操作系统原理

黄俊杰

2024年9月

第5章 存储管理

内存的基本目的是什么?

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);
}
```

```
Liaos-MacBook-Air:Programs liaojianwei$ ./sum 10000
50005000
Liaos-MacBook-Air:Programs liaojianwei$ ./sum 20000
200010000
Liaos-MacBook-Air:Programs liaojianwei$ ./sum 30000
450015000
Liaos-MacBook-Air:Programs liaojianwei$
```

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

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

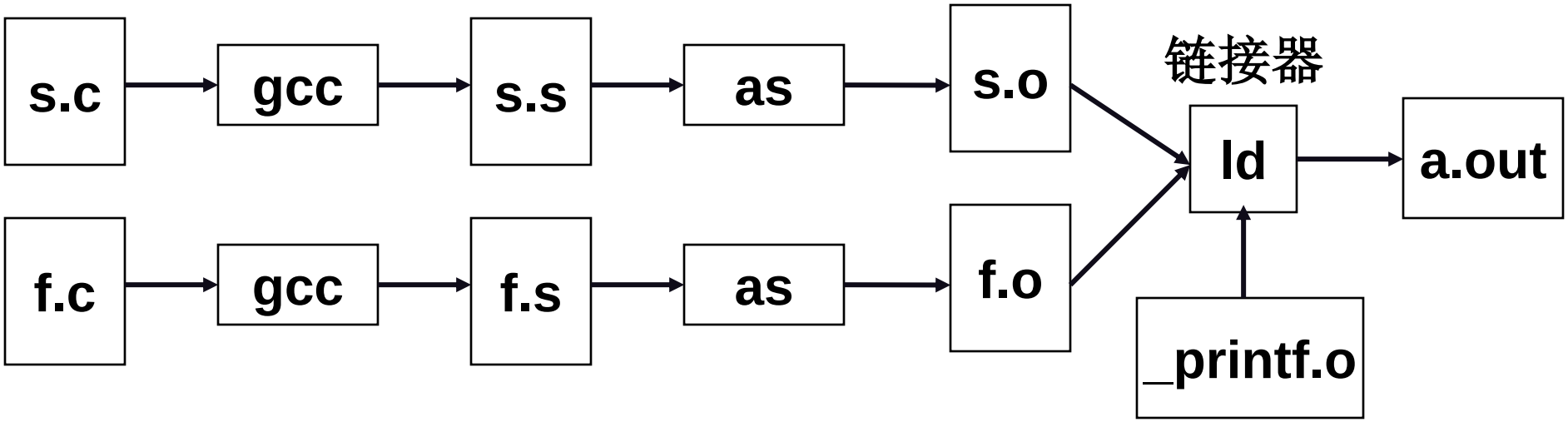
printf.o

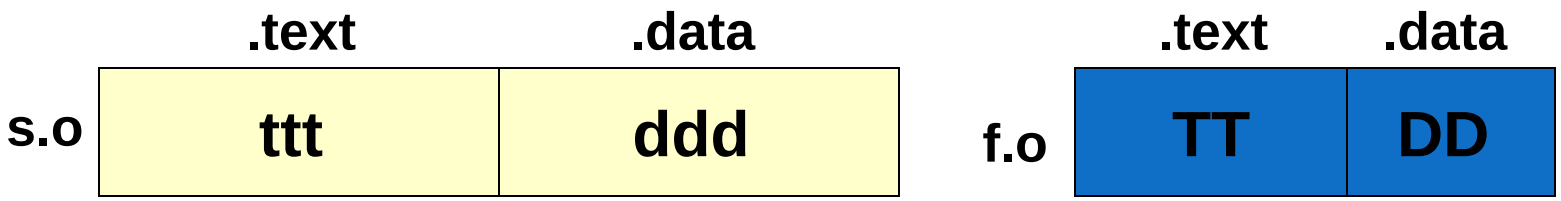
ld

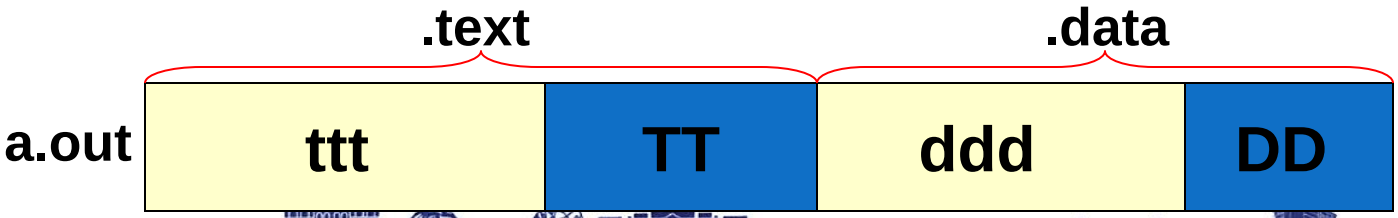
```
0   _entry: //入口地址
      call 40
40  _main:
      mov [300],  0
    1:
      j> 80, [_i], [_to]
80  2:
      call 100
      ret

100 _printf:
      …

      …
300 _sum:  .int  0
```

许多东西有待明确...

- 第二步: 链接——从汇编到可执行程序

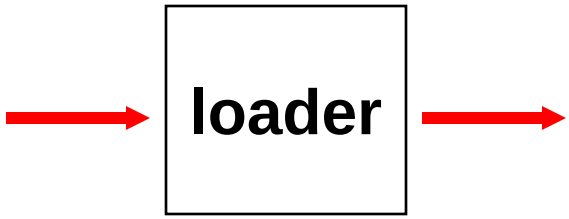


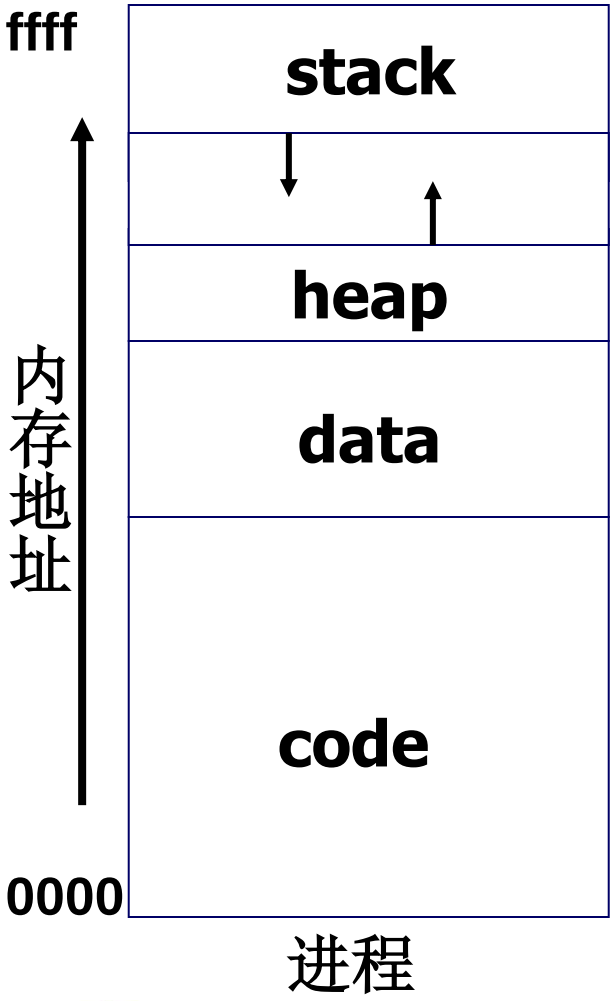




现在还是程序，不是进程

```
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
```

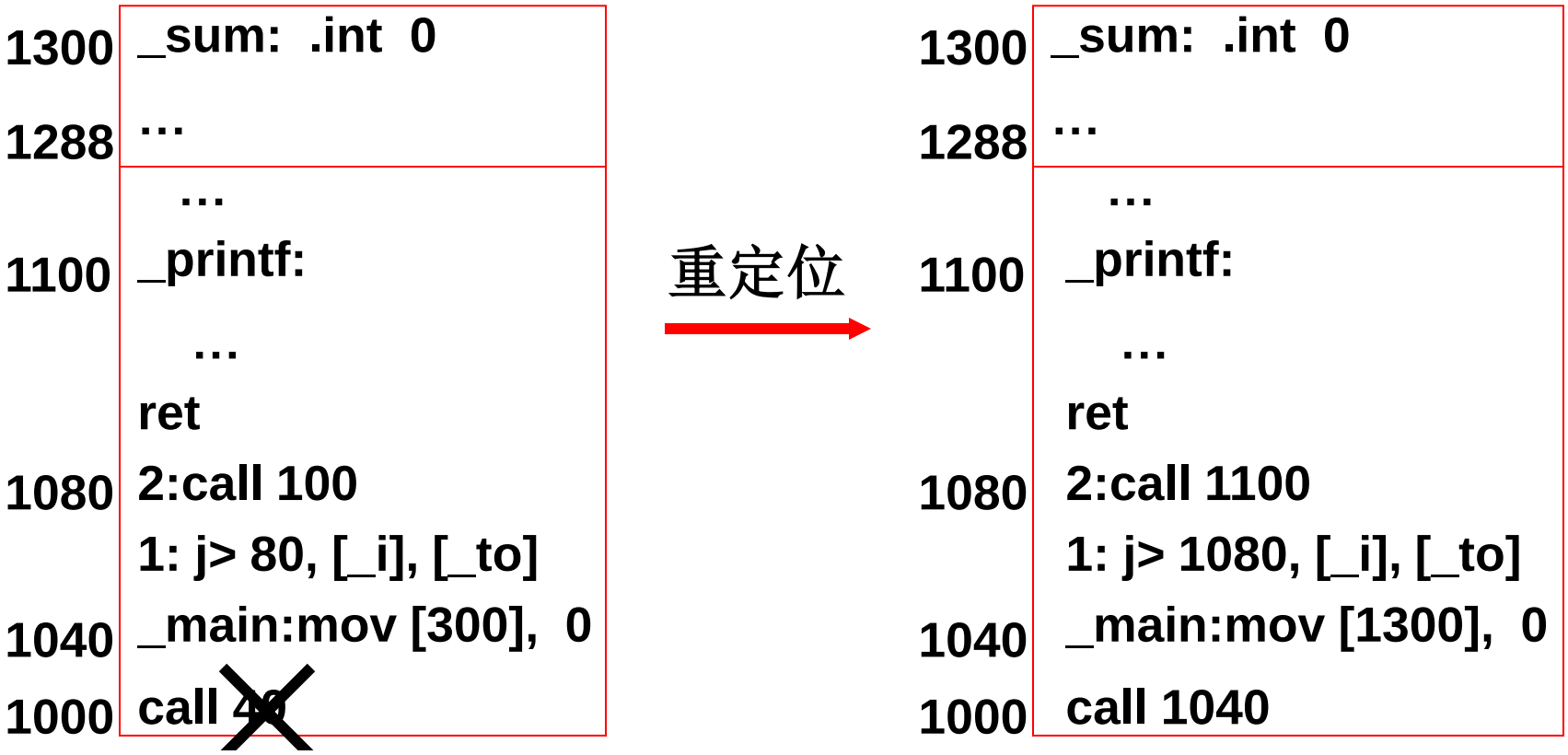




需要重定位!

1000是由硬件和操作系统决定的!

