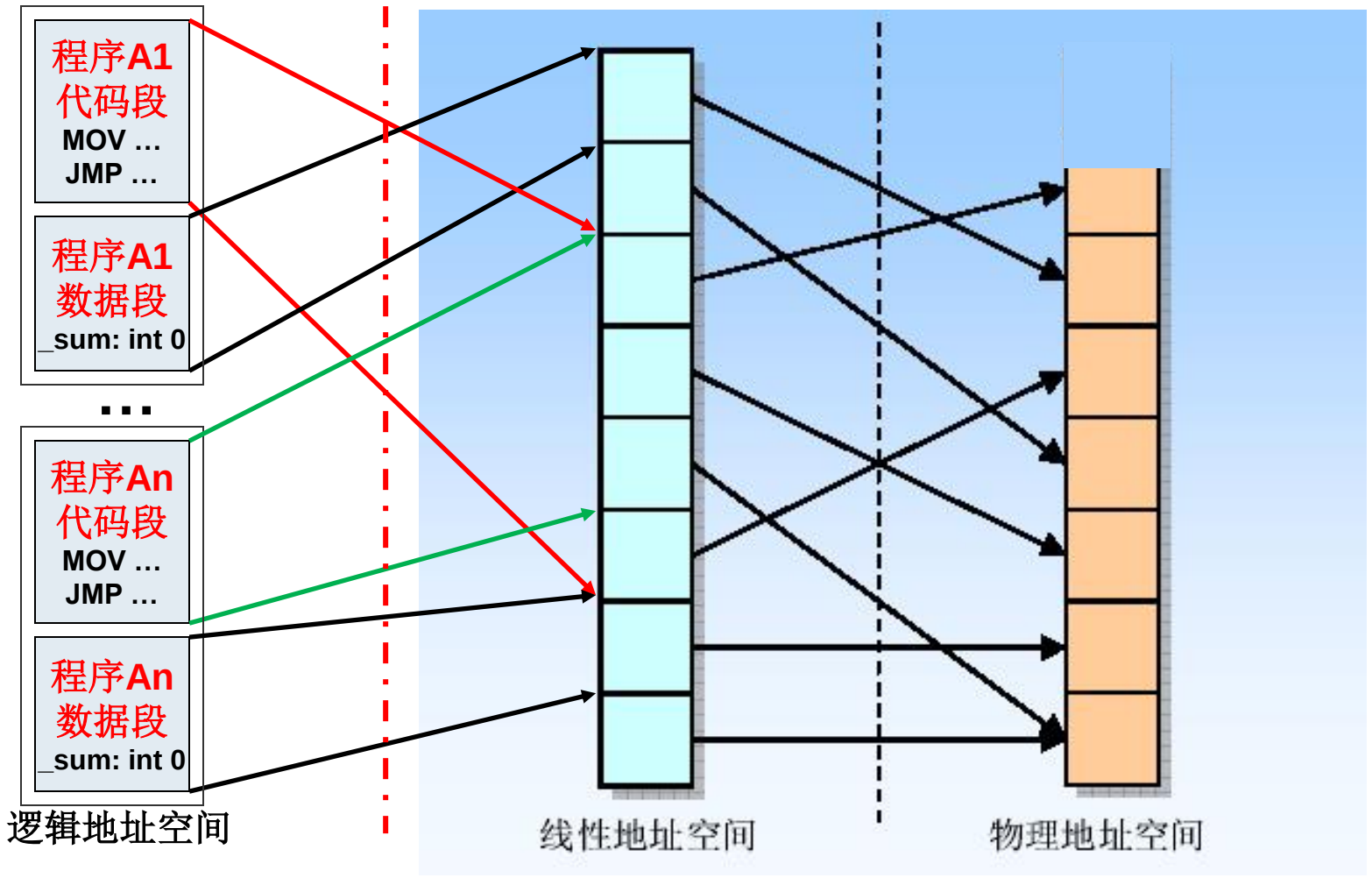
- **线性地址空间：**

  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



CPU可寻址的地址空间

第3组3个重要概念

- MMU(Memery 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

简单总结一下

- 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
- PC和服务器、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

- *p=0x12345
- Int a,b;
- b=&a;
- 考虑在ucos/vxworks;windows/linux上的上述语句执行结果。
- 前两者链接时重定位
- 后两者运行时重定位
- 如果能通过运行时重定位实现进程保护，那么程序中的虚拟地址将不是物理地址!

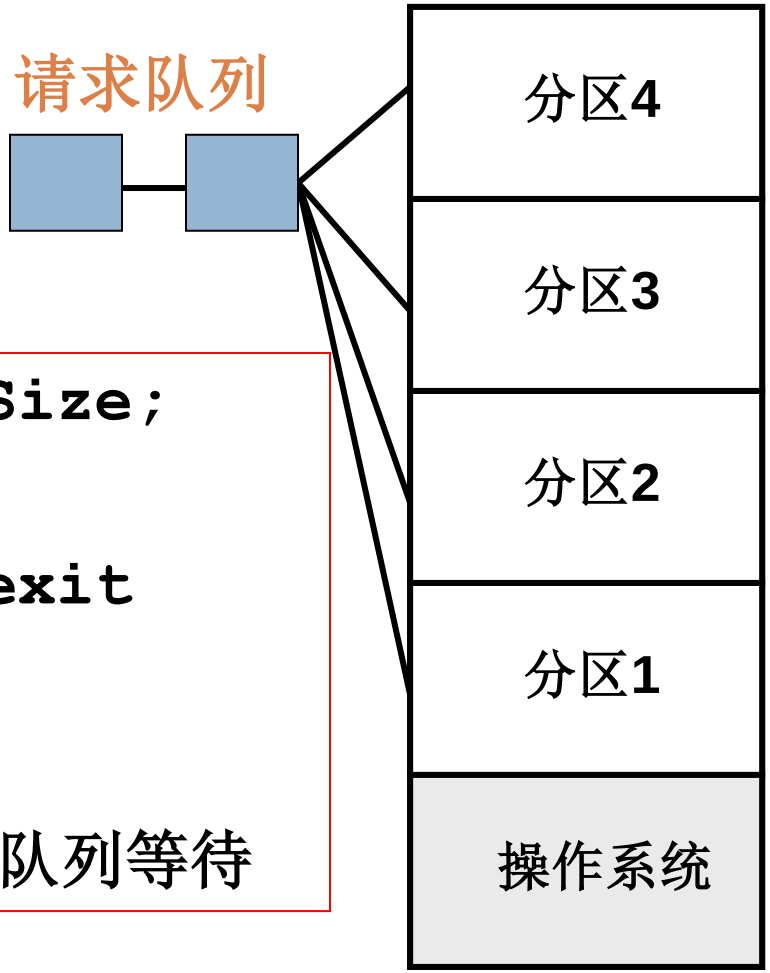
内存分配与管理方案

- 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)

从最简单的分配方案开始!

(1)固定分区--等长分区

- 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



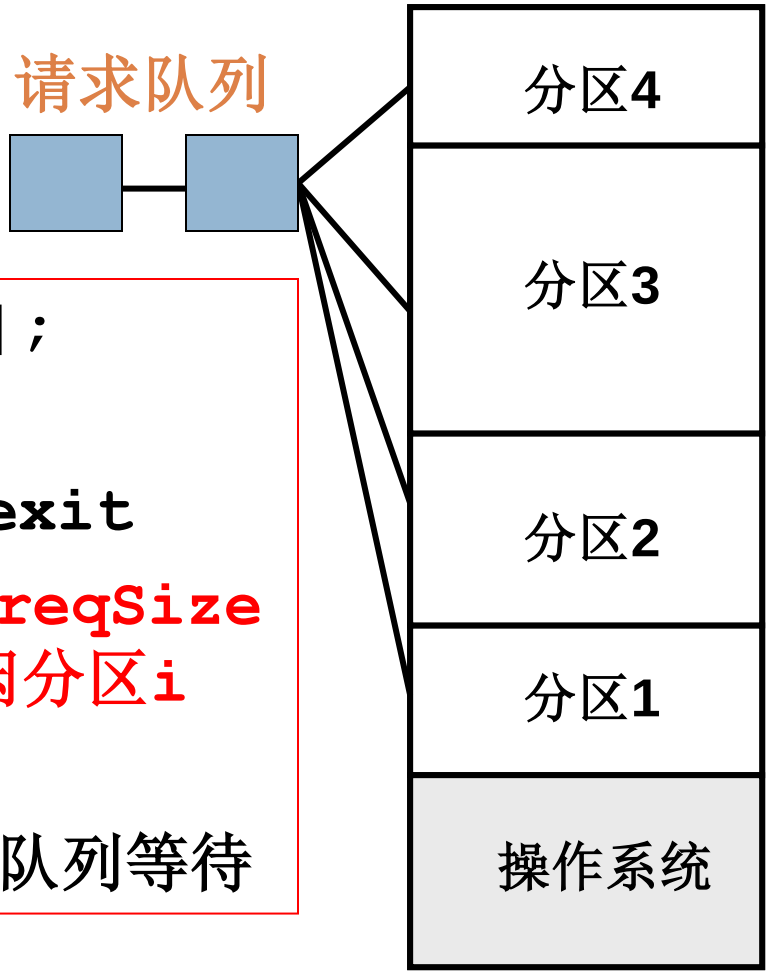

```
bool AvailSec[k]; int SecSize;
```

内存请求算法　//进程创建时

1. if(reqSize>SecSize) exit
2. 找出AvailSec[i]为假的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

