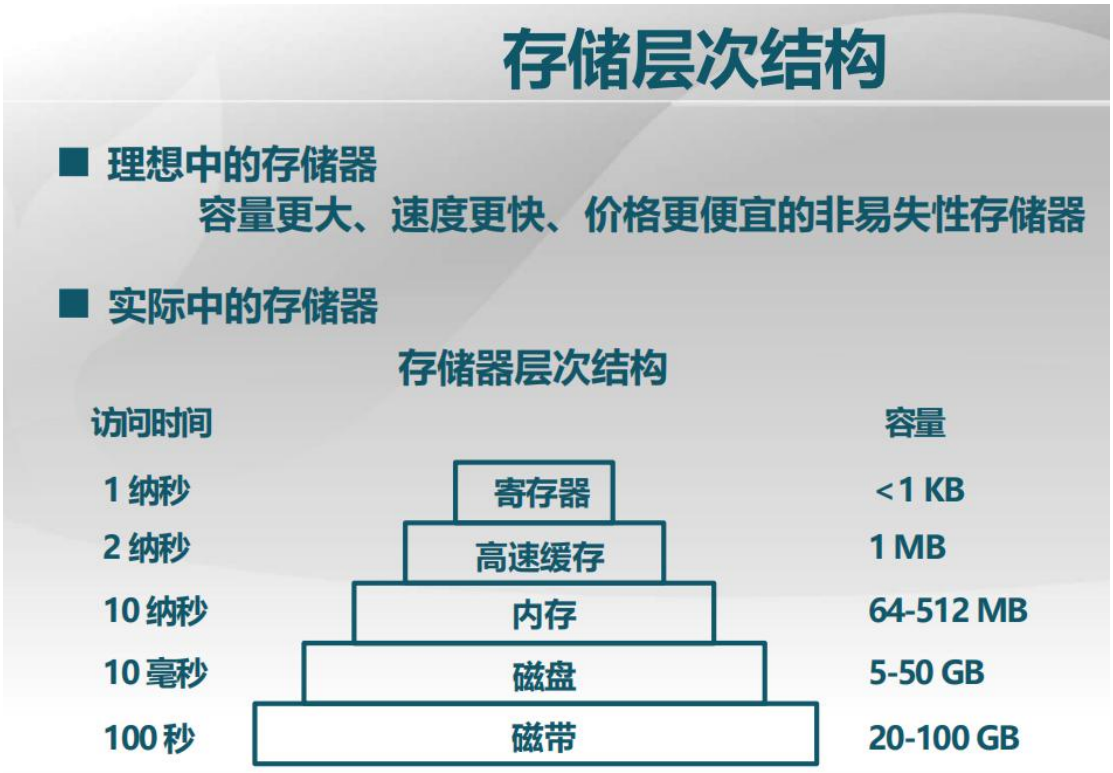
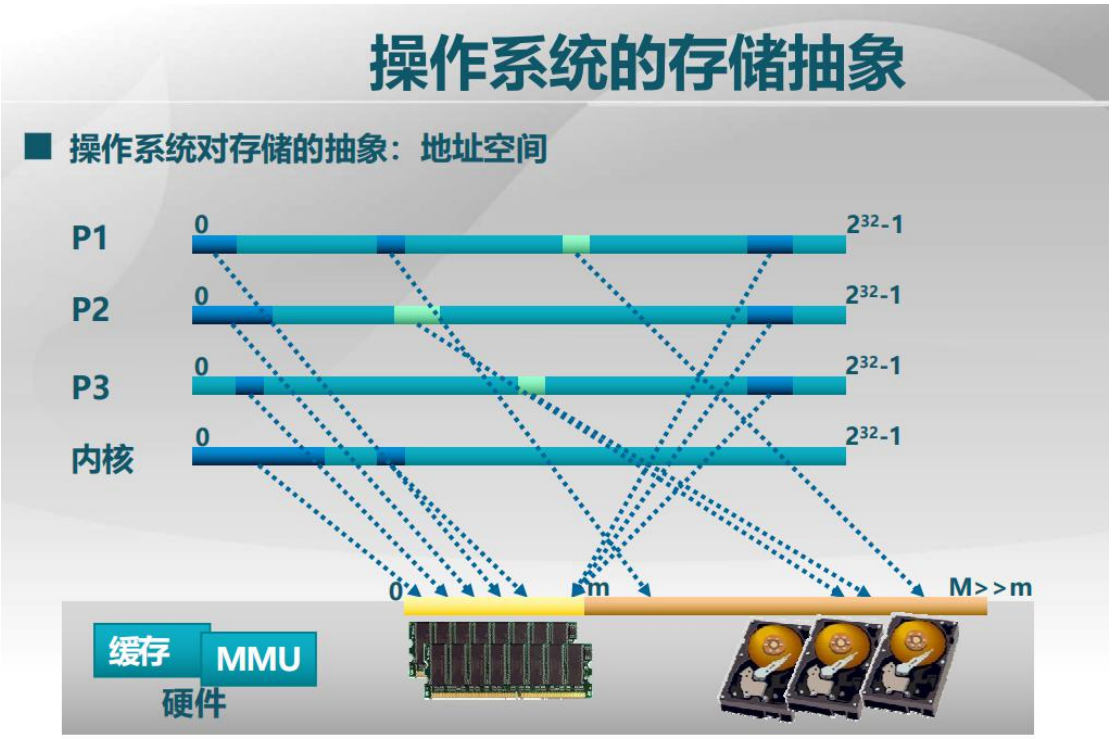
第一讲 虚拟存储的需求背景

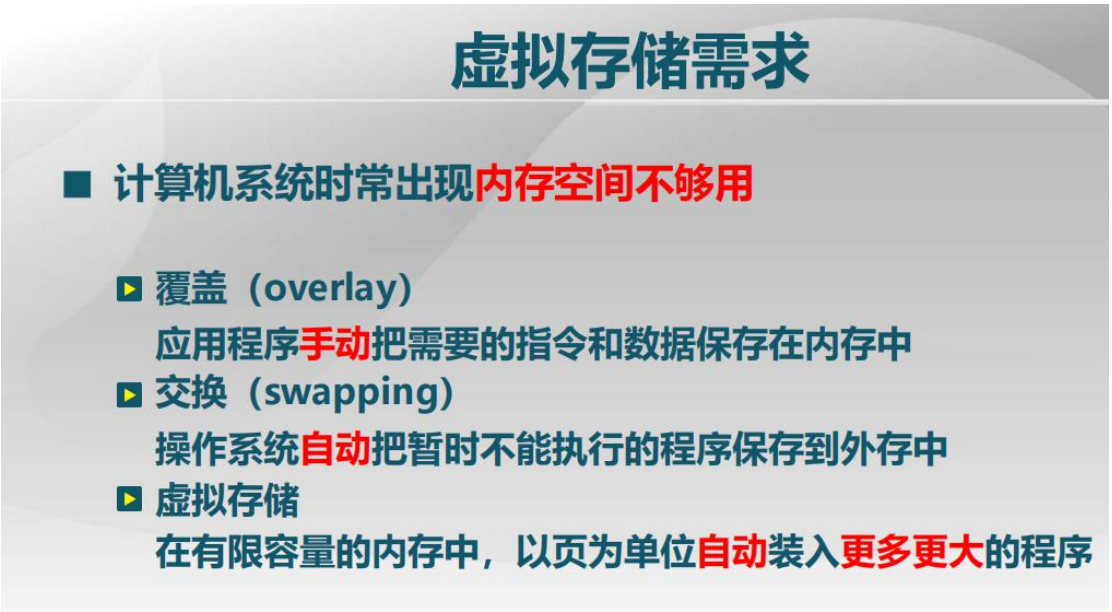
在前面，我们讲过了物理内存的管理，在那个地方我们有(连续)分区、非连续分区，这两种办法是说我们在物理内存中如何找到一块可以给进程使用的内存空间。虚拟存储是前面讲的非连续内存分配(非连续分区)的一个延续，非连续内存分配是在内存里找存储空间，让其可以不连续，而虚拟存储是在非连续存储内存分配的基础上，可以把一部分内容放到外存里的做法，这种做法可以让我们的应用程序有更大的空间可以使用。



实际上，理性中存储器的几个目标是相互制约的，和计算机其他部分比较而言，存储器占的成本不能太高，理想情况是做不到的。实际中的存储器做了上图中的折中，构成存储器层次结构，最快的是 CPU 寄存器，价格也最贵，最慢的是磁带，价格便宜。