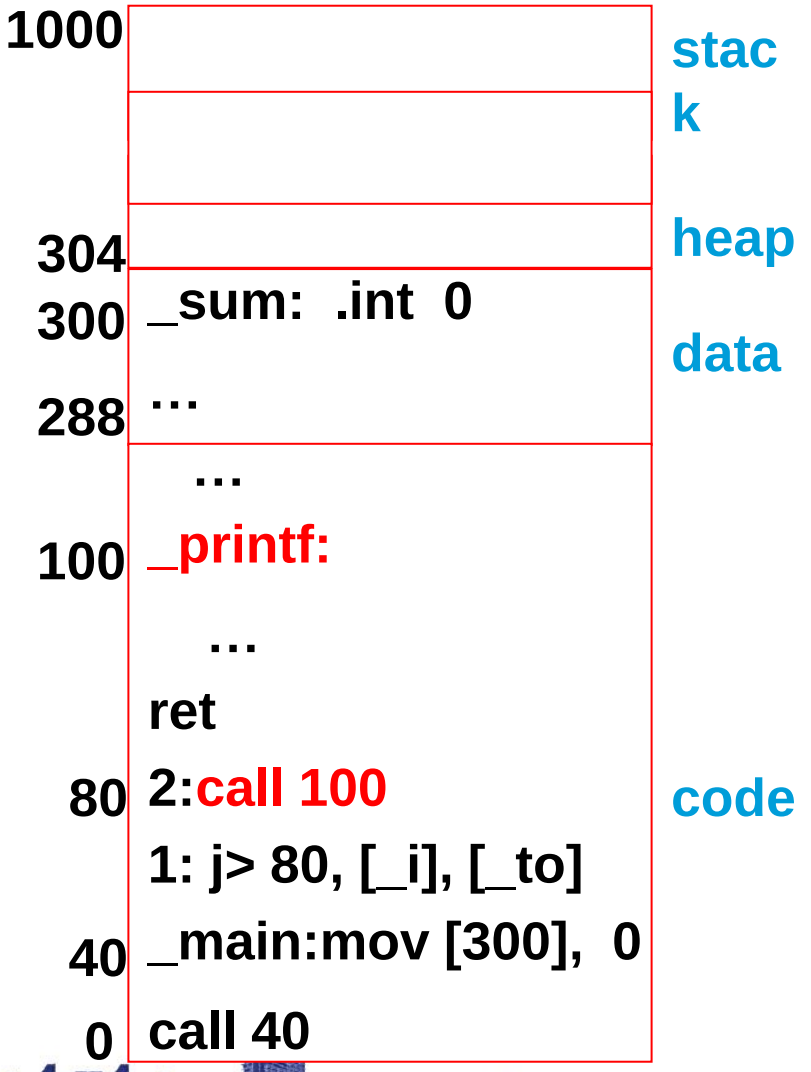
- 假设内存从地址1000以后是可以使用的



- 重定位: 为执行程序而对其中出现的地址所做的修改

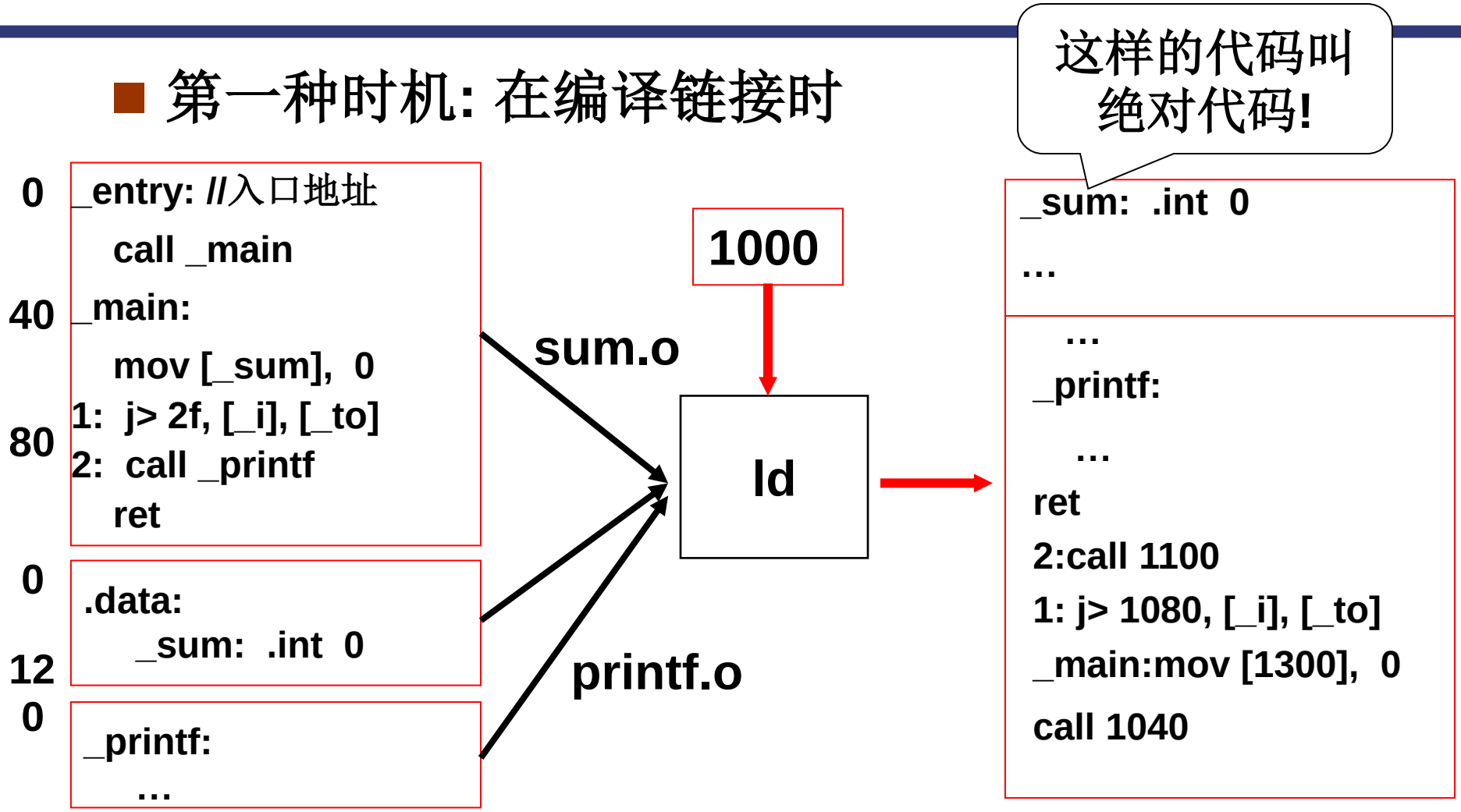
程序可以执行了吗?

- 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



重定位：静态重定位

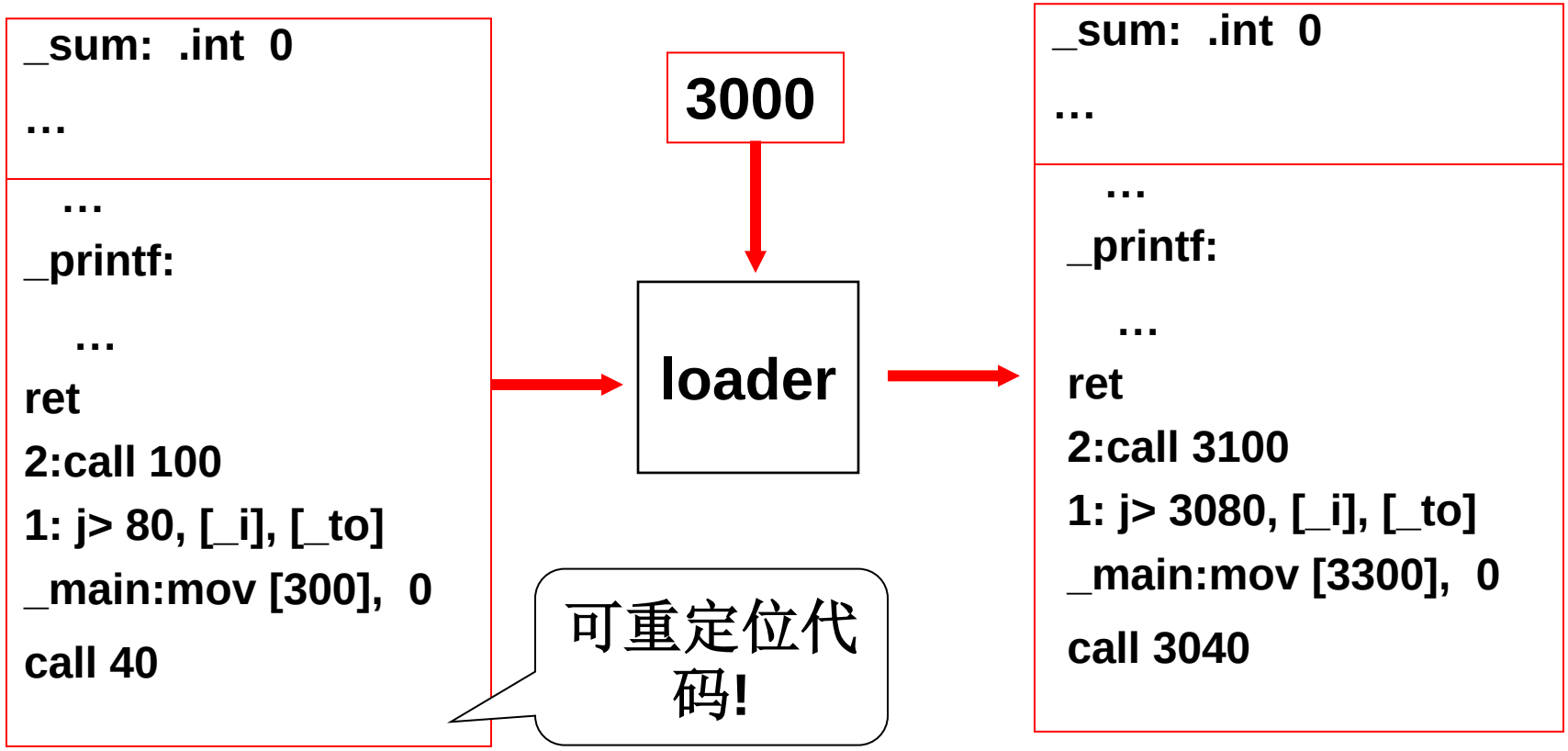
- 第一种时机: 在编译链接时



- 绝对代码: 这样的代码只能放在事先确定的位置上

并发 ⇒ 多个程序同时在内存中

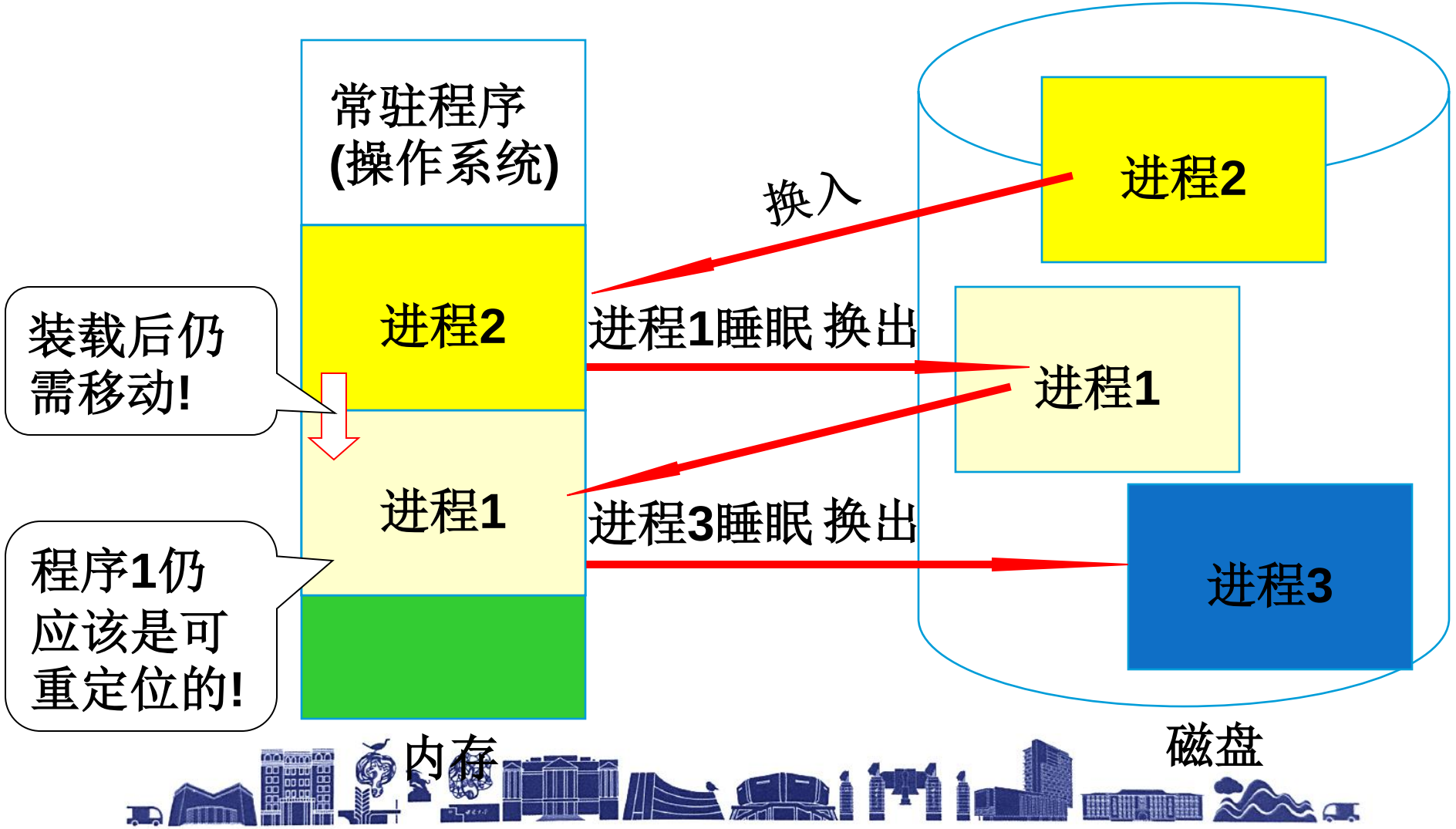
- 分别用1000，2000，… 吗？ 第二种时机: 载入时



- 装载时的重定位仍然存在缺点… 一旦载入不能移动

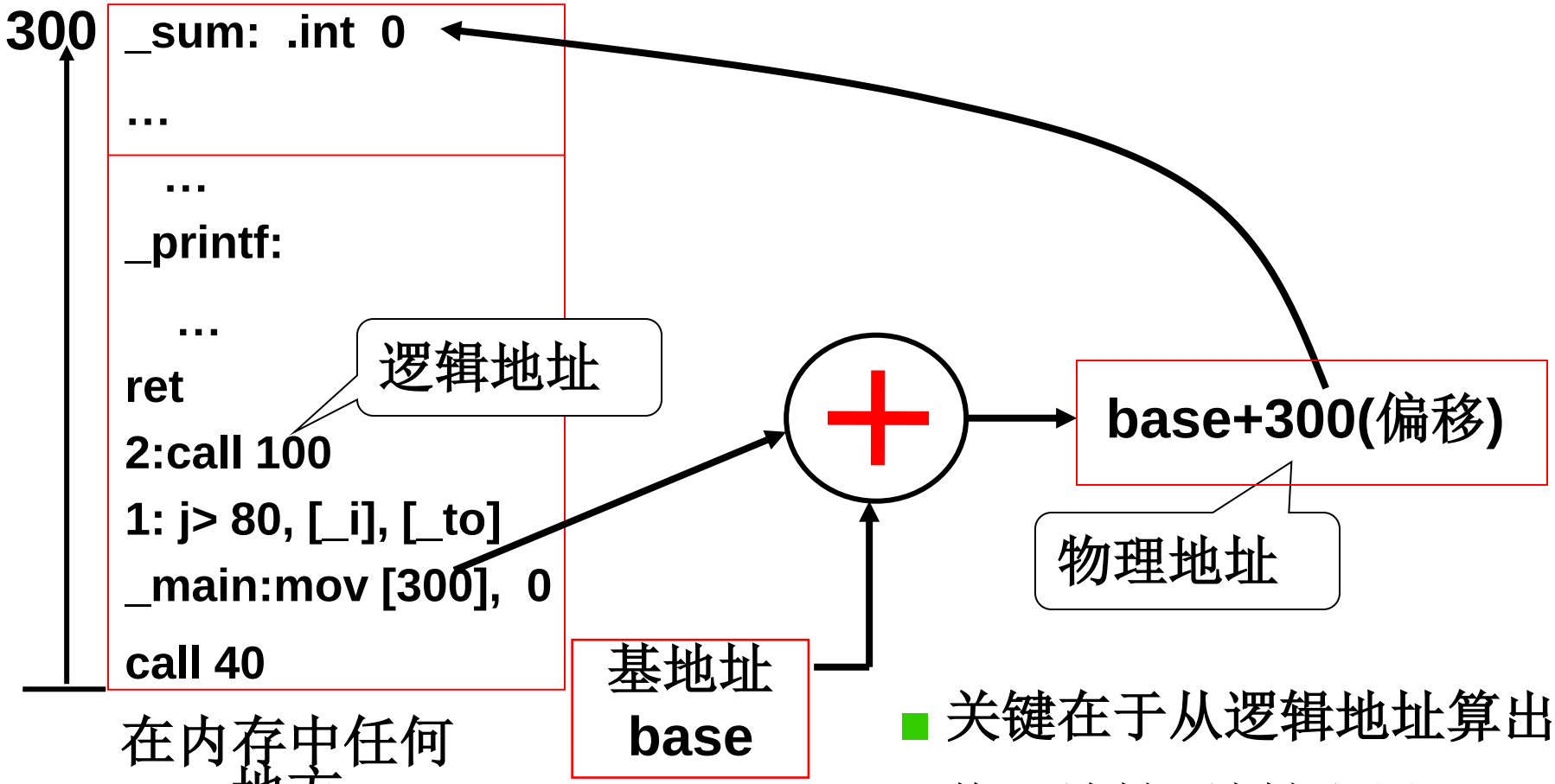
移动也是很有必要的!

