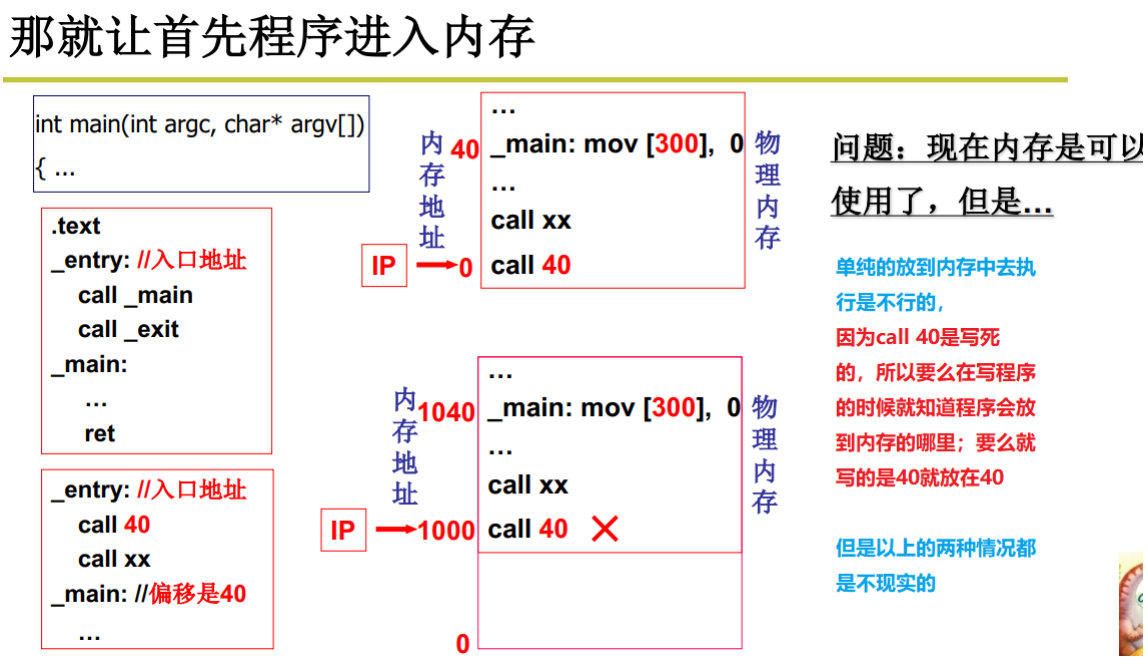
哈工大操作系统-L20内存使用与分段

从直观的内存应该如何的使用，再到如何科学合理的使用内存

1.内存如何使用

- 计算机工作的简单原理
  - 内存使用：将程序放到内存中，PC指向执行序列所在的内存的开始地址
  - 简单来说，我们只需要把程序加载到内存中就行了
- 但是会有什么问题呢?

问题：单纯搬移程序，绝对地址会引起内存地址的冲突



- 硬生生的搬移需要将上面这个程序放到0地址处，然后call40才能正确的执行，但是0地址放的是系统
- 科学的搬移应该是，在内存中找一段空闲的，搬入程序。
- 但是这样的话，call的40就不能是绝对地址了，因此需要处理