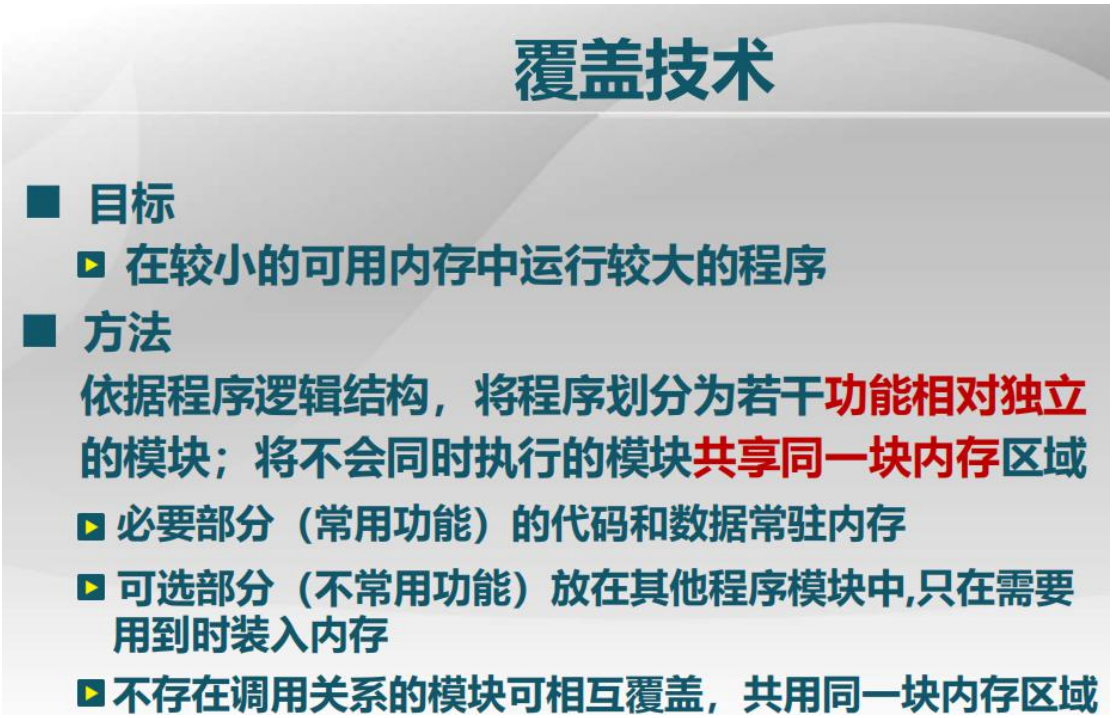


操作系统让用户不用面对各种各样的存储介质，而是将存储抽象为地址空间，并自动完成从地址空间到存储介质的映射，使得用户的使用更加简单。

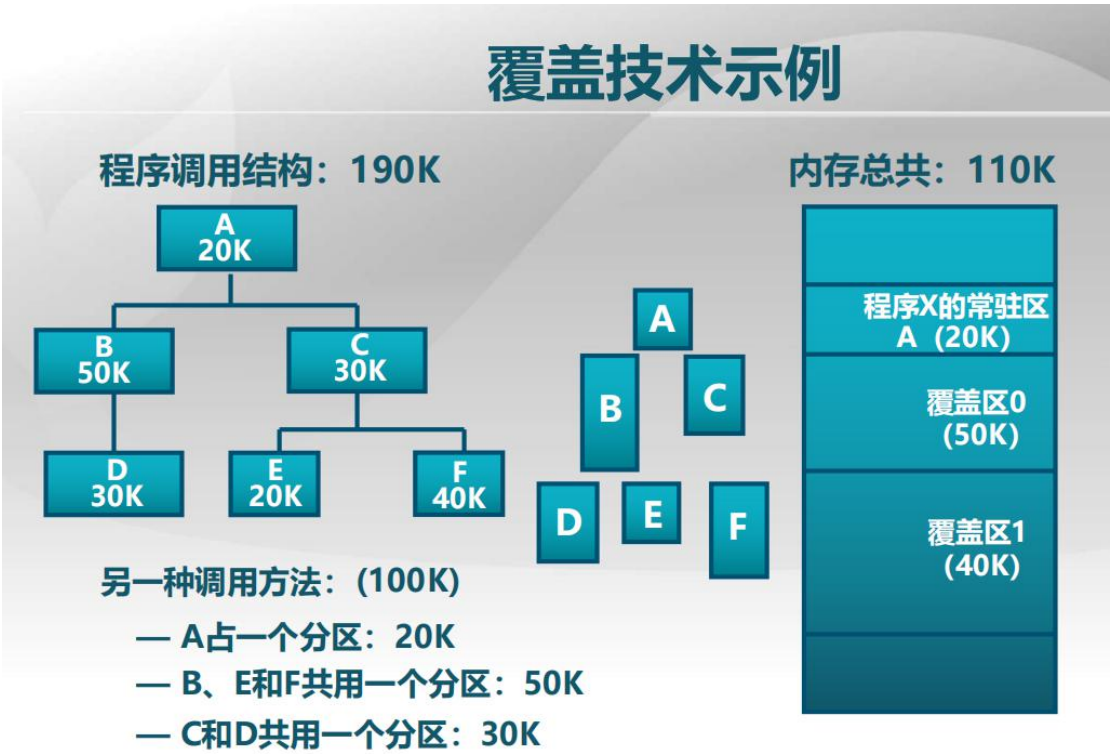


覆盖、交换、虚拟存储是解决内存空间不够用的三种方法。覆盖导致应用开发难度提高；交换：因为交换的单位是一个进程的整个地址空间，开销比较大；虚拟存储：相比于交换，空间和容量更大，也更自动。

第二讲 覆盖和交换

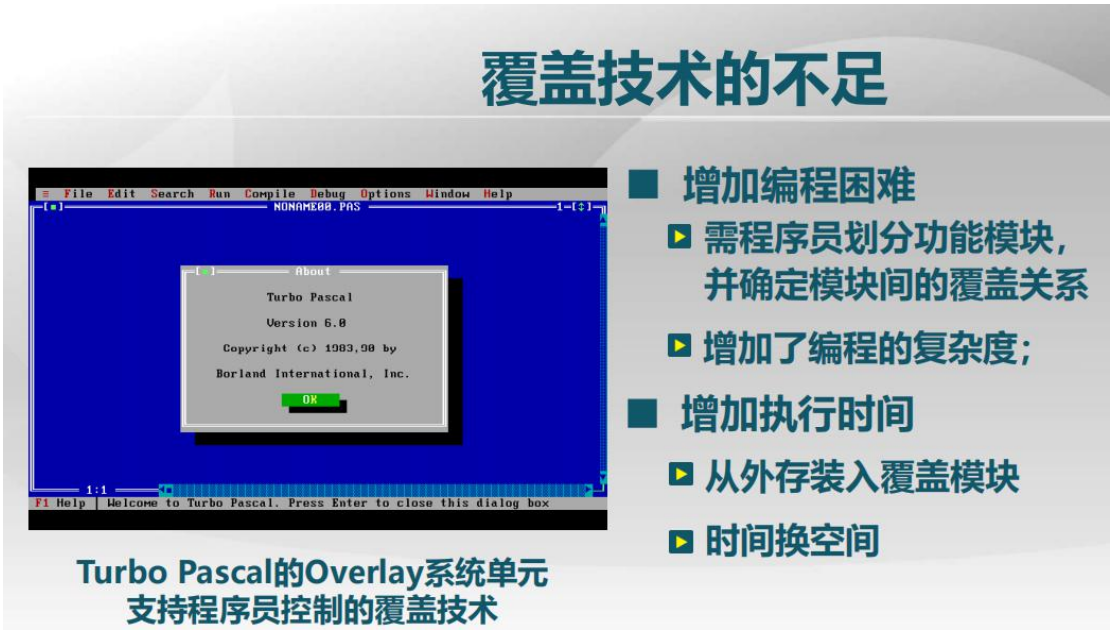


覆盖技术：有一个进程要在内存中运行，但内存较小，而程序比较大。方法见上图。



覆盖技术的具体示例见上图：一个程序总大小为 190K，分成了六个模块，各模块大小见上图，假设在一内存为 110K 的计算机系统上运行该程序，先依照调用关系分组：A 与其它所有模块有关系，它自己一组；B 与 C,E,F 没有关系，将 B,C 分为一组，D,E,F 一组。此时如何分配存储空间？每一组中按照大小最大的来分配，A 给 20K，第二组按照 B 的大小给 50K，第三组按照 F 的大小给 40K.于是程序按照 A,B,D　A,C,E　A,C,F 的顺序执行，每个执行序列都可装载到 110K 的内存当中。(动态的演示见 PPT)　　上面的做法，A 占一个分区，B,C