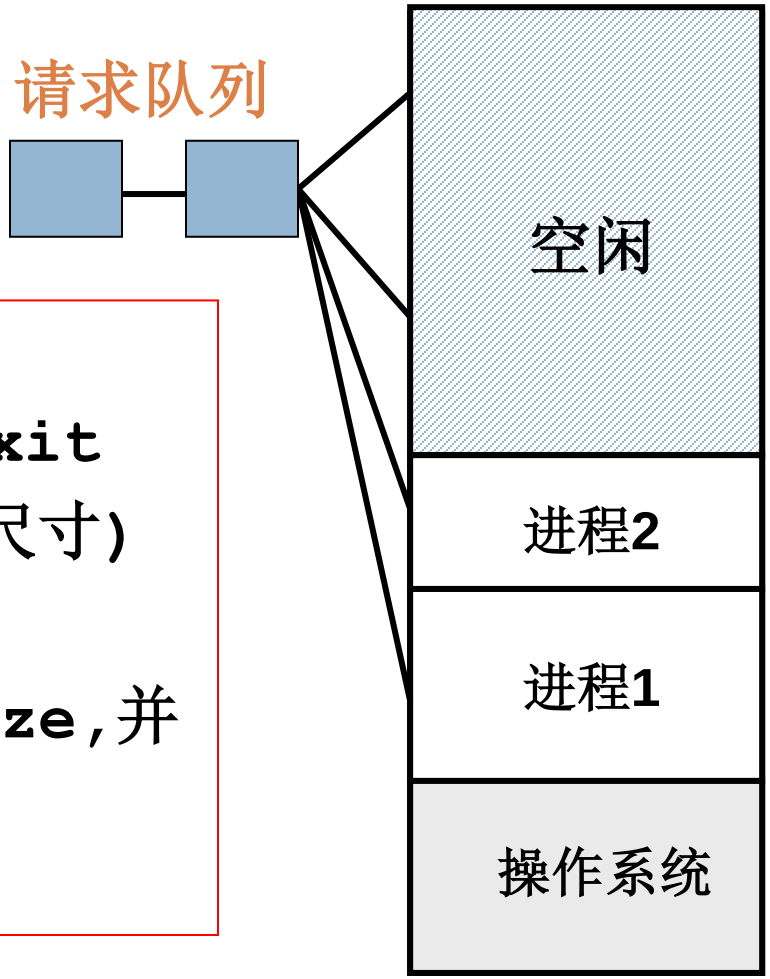
- 孩子有大有小，进程也有大有小...
  - 初始化时将内存分成k个大小不同的分区




```
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

- 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



内存请求算法　//进程创建时

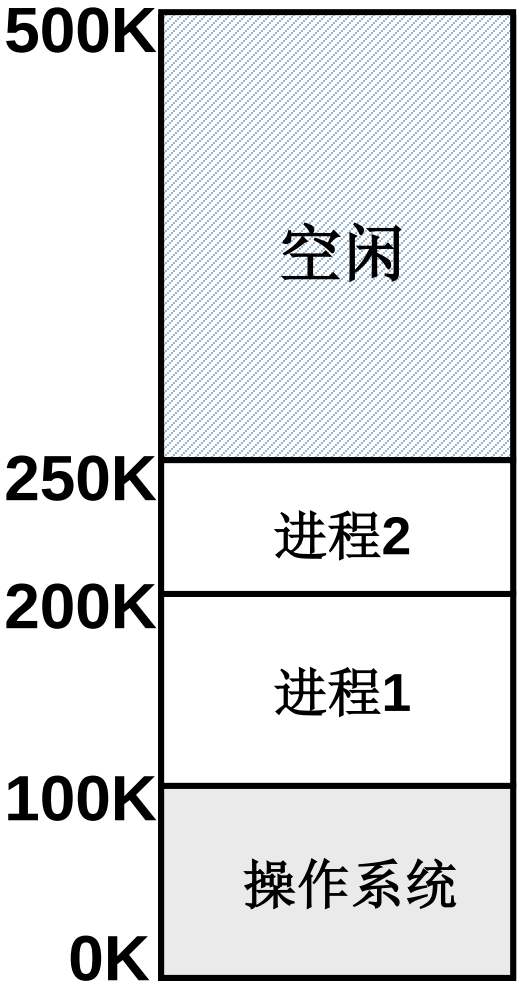
1．if(reqSize>内存大小)exit

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)

- 内存请求: reqSize = 100K

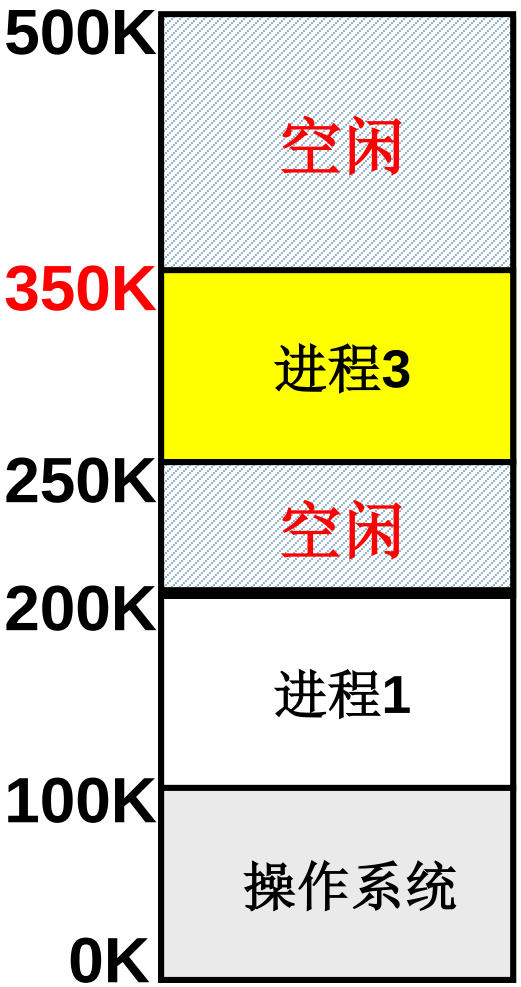
空闲分区表

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)



- 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
250K	100K	P3

(4)分区分配算法 – 找到合适的空闲分区

- 发起请求reqSize=40K怎么办?
  - 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！
  - 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
  - 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

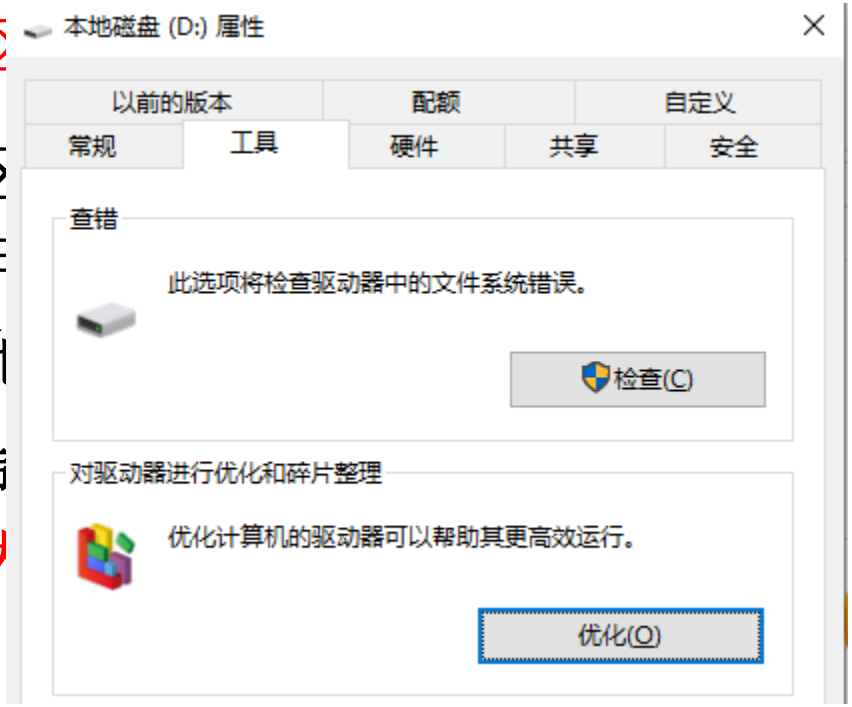
空闲分区表

始址	长度
350K	150K
200K	50K

仍然需要根据应用的特点
来决定选取哪种策略

(5)碎片问题

- 碎片：内存中剩余的无法使用的存储空间
- 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
- 内部碎片：对固定分区某进程使用能分给其他
- 碎片的产生和分配“制
- 碎片问题可以消除（内
- 磁盘也存在碎片问题！



(6)外部碎片处理 – 内存紧缩

- 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，如果复制速度10M/1秒，则1G内存的紧缩时间约为100秒

磁盘碎片是什么？

该值表明连续分配技术不合适!

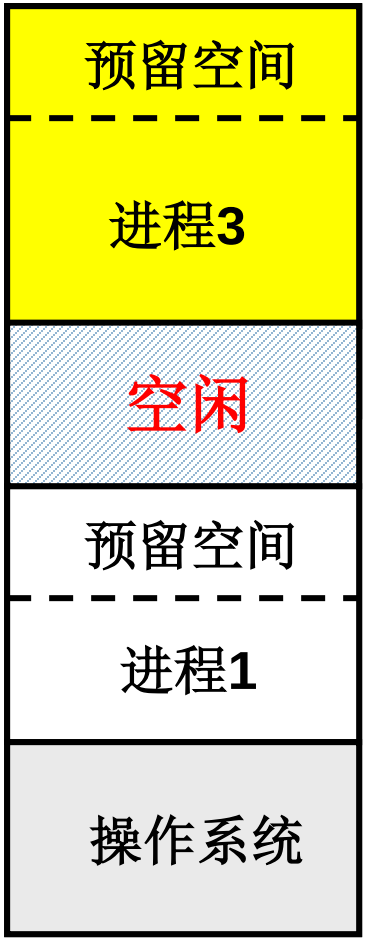
空闲分区表

始址	长度
350K	150K
200K	50K

分区方案中还有一个问题没有搞清楚

- reqSize值怎么确定?
  - reqSize=code+data+stack+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浪费时间
  - 怎么办? 各个段区别对待，分别分配

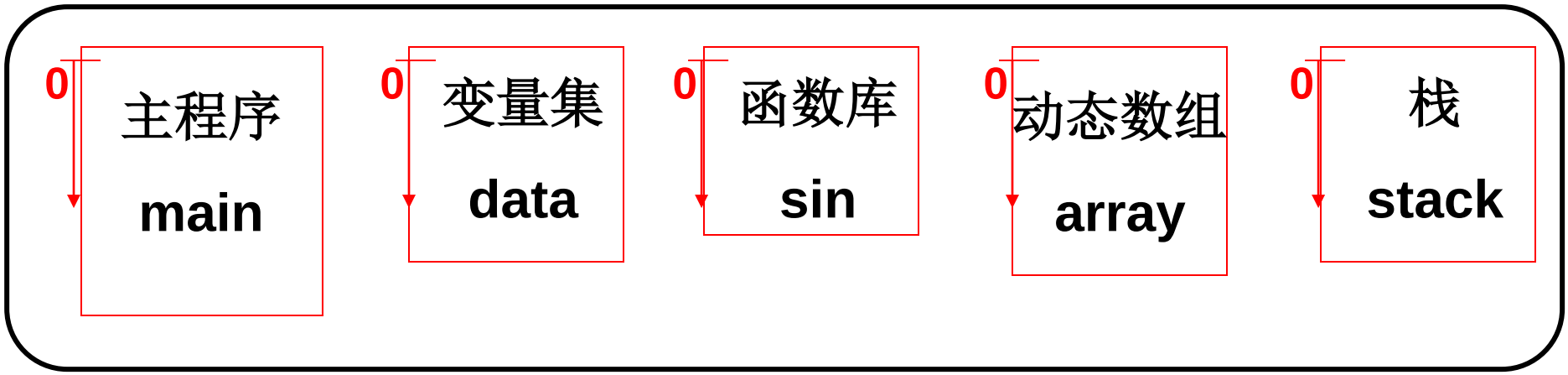
还有利于建立合适的保护策略!



分段(Segmentation)!

程序员眼中的程序（从汇编—高级语言）