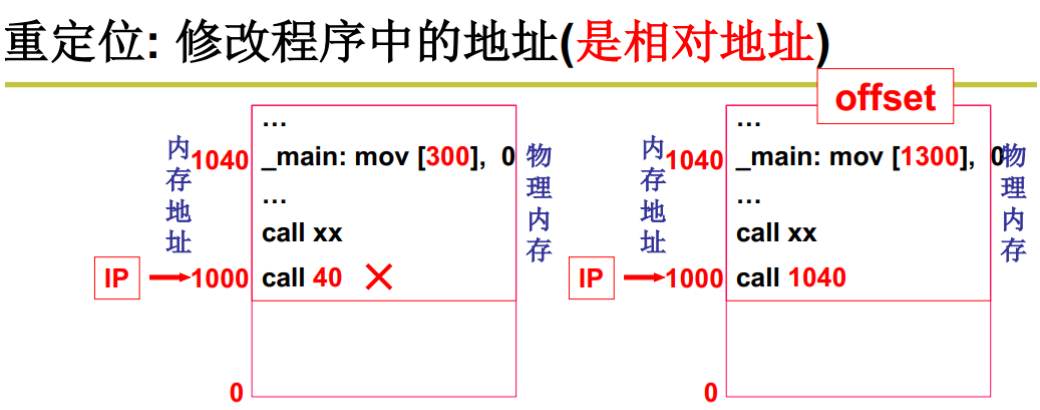
2.重定位--产生逻辑地址和物理地址的映射

- 我们称程序中写死的地址为**逻辑地址**。
- 程序中写死的地址其实是相对于程序段本身而言的。因此，把程序段中的地址认为是**相对地址**

- **重定位**：修改程序中的地址，把地址由相对地址，修改为程序中的物理地址
  - 即如果上面的程序放入的是1000处，那么call40只需要改为1040就行。

两种一般情况的重定位

- 两种一般情况的重定位：
  - **编译时重定位**：只能**载入到固定的地方**，即编译时已经设置好的偏移(要保证那个固定的地方一定是空闲比较难)
  - **载入时重定位**：载入之前任选位置，**一旦载入**，偏移也就是固定的，**程序也不能再移动了**(如果加入的程序多了，可能内存空间也不够)



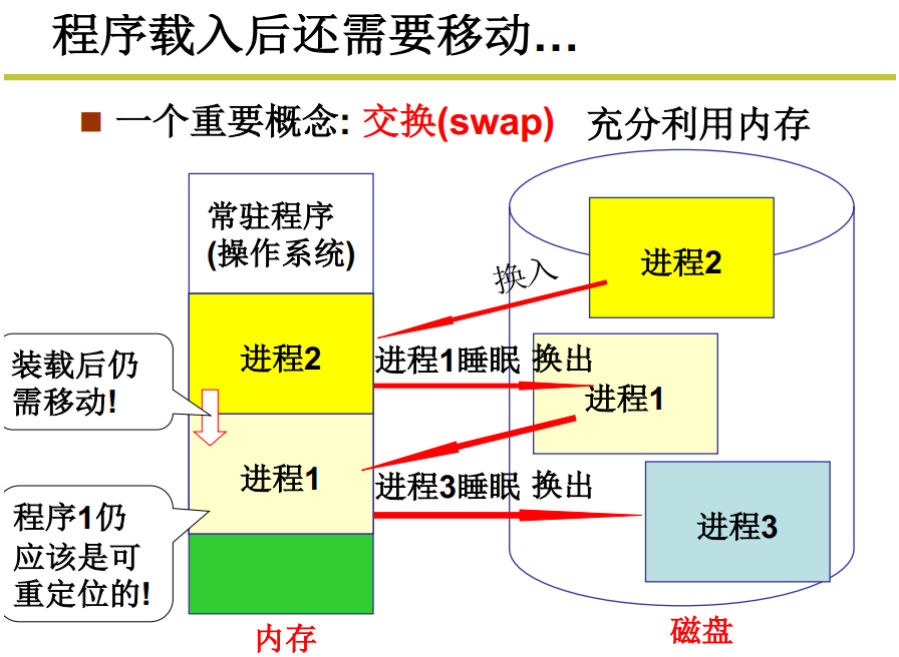
■ 是么时候完成重定位? 编译时 载入时

问题：两种定义各自有什么特点?