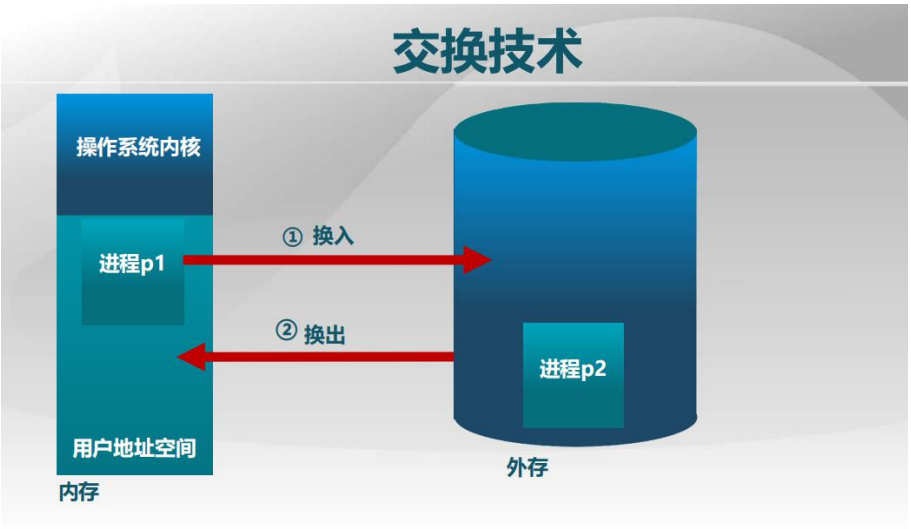
共用一个分区，D,E,F 共用一个分区是最优的吗？上图给出了互相没有调用关系的更优的一个分组，只需 100K.但实际中，确定这样一个最优分组是十分复杂的。因此，覆盖具有下图缺点：



覆盖技术需要程序员手工划分分组，增加了编程复杂度。与此同时，涉及不断从外存读入覆盖模块(需要执行时，同一分组的其他模块)，执行时间增加。

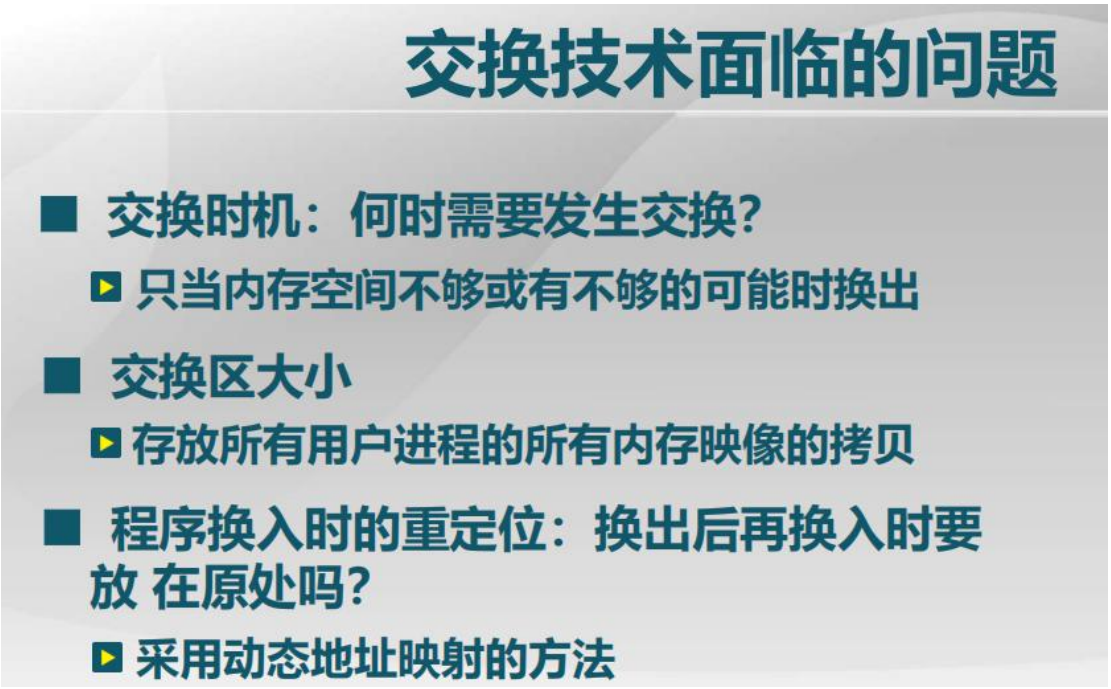
下面看交换技术：

交换技术是增加正在运行的程序或需要运行的程序的空间，这和覆盖技术关心的问题不一样，覆盖关心的是只有一个程序要运行，其内存空间就不够用了；而交换技术关心的是，只有一个程序运行，内存空间是肯定要够用的，只是当前这个程序由于多道程序运行使得另一个应用程序占用了内存空间，造成当前程序的空间不够，交换技术并不讨论只有一个程序运行，内存依然不够的情况。 注意交换技术的换入换出的基本单位是整个进程的地址空间。



(本图摘自Silberschatz, Galvin and Gagne: "Operating System Concepts")

上面是交换技术的示意图。

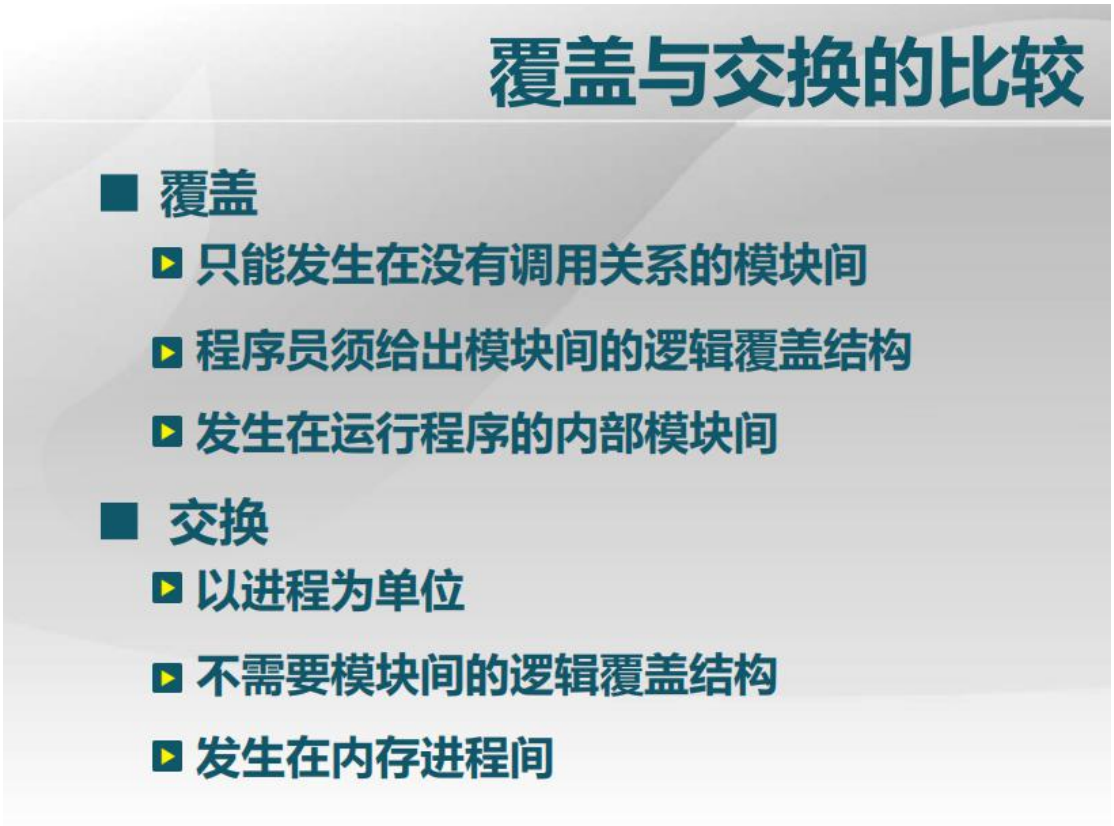


交换时机：一个正在执行的进程，它的内存空间不够用，这时必须把另外一些暂停执行的但在内存里的进程的进程地址空间兑换到外存当中，另一种情况是一个进程要执行，现在在内存中可用的空闲空间不够，这时必须把另外一些暂停执行的但在内存里的进程的进程地址空间兑换到外存当中。

