物理内存管理的第二个内容：非连续内存的分配

为什么需要非连续内存分配？

连续内存分配的缺点：

分配给一个程序的物理内存是连续的；

内存利用率低；

有外碎片和内碎片问题

非连续分配的优点：

一个程序的物理地址空间是非连续的；

更好的内存利用和管理；

允许共享代码与数据（共享库等…）；

支持动态加载和动态链接；

非连续分配的缺点：

如何建立虚拟地址和物理地址之间的转换：软件方案、硬件方案

一般不用开销大的软件方案，两种硬件方案：分段、分页

非连续内存分配方法：

分段（segmentation）

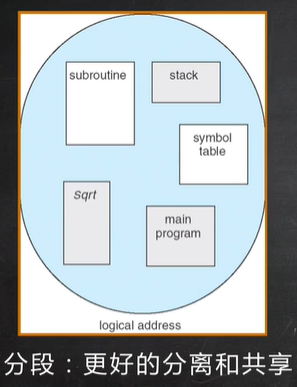
分页（paging）

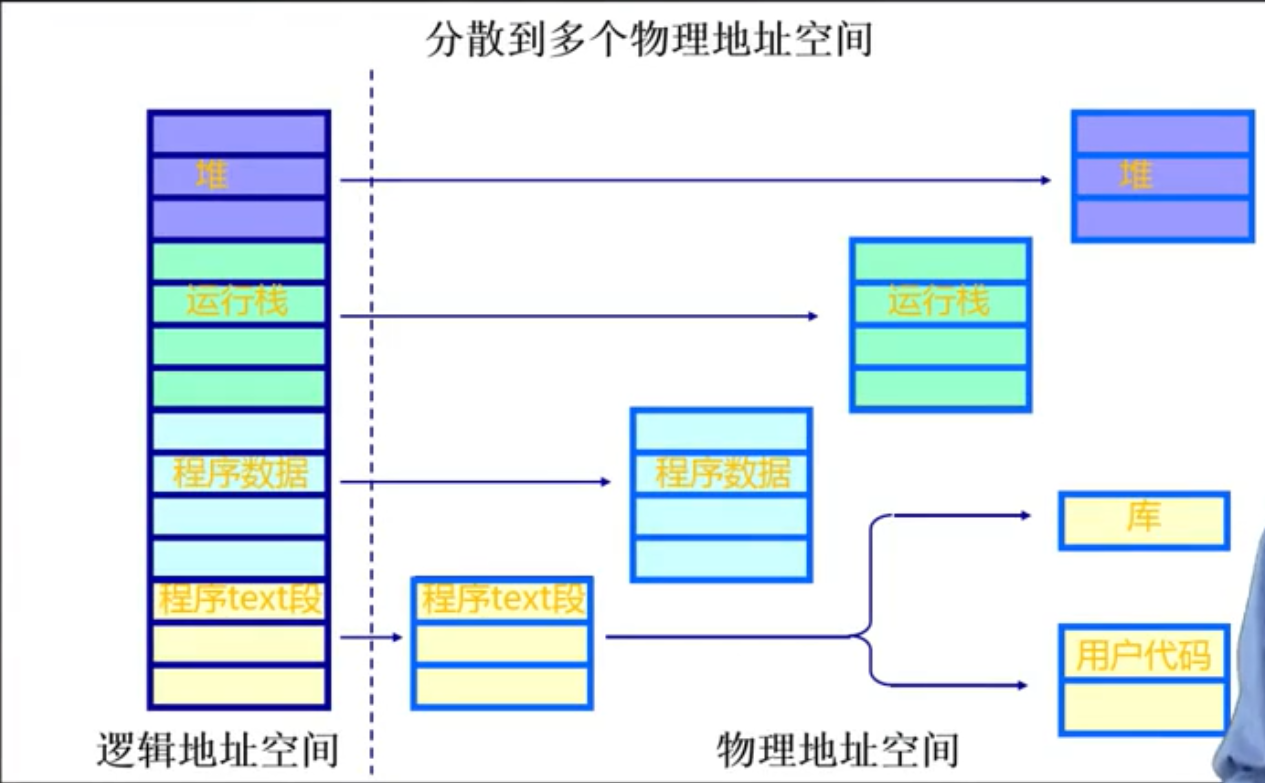
页表（page table）

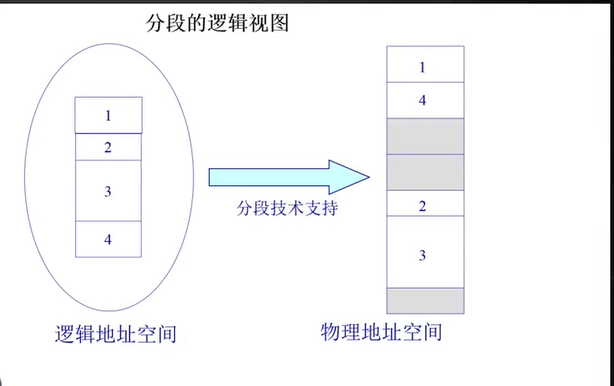