- 一个重要概念: 交换(swap) 能让更多的进程并发



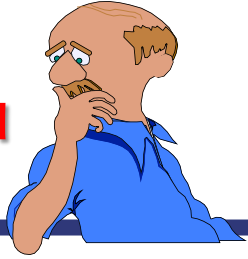
重定位：运行时重定位（动态重定位）

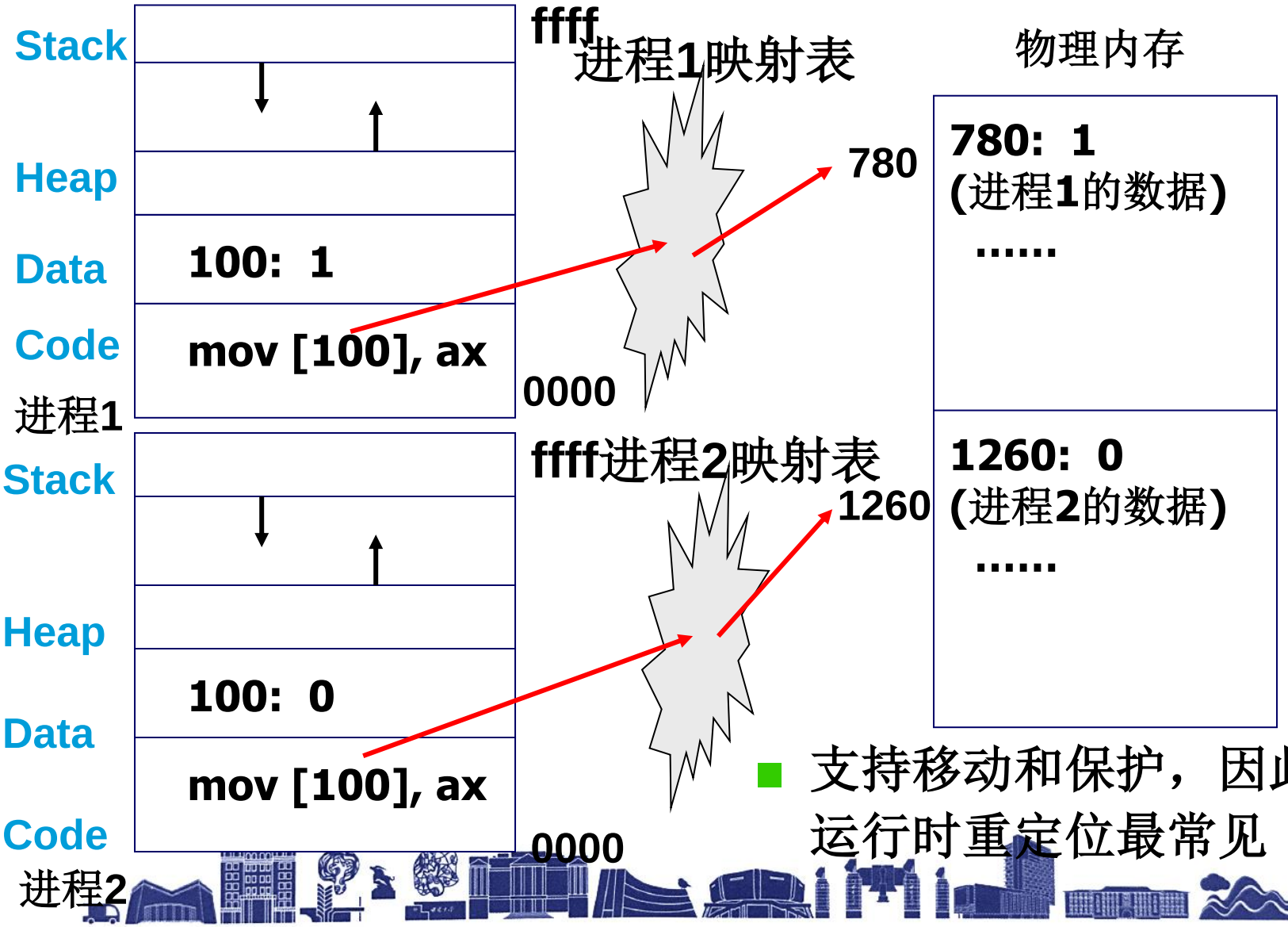
- 内存中的代码总是可重定位的!



- 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



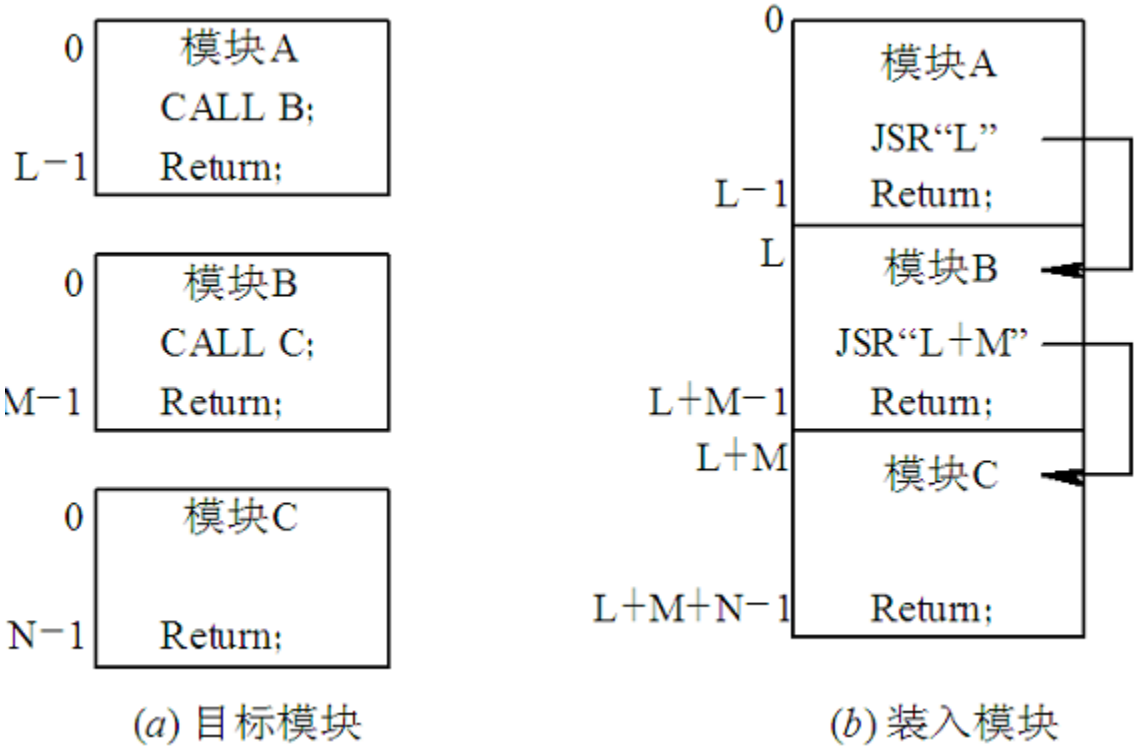


- 支持移动和保护，因此运行时重定位最常见

稍稍复习一下链接

程序的链接：1. 静态链接方式

- 两个问题需解决：相对地址的修改、变换外部调用符号



(*a*) 目标模块　　(*b*) 装入模块

静态链接的优点和缺点

- 这种链接所形成的一个完整的装入模块称为可执行文件，简单
- 通常不再拆开它，要运行时可直接装入内存
- 若要修改或更新其中的某个目标模块，则要求重新打开装入模块，麻烦

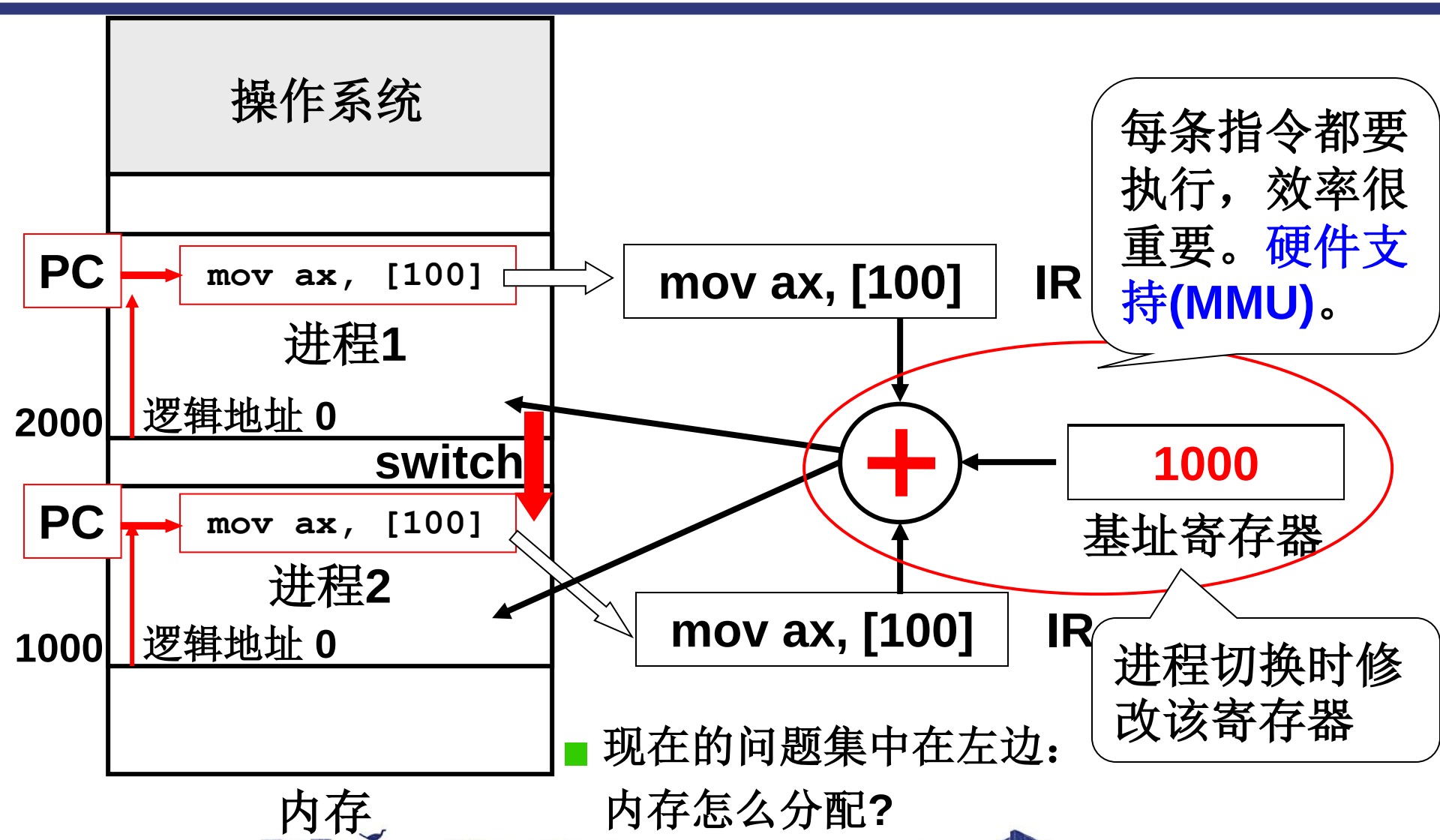
2. 装入时动态链接

- 装入目标模块时，边装入边链接。
- 装入时动态链接方式有以下优点：
  - 便于修改和更新。
  - 便于实现对目标模块的共享。

3. 运行时动态链接

- 将对某些模块的链接推迟到执行时才执行，即，在执行过程中，当发现一个被调用模块尚未装入内存时，立即由OS去找到该模块并将之装入内存， 把它链接到调用者模块上。
- 凡在执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，这样不仅可加快程序的装入过程，而且可节省大量的内存空间。

整理一下思路...