外存中的交换区保存有所有的暂停的用户进程内存映像的拷贝，这要占用一定存储空间。

换出后再换入时要放在原处吗？若不放回原处，原来的函数调用或者跳转怎么办？解决

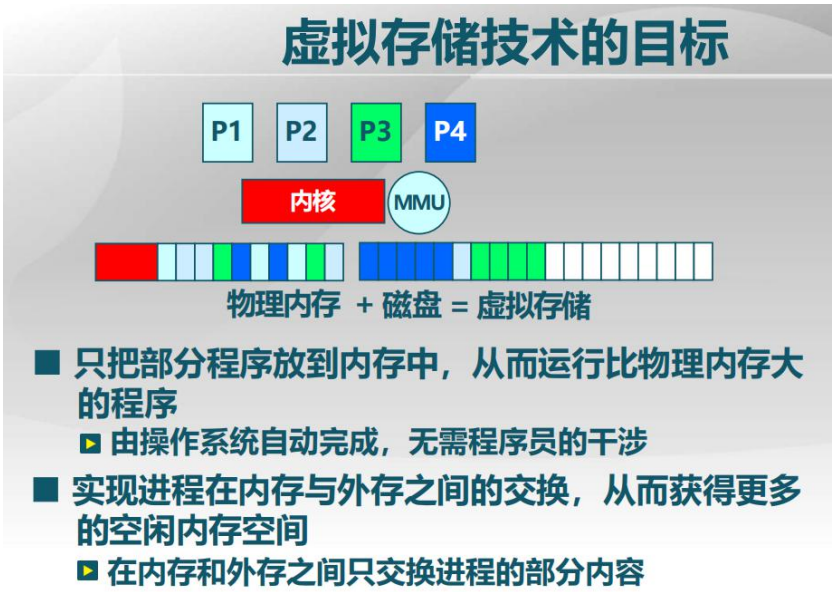
办法：不放回原处，采用动态的地址映射的办法。



覆盖中的逻辑关系确定，操作系统无法准确掌握，因此需要程序员手动给出。对于交换，操作系统是可以自动进行的。那么能否不以整个进程为单位，而是以进程地址空间的一部分为单位，导入外存中去，并且由操作系统自动完成呢？能，这是虚拟存储可以实现的。

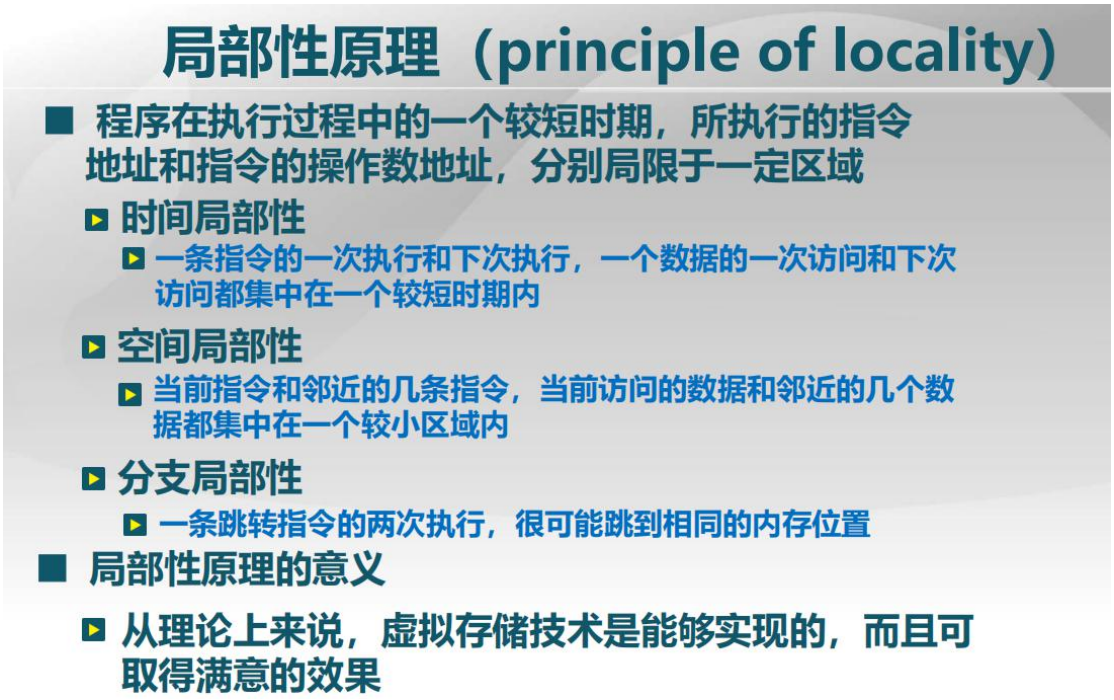
第三讲 局部性原理

虚拟存储是想把原来放到内存里的一整个进程地址空间的信息的一部分放到外存中，为实现这一点，需要有一系列的准备工作。首先看一下虚拟存储技术的目标：



虚拟存储可实现：一部分程序加载到内存中来就可以让程序运行，而之前要想让一个程序运行，必须把整个进程加到地址空间中。只加载一部分，如何能运行起来？这是操作系统管理的，操作系统可自动完成需要部分的加载，即操作系统**自动决定要把哪些东西放进来，**而不需要程序员的参与，与之对比，覆盖技术是需要程序员参与的。　　另一方面，我们可以把一部分内存空间中的信息放到外存当中去，这让正在运行的进程获得了更多的空闲空间，这也是操作系统自动完成的，它在内外存之间进行交换，操作系统可**自动决定要把哪些东西放出去，**实际上是把不常用的放到外存当中，这就是后面要讲的置换算法。