- 由若干部分(段)组成，每个段有各自的特点、用途!

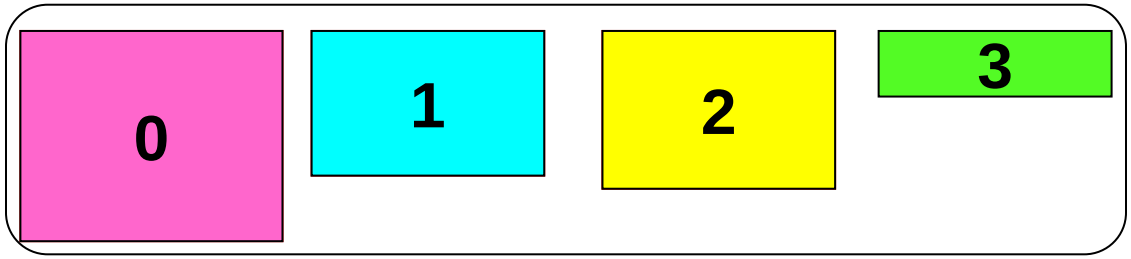


程序员眼中的一个程序

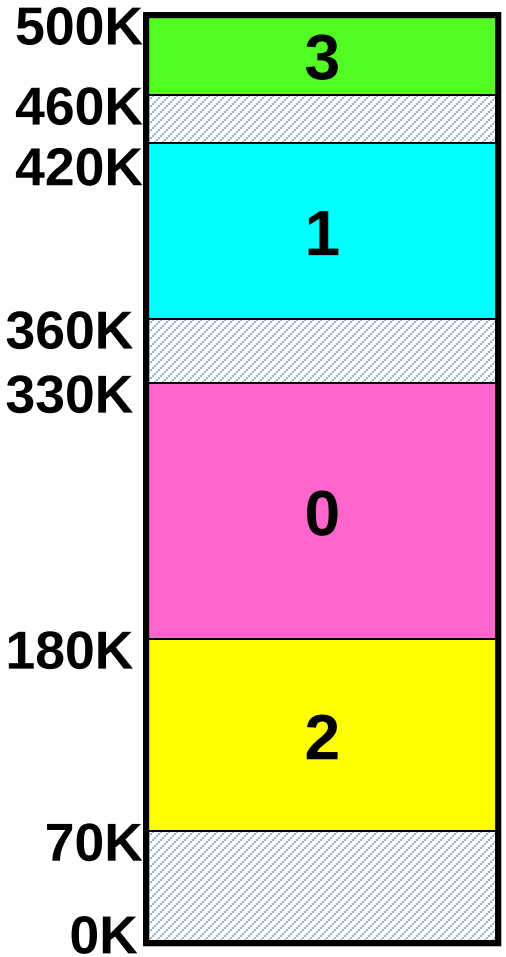
- 程序员怎么定位具体指令(数据): <段号, 段内偏移> CS:IP
- 分段符合用户观点: 用户可独立考虑每个段特点(分治)

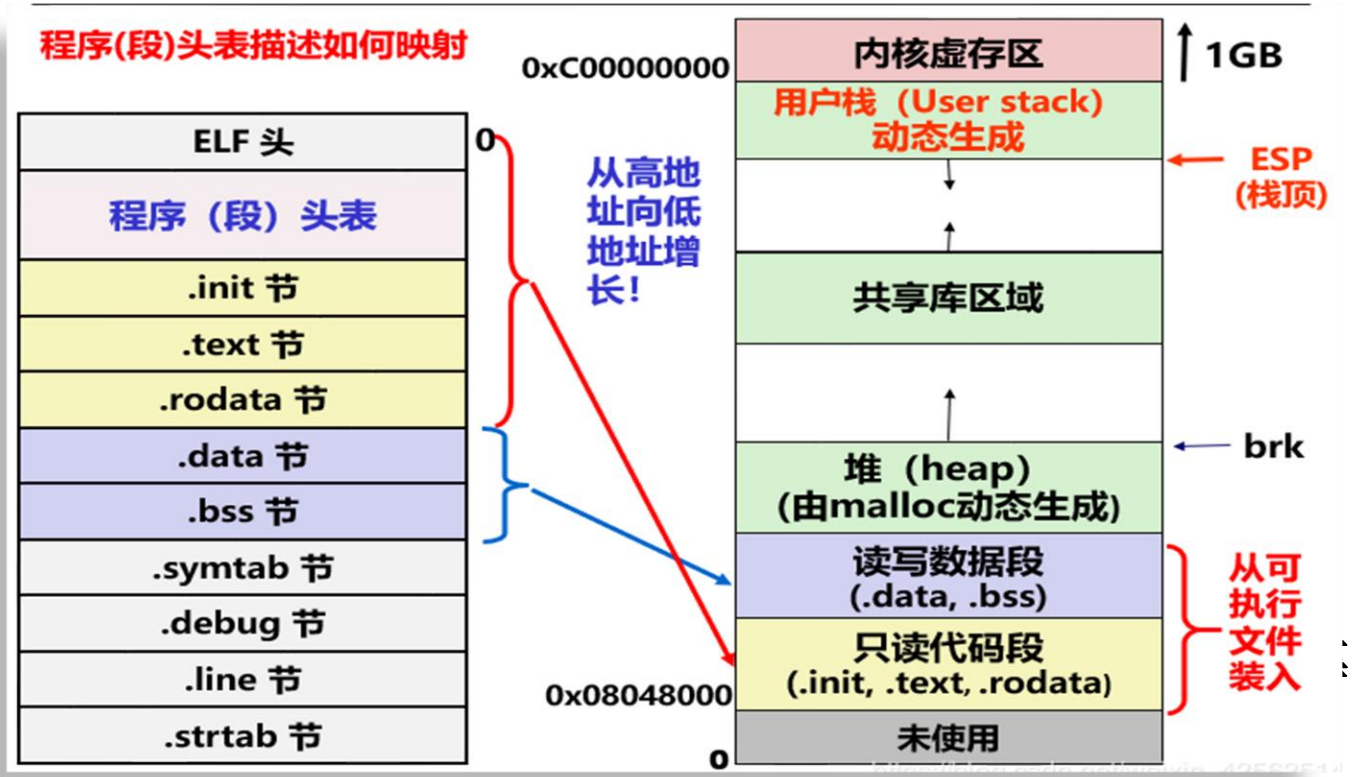
将段放入内存　引入段表

- 分段制造了二维空间，而内存是一维的



各个段可分散





分段的地址翻译

- 来看一个例子!

段号	Offset

15 14 13 0

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

PC = 0x240 ➡ 段号:0, 偏移240

逻辑地址

异常

0x4240 `mov ax, _var` 0x240

`…` `…`

`_var dw 0x314159` 0x4050

取出指令

800>240

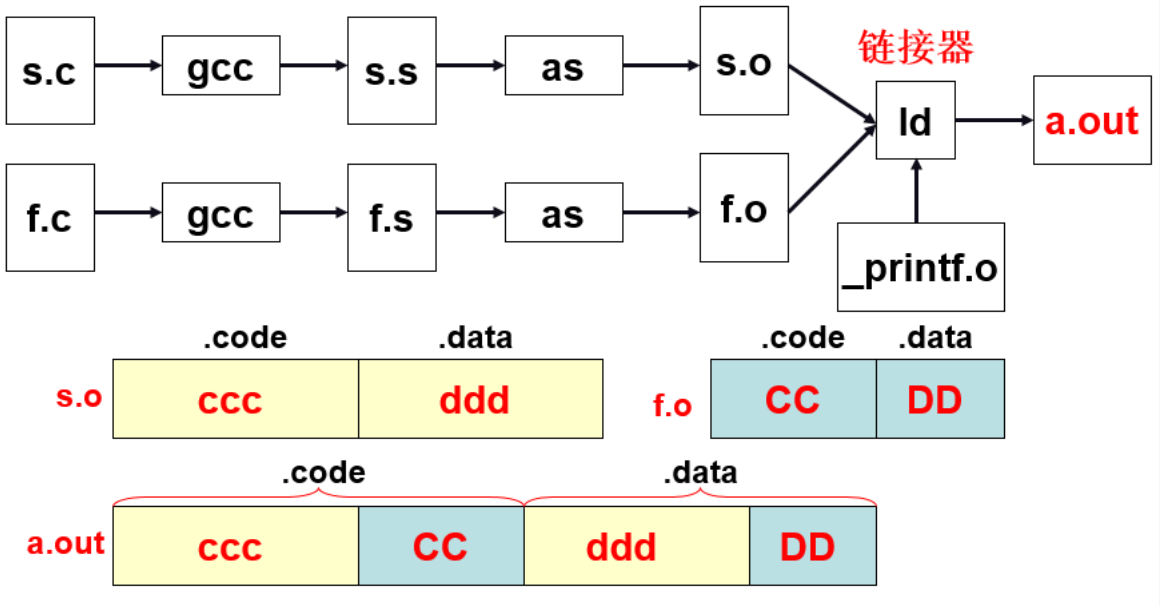
0x4000+240

物理地址

- 同样可算出_var应该在0x4850处

分段技术总结

■ 第二步: 汇编与链接——从汇编到可执行程序



■ 实现原理

- 将程序按含义分成若干部
- 各目标文件段从0开始编址
- 创建进程(分别载入各个段
- 内存仍用可变分区进行管理，载入段时需调用分配算法
- PC及数据地址要通过段表算出物理地址，到达内存
- 进程切换时，进程段表也跟着切换

进程、内存、编译环境、编程思想被融合在一起了，
这正是操作系统的复杂之处!

分段技术优缺点分析

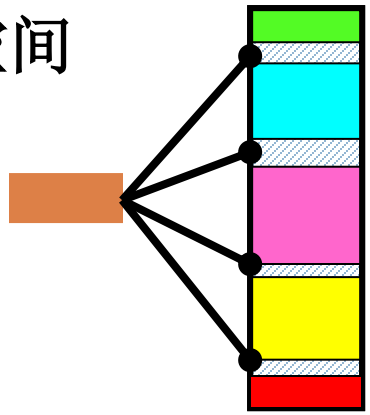
- 优点:
  - 不同的段有不同的含义，可区别对待
  - 每个段独立编址，编程和管理容易(分治)

- 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



8.4 分页内存管理

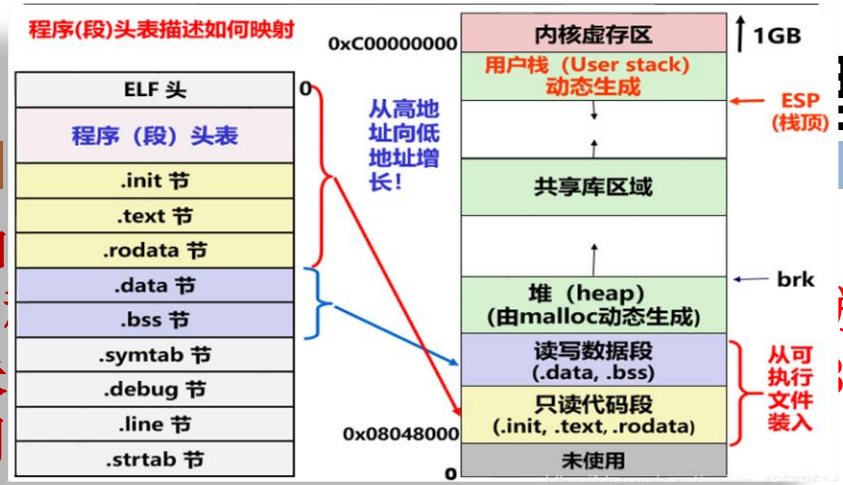
分页(Paging)!

思考：按需供给内存

从连续到离散

- 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（函数调用链）
  - 代码和数据段没必要同时加载到内存，一次分配给一点(没有外部碎片，内部碎片有上界)
  - 将面包切成片，将虚拟/物理内存等分

思想示意图



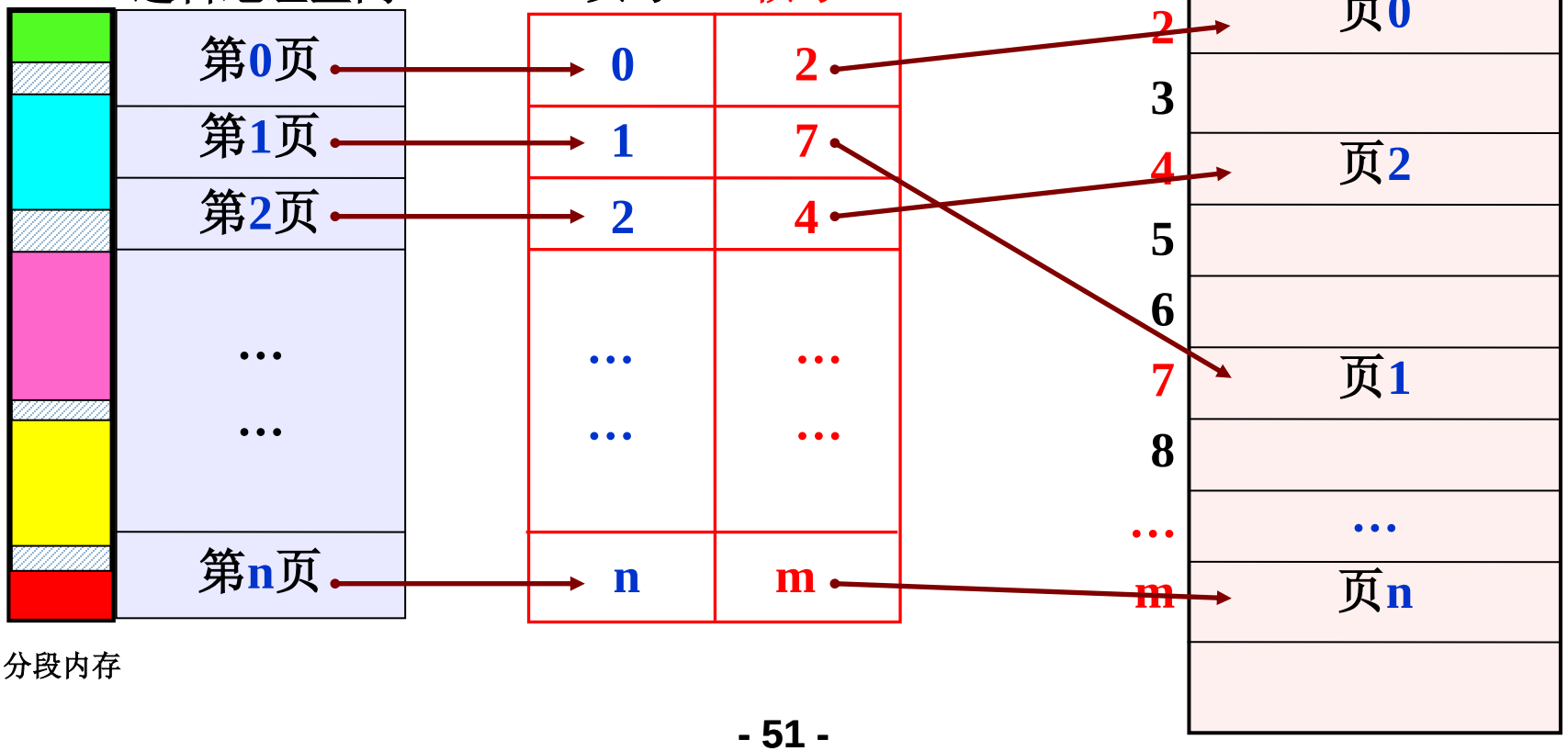
如
1)
未
问

访问？ 2)某些
)长时间未访

用户程序
逻辑地址空间

页 表
页号 帧号

帧号 物理地址空间



分段内存

分页机制中的页表

- 和分段类似，分页依靠页表结构

第0页
第1页
第2页
…
…
第n页

逻辑地址格式

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(赵炯书) = 帧(教材)

分页的地址翻译

_var地址也需要翻译，因此执行该指令需要查两次页表!

- 一个实例!

