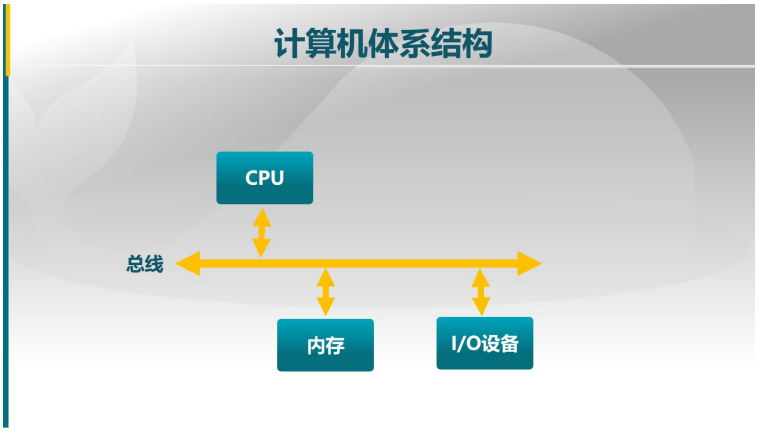
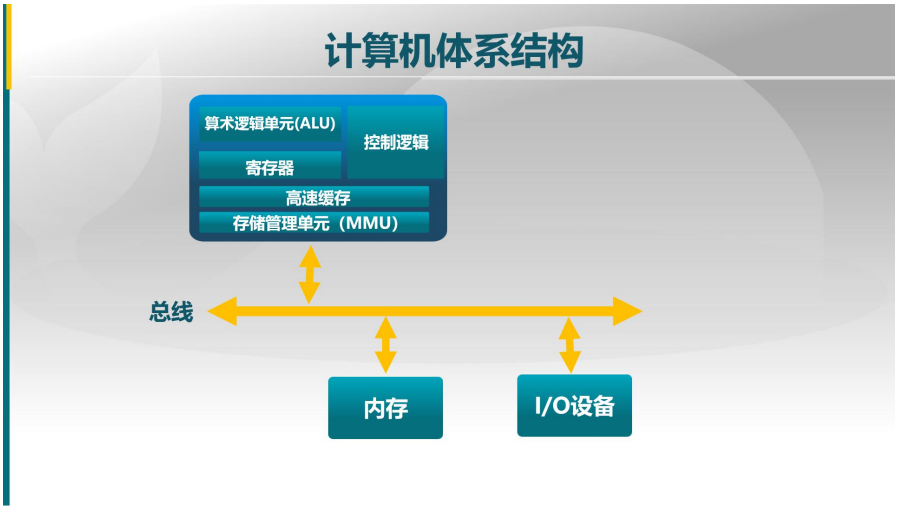
第五讲 物理内存管理：连续内存分配

第一节 计算机体系结构和内存层次

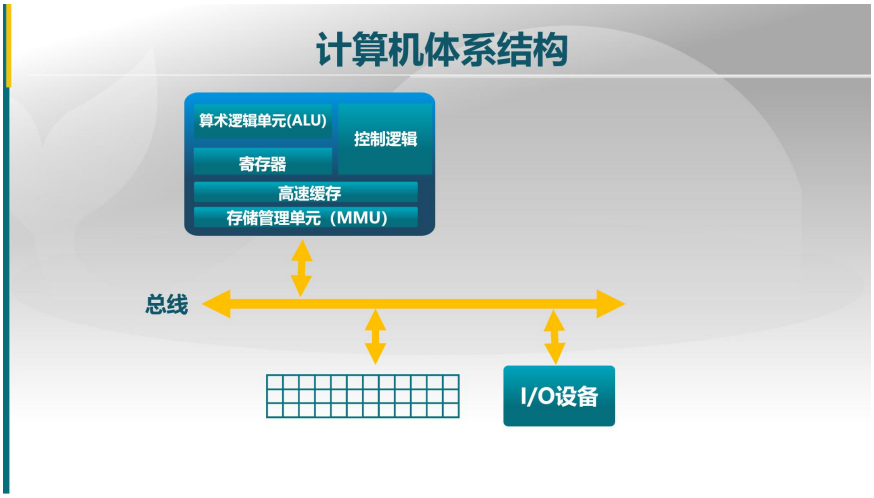
从这节开始是存储管理，计算机除了处理能力，还有存储能力，存储能力相当于我们有一系列基本存储介质，我们要在这些介质中存储代码和数据。为此，计算机系统在体系结构中约定了哪些地方可以来存数据。存数据的地方，包括 CPU 中的寄存器，内存，外存，这几种不同的存储介质，它的容量、速度、价格都是不一样的，为了组织一个合理的系统，我们把计算机系统当中的存储组织成了一个层次结构。针对这种层次结构下的存储单元，操作系统需要对它进行管理。操作系统中的存储管理，实际上就是用来管理这些存储介质的，最基本的管理要求是说我们一个进程要使用存储单元时，需要从操作系统分一块给它。当它不用，就还给操作系统。这是最基本的分配和释放的管理要求。