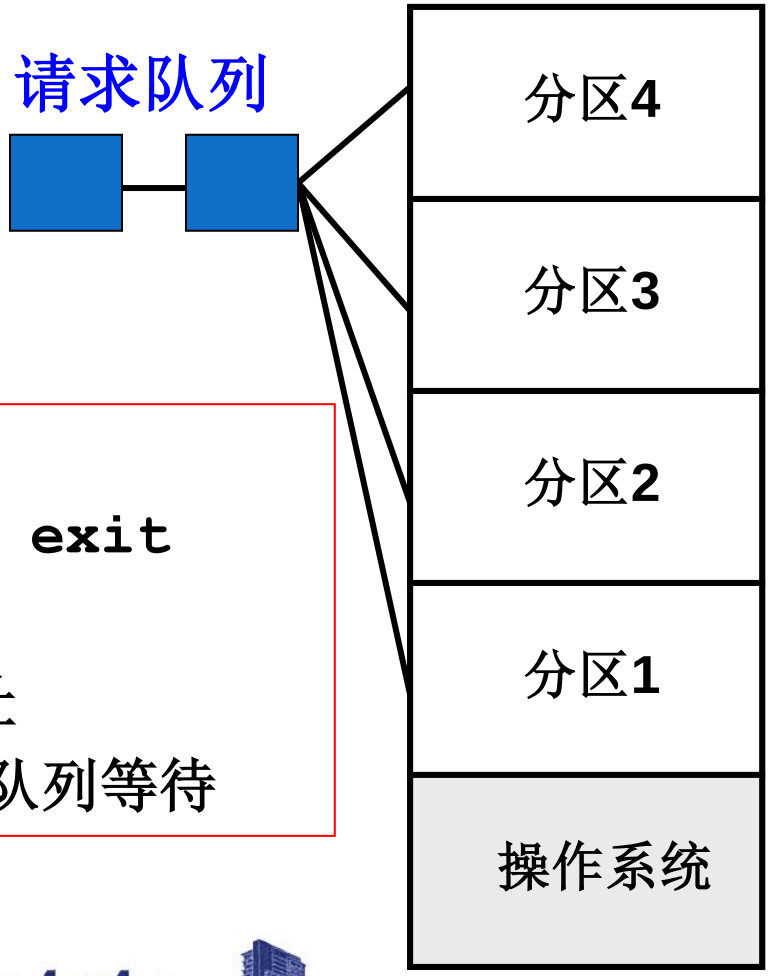
- 现在的问题集中在左边：内存怎么分配?

内存分配方案!

从最简单的分配方案开始!

（1）连续内存分配 – 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

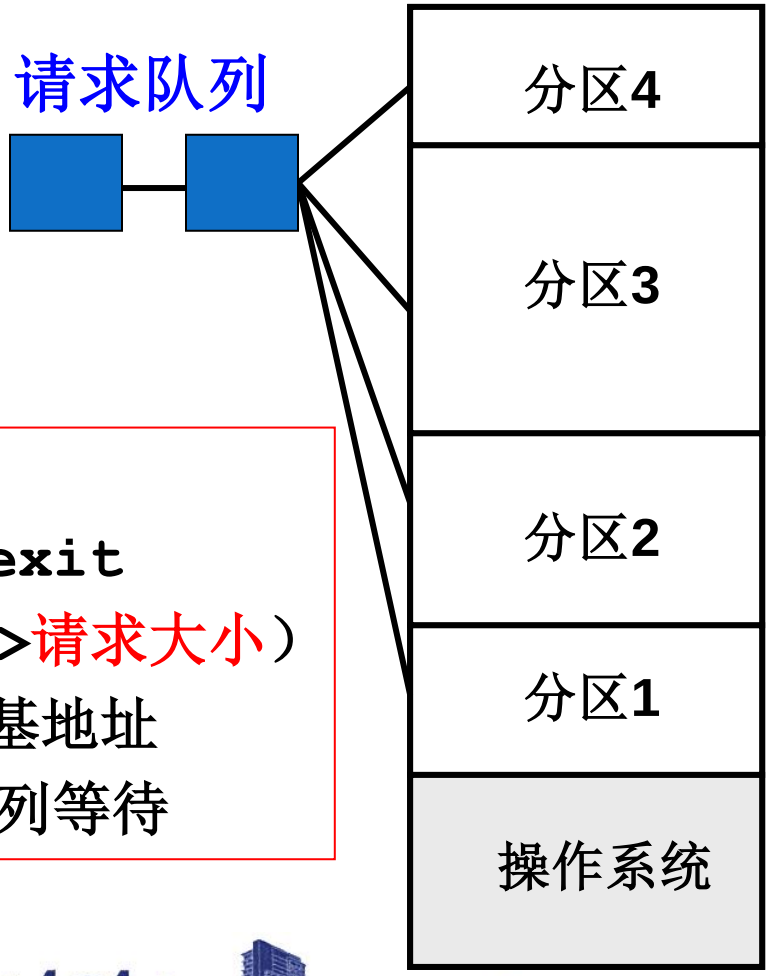


内存分配算法　//进程创建时

1. if(请求大小 > 分区大小) exit
2. 找出空闲的内存分区
3. 如果有，返回分区i的基地址
4. 否则，将请求进程加入请求队列等待

（2）等长分区到不等长分区

- 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

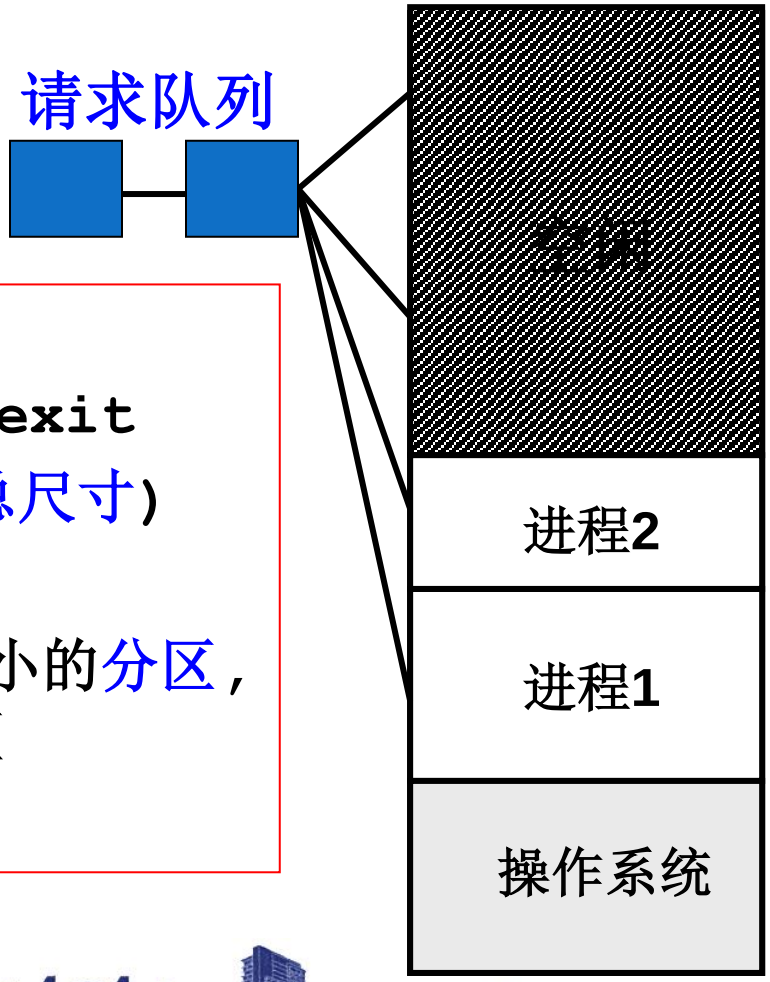


内存分配算法　//进程创建时

1. if(请求大小 > 最大分区) exit
2. 找出空闲内存分区（分区大小>请求大小）
3. 如果有，返回[最小]分区i的基地址
4. 否则，将请求进程加入请求队列等待

（3）固定分区到可变分区

- 合理的方法应该是根据孩子饥饿程度来分割
  - 根据请求大小进行动态分割



内存分配算法　//进程创建时

1．if(请求大小 > 内存大小)exit

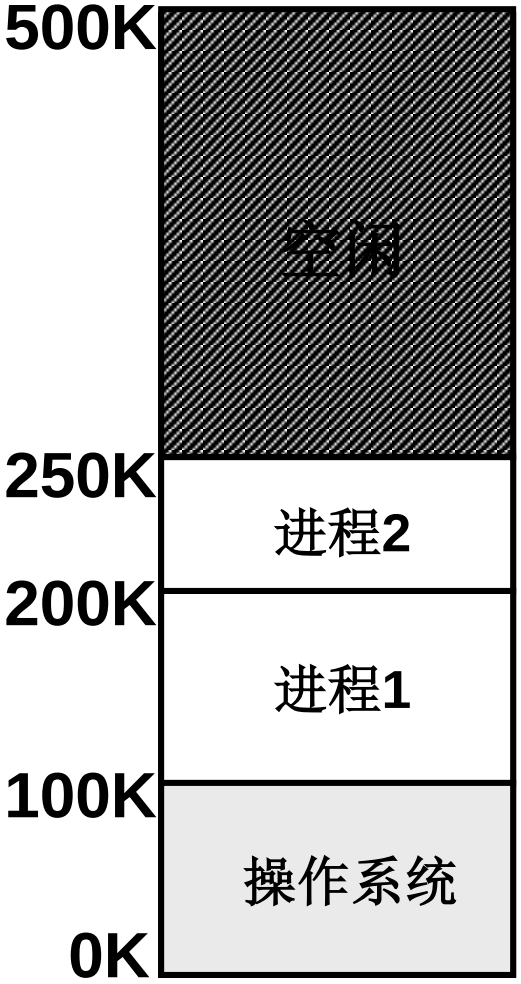
2．if(请求大小 > 空闲空间总尺寸)

将请求进程加入请求队列等待

3．从空闲分区划出一个请求大小的分区，并返回其基地址　//哪个空闲分区

4．修改分区数据结构

可变分区的数据结构



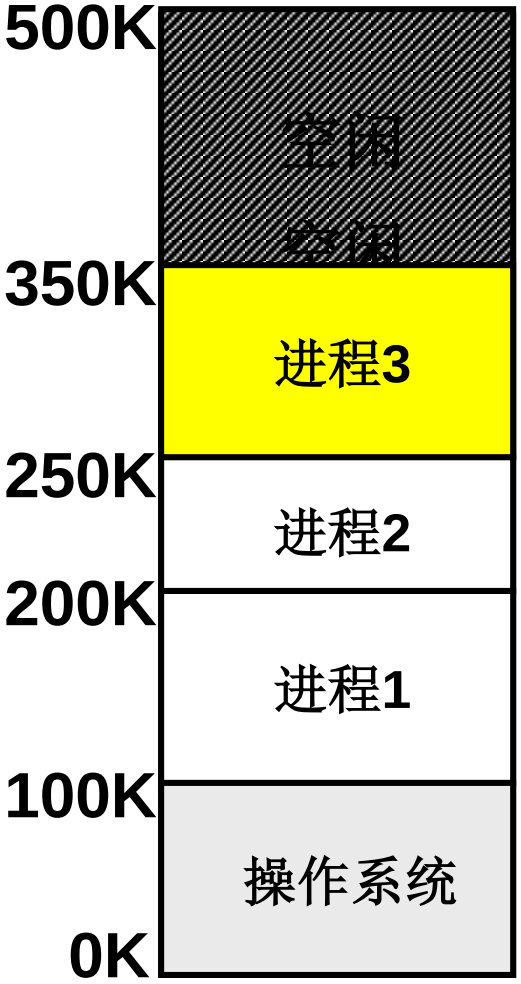
空闲分区表

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

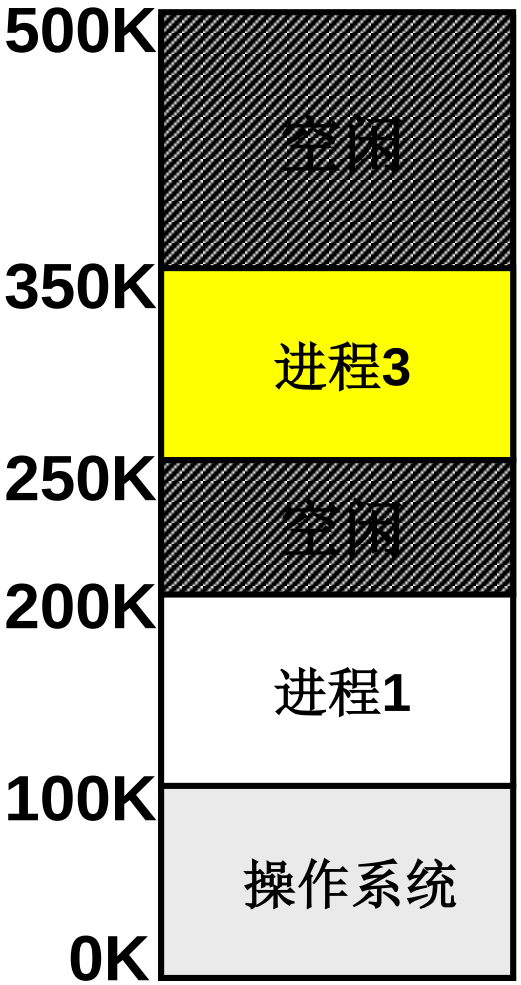
空闲分区表

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

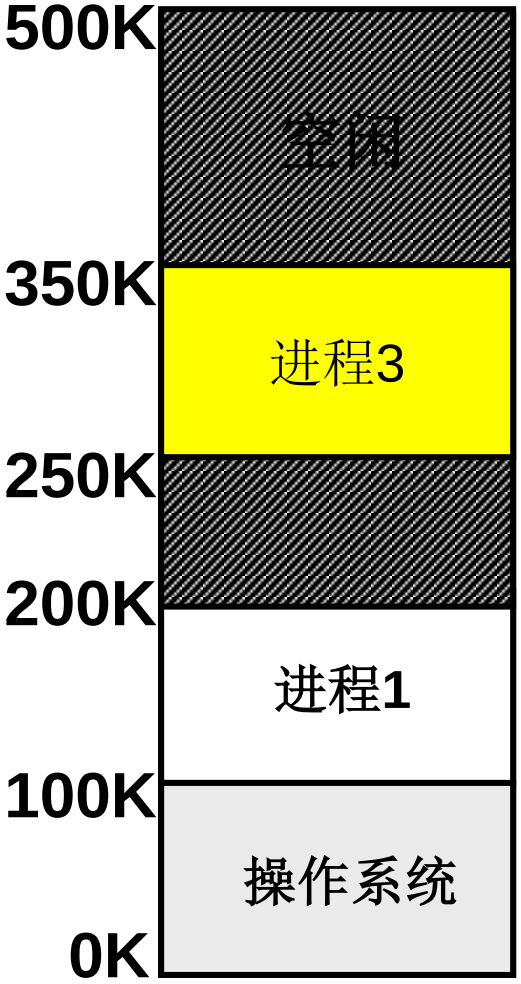
空闲分区表

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
200K	150K	P2
250K	100K	P3

可变分区数据结构的变化(3)



- 进程3执行完毕，释放内存
- 合并空闲分区

空闲分区表

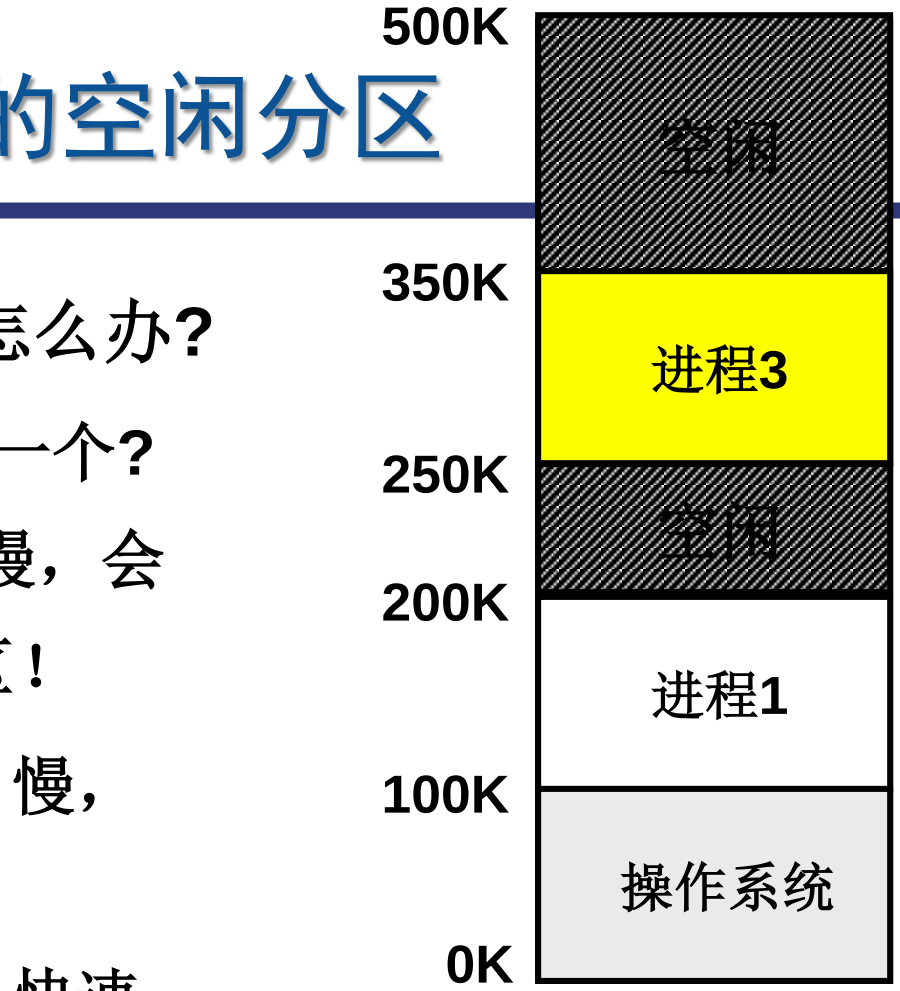
始址	长度
~~350K~~ 200K	~~150K~~ 50K
200K	50K

已分配分区表

始址	长度	标志
0K	100K	OS
100K	100K	P1

（4）再谈分配 – 合适的空闲分区