在具体讨论虚拟存储前，需要先讨论程序访问的一些特征，也就是局部性原理：

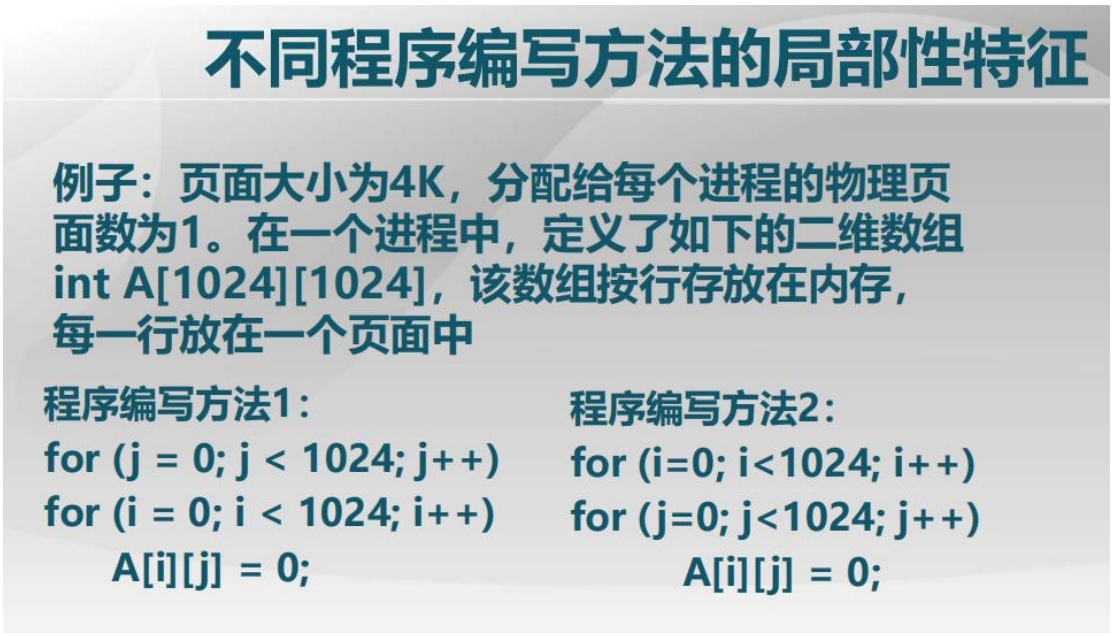


通常情况下，指令是存在代码段里的，而指令所访问的操作数是存在数据段里的，代码段和数据段所在区域不同，所以用了分别一词。

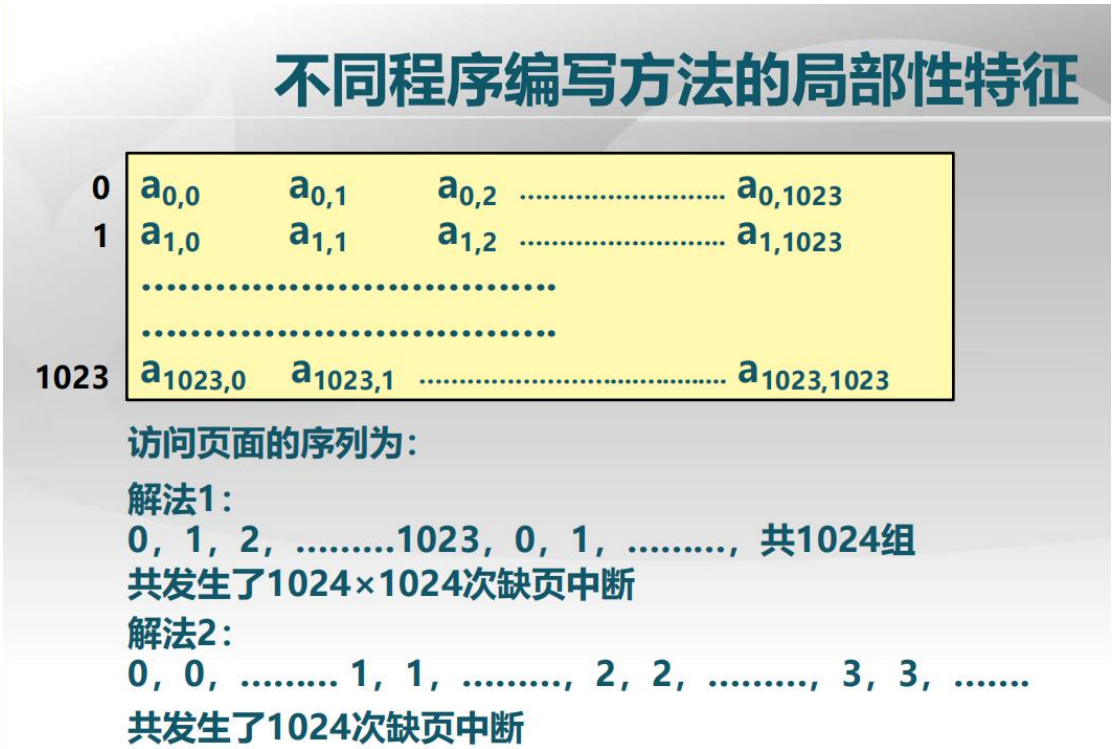
时间局部性：短时间内多次访问同一数据/指令。

分支局部性：例如一循环，进入循环体内都执行(跳转到)一个函数。

由局部性原理，我们所运行的程序它所执行的指令及访问的数据，有很好的集中特征，它们会集中在一个局部的区域里，这样，如果我们能判断清楚局限在的那个区域到底是哪些，我们能对其做很好的预测的话，这时就可以把这些内容放到内存里，而把那些不常用的放到外存当中。这种经常用的放到内存里后，我们程序执行的性能就不会有大幅度下降。也就是说，局部性原理从理论上保证了虚拟存储是可以实现的，且会有很好效果。

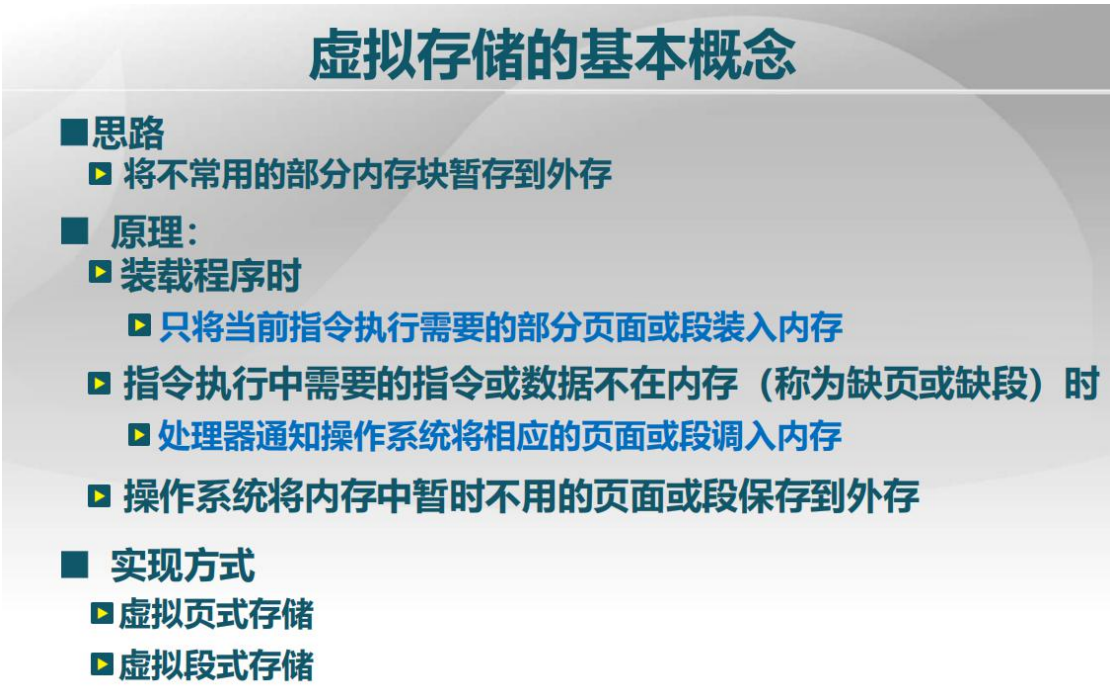


通过上图这个例子，可以说明局部性体现在哪里：一个 int 占 4 字节，那么 1024 个 int 就占用 4*1024=4028 字节=4KB. 即内存只能存 1024 个 int，于是：

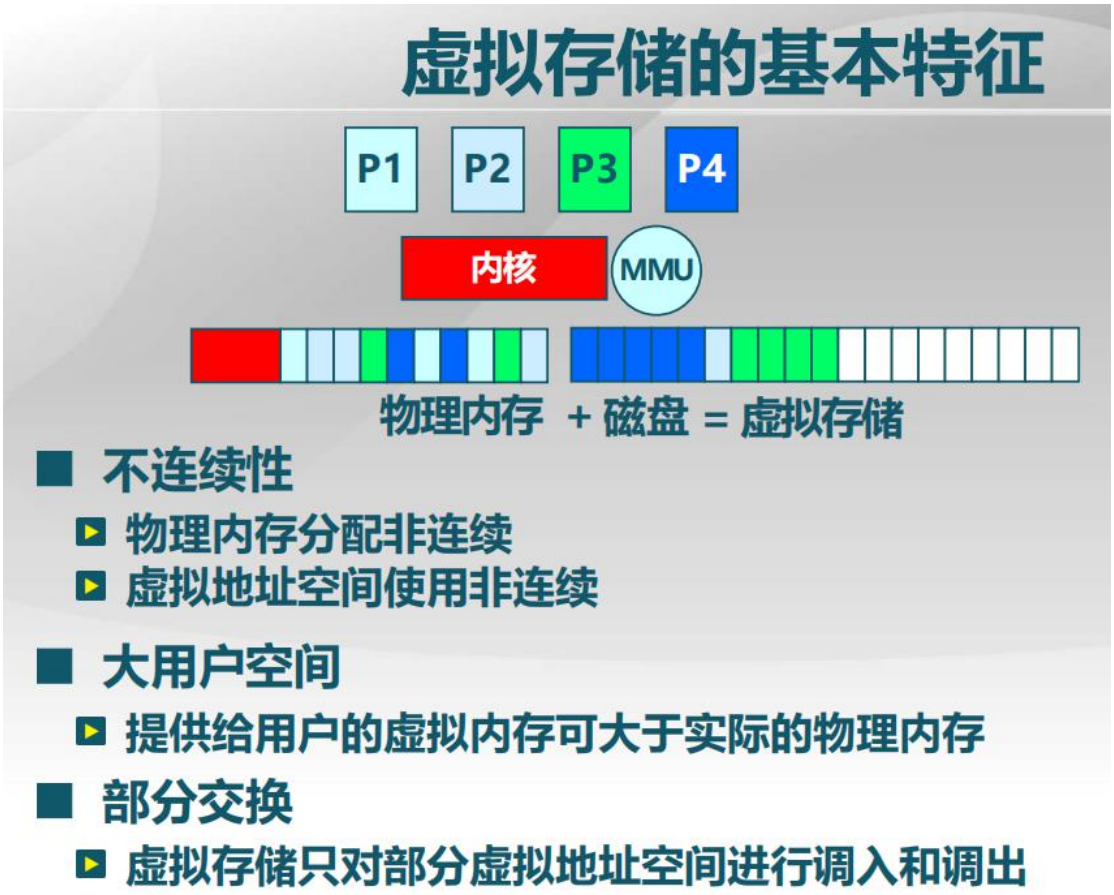


解法一每次内循环都要发生页的换入换出。解法二每外循环一次才发生页的换入换出。解法二提高了局部性特征，这有利于提高程序的性能。下面，我们看如何基于这种局部性特征，实现虚拟存储系统。