非连续内存分配方法——分段，要解决以下两个问题：

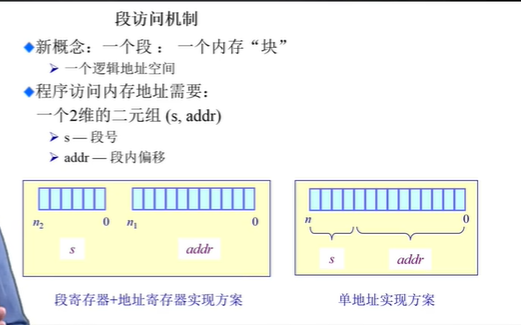
程序的分段地址空间

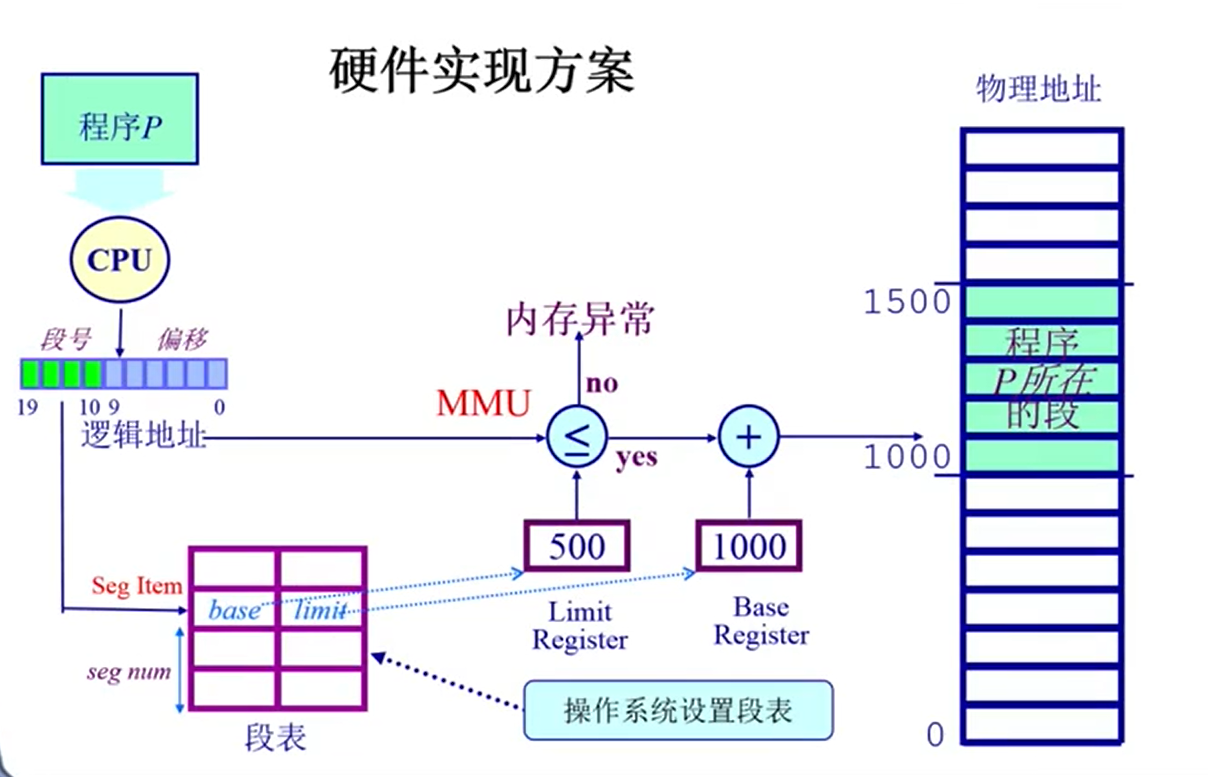
分段寻址方案





大小位置不一样



箭头反了

操作系统在寻址之前就建立好了段表

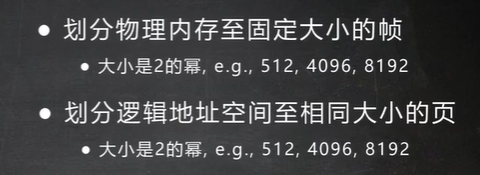
已经很少使用分段了，一般主流用分页

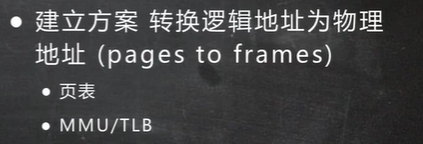
CPU的分页寻址方案、分页地址空间

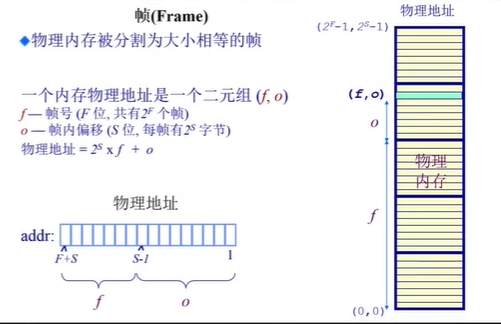
分页和分段的主要区别是偏移尺寸大小不一样，分段的偏移尺寸是可变的，而分页的偏移尺寸式固定的，其实就是页的大小；