```
mov ax, _var      0x240
…                 …
_var  dw 0x314159 0x3050
```

逻辑地址格式

Page#	Offset
31 … 12	11 … 0

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

为什么?

物理地址：0x5240

权限检查

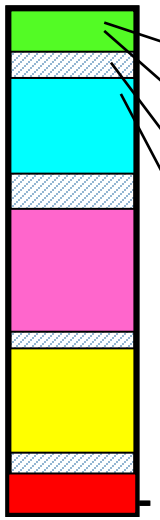
访问错误

来考虑一些细节问题

- 和段表不一样，页表可能会很大!
  - 页是用来解决碎片问题的 ⇒ 页面尺寸应尽量小
  - 页面尺寸通常4K(2^{12})，32位机， 2^{32} / 2^{12}=2^{20}个页
  - 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

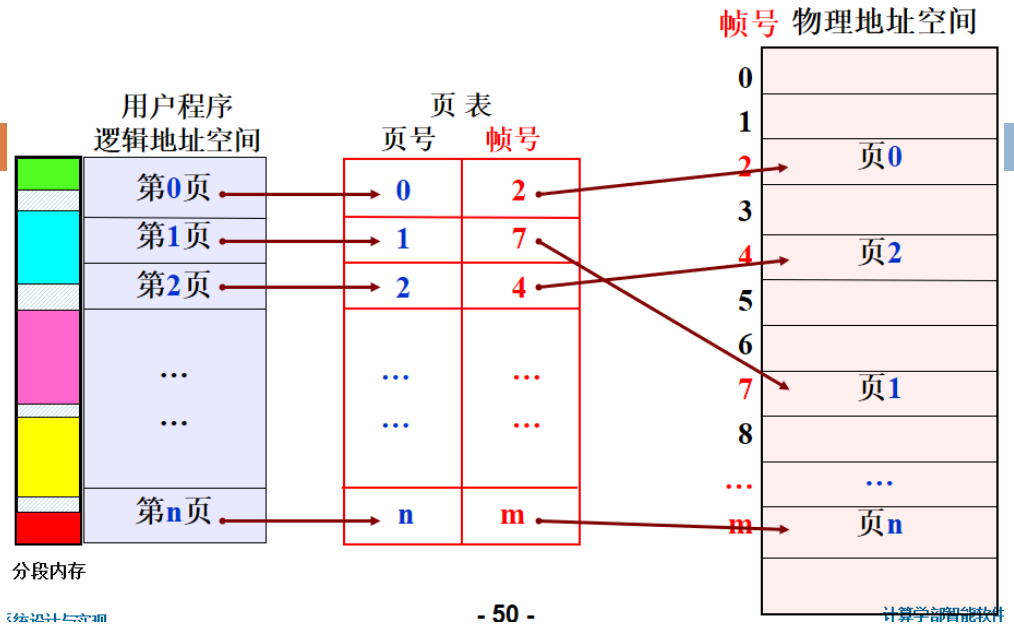
页号	页框号	保护
0	5	R
1	1	R/W
3	3	R

应该是连续的!

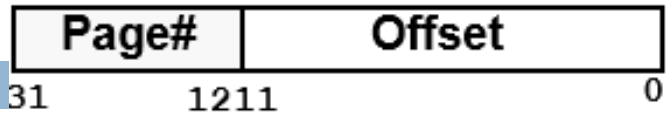


页号	页框号	保护	有效
0	5	R	1
1	1	R/W	1
2			0
3	7	R	1

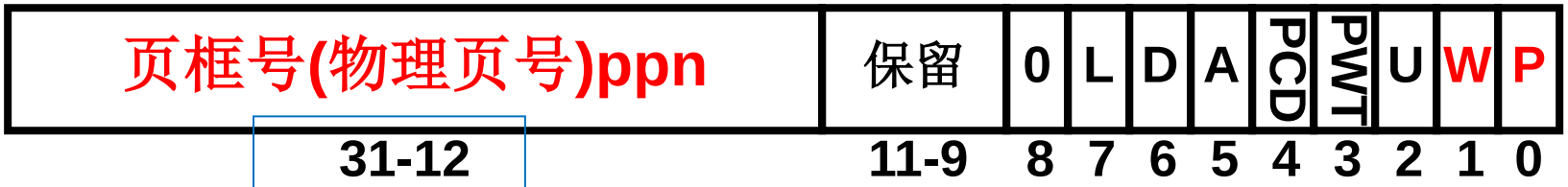
可以删掉!



逻辑地址格式



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va找到物理地址，一级页表，每页4k。查找页和页框的对应

假设PC :0x3422

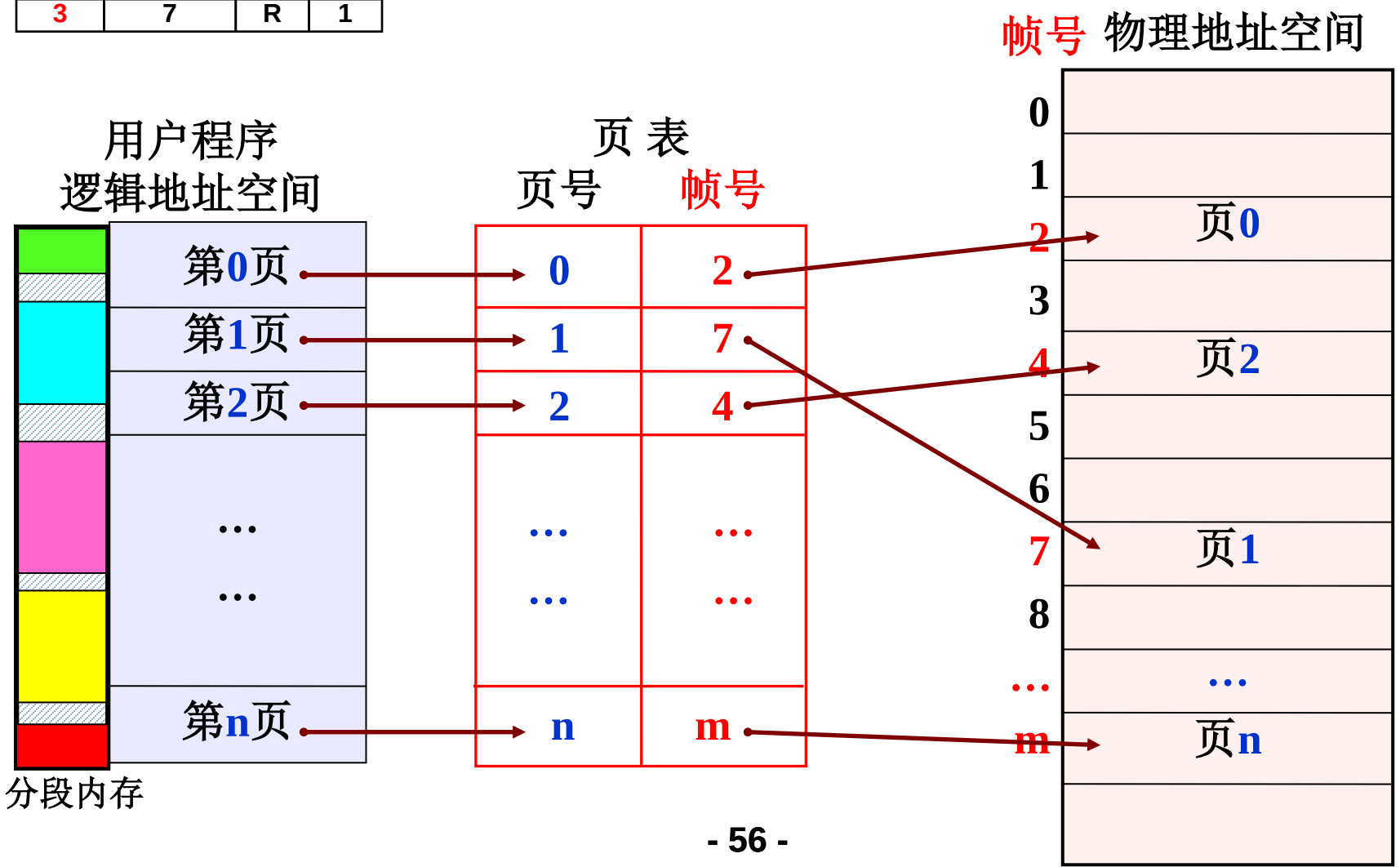
```
unsigned translate(unsigned va, int wr) {
  struct pte *pte = &page_table[va >> 12];//页号
  if(!pte->valid || (wr && !pte->writeable))
          throw address_fault;
    return (pte->ppn << 12) | (va & 0xfff); }
```

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

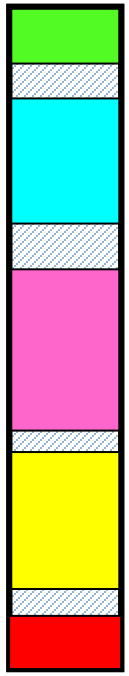


多级页表

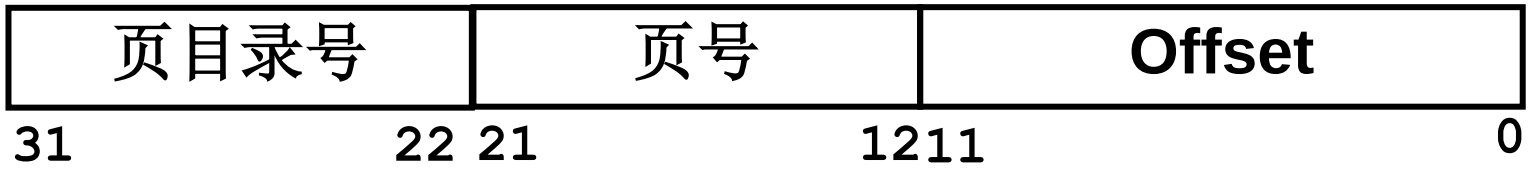
- 32位地址空间+ 4K页面+页号连续⇒2^{20}个页表项
  - 2^{20}个页表项，每个4字节，都放在内存，要4M内存
  - 系统中并发10个进程，需要40M内存
  - 实际上大部分逻辑地址根本不会用到
  - 引入多级页表，顶层页表常驻内存，不需要映射的逻辑地址不需要建立页表项

为什么每个进程需要独立一个页表？

32位: 总空间[0，4G-1]!



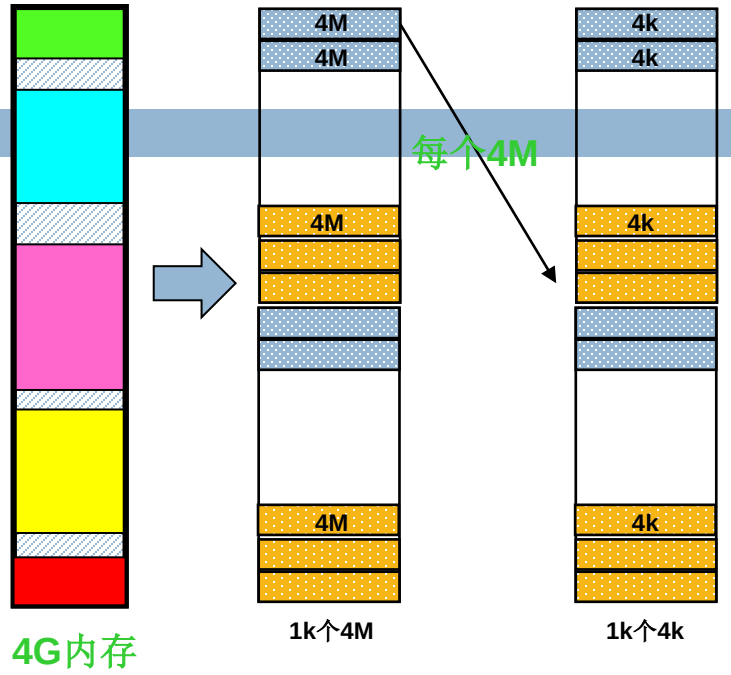