计算机系统包括 CPU,内存和 I/O 设备，CPU 在加电的时候，我们关心各个寄存器的初始状态，现在我们关注更多与存储有关的内容。

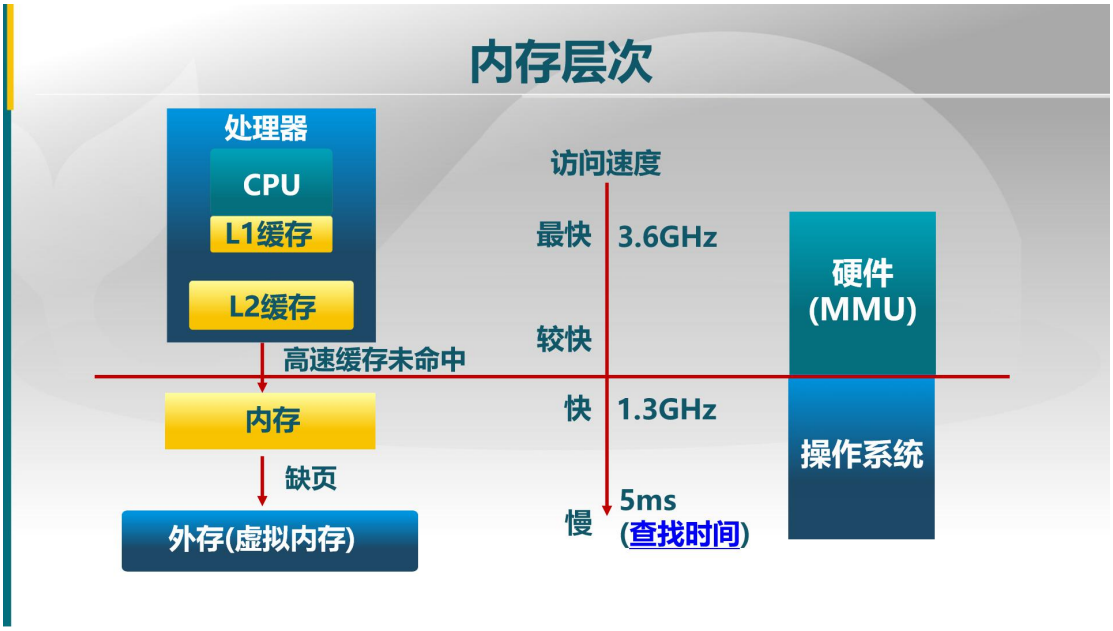


比如在 CPU 中，可以往寄存器里存内容，寄存器可以存数据，但是寄存器的容量是非常小的，通常是 32 位、64 位的寄存器，能存的数据也就几十个字节或者几百个字节这种尺度。



我们说内存是更多的存数据的地方，计算机系统中内存的最小访问单位是字节，也就是8bit，而我们通常所说的计算机系统是 32 位的总线，所谓 32 位总线，相当于一次读写可以从内存当中读或写 32 位，即 4 字节，这样一来读写的速度就快了。针对这种特点，一次读写 32 位有地址对齐的事，因此在访问时就不能从任意的地方开始一个四字节，有可能这个读写会被分成两次。 可以看到，在 CPU 中，还有高速缓存：在进行读写指令或指令执行过程中，访问数据都需要从内存里读数据，这时如果有大量数据要读写，而且这些数据会重复利用的话，在 CPU 中加上高速缓存，读写速度就会更快，读写效率会提高。以上部分都对存储管理有至关重要的影响。

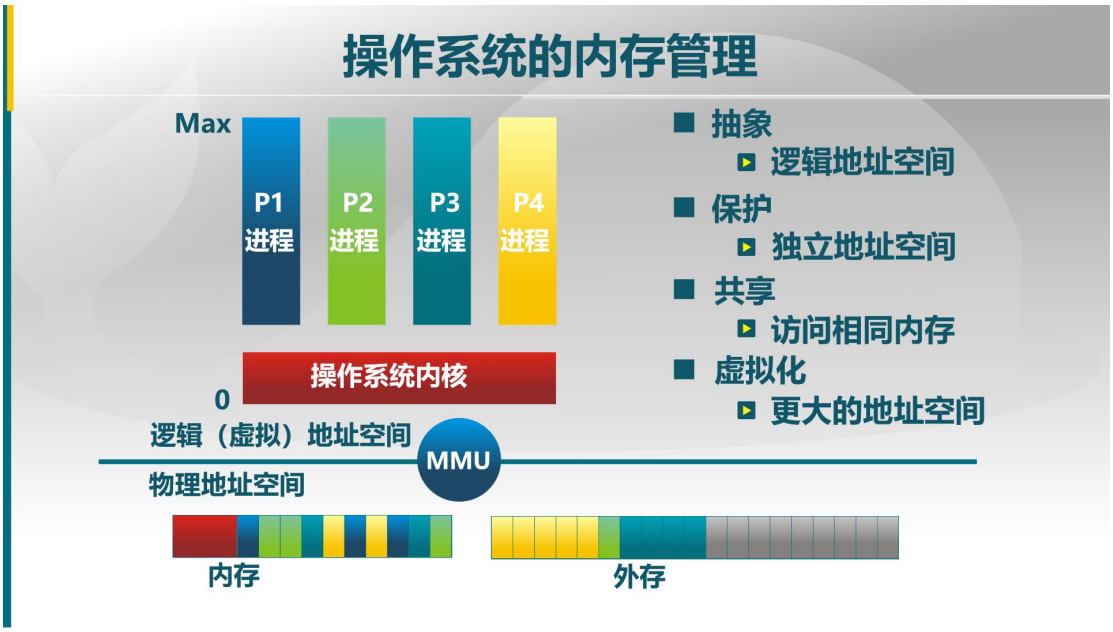
更详细的存储层次结构：



在 CPU 中有两级缓存，这两级缓存，如果在读写数据或者指令时，在缓存中已经有相应的内容(事先已经读过)，那就可以直接从缓存中拿到，这时速度是最快的。如果在这里缓存不命中，就必须去内存中读，而上面这部分 L1 缓存，L2 缓存，在写程序时是感受不到它们的，为什么呢？因为这一部分完全是硬件在做控制，我们写的程序不能显式的用到它们。而内存的访问需要操作系统的控制。如果在内存中访问时仍然找不到，这是便可能存在外存里面，从外存中读取也需要操作系统的控制。在这个体系结构中，从 CPU 的外部一直到硬盘外部，速度相差是非常大的。最快和 CPU 主频是一样的 3.6GHz(几纳秒)，最慢的是几个毫

秒(毫秒 微妙 纳秒)，这两者间差了百万的数量级，因此要在这个体系中，要把它协调成一个有机的整体，对于存储管理来讲，挑战是很大的。

存储管理最后要达到什么效果？描述如下：



首先：系统当中的存储内存是以字节为单位进行访问，每一个字节有自己的一个地址，这个地址是物理地址。如果数据存到外存里了，比如像磁盘，磁盘的访问有扇区编号，每一个扇区是 512 字节最小单位，这是能读写存储的最基本的内容。而写程序时我们希望看到的情况是：我有若干个进程，每一个进程它们都有共同的一部分地址空间，是操作系统的内核。而每一个应用程序自己又是不一样的，它们各自有各自的内容，我们希望在各自写这些内容的时候，它们的地址是可以重叠的，相互之间是不干扰的，为此在中间加一层存储管理单元，存储管理单元就把逻辑地址空间转变成物理地址空间。实际的操作系统代码通常存在内存里，而进程的地址空间随着它们运行的转换，有些在内存里，有些是放在外存里的，这个转换的过程就由中间的存储管理单元来完成。