页的大小可以是2的幂次方，逻辑地址空间和物理内存一样

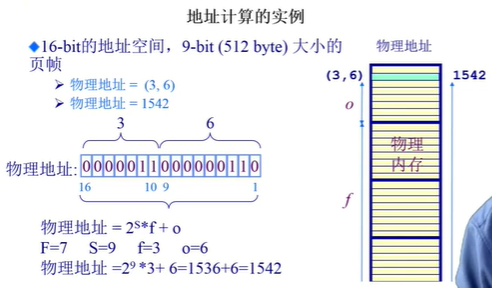
物理上是帧，逻辑上是页，建立映射关系

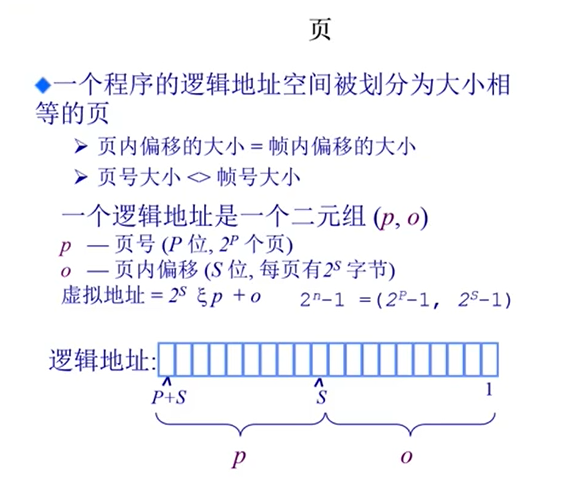






上图是由帧号和偏移得到物理地址

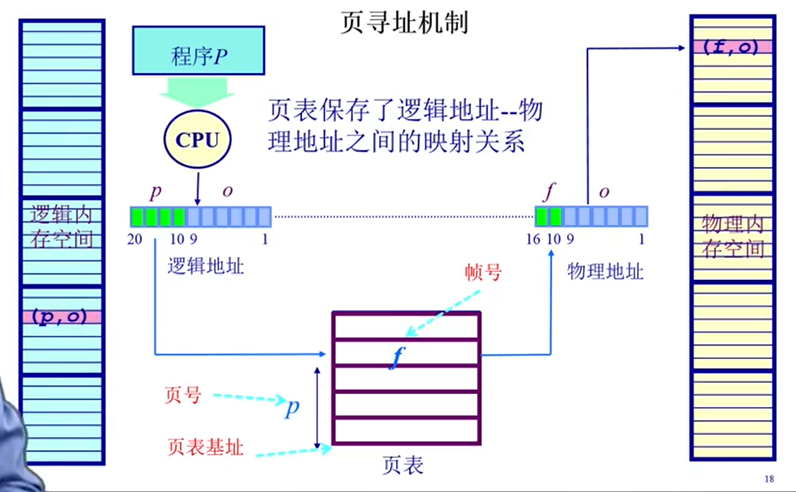




虚拟地址就是逻辑地址的意思，上面是2^S \* p + o

S代表一个页的大小，这个大小可能与帧号大小不一样

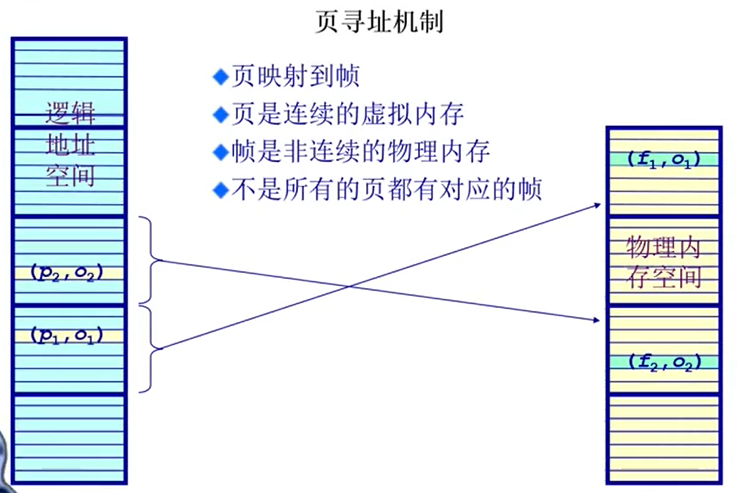