32位逻辑地址格式(多级页表)



多级页表

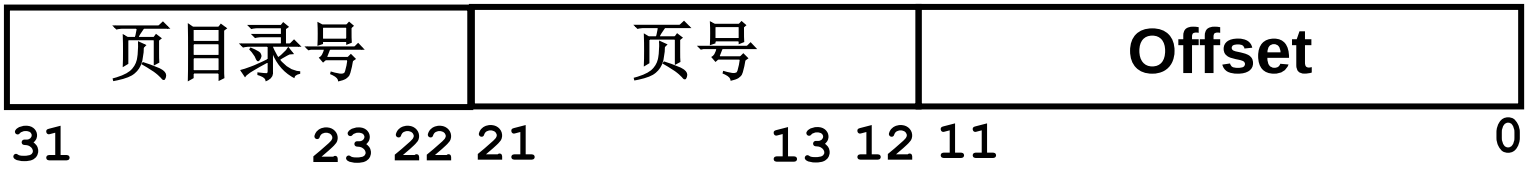
林力帆 113 37 1 02 01
伽晗 113 03 1 02 19

楼房：楼号-单元号-房间号

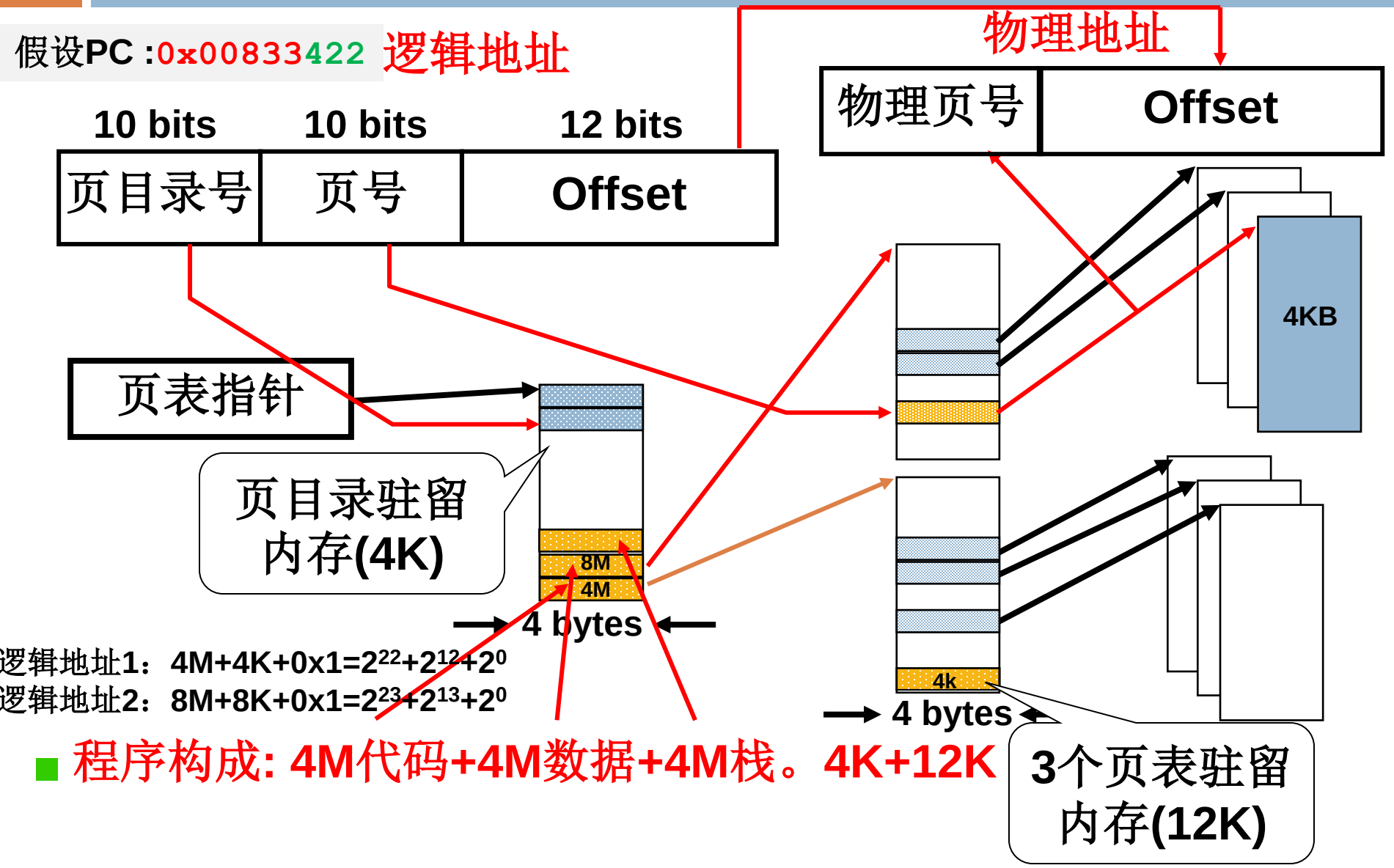


32位地址，4G内存：先按照4M分割，每个4M内再按照4K分割。

32位逻辑地址格式(多级页表)



多级页表时的地址翻译


假设PC :0x00833422
逻辑地址
物理地址
10 bits
10 bits
12 bits
页目录号
页号
Offset
物理页号
Offset
4KB
页表指针
页目录驻留
内存(4K)
8M
4M
4 bytes
逻辑地址1：4M+4K+0x1=2^22+2^12+2^0
逻辑地址2：8M+8K+0x1=2^23+2^13+2^0
4k
4 bytes
程序构成: 4M代码+4M数据+4M栈。 4K+12K
3个页表驻留
内存(12K)

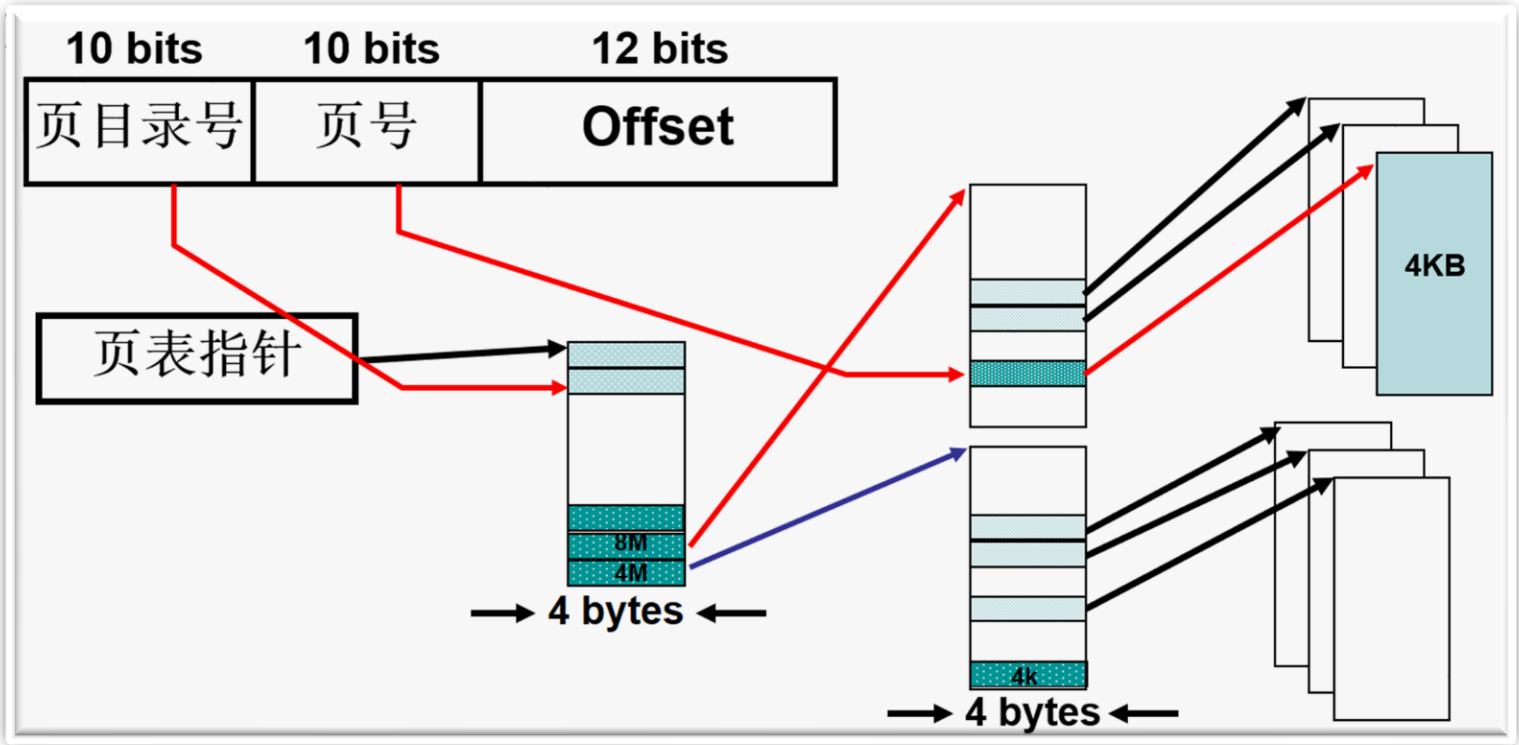
多级页表使得地址翻译效率更低

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%

访问页表计算内存物理地址，从物理地址取内容

需要注意的事实：内存相比CPU本来就很慢!

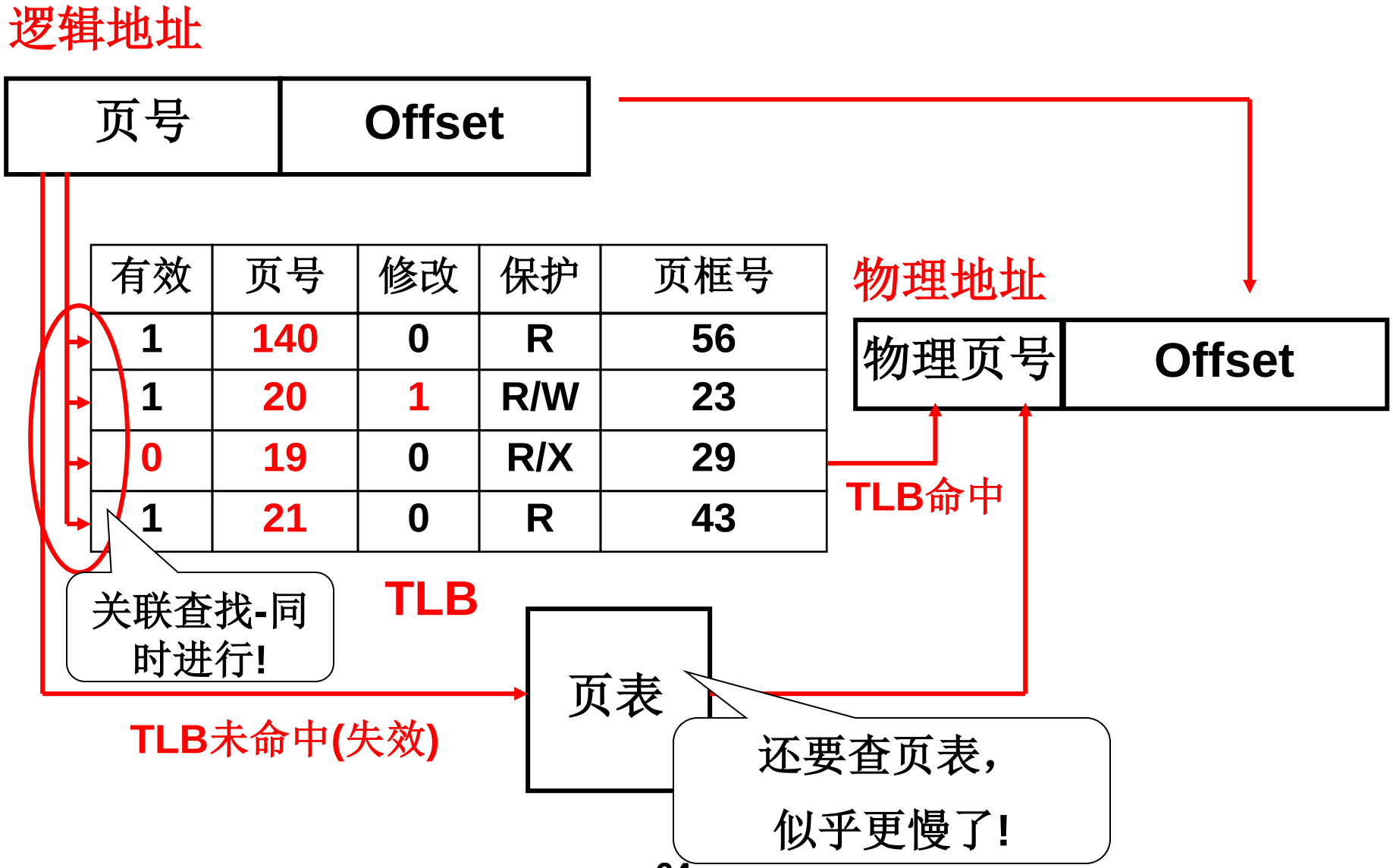


提高地址翻译的效率

- 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⇒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



TLB得以发挥作用分析

- 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:

有效访问时间 = HitR×(TLB+MA) + (1-HitR)×(TLB+2MA)

命中率!

内存访问时间!
假设100ns

TLB时间!
假设20ns

有效访问时间=80%×(20ns+100ns) + 20%×(20ns+200ns)=140ns

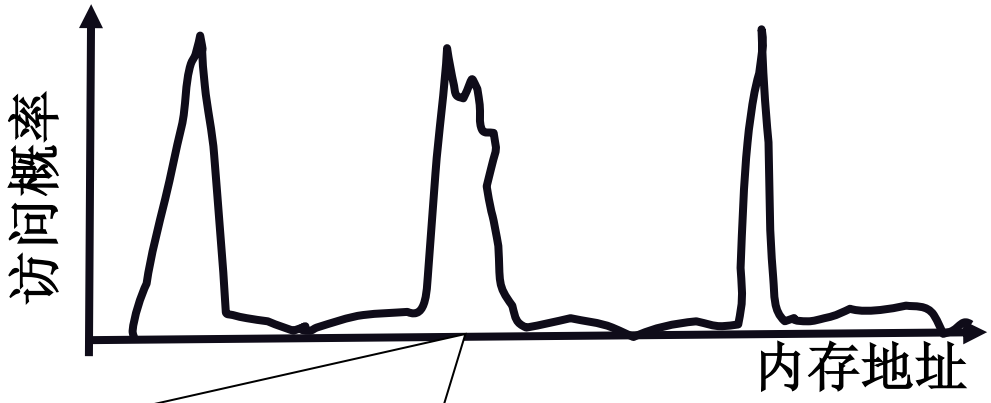
有效访问时间=98%×(20ns+100ns) + 2%×(20ns+200ns)=122ns

平均慢了22%!

- TLB要想发挥作用，命中率应尽量高
- 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?

- 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、局部长时间运行

局部性又是计算机
的一个基本特征

TLB条目少，页表项多 ⇒ 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。

许多东西都与操作系统管理有关，提升资源管理的效率和质量

分页技术总结

- 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