以上也就是说：存储管理要达到的效果是抽象，即把线性的物理地址编号转变成抽象的逻辑地址空间，在这里需要对地址空间进行保护，每一个进程只能访问自己的空间，尽管在内存里它们是相邻存放的。与此同时，还要方便共享，比如操作系统内核的代码，各个进程都是一样的，要实现此部分的共享，而不是保护，因此要把上面说到的保护和共享统一起来。这个目标是有一些矛盾的，与此同时，还希望实现更好的虚拟化，即每个进程的地址空间其地址空间编号都是一样的，但实际上每个进程相应的用户地址空间对应的物理地址空间是不一样的，但给每个进程看到的都是一个区域一致的一个地址空间，甚至说逻辑地址空间里看到的可以存数据的地方大小是大于物理内存的总量的。

操作系统中采用的内存管理方式：

重定位(relocation)，实际上在最早的计算机系统中是直接使用总线上的物理地址来写程序，我要想读写某个内存单元，它在什么位置，我们在程序里见到的就是它的物理地址，但这种做法有很大局限，即你写的程序只能在指定类型的机器上运行。重定位就是实现段地址加一个偏移来表示的，有了重定位，通过移，我只需要改变段寄存器的地址，为了实现它，在操作系统和程序中都要有相应的支持。

分段(segmentation)，在重定位时，每一个进程分的存储空间是一个连续的空间，这

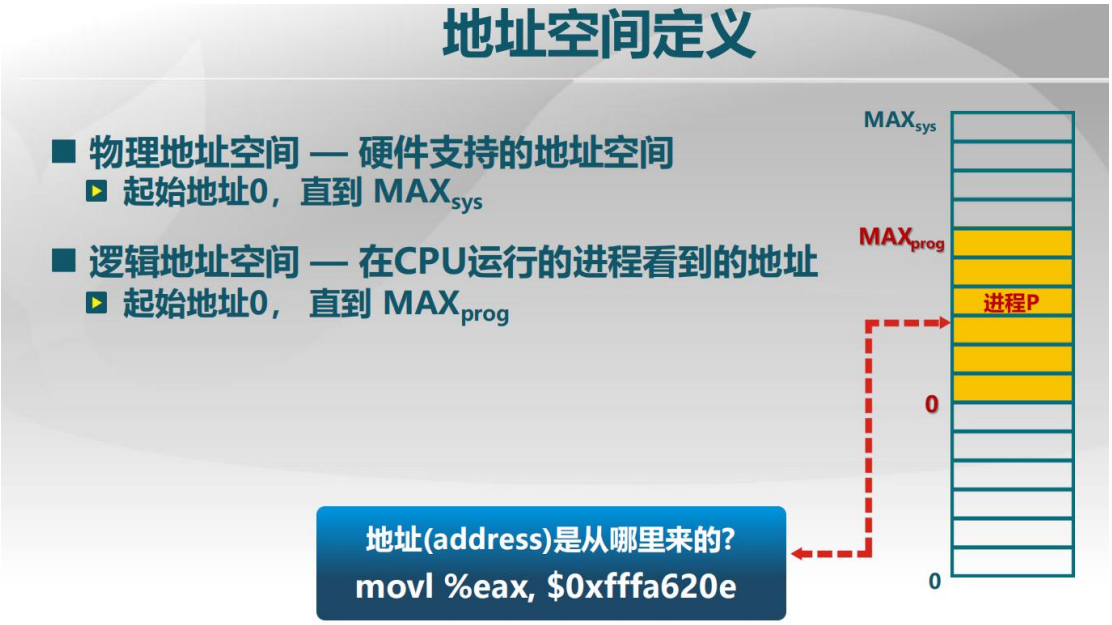
其实是一个很大的限制，我们希望这个空间能够不连续，实际上我们在写程序时，它的逻辑结构并不是一个必须连成一片的区域，而是把程序分成了数据、代码、堆栈，这三个部分是相对独立的，不会从堆栈里直接去访问代码段里的内容，也很少有从代码段里直接去访问数据段的内容，依据这种情况我们至少可以分成数据、代码、堆栈三块，每一块要的空间就变小了，这就是分段，分段仍然需要一段的内容是连续的，这个要求依然足够高，接下来就有了分页：

分页(paging),分页就是把内存分成最基本的单位，我们从页来构建所需的存储区域。**问题：那用字节不就行了吗？答：**这样的话，在访问时开销粒度太细，以至于管理时难度很高，所以我们要选一个合适的大小，这一块最基本的单位是一个连续区域，即页，基于这个来构造所需的存储空间的内容。在分页基础之上，我们希望把数据存到硬盘上，而硬盘外存和内存上的数据之间的倒换是由操作系统来实现，这时希望程序看到的是一个逻辑的地址空间，甚至于这个逻辑地址空间是大于物理内存空间的，于是：