而，页内偏移的大小 = 帧内偏移的大小，（页内偏移大小是固定的）



上图的逻辑内存地址空间是连续的，一页一页大小相同，但整个逻辑内存空间可能和物理内存空间不一样大小，主要看映射关系

页表是一个可以通过页号查询到帧号的工具

操作系统建立的页表

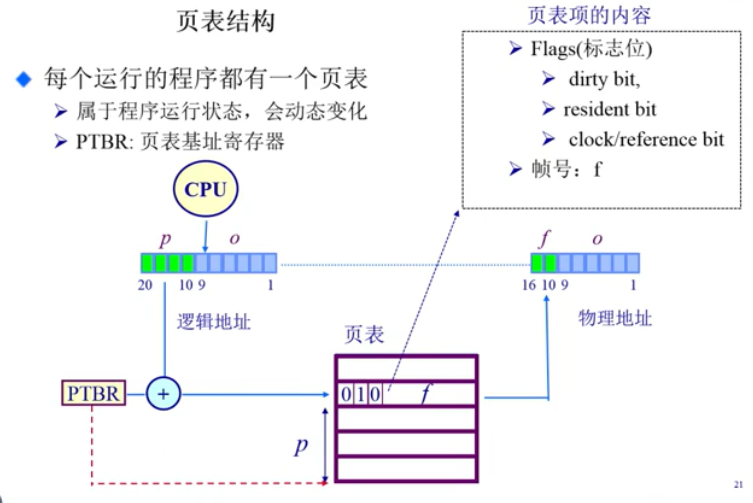


左边的逻辑地址是连续的，映射到右边物理地址时不一定是连续的，这样有助于减少碎片

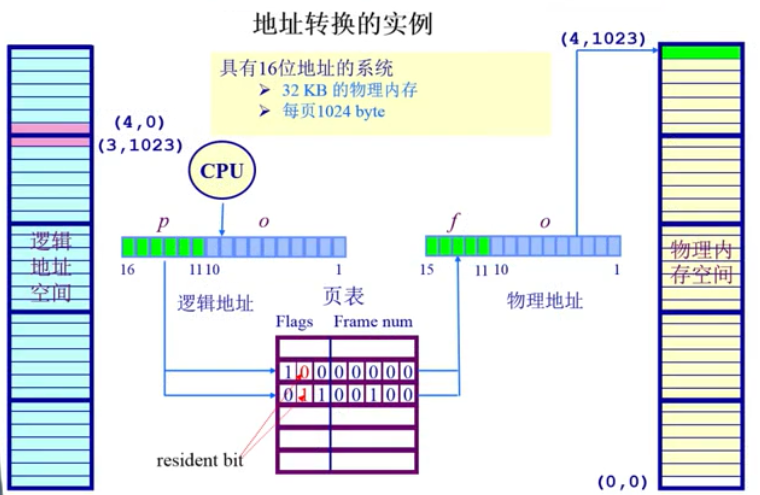
页表是提高页到帧转换效率的关键

页表概述

大数组，索引是页号page num，内容元素是帧号frame num



页表的作用可以检查要查找的帧号是否真的存在，是否读过等等