- 发起请求reqSize=40K怎么办?
  - 有2个空闲分区，选哪一个?
  - 最佳适配: (200,50)，慢，会产生许多小的空闲分区！
  - 最坏适配: (350,150)，慢，没有大的空闲分区！
  - 首先适配: (350,150)，快速



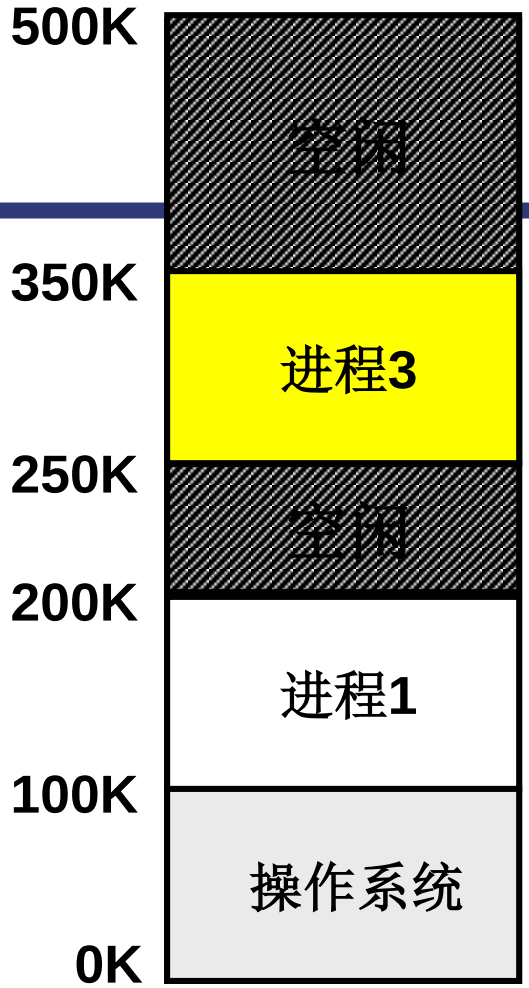
仍然需要根据应用的特点来决定选取哪种策略

空闲分区表

始址	长度
350K	150K
200K	50K

（5）再谈分配 – 内存紧缩

- 发起请求需要160K内存怎么办?
  - 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100M/1秒，则1G内存的紧缩时间为10秒。



空闲分区表

始址	长度
350K	150K
200K	50K

该值表明连续分配技术不合适!

（6）内存碎片

- 内存碎片是指描述一个系统中所有不可用的空闲内存区域（事实上，磁盘也存在碎块问题！）。
  - 内部碎片：分配给进程的内存空间中未被使用的内存部分。
  - 外部碎片：系统中尚未分配给任何进程，但无法利用的小存储区域
- 解决碎片问题
  - 内存紧缩？
  - 其它更为有效地的内存管理机制？

解决外部碎片&
时间成本太高！

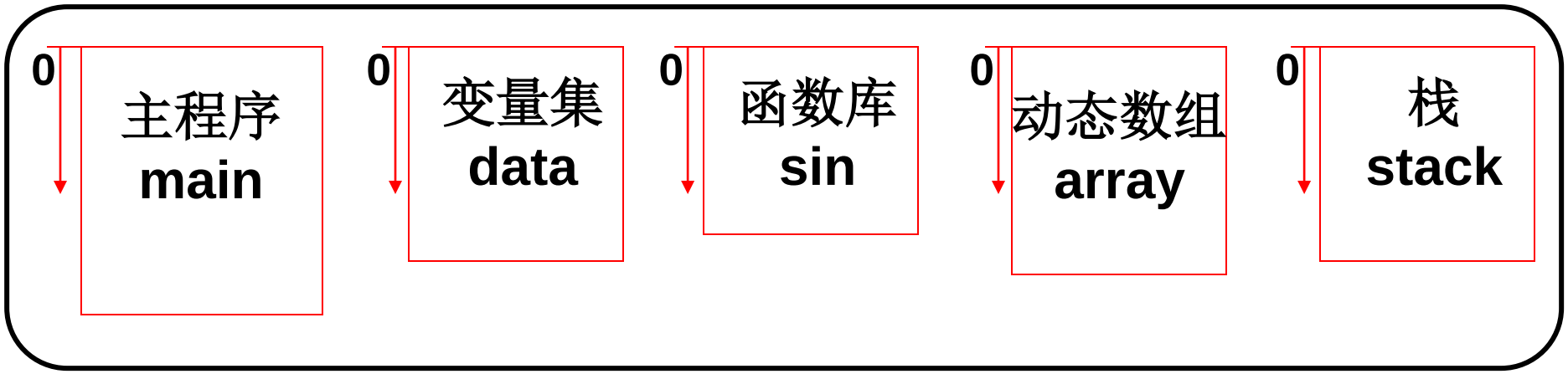
固定和可变分区存储管理方案小结

- 连续分区管理的缺点：
  - 内存利用率不高，存在严重的碎片
    - 内存碎片是指在系统中不可用的空闲内存空间
  - 由于要求连续存储，并占有一个独立分区，故进程的大小受分区大小的限制（固定分区），或受内存可用空间的限制（可变分区）。
    - 内存紧缩等技术只是对分区管理的有限改进。
  - 不利于程序段和数据的共享

分段(Segmentation)!

程序员眼中的程序

- 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

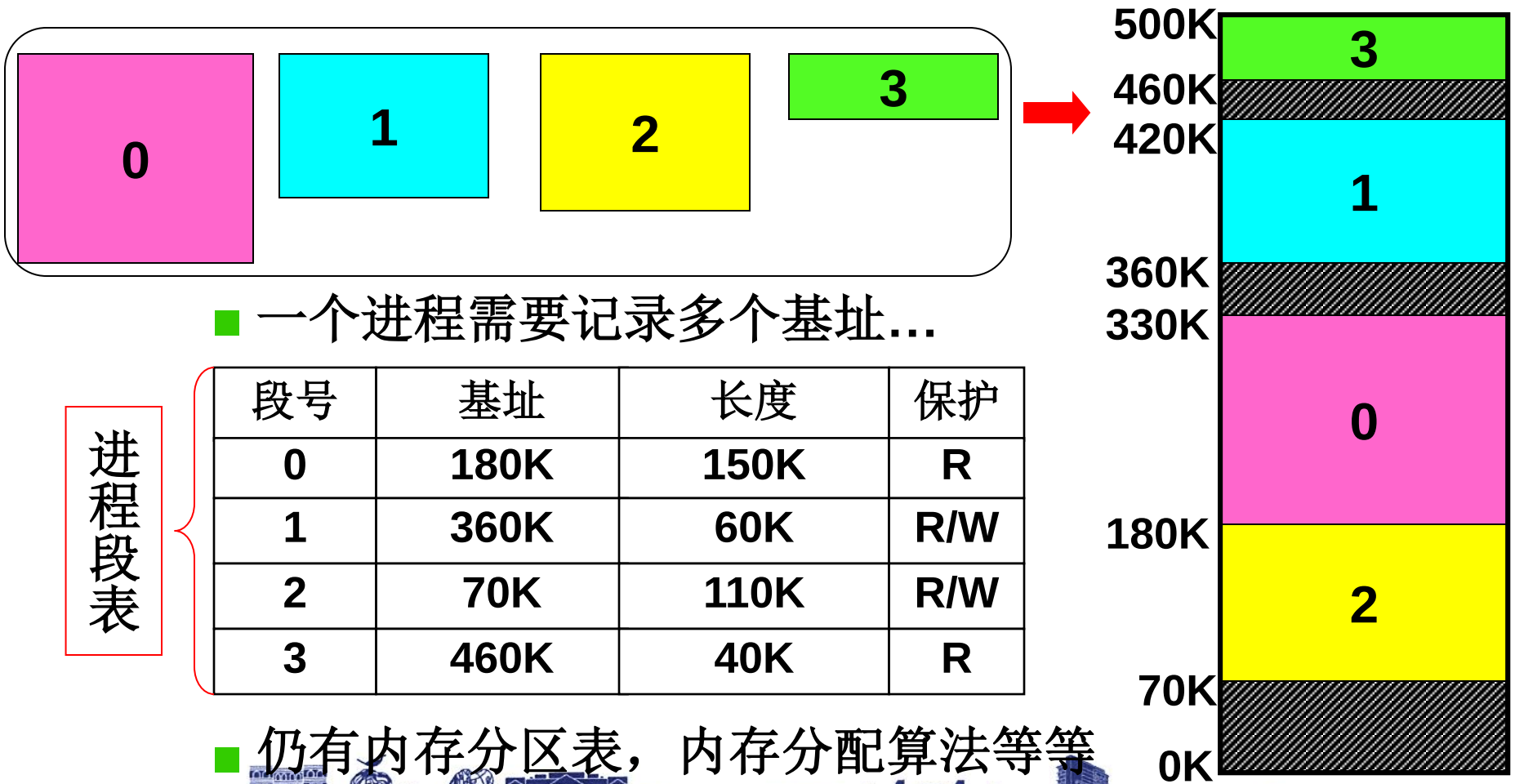
- 程序员怎么定位具体指令(数据): <段号, 段内偏移>

如mov [es:bx], ax

- 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

- 分段制造了二维空间，而内存是一维的



- 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

- 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等

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

异常

<

800>240

逻辑地址

0x4240 mov ax, _var 0x0:240

... ...