虚拟存储(virtual memory)出现了：目前多数系统(如 Linux)采用按需页式虚拟存储。

以上实现高度依赖硬件，与计算机存储架构紧耦合。比如说 MMU(内存管理单元)是处理 CPU 存储访问请求的硬件。

第二节 地址空间和地址生成

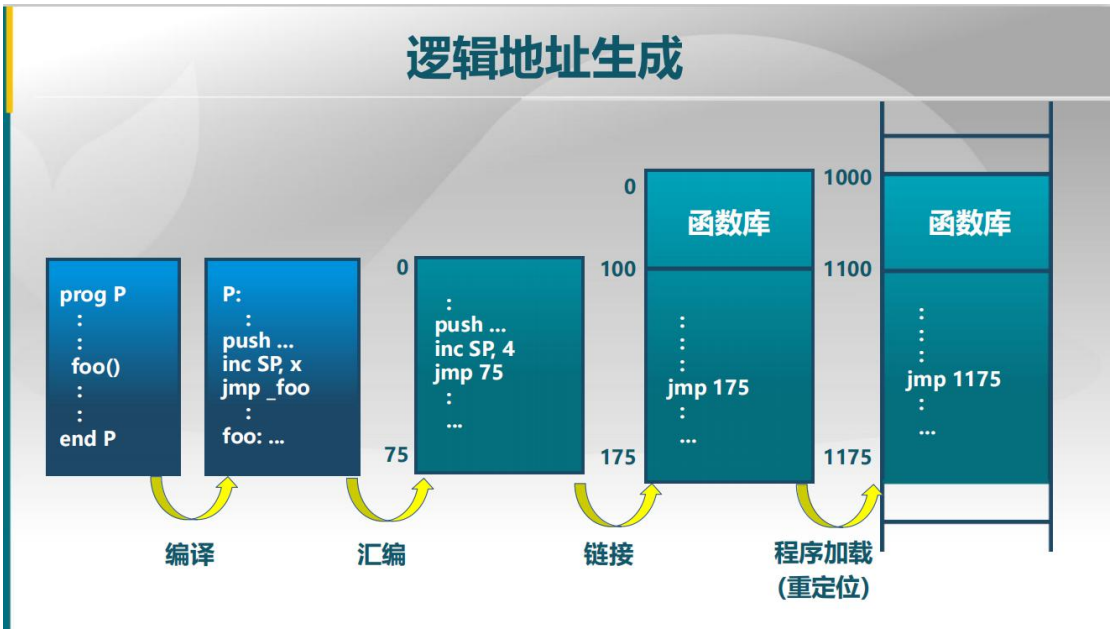


地址空间的定义：我们在机器里总线上看到的地址是物理地址，所有的物理地址所构成的空间叫做物理地址空间，它是由硬件支持的，通常情况下我们说有多少位地址总线，指的就是这里的物理地址总线的条数，比如 32 位的，通常情况下就是 32 条地址线，它的标号应是从 0 到 4G-1(MAX sys),这个编号在存储单元角度来讲是唯一的，但这种唯一对于我们写程序来讲是不太容易用的，因为我到底用哪个地址，在写程序之前或者运行之前可能不知道。

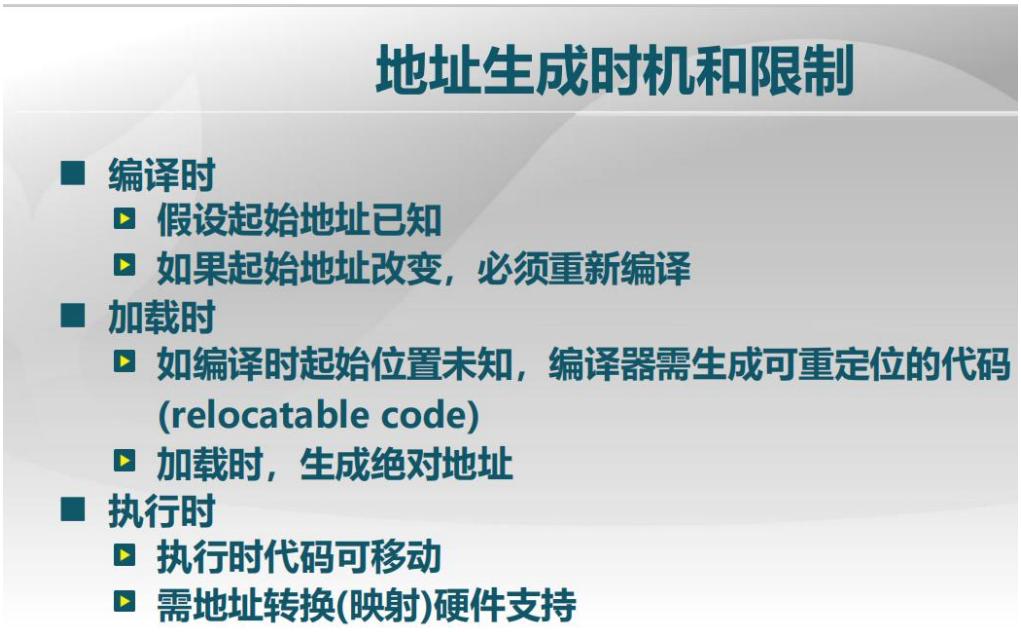
这样便用到逻辑地址空间：在 CPU 运行的时候，进程看到的地址空间，通常情况下，对应可执行文件里的那一段区域，加载程序的时候，你的程序加载到内存中变成进程，这时在你的可执行文件里的 0 到它最大值(MAX prog)，这个地方在相应地址空间里有一段区域，这段区域就是我们进程的逻辑地址空间，逻辑地址转换成物理地址就是我们后面要讲到的。

那么这时我们这里访问到一条指令：movl %eax, $0xfffa620e，这条指令在执行的过程中会访问相应的内存单元，**这些内存单元的地址从哪里来？答：**就是从逻辑地址根据后面讲的方法转换成物理地址，最后在总线上访问相应存储单元。

下面看逻辑地址的生成：



通常情况下，我们写程序都是用高级语言，上面是个例子：一个程序，prog 和 end 是程序的开始和结束标志，中间调用了一个函数 foo()，这个函数实际上就是一个地址。程序的源代码 CPU 是没法直接认识的,为此进行一次编译，转变成机器能认识的指令：汇编码。之后对汇编码进行一次汇编，汇编之后就变成二进制代码了，这时才是实实在在机器能认识的指令，这时里面的符号就不能再是前面的字符串如_foo 了，而是变成了 75 这些(地址空间里的某一个位置,jmp 75 即从当前位置蹦到 75，这用的就是编号)。另外，我们可能用到别的符号(别的地址),比如我有一个函数调用，从模块 A 调用了模块 B 中的一个函数，在这个调用过程中，在你做汇编的时候，另一个模块的位置你并不知道，这是要有一个链接的过程，把多个模块和你用到的函数库，放在一起，排成一个线性的序列，这时便可以知道跳转的另一个符号的位置在哪里，从而变成上图第四个样子，函数库被加到前面 0-100，foo 跳转的位置由 75 变为 175.这里的 0-175 只是在这一个文件内的表述，如果这个程序现在去运行，那么运行时不一定正好能放到 0 的位置，那么在加载的的时候，还会有一个重定位，重定位是说原来的 0-175 加载进来之后放到了 1000-1175 的位置，相应的，jmp 175 变为 jmp 1175,这是程序加载时由操作系统提供的重定位的功能要做的事，这样，我们程序在跑的时候就变成实实在在的地址了。以上是逻辑地址。