第四讲 虚拟存储概念



原来的加载是把整个进程的地址空间内容全部加载进来，现在虚拟存储变成只将当前指令执行需要的部分页面或段装入内存。操作系统会监控内存中页面或段的使用情况，将内存中暂时不用的页面或段保存到外存。

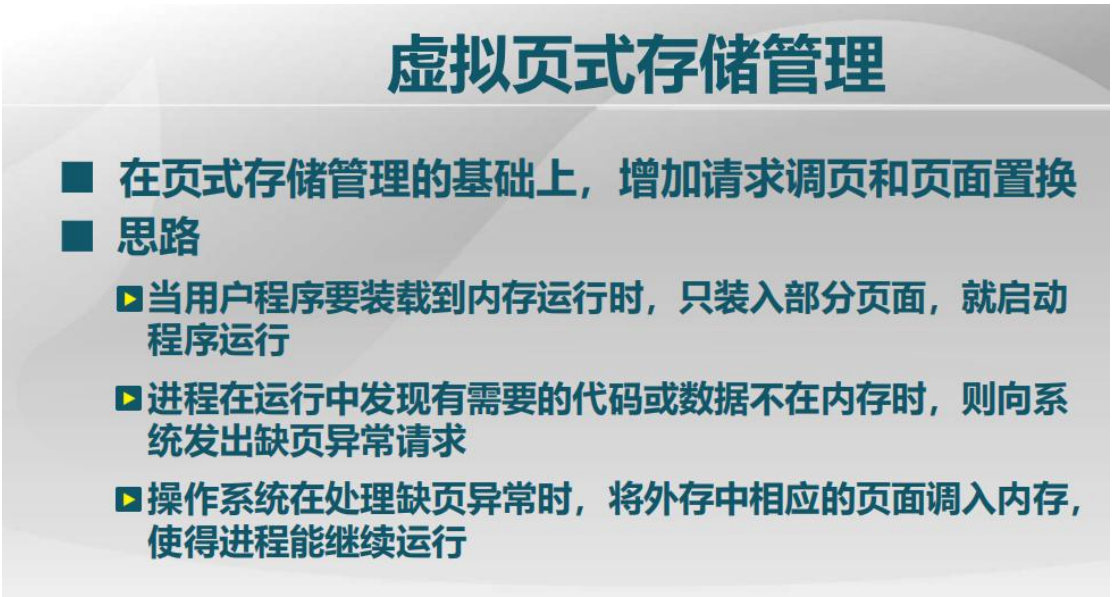


大用户空间和部分交换体现了对覆盖和交换技术的改进。

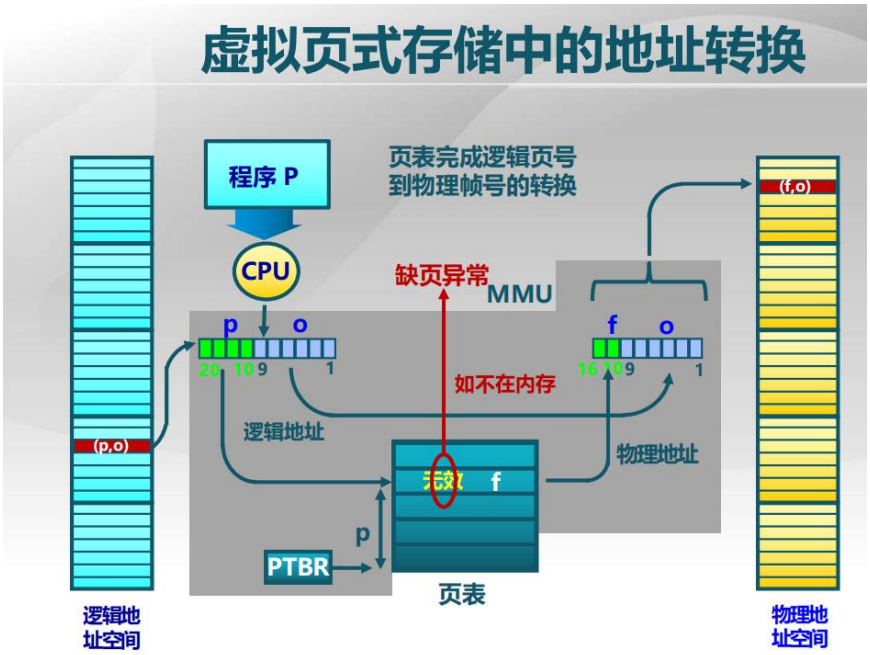
上图是实现虚拟存储需要的技术。硬件：地址转换，以及如何知道所需部分在内存还是外存。　　操作系统：执行指令时，出现异常，便可进行换入换出操作。

第五讲 虚拟页式存储

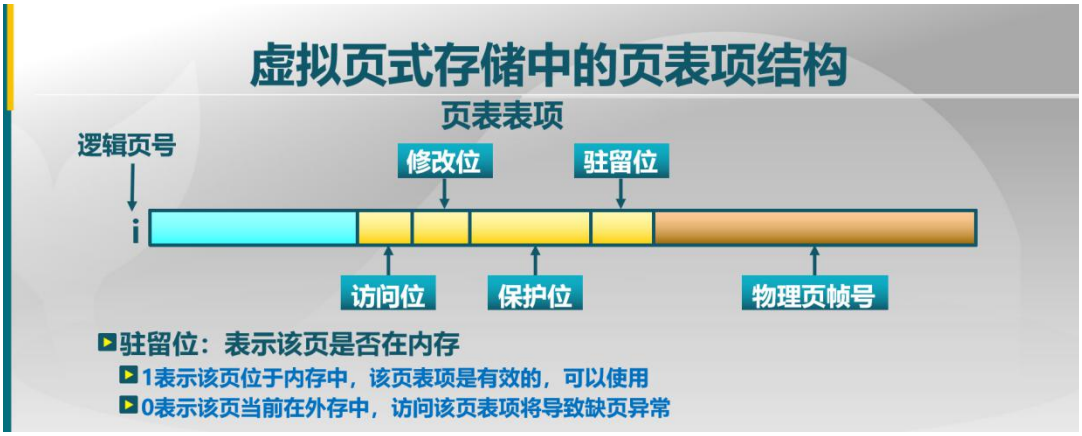
换入换出的单位设置为页之后，就变成我们本节要讲的虚拟页式存储。基本思路如下：



页式存储管理加载时是把所有页面加载到内存当中，它只是可以实现存储的不连续，即：可以找到相应页面，有空闲页面就足够了，而虚拟页式存储管理只装入部分页面。操作系统在处理缺页异常时，将外存中相应的页面调入内存，并且将相应页表项进行修改，修改完毕后，便于重新执行这条指令，使得进程能继续运行。