  - 进程创建时申请页框，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)!

- 优点：靠近硬件，结构严格，高效使用内存
- 缺点：没有结合程序的分段（分治）思想

8.5 段页结合内存管理

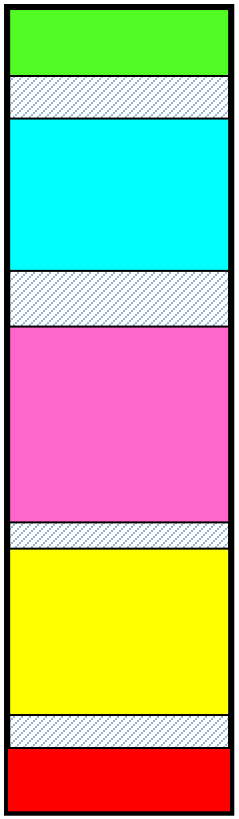
段、页结合!

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 *optional*
Section 1
. . .
Section *n*
. . .
. . .
Section header table

Execution View
ELF header
Program header table
Segment 1
Segment 2
. . .
Section header table *optional*



让段面向用户、让页面向硬件

段号+偏移(cs:ip)

逻辑地址

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

页号 | 偏移

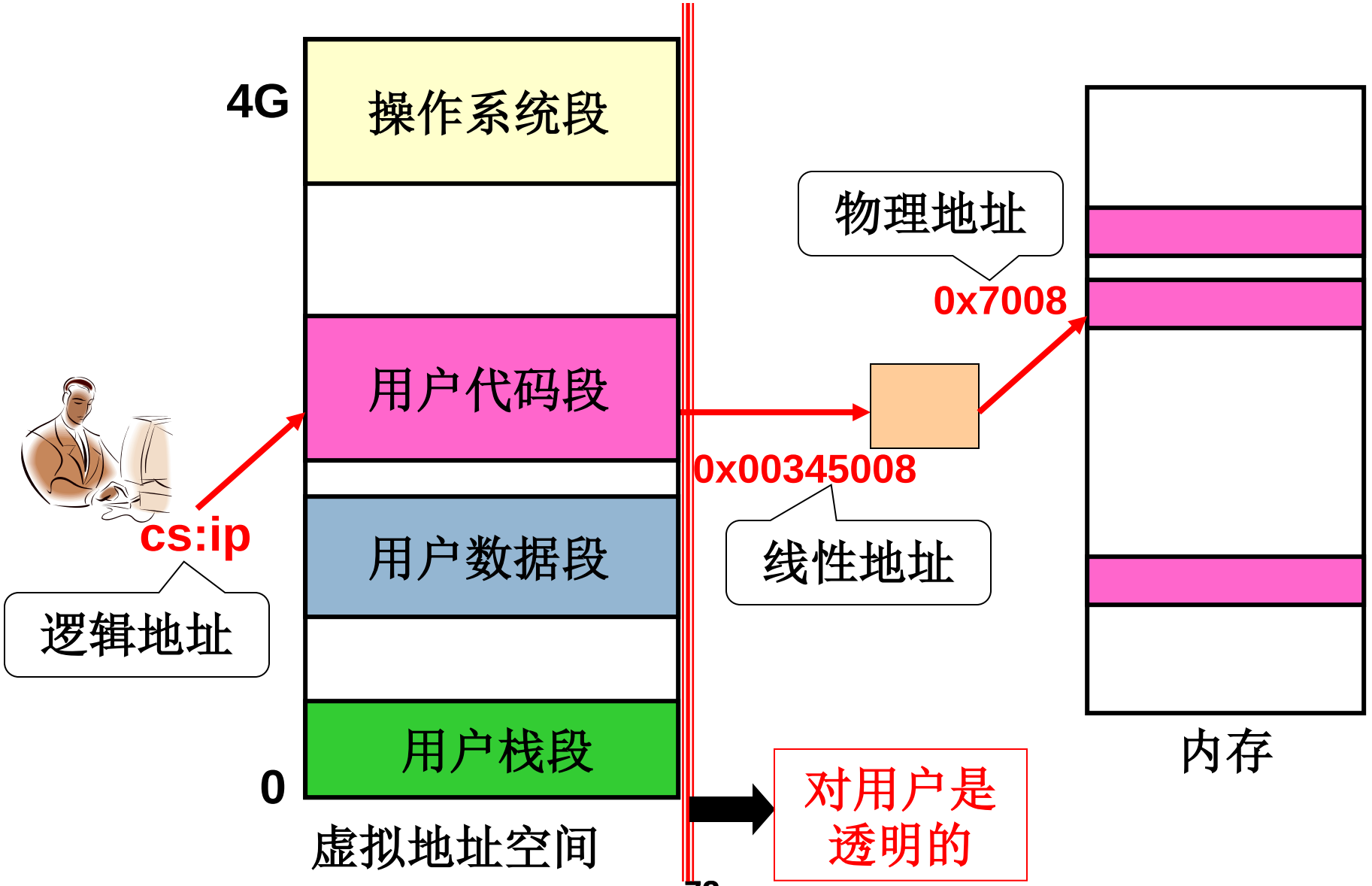
物理地址

物理帧号 | 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

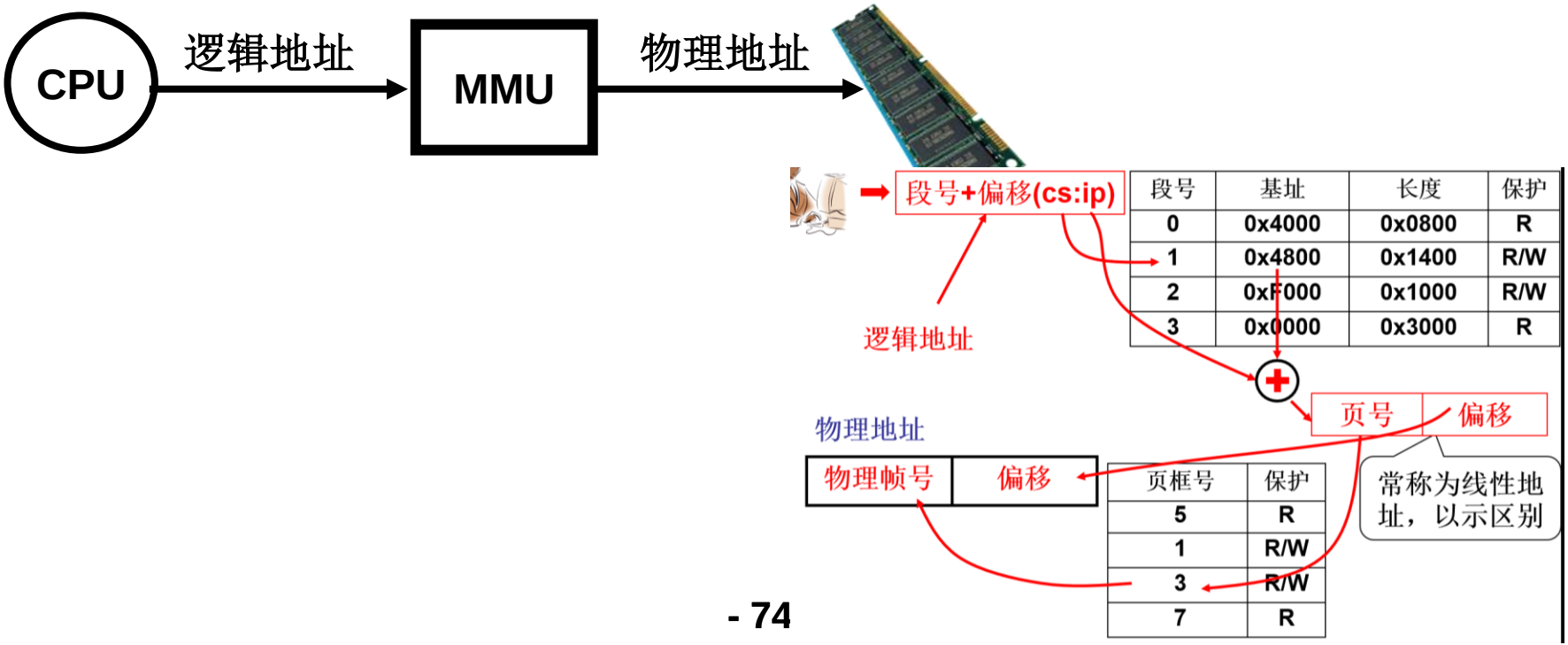


段页结合技术总结

- 实现机理
  - 程序的段和偏移计算线性地址空间(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
- 优点：符合程序员习惯，并可高效利用内存
- 缺点：复杂，访问一次地址需要查表好多次…

操作系统课讲硬件，大家可能觉得奇怪

- 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



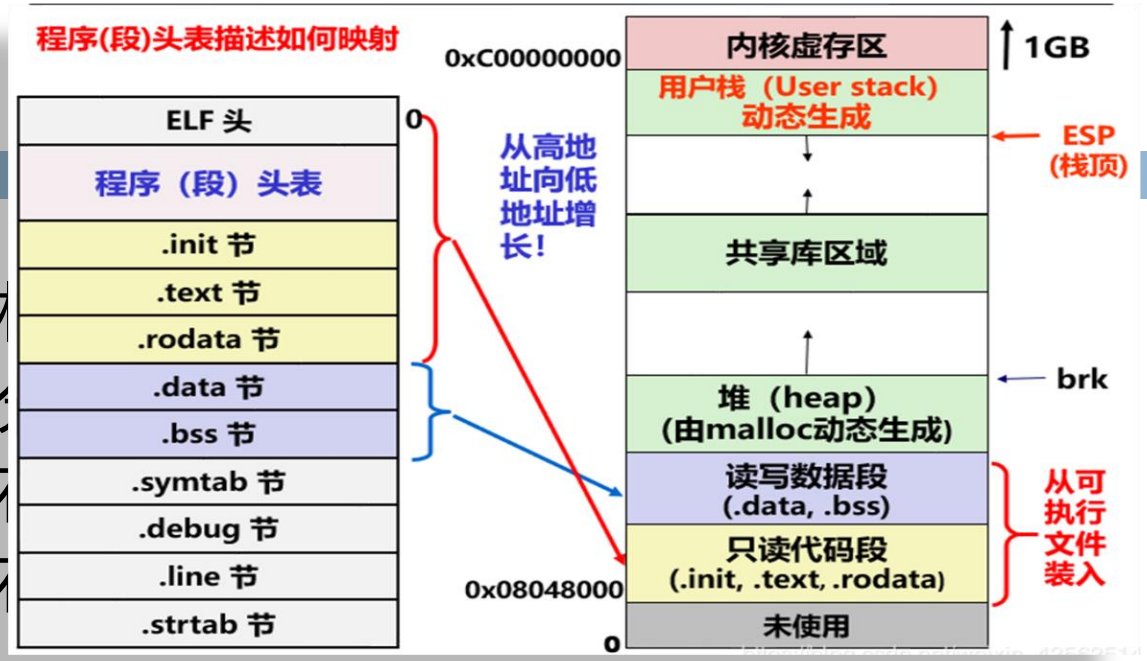
Intel x86的内存

Intel x86的内存管理硬件!

- 实模式，内存地址20位，1M。
- 保护模式，地址32位，4G。
- CR0.PE位=1进入实模式。

X86地址变换

- 保护模式开启分页
- 实模式和保护模式
- 实模式段寄存器中
- 保护模式段选择器



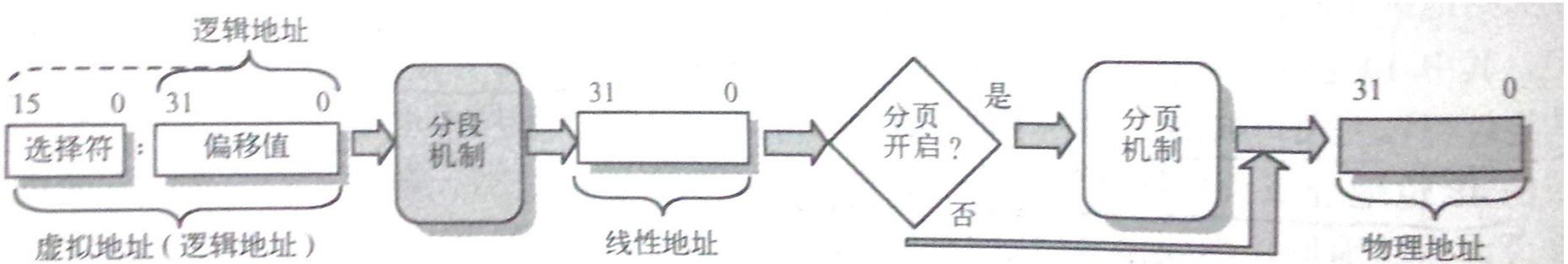
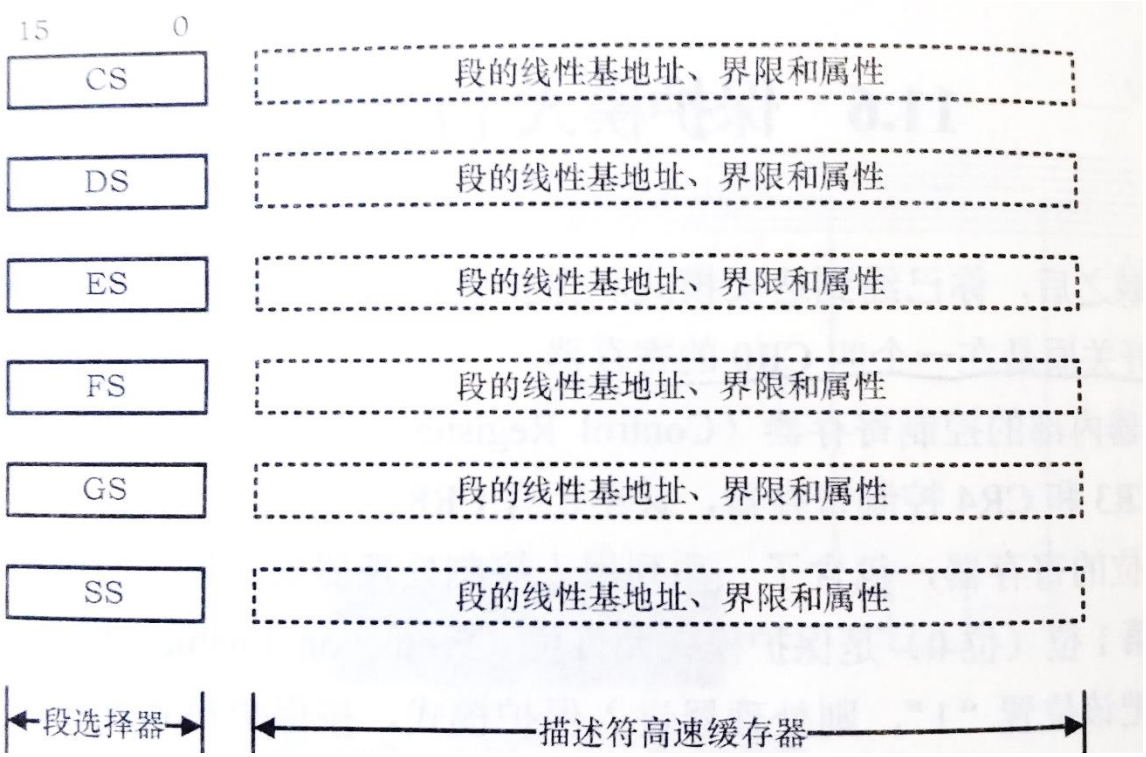


图 4-4　虚拟地址（逻辑地址）到物理地址的变换过程

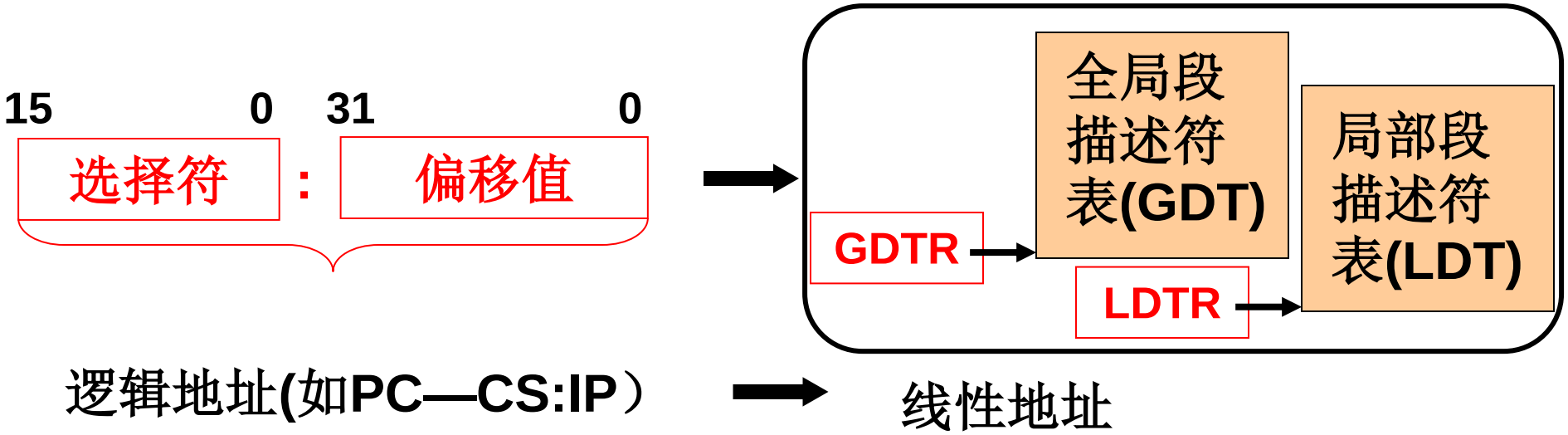
Intel x86的内存管理硬件!

- 实模式称为：段寄存器(CS,DS,ES,SS)，16位
- 保护模式称为：段选择器（6个），16位



段寄存器或段选择器中的内容如果不变，不用查段表，直接用高速缓冲寄存器。

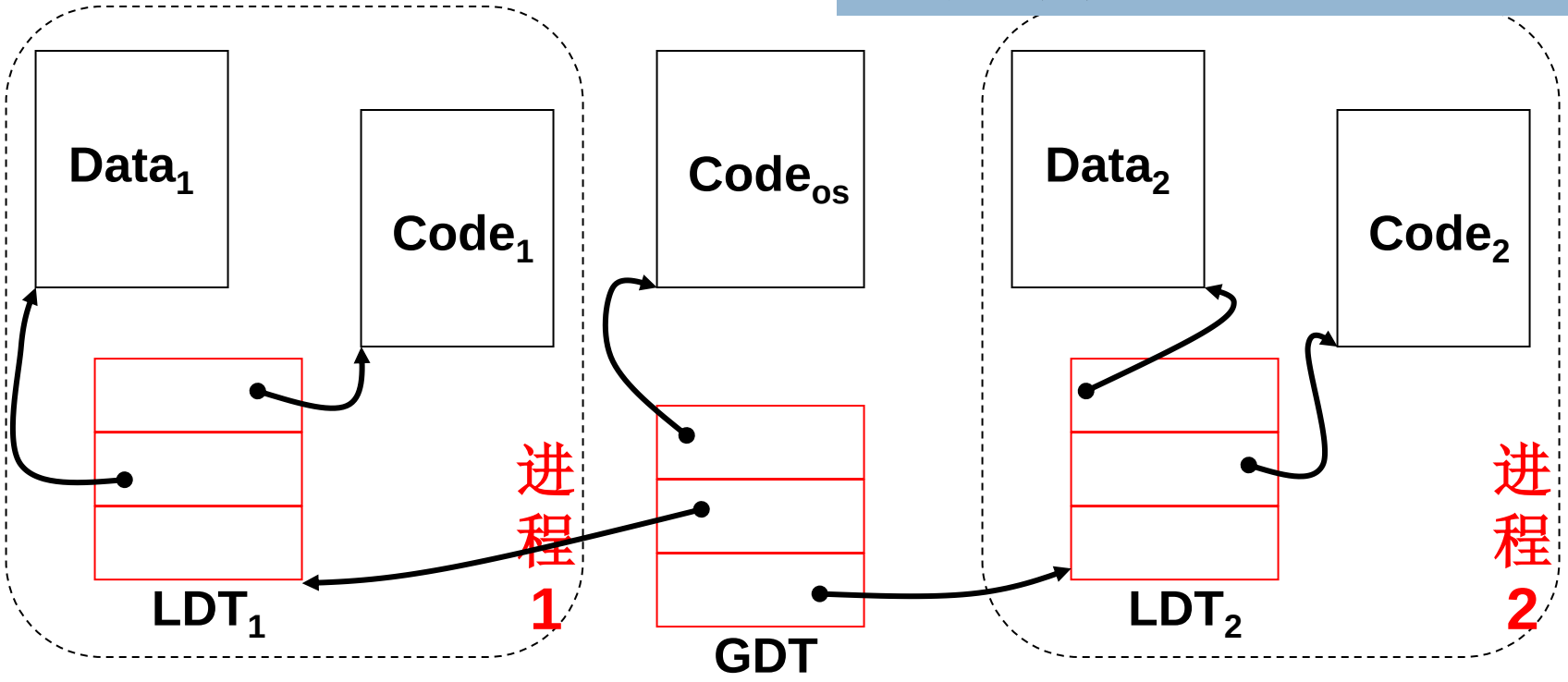