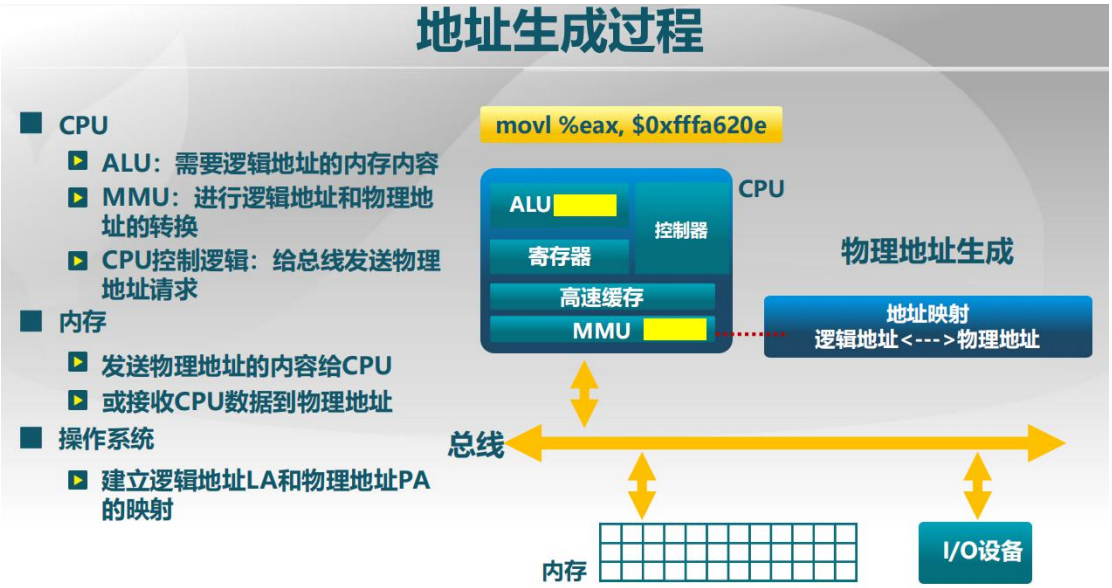
上图说明了地址生成的时机的几种情况：

1.编译时：假设起始地址已知(假定我知道我最后要放的位置，那在编译时就可以把这个地址写死) 但，如果起始地址改变，必须重新编译。这种情况，现在在何处出现？我们用到手机，如果手机是功能机，不是智能机，这时里面程序通常是写死的。

2.加载时：(允许你加载到不同地方，比如智能机，可以在买到后向其中再加我的程序，这时写程序的人无法知道程序最后会加载到系统的什么地方去)如编译时起始位置未知，编译器需生成可重定位的代码 (relocatable code) (通常，可执行文件里前有一个重定位表，它其中就包含了这个程序中到底有哪些地方需要去改的，加载时，都改成绝对地址，这时程序就能跑了，也就是我们上上图分析的情况)加载时，生成绝对地址

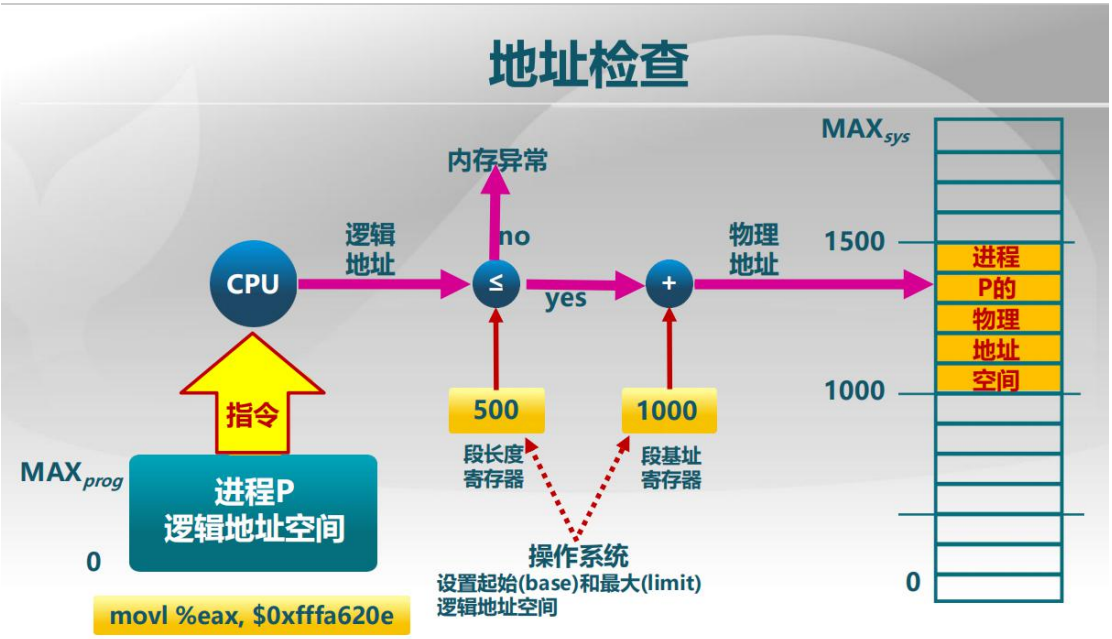
3.执行时：(相当于我们前面用的一直是相对地址，执行到这条指令时，才可以确切的知道它访问的是什么地方，这种情况出现在我们使用虚拟存储的系统中)执行时代码可移动需地址转换(映射)硬件支持

优点：在程序的执行过程中，可以把它在物理存储里的位置进行挪动。而如果是前面两种情况的话，不但要求你在地址空间是连续的，同时运行起来之后，不能再进行改动，比如在加载时做了重定位，已经写了绝对地址了，由于存储空间不够或别的程序存储空间不够，它的位置被往后挪了一段，这么挪完之后，程序中已有的地址就不对了。所以从灵活性的角度来讲，我们在执行时生成这个地址是最好的。而前面两种的好处是简单，所以在这里，不同的系统这几种做法现在都是有采用的。