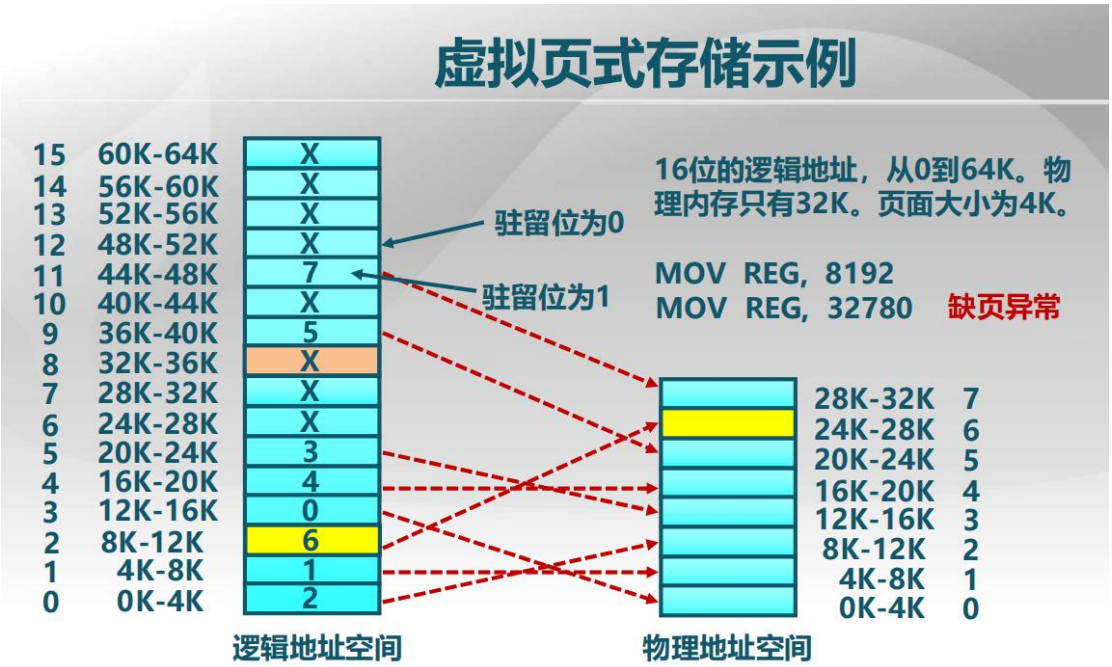
虚拟页式存储中的地址转换过程如上图所示，可见它和前面页式存储的地址转换大致是完全一样的：某一条指令访问时，通过页号 p 找到相应页表项，如果是页式存储管理，该页表项一定会有一个物理页号(页帧号)，找到页帧号再加上页内偏移 o 即得到相应物理内存单元的内容。而对于虚拟页式存储，页表中会多出一个标志位，该标志位表示对应的那一页是否在物理内存中，如果不在，刚才上面说的过程就无法进行，这时会产生缺页异常，此时操作系统接管，操作系统找到相应页，换入内存，写好页表项中的 f，将标志位变成有效。



- 修改位：表示在内存中的该页是否被修改过
  - 回收该物理页面时，据此判断是否要把它的内容写回外存
- 访问位：表示该页面是否被访问过（读或写）
  - 用于页面置换算法
- 保护位：表示该页的允许访问方式
  - 只读、可读写、可执行等

原来的页式存储管理中，页表项以逻辑页号为序号，找到的就是物理页帧号，有了物理页帧号就可以转换出相应的物理地址。但在虚拟页式存储中，我们会加上一些标志位，上图是在虚拟存储中需要用到的几个标志位。驻留位：1 即可找到页帧号，可转化为相应物理地址。0 即页在外存中，将缺页。　　修改位：在驻留位为 1 时，该位才有效，表示在内存中该页是否被修改过，若修改过，在将该页淘汰放到外存中时，就必须把内存中修改的内容写

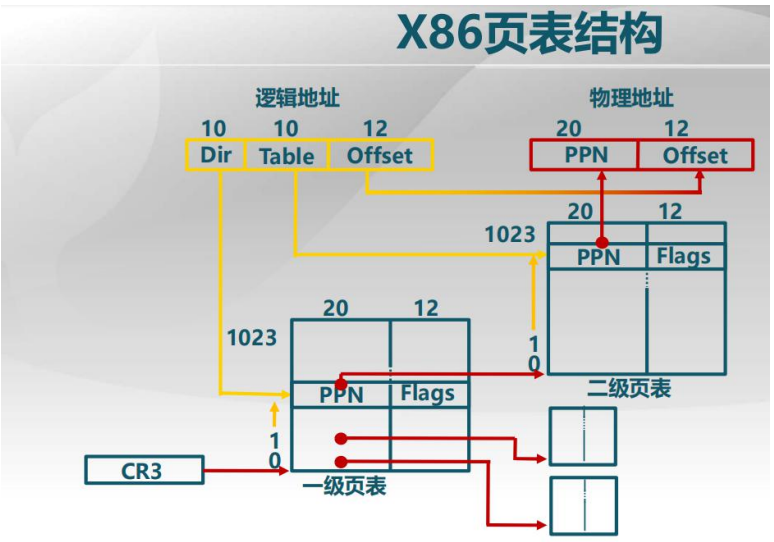
回到外存当中。若没修改过，这时外存单元中有相应的内容，这时在替换时，只需在内存中将该页作废即可，不需将其内容写回外存。　　访问位：确定该页面是否常用，近似统计出这一页是否被经常访问，1 表示经常访问，0 表示不经常访问。



16 位逻辑地址空间有 2^10=1024B=1KB　1KB*2^6=64KB 的大小，在上图中，映射到物理内存中的逻辑页上有 0-7 的物理帧编号，没有映射过来的逻辑页上写的 X.这时逻辑页上写 X 的隐含其页表项中驻留位是 0.

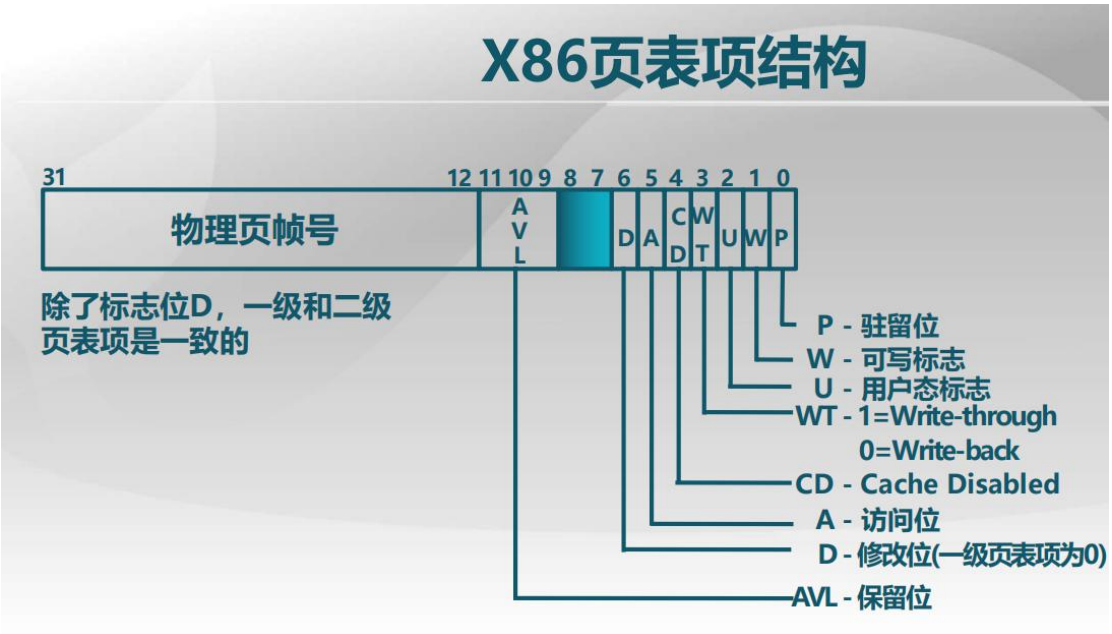
此时执行指令：MOV REG,8192 即把指定存储单元内容移到寄存器里，由于 8*1024=8192，因此找逻辑页号为 2 的逻辑页，看图，于是访问帧号为 6 的物理帧。

再执行指令：MOV REG,32780 由于 32*1024=32768 因此因此找逻辑页号为 8 的逻辑页，看图，产生缺页异常，缺页后，需要把现在在内存里某一页去掉，然后把逻辑页号为 8 对应的内容写到内存中，并修改其对应驻留位为 1 即可。



以现在常用的 X86-32 的 CPU 页表结构为例，在 32 位的 X86 系统中，逻辑地址：有 12 位的页偏移(offset),有两个 10 位的二级页表项(dir,table)，共 32 位地址。　　物理地址也