这里在查找（4，0）时Flags的resident bit不存在，会返回一个异常

分页机制的性能问题

访问一个内存单元需要2次内存访问，一次用于获取页表项，一次用于访问数据

页表可能非常大：64位机器如果每页1024字节，那么一个页表大小会是多少？

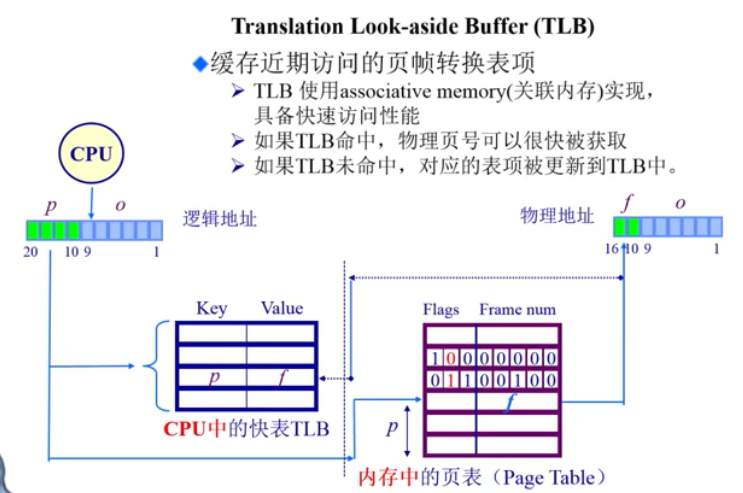
每一个应用程序需要有自身对应的页表

解决：

缓存（Caching），把最常用的内容缓存到离CPU很久的地方

间接（Indirection）访问

转换后备缓冲区（LTB）：解决时间效率问题



TLB存在于MMU内