上面用图示展示了地址的生成过程：CPU 当前正在执行一条指令：

movl %eax, $0xfffa620e，这条指令在执行时里面有地址，这个地址在 CPU 里先看到了，这时 MMU 依据页表完成从逻辑地址到物理地址的转换，翻译成物理地址之后，CPU 中有一个控制器，控制器负责把你得到的物理地址和相关的总线控制信号送到总线上。这时，存储单元，存储芯片这时会识别总线上的地址和控制信号，依据控制信号到底是读还是写，总线上有一组相应的持续逻辑的交互。如果是写，就会把 CPU 送来的数据写到内存当中指定的存储单元上。如果是读，那就从内存当中指定的内存单元中读出数据放到数据总线上，然后 CPU 拿回去。（**在这个交互过程中，CPU 做什么？** CPU 在地址转换过程中，它有影响？？？），实际上在每一次访问的时候，它是不依赖于软件的，是由硬件来完成这个转换的。但这个转换的表，是可以通过操作系统建立逻辑地址和物理地址两者之间的关系(这是页表的功劳)。

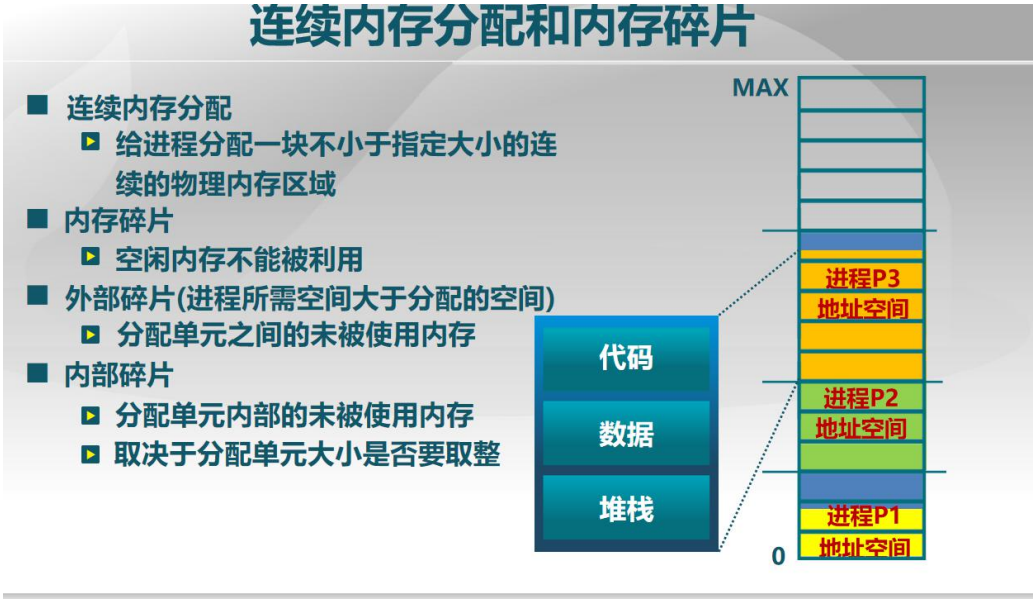


接下来讨论地址生成过程中的地址检查。一条指令：movl %eax, $0xfffa620e，在执行这条指令的过程中，会产生逻辑地址，这个逻辑地址，比如我访问的是数据段的数据，这时数据段有一个段基址和段的长度，若从数据段去访问的偏移量超过这个长度，这时这个访问应

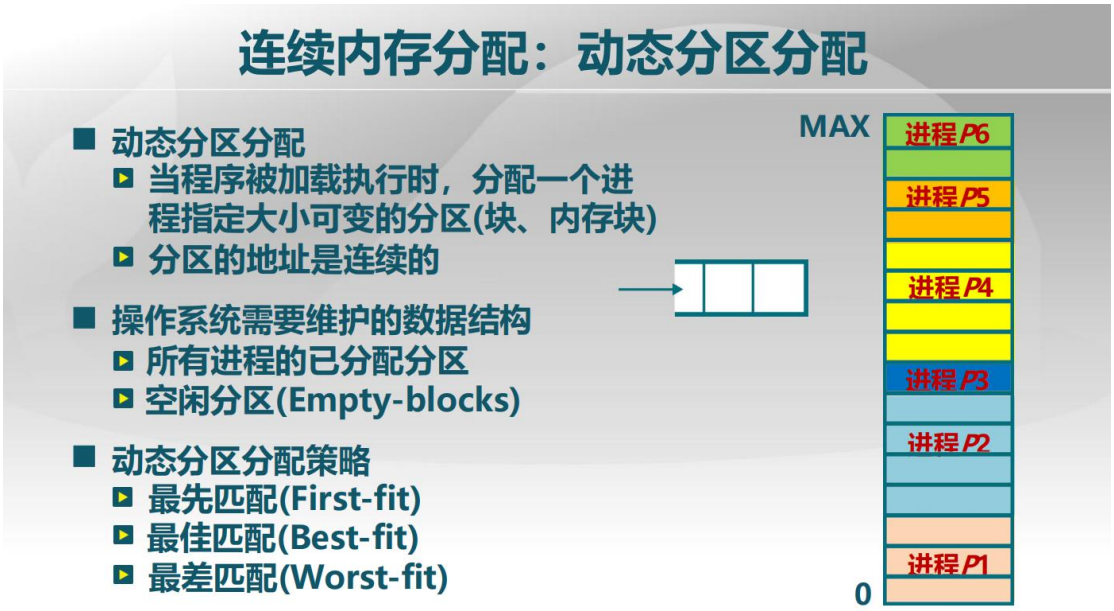
该是非法的。否则访问的偏移量在 0 和最大长度(上图是 500)之间，那便是合法的，这时偏移便会和段基址加在一起得到物理地址，从而访问到对应进程的物理地址空间中去。在这个过程中，操作系统可以用指令来设置相应的段长度和段基址，这也就通过软件的方法来影响到了做相应检查的过程。有了这个检查之后，我们就有了从符号到逻辑地址，逻辑地址在执行过程中转变成物理地址，并且在这个过程中有相应的检查机制。