取出指令

0x4000+240

物理地址

- 同样可算出其它变更的物理地址

分段技术总结

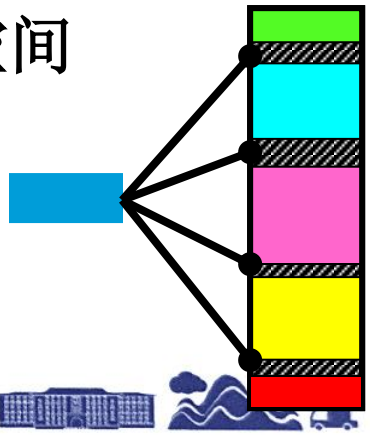
- 实现机理
  - 程序员将程序按含义分成若干部分，即分段
  - ld从0开始编址每个段(链接速度会很快)
  - 创建进程(分别载入各个段)时，建立进程段表
  - 内存仍用可变分区进行管理，载入段时需调分配算法
  - PC及数据地址要通过段表算出物理地址，到达内存
  - 进程切换时，进程段表也跟着切换

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

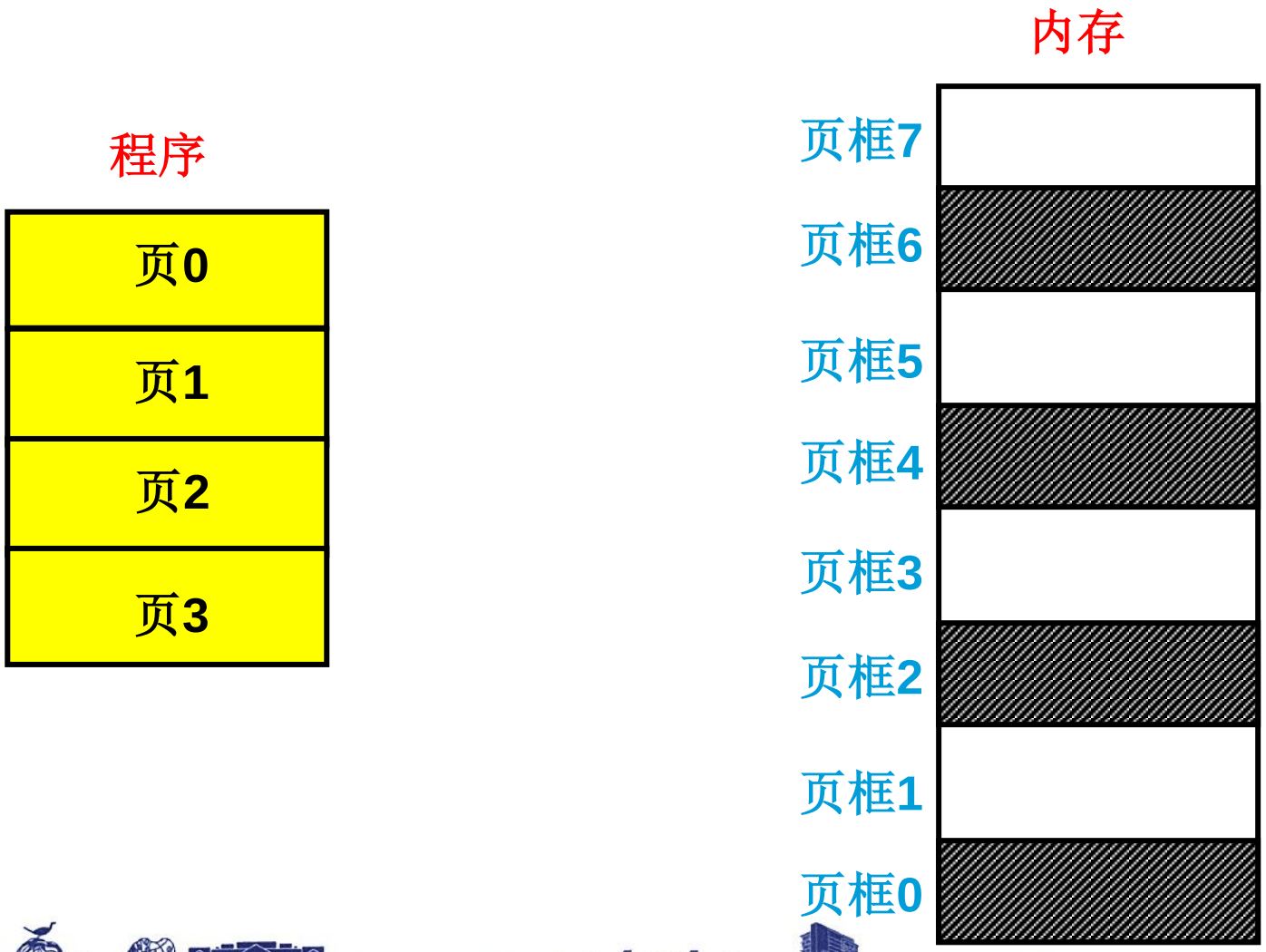
- 优点：符合人的习惯，程序员感觉舒服
  - 不同的段有不同的含义，可区别对待
  - 每个段独立编址，编程容易(如果是一个大的一维地址空间，程序员一会儿就糊涂了!) (分治)
- 缺点：靠近了我们，必然会远离…　空间低效
  - 空间预留；空闲空间很大却不能分配；内存紧缩
  - 著名的碎片概念: 空闲的却用不上的空间

内部碎片　外部碎片

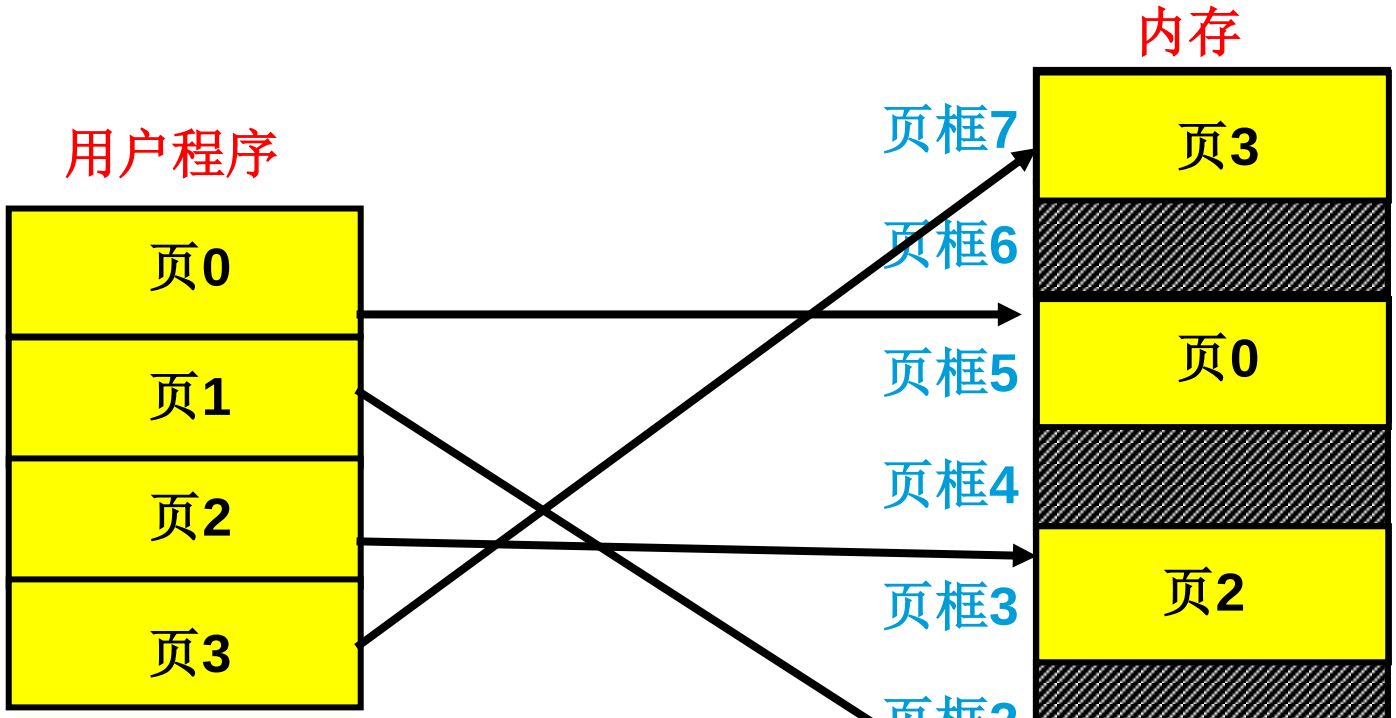


分页(Paging)!

从连续到离散



程序装入内存（页→页框）

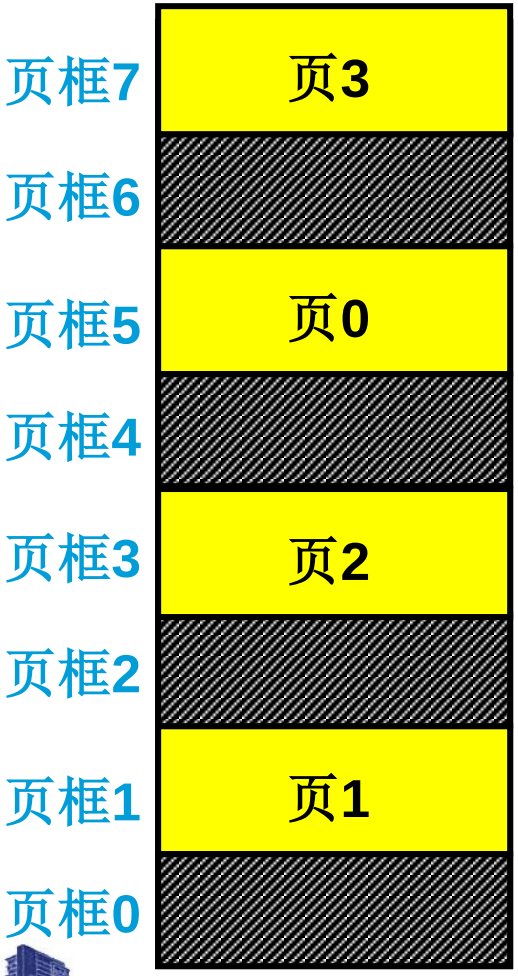


进程页表

页号	页框号	保护
0	5	R
1	1	R/W
2	3	R
3	7	R

分页机制中的页表

- 和分段类似，分页依靠页表结构



程序中的页号！

如何将这样的逻辑地址转换成的物理地址呢？

你还记得页表么？

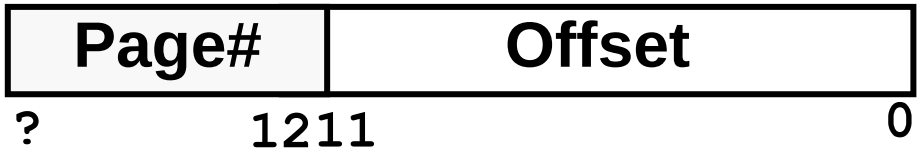
分页的地址翻译

- 一个实例!

执行该指令需要查两次页表!