Intel x86的分段硬件



- 保护模式下每个进程可以有自己的段表
- 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

G颗粒度，如为0表示段限长以字节为单位，如果为1表示以4k为单位。
P表示段是否在内存中。
DPL段的特权级。

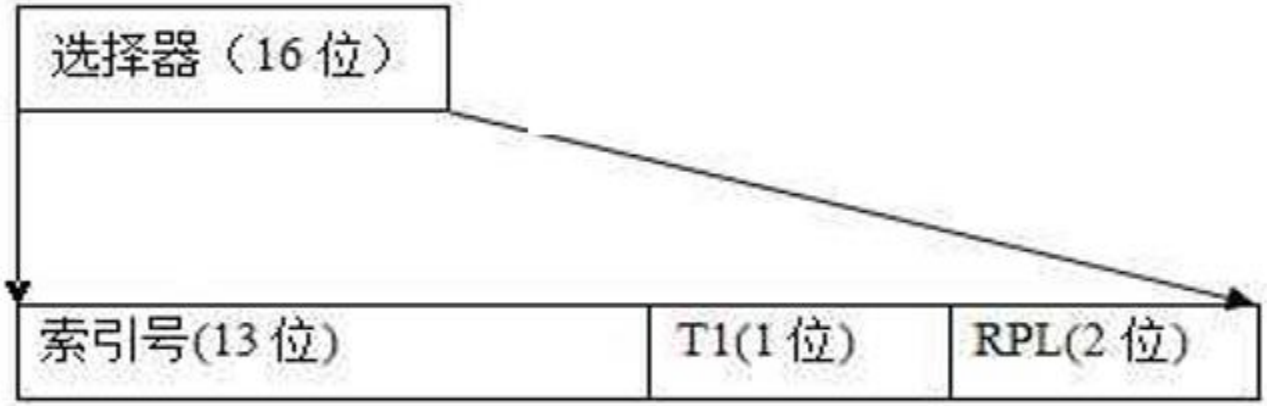


- 段描述符: LDT(GDT)中的表项，全局和局部表，

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

- 保护模式：(CS,DS,ES, FS,GS,SS)
- 存放段描述符在段描述符表中的索引值；
- D3-D15位是索引值,D0-D1位是优先级(RPL)用于特权检查,D2位是描述符表引用指示位TI；
- TI=0指示从全局描述表GDT中读取描述符，TI=1指示从局部描述符中LDT中读取描述符。



备注：
选择器的 T1=0时 指向 GDT ,T1=1 时 指向 LDT
RPL 为请求特权级

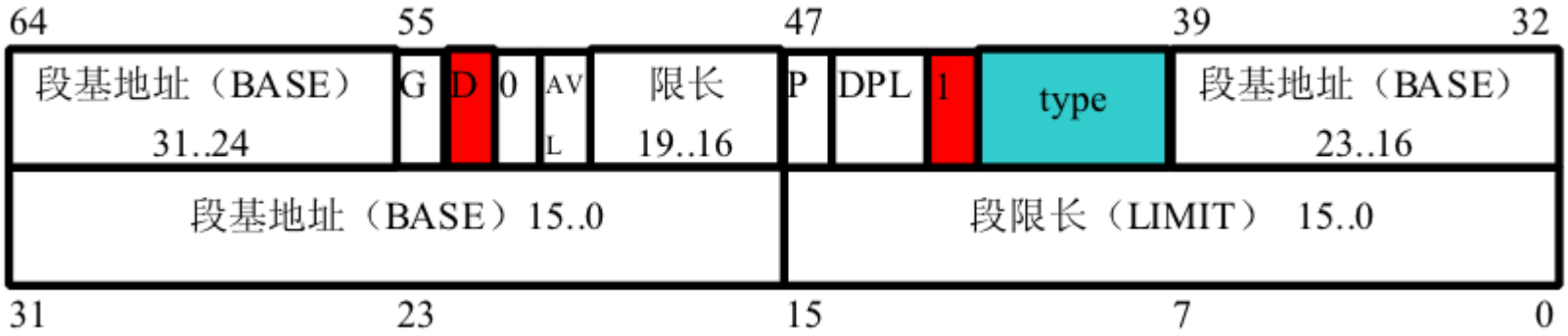
Intel x86的分段硬件

- 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（64位寄存器）: 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 – LDT、GDT

- 段描述符: LDT(GDT)中的表项，type含义



	TYPE			说明
十进制值		E	W	A	数据段
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

32位保护模式下的寻址方式

	TYPE				说明
十进制值		C	R	A	代码段
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。
- 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）。
- OS对每个进程的LDT也看成一个段,用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

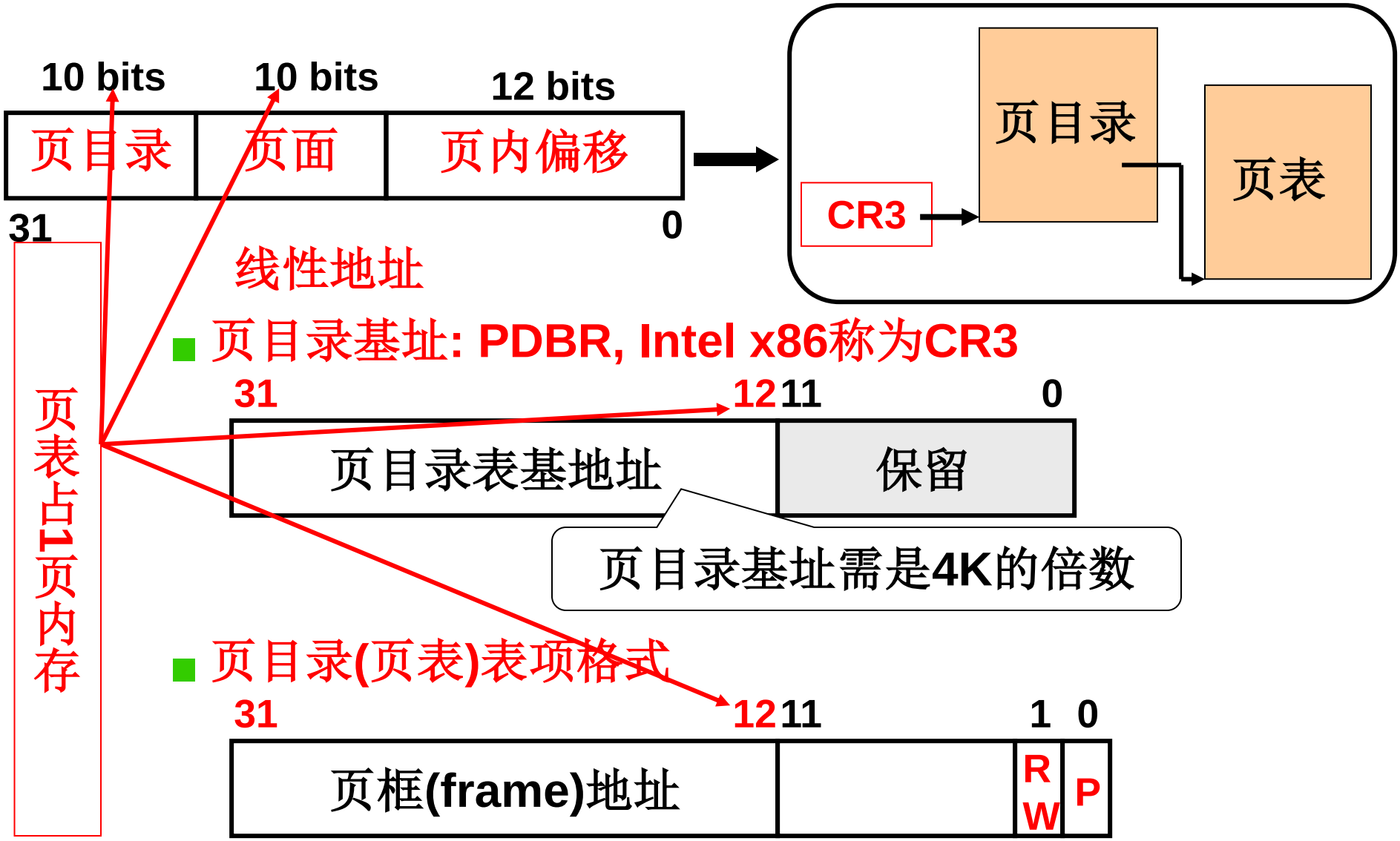
- 任务状态段（Task State Segment）：
  - 每个任务都有一个任务状态段TSS，描述任务状态段的描述符称为TSS描述符
  - 所有任务的TSS描述符均被存放在GDT中。
  - 系统可通过TR寄存器在GDT中找到正在执行的任务的TSS描述符，从而找到相应任务的TSS。
  - 任务状态段TSS中包含：
  - 1.任务的CPU现场（通用寄存器、段选择寄存器、指令指针、标志寄存器等）；
  - 2.特权级分别为0、1、2时的堆栈段选择符和栈顶指针；
  - 3.该任务被调用时，前一个任务TSS的返回连接选择符；
  - 4.I/O允许位图等。

Intel x86的分段硬件linux0.11任务控制块