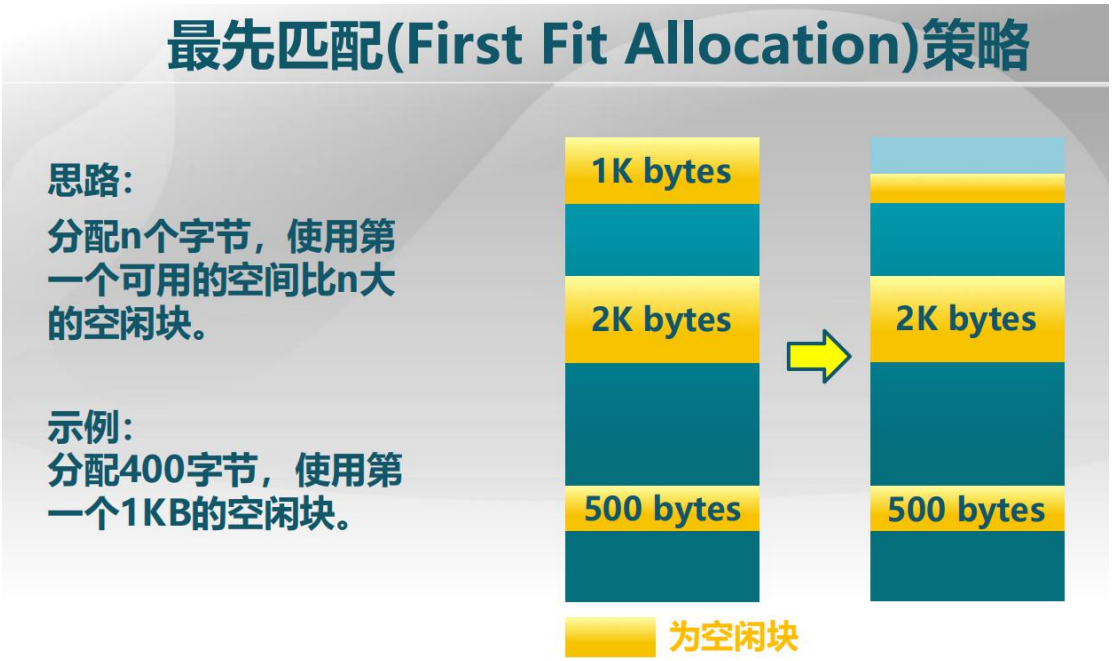
第三讲 连续内存分配

在分配内存空间的时候，没有其他技术支持情况下，分配给一个进程的地址空间必须是连续的。为了提高利用效率，希望分的位置有适度的选择。动态分配算法：最先匹配、最佳匹配、最差匹配实际上就是在选择分配内存空间的方法。分配完内存之后，每个进程利用内存时间的长短可能不同，有些进程先结束，有些后结束，这样先结束的就会留出一些空位，这样有些进程在后面，中间就会留下一些碎片，这些碎片对于我们后续的分配是会有影响的。



详细的内存碎片，内部碎片，外部碎片图片示意见 PPT。





是从上往下找的，分配完之后如上图最右边所示。(400 字节=400bytes)

最佳匹配：

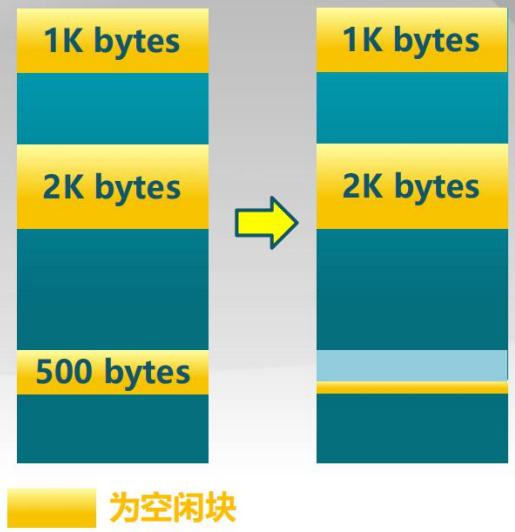
最佳匹配(Best Fit Allocation)策略

思路:

分配n字节分区时， 查找并使用不小于n的最小空闲分区

示例:

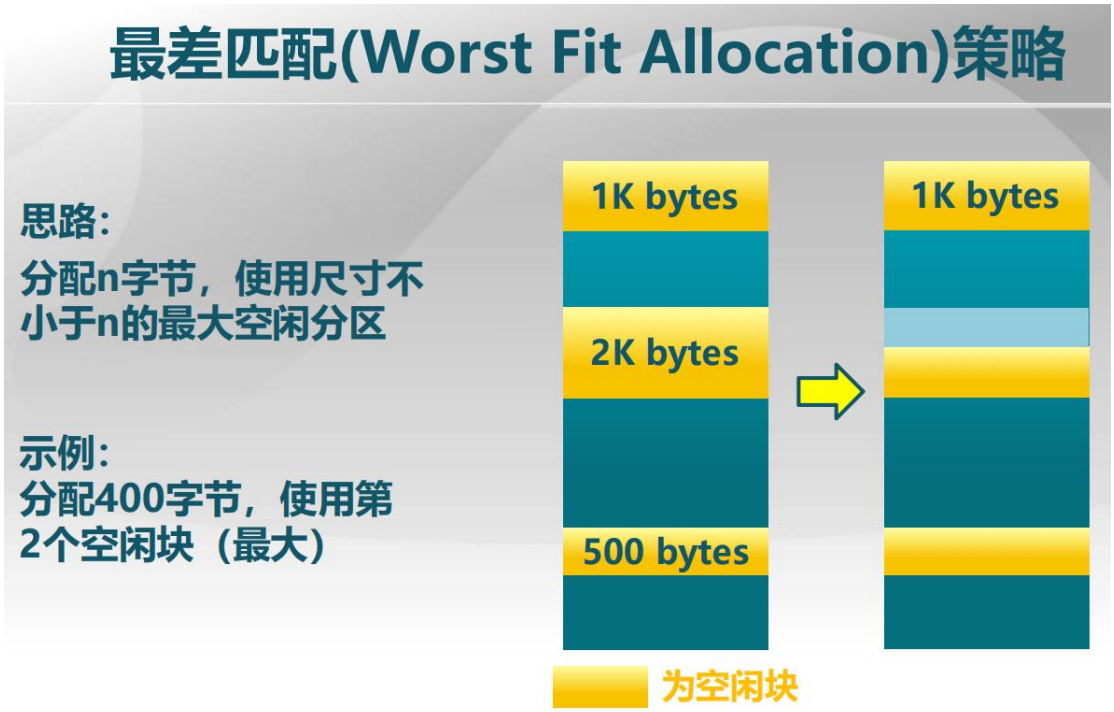
分配400字节， 使用第3个空闲块(最小)