```
mov ax, _var 0x240
…            …
```

逻辑地址格式

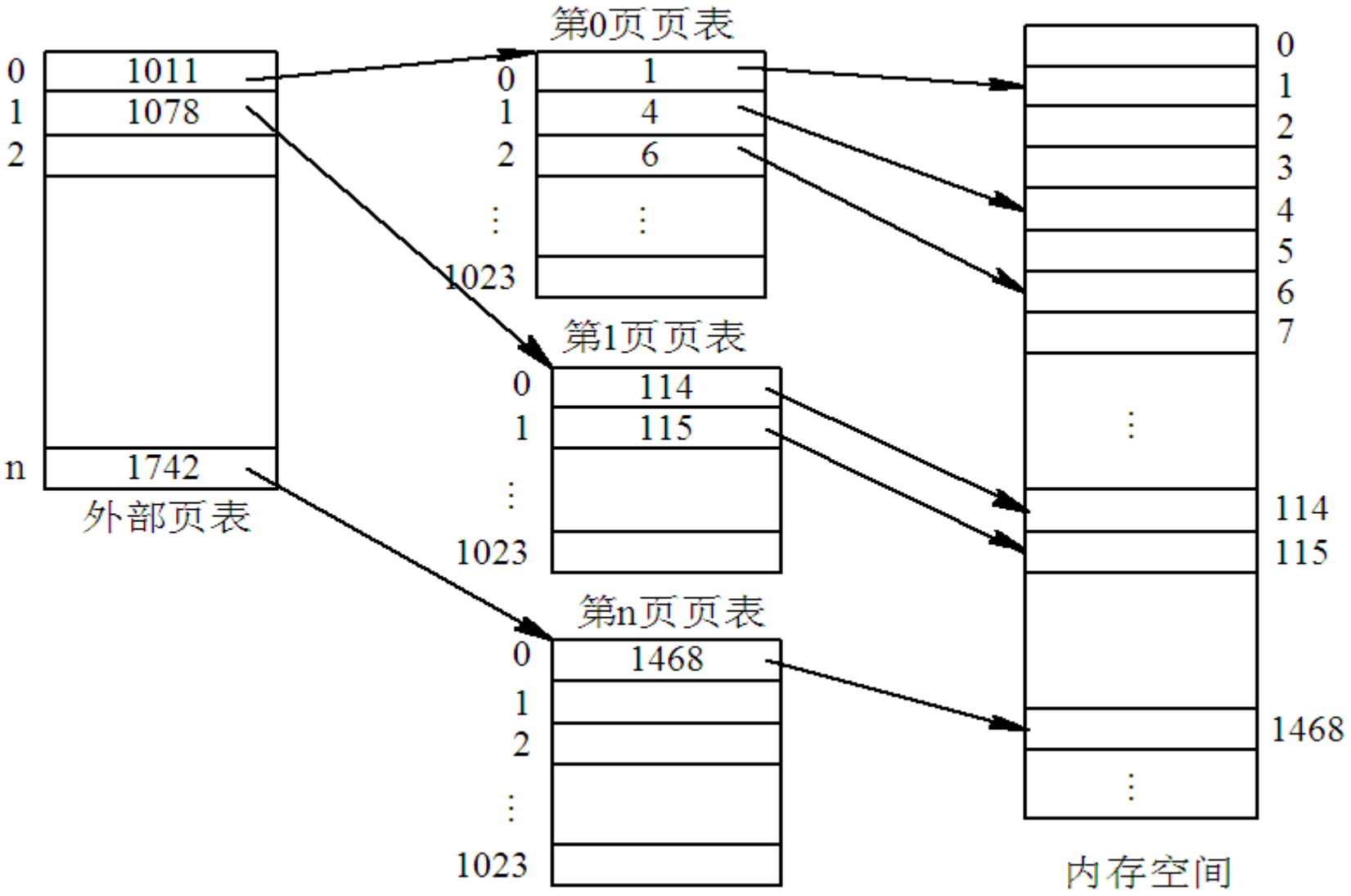


二级和多级页表

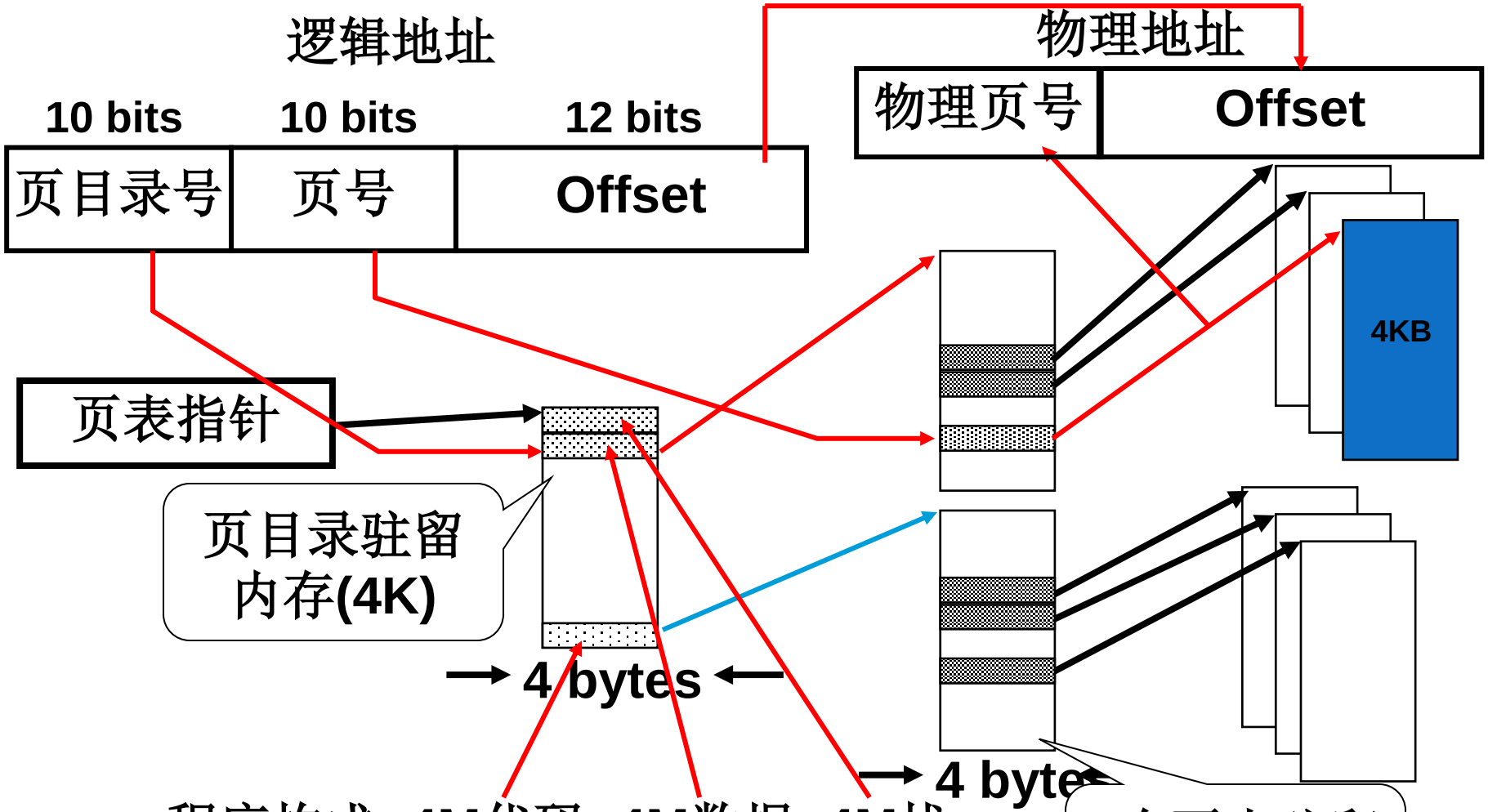
- 32位地址空间+ 4K页面+页号连续⇒2^{20}个页表项
  - 2^{20}个页表项都得放在内存中，需要4M内存
  - 系统中并发10个进程，需要40M内存
  - 最关键我们还需要连续的内存空间来存放它们
  - 引入多级页表或者只将页表的一部分调入内存，需要时再调入

32位逻辑地址格式(多级页表)

二级页表的例子


第0页页表
第1页页表
第n页页表
外部页表
内存空间
0
1
2
n
1011
1078
1742
1
4
6
114
115
1468
1023
0
1
2
3
4
5
6
7
114
115
1468

多级页表时的地址翻译



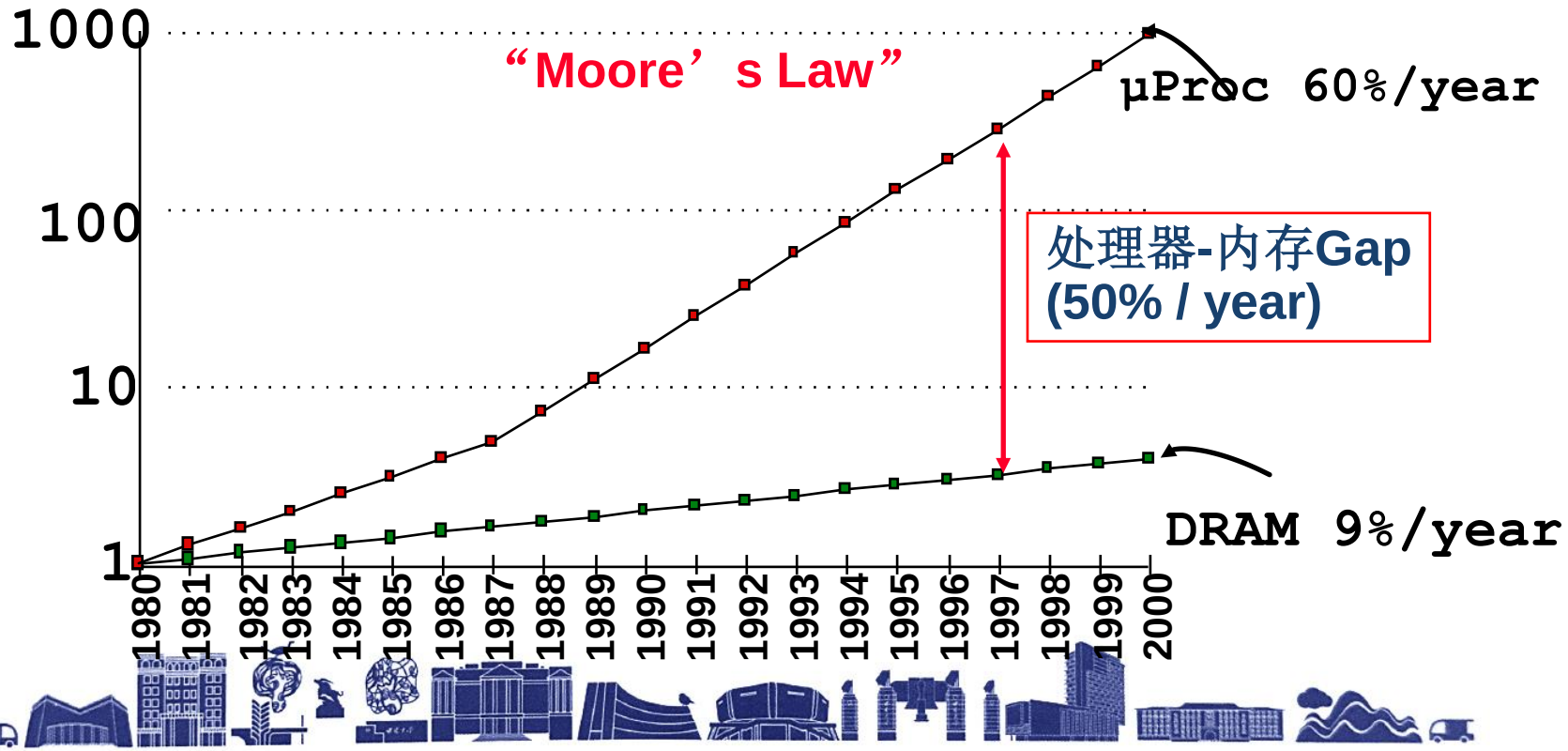
- 程序构成: 4M代码+4M数据+4M栈

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%

需要注意的事实:
内存相比CPU本来就很慢!

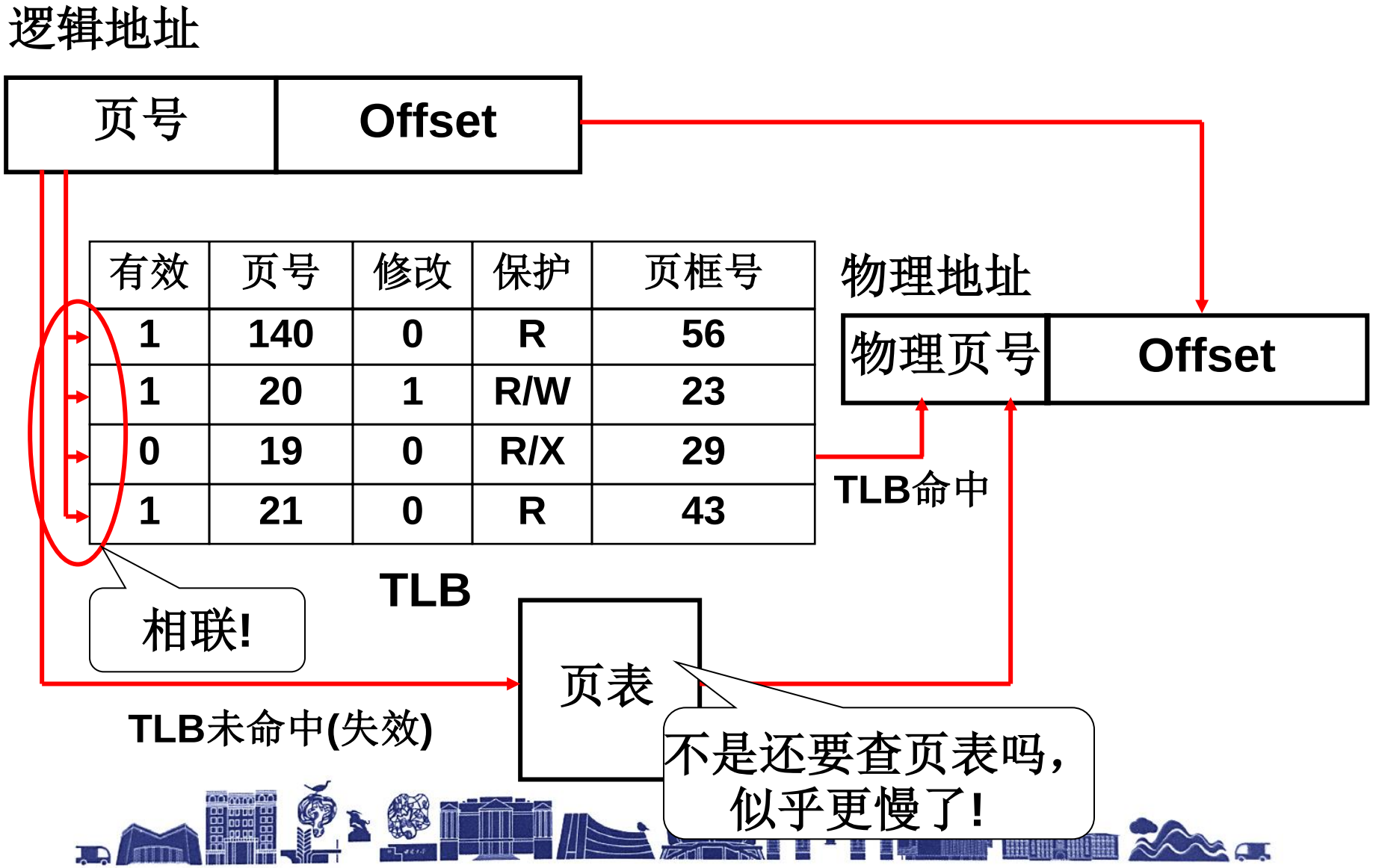


提高地址翻译的效率

- 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：这个硬件访在哪里？ 寄存器
  - 页表小⇒寄存器可行，但如果页表很大呢?
  - **TLB(Translation Look-aside Buffer)**是一组相联快速内存，也称为快表。

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得以发挥作用的原因

- **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! TLB时间!