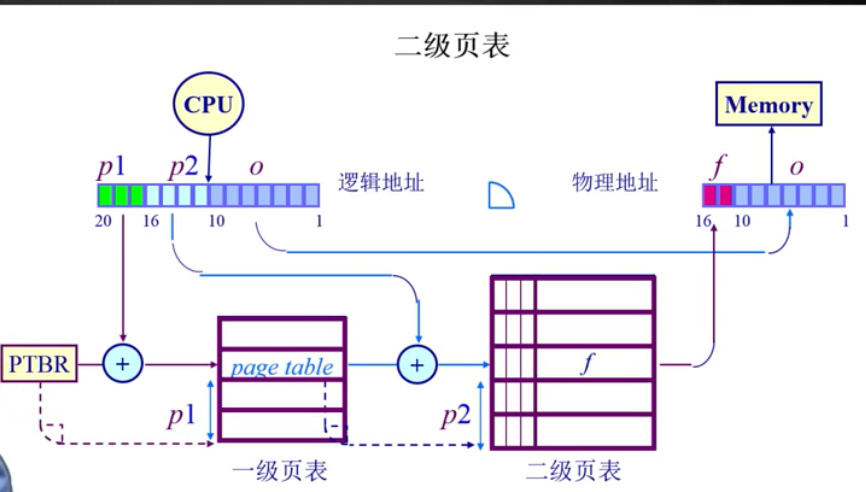
LTB是一个cache缓冲，缓存着页表里面的内容

LTB减少了对内存页表的访问，提高了效率

当LTB查找不到后，才会去内存页表中查找，并且把那一项存到TLB中

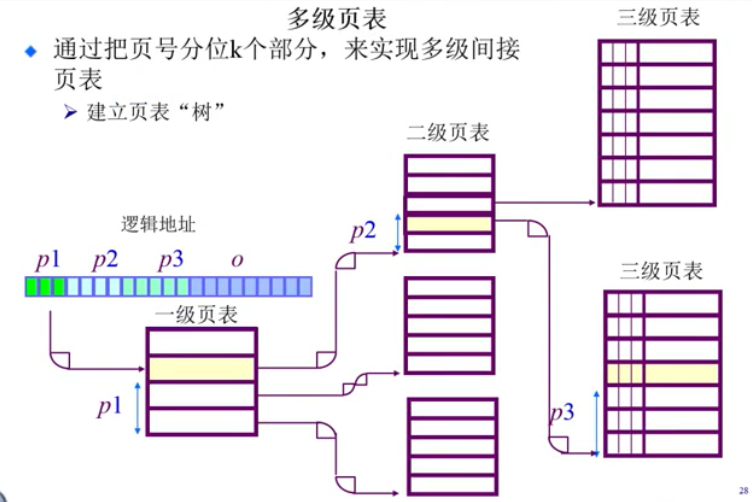
二级/多级 页表：为了减少空间

Offset还是不变，page num细微改变



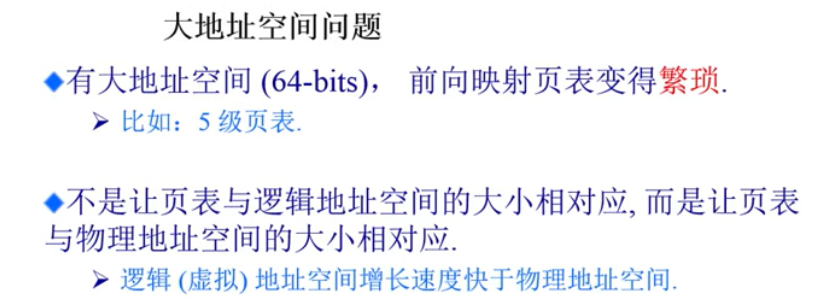
一级页表存的是二级页表的其实地址

分级过程多了查找步骤，且这些页表均存在于内存中，开销较大是事实，但这样使得不存在映射关系的页表项不会占用内存了