```
    long alarm;    // 报警定时值
    long utime,stime,cutime,cstime,start_time;
        // utime 用户态运行时间
        // stime 内核态运行时间
        // cutime 子进程用户态运行时间
        // cstime 子进程内核态运行时间
        // start_time 进程开始运行时刻
    unsigned short used_math;    // 标志，是否使用了 387 协处理器
/* ----------------------------file system info---------------------- */
    int tty;        // 进程使用 tty 的子设备号，-1 表示没有使用
    unsigned short umask;   //文件创建属性屏蔽码
    struct m_inode * pwd;   // 当前工作目录的 i 节点
    struct m_inode * root; // 根目录的 i 节点
    struct m_inode * executable;     // 可执行文件的 i 节点
    struct file * filp[NR_OPEN];     // 进程使用的文件
    unsigned long close_on_exec; // 执行时关闭文件句柄位图标志

/*-------- ldt for this task 0 - zero 1 - cs 2 - ds&ss ----------*/
    struct desc_struct ldt[3];//本任务的 ldt 表，0-空，1-代码段，2-数据和堆栈段
/* ----------------------tss for this task ----------------------*/
    struct tss_struct tss;       // 本任务的 tss 段
};
```

Intel x86的分页硬件



Intel x86的内存管理硬件!

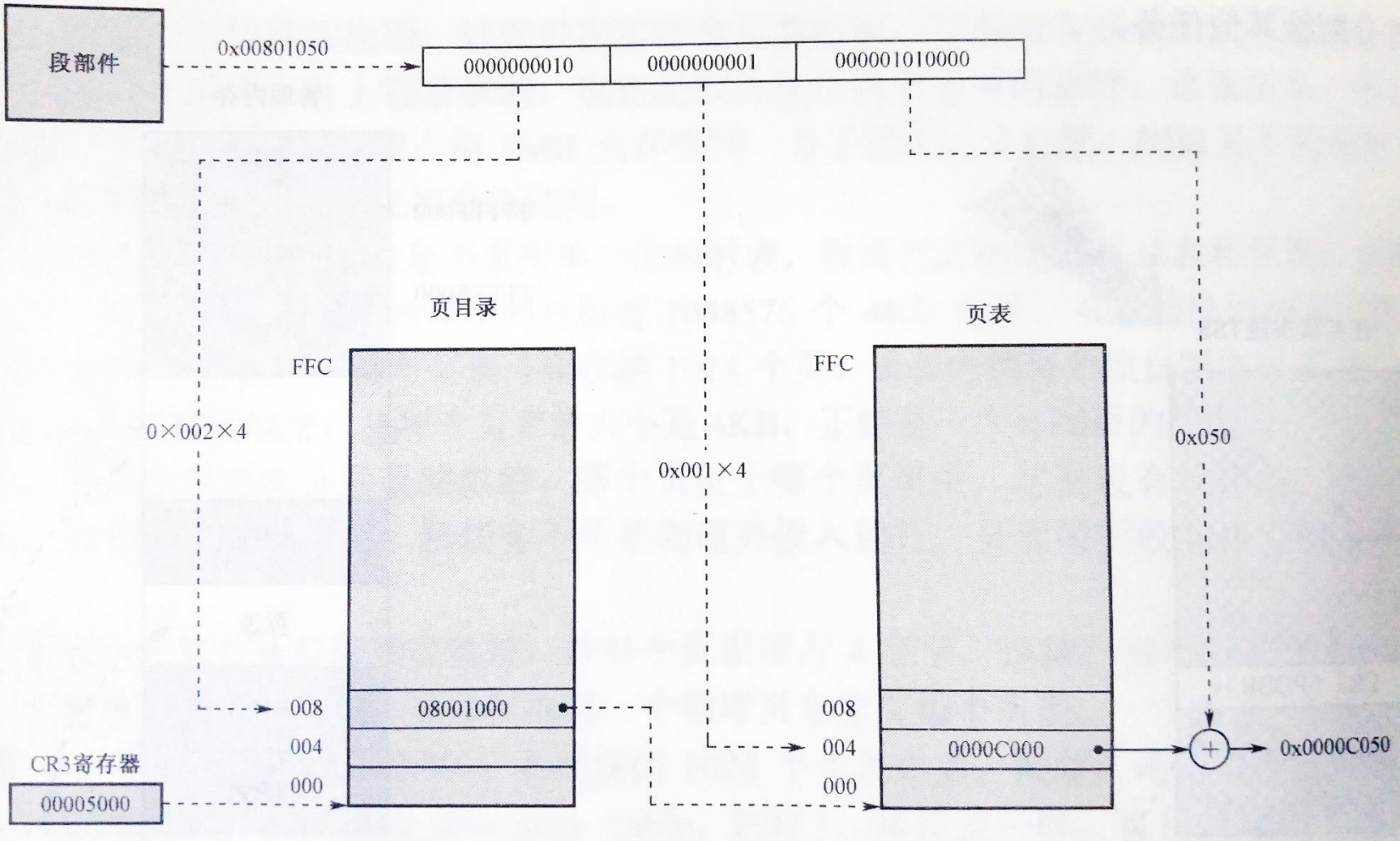


图 16-8　页部件把线性地址转换为物理地址的例子

Intel x86的内存管理硬件!

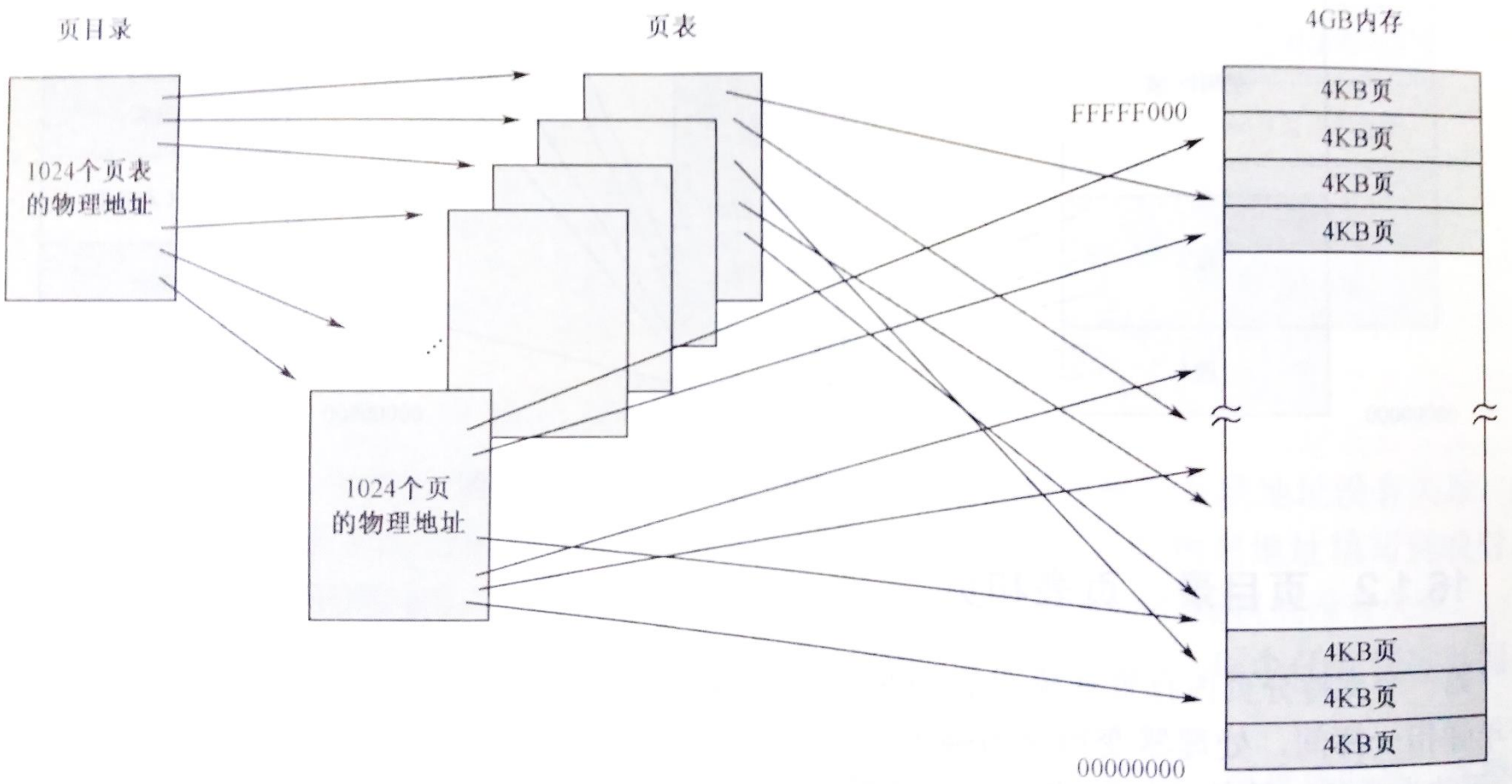


图 16-6　页目录、页表和页的对应关系

内存管理总结

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