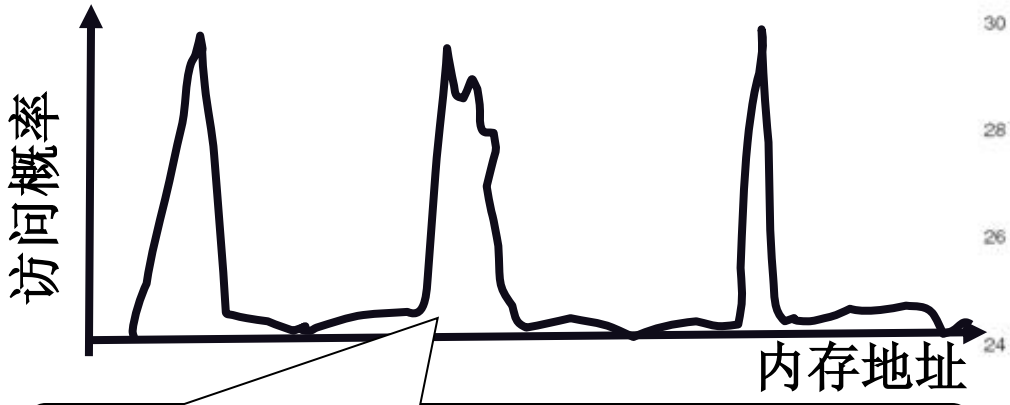
有效访问时间=80%×(20ns+100ns) + 20%×(20ns+200ns)=144ns

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

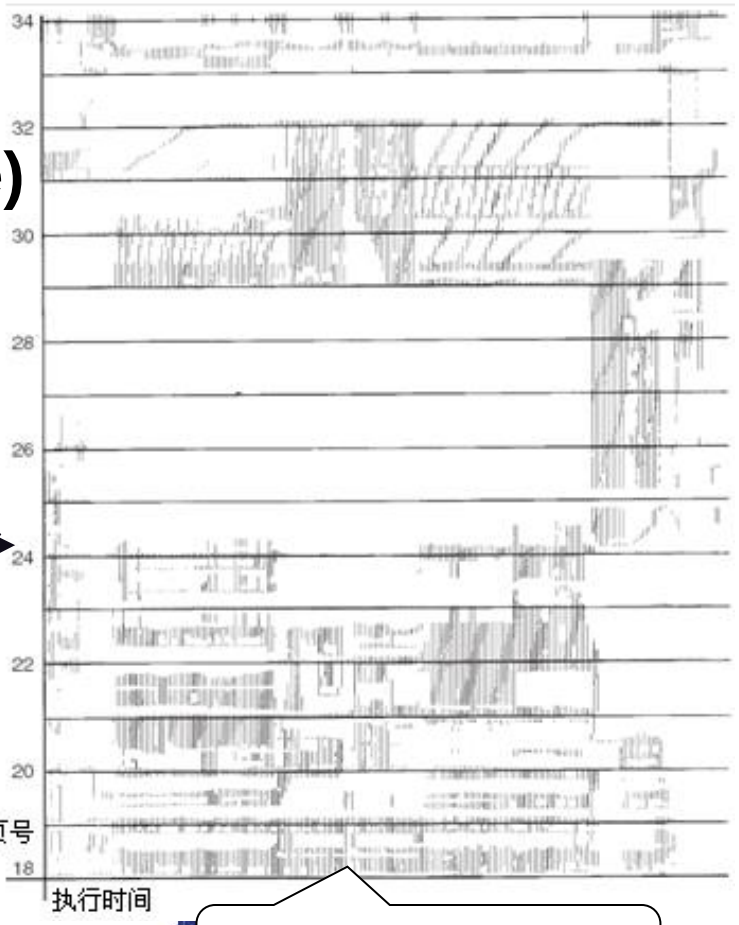
- **TLB要想发挥作用，命中率应尽量高**
- **TLB越大越好，但TLB价格昂贵，通常[64, 1024]**

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



某内存引用模式

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)，因为这些页会经常被用到且不会被换出。某些TLB的设计有这样的功能，不被选择替换
- 进程切换后，所有的与之前进程相关的TLB表项都变为无效(flush)

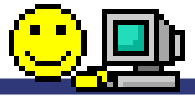
分页技术总结

- 实现机理
  - 地址空间和内存都分开大小相等的片(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)!

- 优点：靠近硬件，结构严格，高效使用内存
- 缺点：不符合程序员思考习惯

分页和分段的区别



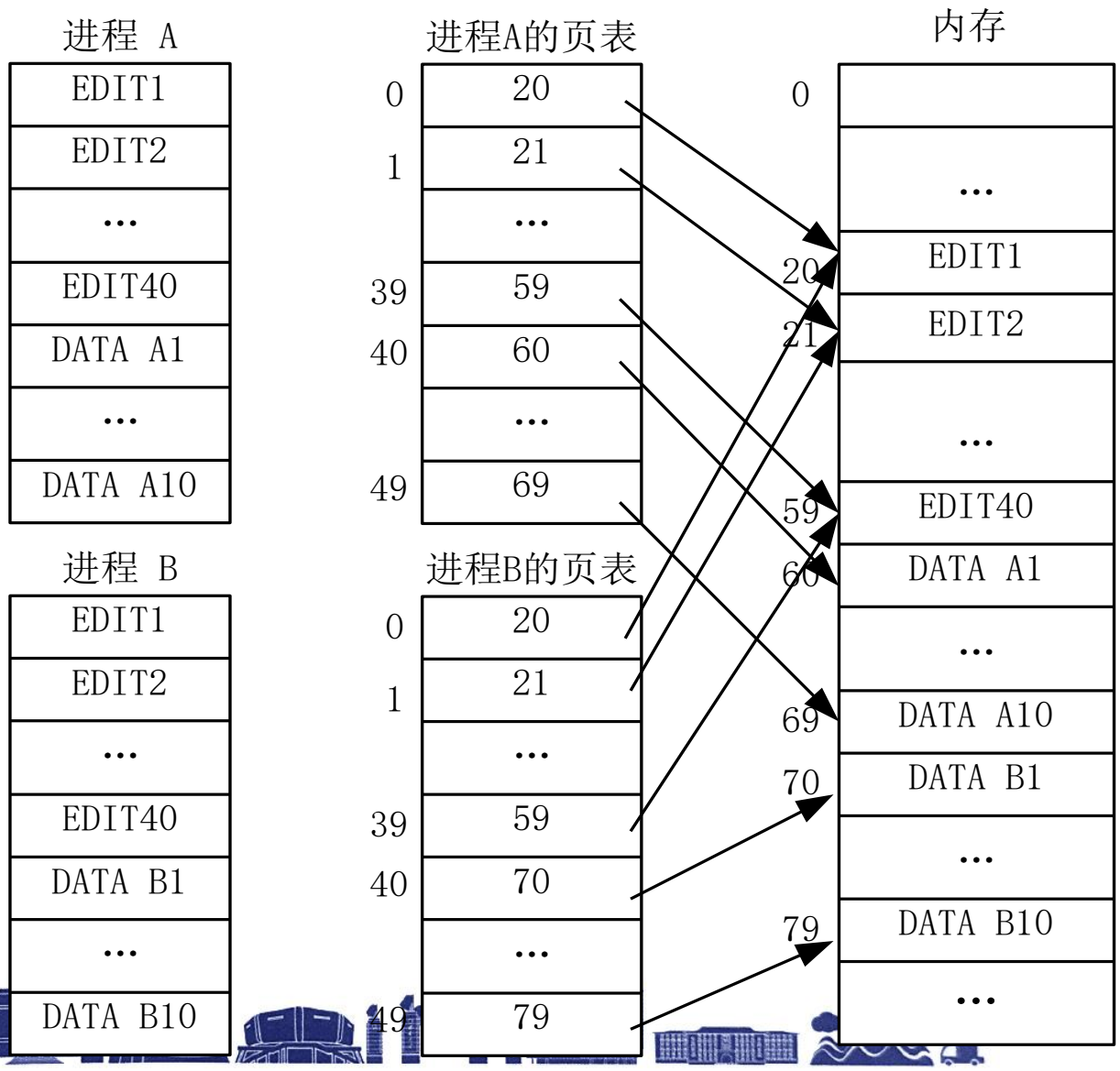
- 分页和分段的目的
  - 页是信息的物理单位，分页是系统管理的需要，而不是用户的需要。
  - 段是信息的逻辑单位，它含一组意义完整的信息。分段是为了更好地满足用户的要求。
- 页和段长度
  - 页的大小固定，由系统确定。
  - 段的长度不固定，决定于用户所编写的程序。
- 地址空间
  - 分页的作业地址空间是一维的，即单一的线性地址空间。
  - 分段的作业地址空间是二维的，程序员在标识一个地址时，需给出段名和段内地址。

段的共享与保护

- 页共享与段共享的比较
  - 由于段是信息的逻辑单位，用户易于实现对段的共享，也容易对段进行保护。
  - 页虽也可共享，但不方便。
- 举例
  - 例如有一个多用户系统，可同时容纳40个用户，它们都执行一个文本编辑程序，该文本编辑程序含有160KB的代码和40KB的数据，
  - 如不共享，共需160*40+40*40=8MB的内存空间来支持40个用户。
  - 若代码是可重入的，则无论是分页系统还是分段系统都可以共享该代码段，因此内存只需留一个文本编辑程序，所需空间为160+40*40=1760KB。

页的共享实现方式

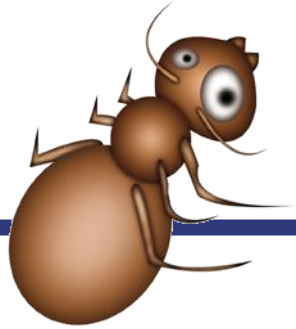
- 使用分页系统，每个页面的大小是**4KB**，则：
  - 代码段占**160/4=40**个页面
  - 数据段占**40/4=10**个页面
- 注意：
  - 页的共享要求作业地址空间的共享页必须具有相同的页号。
  - 不容易做到，父子进程倒是可以。



段式管理的优缺点

- 分段是支持用户内存观点的一种内存管理模式
  - 优点：
    - 便于动态申请内存
    - 管理和使用统一化
    - 便于共享和保护√
    - 便于动态链接
  - 缺点
    - 有碎片问题

段页式内存管理



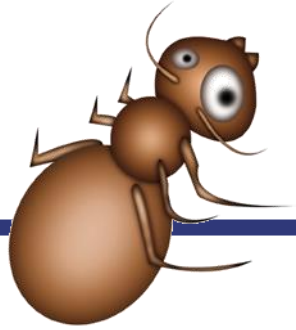
产生背景

段式存储管理：满足程序和信息的逻辑分段要求，反映了程序的逻辑结构，有利于短的动态增长，充分实现共享和保护

页式存储管理：等分内存，有效克服碎片，提高了存储器的利用率

结合页式段式优点，克服二者的缺点

段页式内存管理



- 管理机制
  - 段表：记录了每一段的页表始址和页表长度
  - 页表：记录了逻辑页号与内存块号的对应关系
    （每一段有一个，一个程序可能有多个页表）

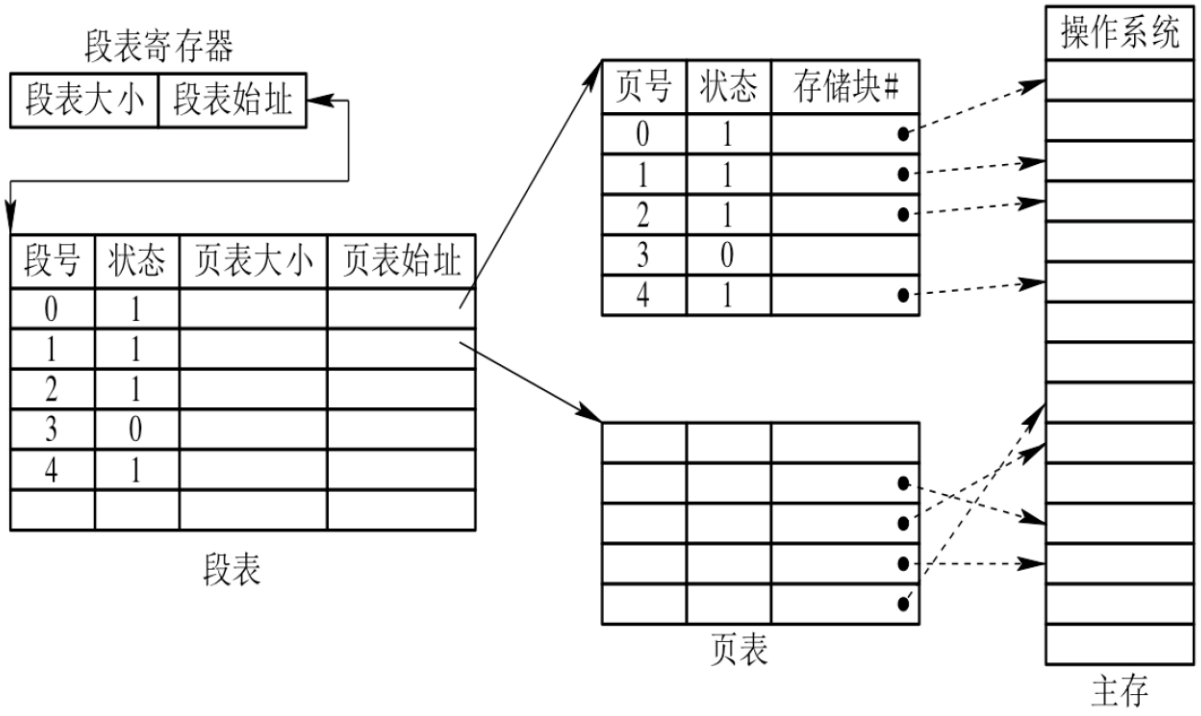
空块管理：同页式管理

内存划分：按（选择：段式/页式）存储管理方案

内存分配：以（选择：段/页）为单位进行分配

段页式内存管理

- 基本思想
  - 用户程序划分：按段式划分（对用户来讲，按段的逻辑关系进行划分；对系统讲，按页划分每一段）



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映射
- 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
- 映射、保护、内存分配是内存管理的三个核心词!

参考资料

- 廖剑伟，操作系统，西南大学，重庆，2023