■ 编译时重定位的程序只能放在内存固定位置

■ 载入时重定位的程序一旦载入内存就不能动了

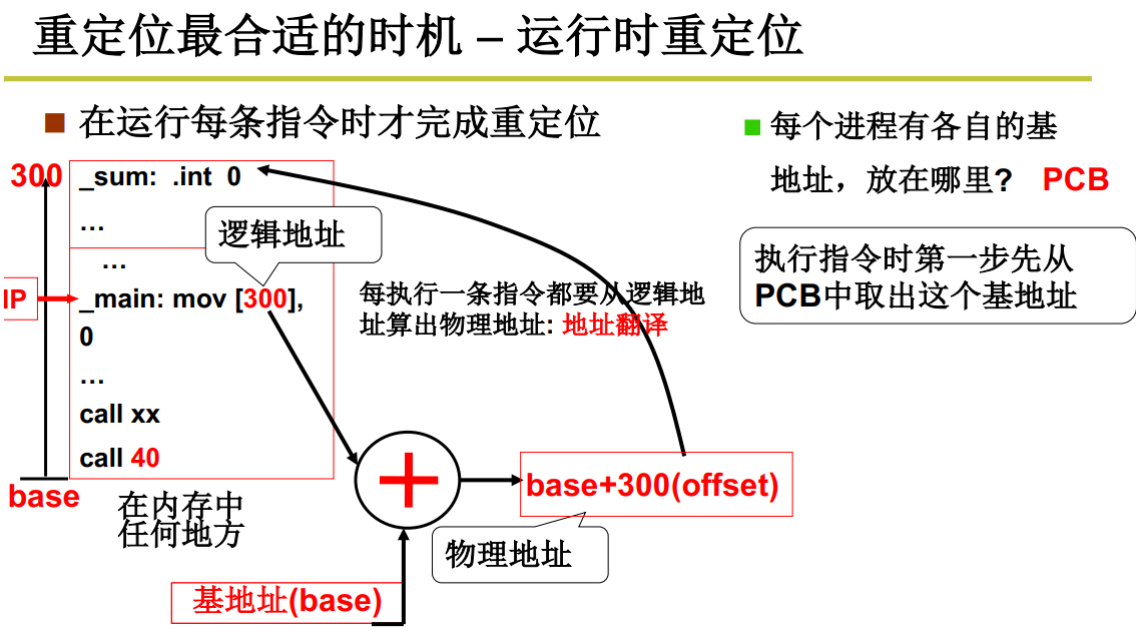
3.交换--将长期阻塞的进程换出内存，节省内存空间

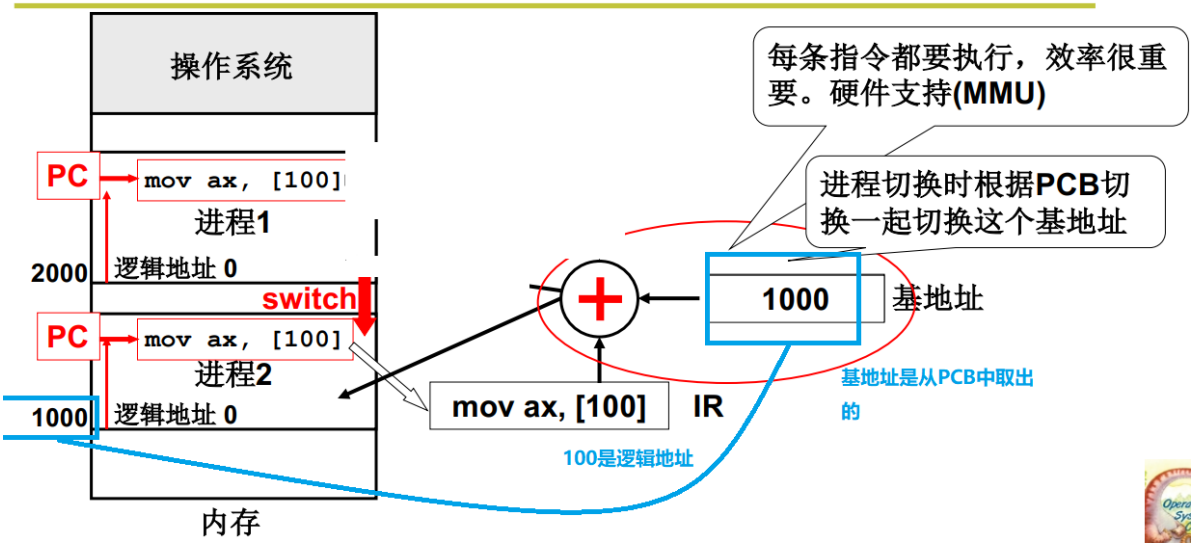


- 交换表面，程序在载入之后，内存位置也有可能发生移动
- 因此，载入/编译时重定位都不行

4.重定位最佳时机--运行时重定位

- 运行每条指令时，才对每条指令上的地址执行重定位
  - 从逻辑地址算出物理地址的过程，叫**地址翻译**。基地址+偏移
  - 基地址存在PCB中
- 这样无论程序何时被换出，换入之后再执行指令，指令中含有的地址都是对的