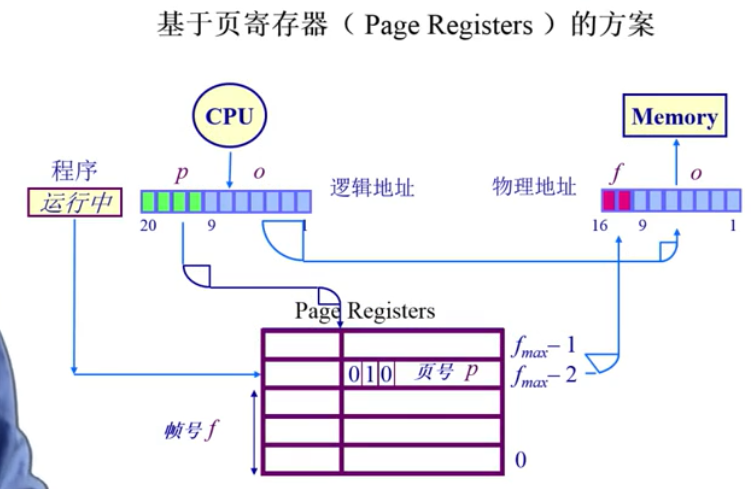


多级页表

反向页表：让页表大小和逻辑空间大小关系不那么密切

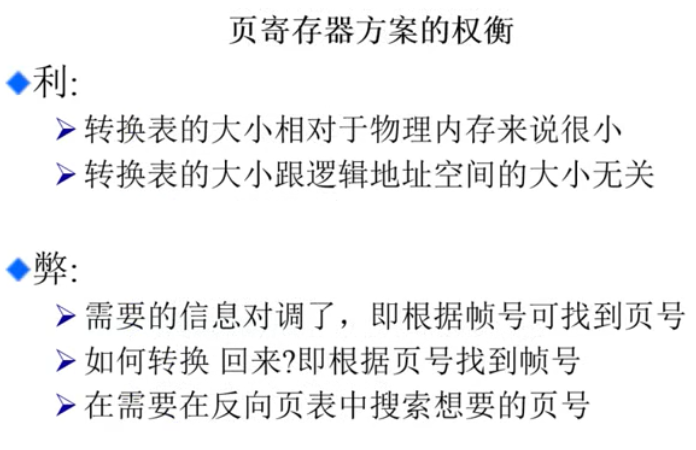


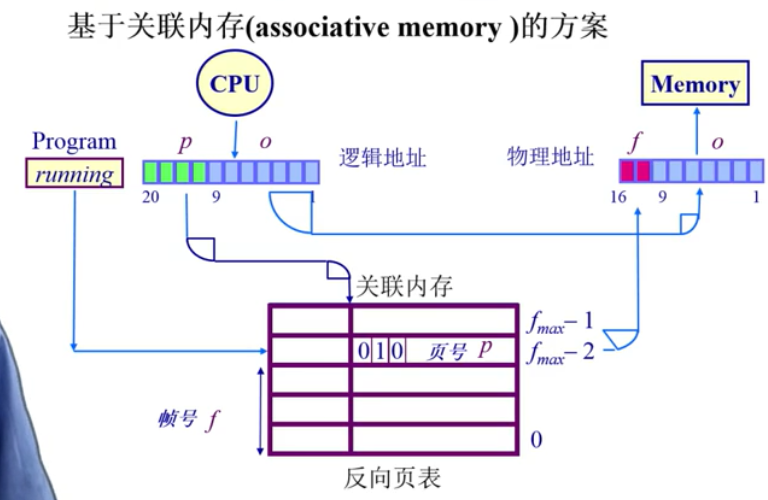
以物理帧号作为index，查找逻辑页的页号



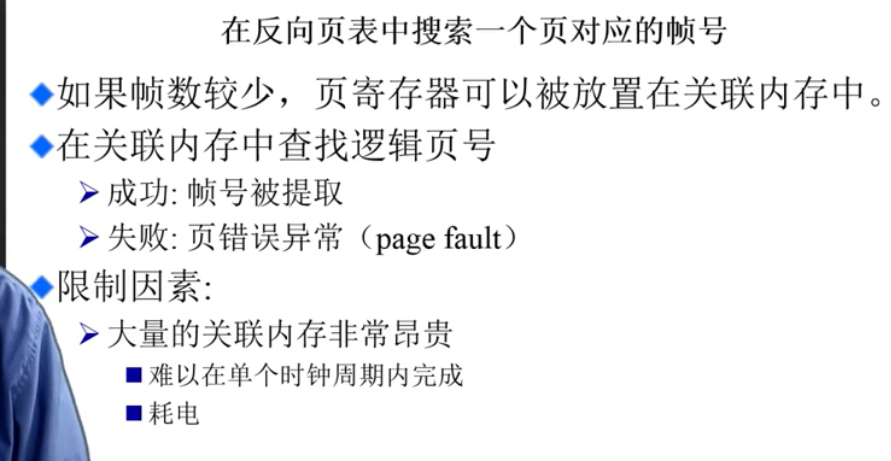
要根据帧号去找到叶号，怎么做了？已经反向了

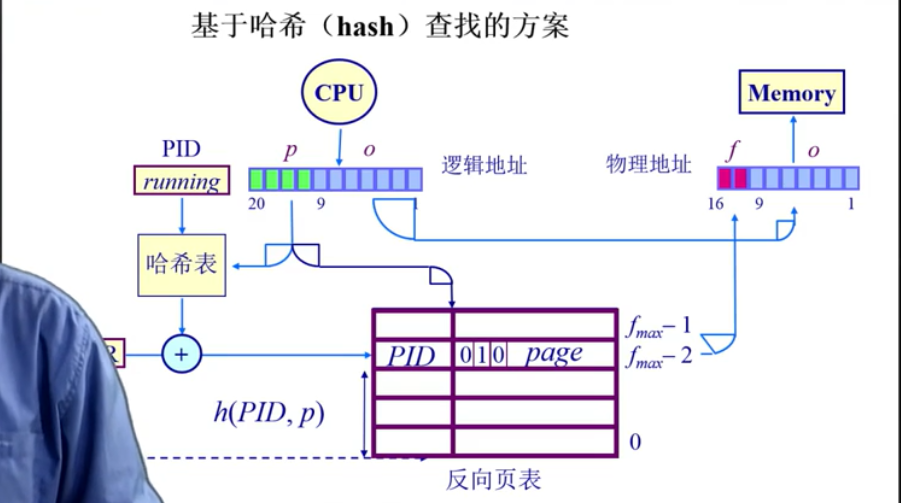






开销成本很大





PID是为了缓解哈希的冲突