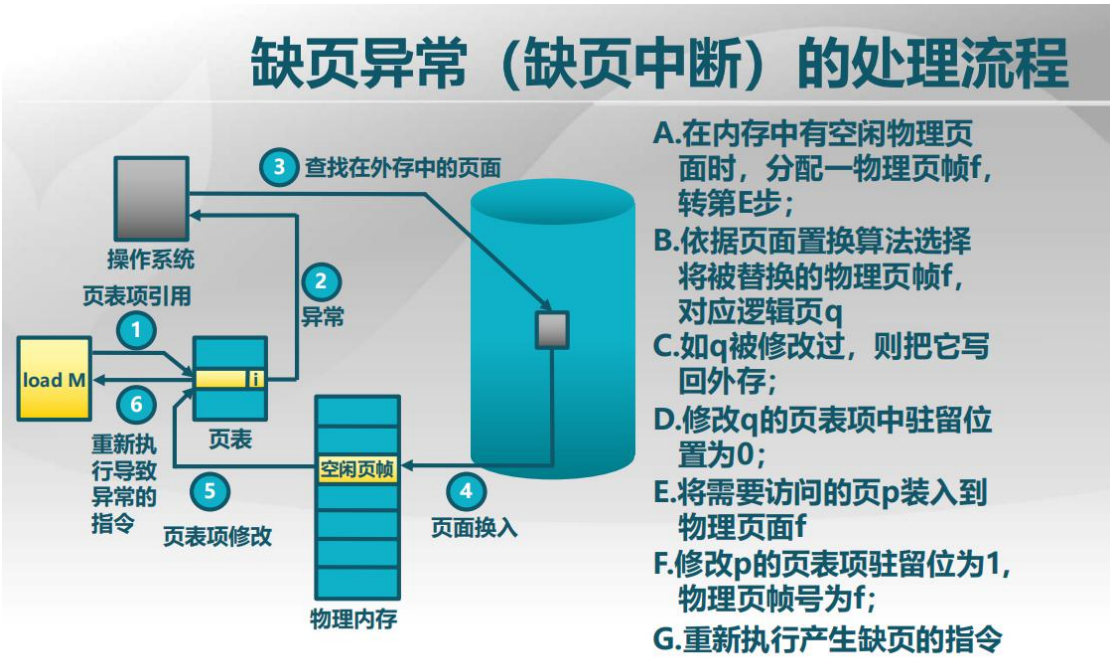
是 32 位，其中 20 位是物理页帧号(PPN)。页表结构中，一级页表的起始地址在 CR3 中，即 CPU 中的一个寄存器指出一级页表的起始地址，再利用 dir(一级页号)作为偏移,得到相应一级页表项，这个页表项中有第二级页表的物理页号 PPN，再利用 Table(二级页号)，以二级页号作为偏移找到相应页表项，得到物理页面的物理帧号 PPN，该帧号和偏移 offset 放在一起，就得到物理地址。这个和之前讲的页式存储完全一样。



变化的地方在于页表项里的东西，上图是 X86-32 以 4K 为页面大小时，页表项的定义格式。这里 12-31 位是 20 位的物理页帧号，我们更关心后面这一段，驻留位、可写位、访问位、修改位前面已经介绍过。用户态标志表示页表项是否可以在用户态访问，保留位 AVL 是标志位没用完剩下来几位，原因在于如果我们在 32 位的 X86 系统中上图这个页表项是好用的，但在实际系统中，物理地址空间是在不断变化的，如 32 位的系统，最大物理内存地址空间是 4G，实际上我们现在用到的一些 32 位系统，已经不是 4G 了，大于 4G 怎么办？相应的页表项就要改变，这样保留位为后续的改动留有空间。CD:在内存与 CPU 之间有一个高速缓存，缓存在读写数据的时候，会把写出的数据先写到缓存里，高速缓存再慢慢把数据写到内存中，如果我要有一些时效性的操作，比如对 I/O 端口的操作，这种缓存会影响我的语意，此时便可以用 CD 标志来做控制，另外，可控制在读的时候高速缓存是否有效，如果前面读了一次，这时，后面再读时可从高速缓存中得到这个数据，但若这个数据它是时时在变的，就要禁止高速缓存，而是去 I/O 端口实实在在的读。

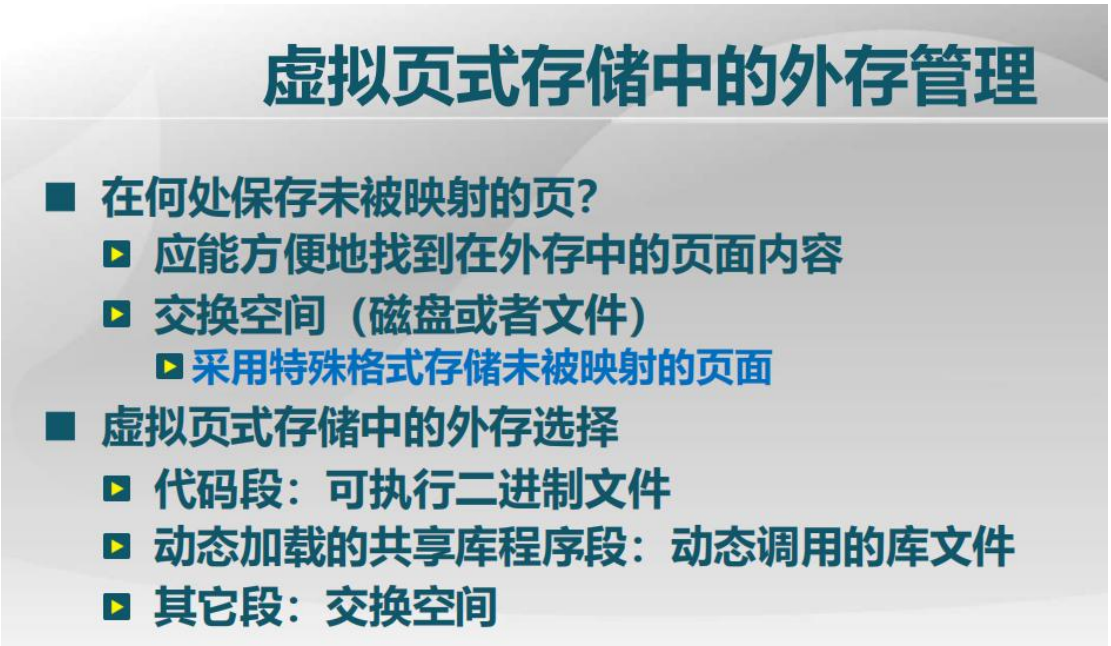
第六讲 缺页异常

缺页异常就是发现页表项中这一页不在物理内存当中，产生缺页异常后，会把相应内容读到内存中来再重新执行这条指令。实际上，在缺页异常中还有很多要处理的内容。



在 CPU 中要访问一条指令：①load M，这条指令会去找 M 所对应的页表项，找到这一项后，如果这一项是有效的(驻留位为 1)，那就直接去物理内存中访问去了。②如果这一页无效(驻留位为 0)，这是就会产生缺页异常，缺页异常就会导致操作系统的缺页异常服务例程的执行。③缺页异常产生后，缺页异常服务例程首先要找到对应那一页在外存中的什么地方 ④当在外存中找到这一页，需要将页面换入物理内存 ⑤换入完成后，要修改相应页表项(修改页帧号，驻留位改为 1) ⑥修改完后，重新执行这条指令。

上述过程中，④页面换入需要进一步细化，细化后的流程如 A,B,C,D,E,F,G 所示。



上图介绍了，在缺页异常产生后，如何去找外存中的页。在现行的操作系统中，有两种做法：①直接做一个分区叫做兑换区，在 Linux Unix 中都是这么做的 ②用一个文件来存这些东西，在文件中采用特殊的格式来映射这些页面:因为这里通常都是固定大小的页面，所以可以针对这一点做优化。　　以上说了外存中的页放在哪儿。

但也不是所有东西都放上面说的那些里。因为我们在外存当中的这些进程地址空间里的内容，有一些是可以放到兑换区、特殊文件里的，而另外一些，可能选择另外地方：①一种是代码，代码本来是在可执行文件里的，如果将它复制到兑换区和特殊文件里保存是没有必要的，所以这时代码是直接指向可执行二进制文件。 ②还有一些就是共享库，这些库实际上也有相应目标文件来存放，没必要把它们复制到兑换区及文件中，因为上面说的这两项内容(代码、库)我们都是不去修改的。 其他的数据段、堆栈段这些就可以放到兑换区(交换空间)中去。

最后，我们讨论虚拟页式存储管理的性能，我们用有效存储访问时间来衡量虚拟页式存储管理的性能。注意后一项，5000000p(1+q)是怎么来的，缺页发生后，一定是要从外存中读入缺失的页的，这一部分是 5000000p,而换出的页如果修改过是不能直接丢弃，而是需要写入外存的，这一项对应 5000000pq. 注意 ns-us-ms 之间都是 10^3 的数量级换算。

如果最后想让有效存储访问时间和 10ns 不相上下，就要求缺页率 p 必须足够小。