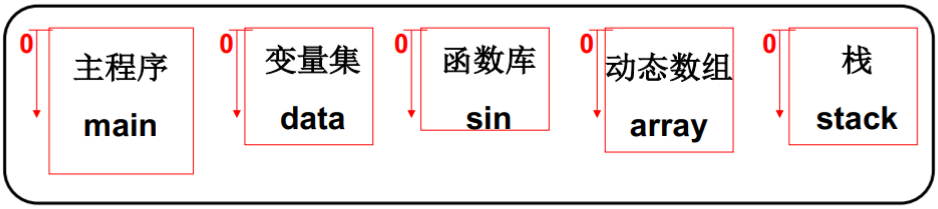


5.分段

通常的程序都是分段的，程序中的地址其实是基于段的偏移。

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：代码段只读，代码/数据段不会动态增长…



程序员眼中的一个程序

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

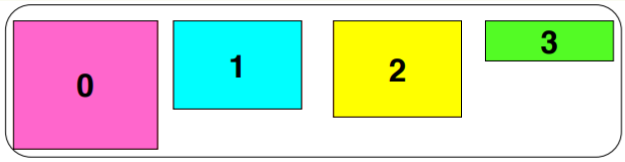
如mov [es:bx], ax

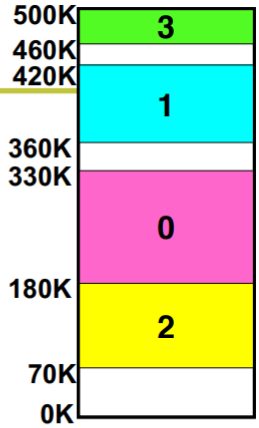
因此，程序中的地址是每个段中的偏移，而不是整个程序中的偏移

- 分段放入内存的好处：更高效的利用内存
  - 不会增长的程序段，可以固定放在一些区域
  - 会增长的程度段，如果放置的内存区域不够了，需要移动这个段到其他区域
    - 如果整个程序放入内存的话，由于增长导致内存区域不够需要移动整个程序

段表

不是将整个程序，是将各段分别放入内存





- 再进行运行时重定位会怎么样?

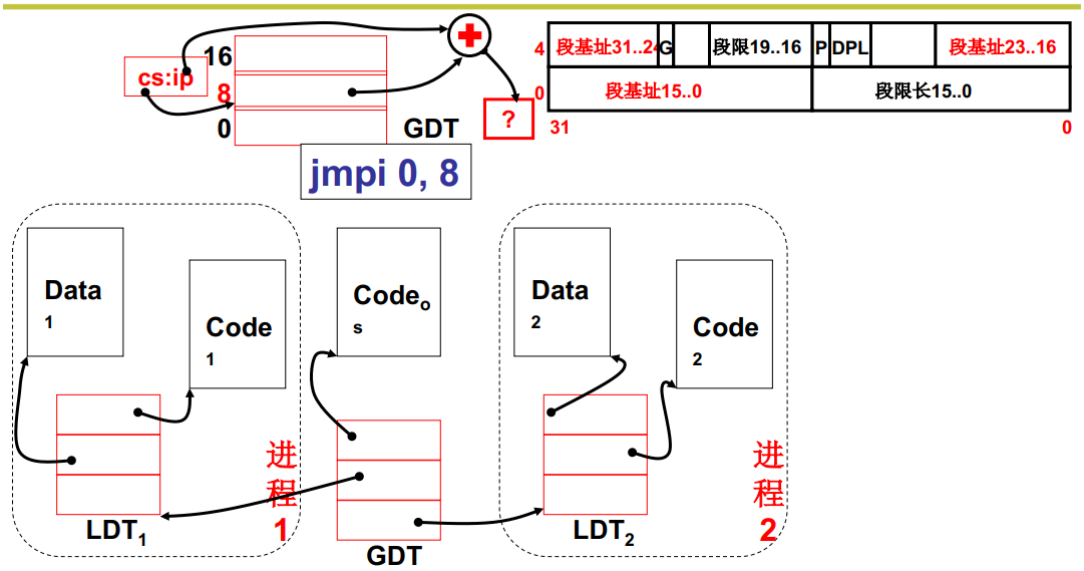
mov [DS:100], %eax　　jmpi 100, CS

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

问题：假设DS=1，CS=0，上面两条指令运行时重定位成什么？那么jmp 500K呢?

这个表似曾相识… 真正故事:GDT+LDT



- **LDT表**：因为是分段的，因此我们需要一个**段表**去查段基址。
- GDT表可以视为是LDT表的特例，GDT表是操作系统的LDT表。