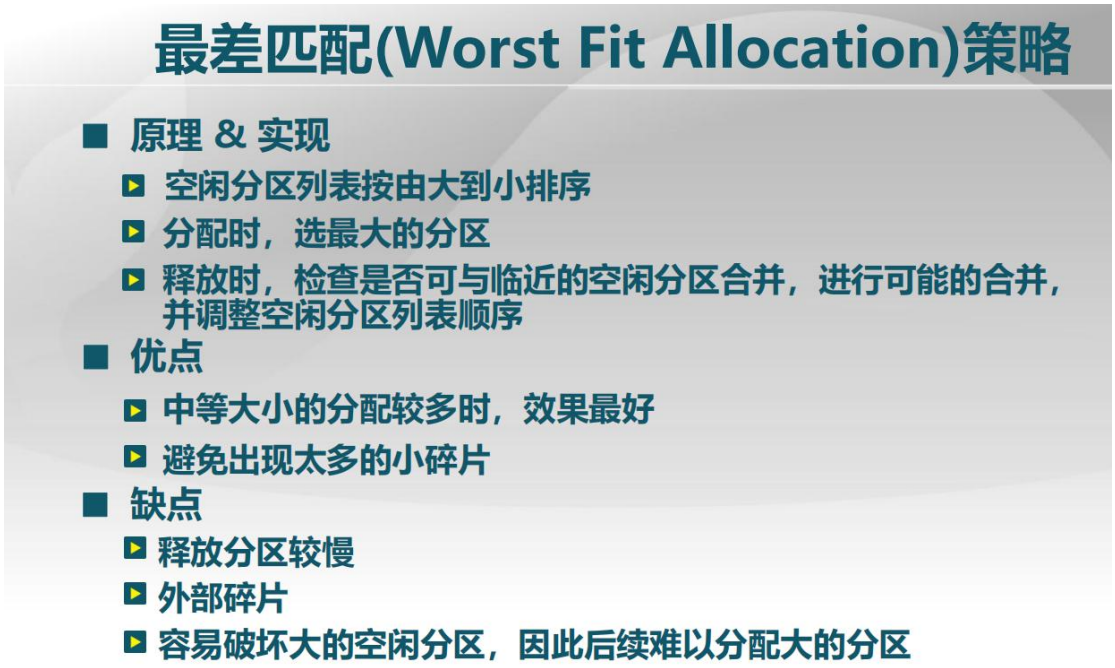


最佳匹配(Best Fit Allocation)策略

- 原理 & 实现
  - 空闲分区列表按照大小排序
  - 分配时，查找一个合适的分区
  - 释放时，查找并且合并临近的空闲分区（如果找到）
- 优点
  - 大部分分配的尺寸较小时，效果很好
    - 可避免大的空闲分区被拆分
    - 可减小外部碎片的大小
    - 相对简单
- 缺点
  - 外部碎片
  - 释放分区较慢
  - 容易产生很多无用的小碎片

上图注意，释放时是按地址临近进行合并的，不是按照空闲分区表的大小进行合并的，因此释放时比较复杂。



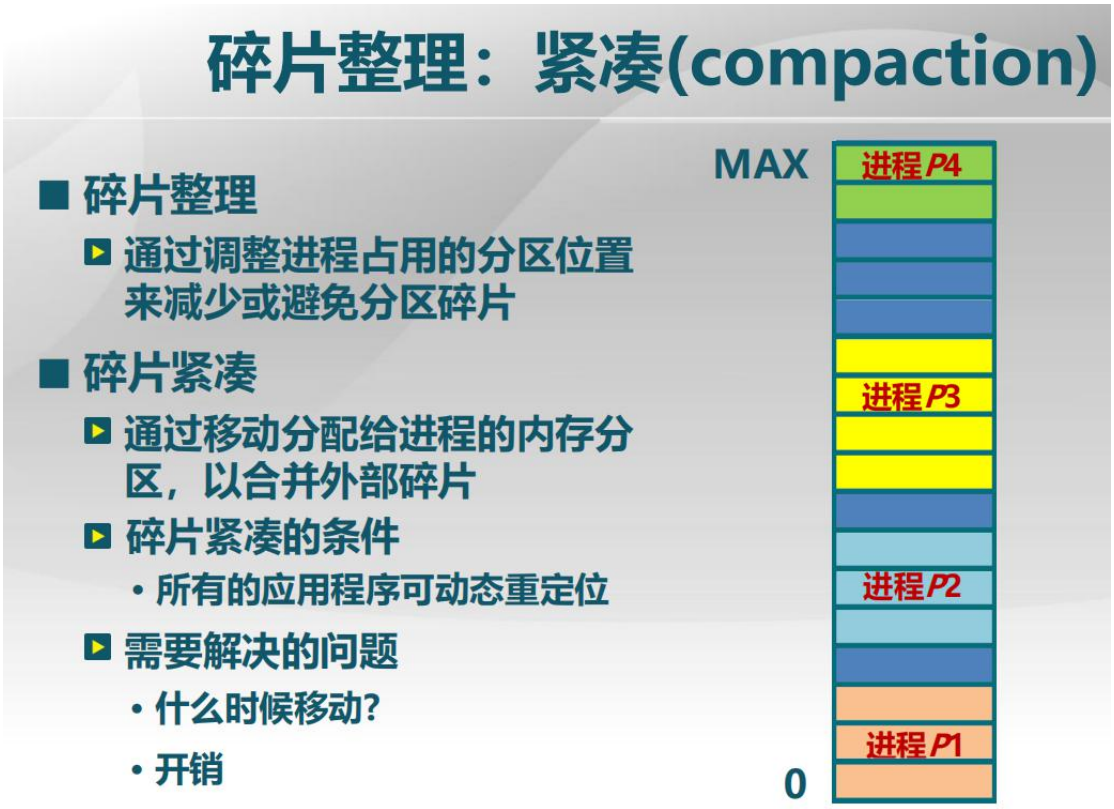


释放时和最佳匹配一样，也是按地址临近进行合并，由于空闲分区列表不是按照地址排的，找临近时就需要按顺序去找了，释放分区会较慢。

第四节 碎片整理

碎片整理是说我们已经把内存分区分配给了进程，这时，正在执行的应用或新创建的进程需要内存空间，这时内存空间没有了或只剩下小的碎片了，这时想要内存空间应该怎么办？我们可以通过碎片整理来获得更大的可用内存空间，以便于满足进程的应用空间需求。

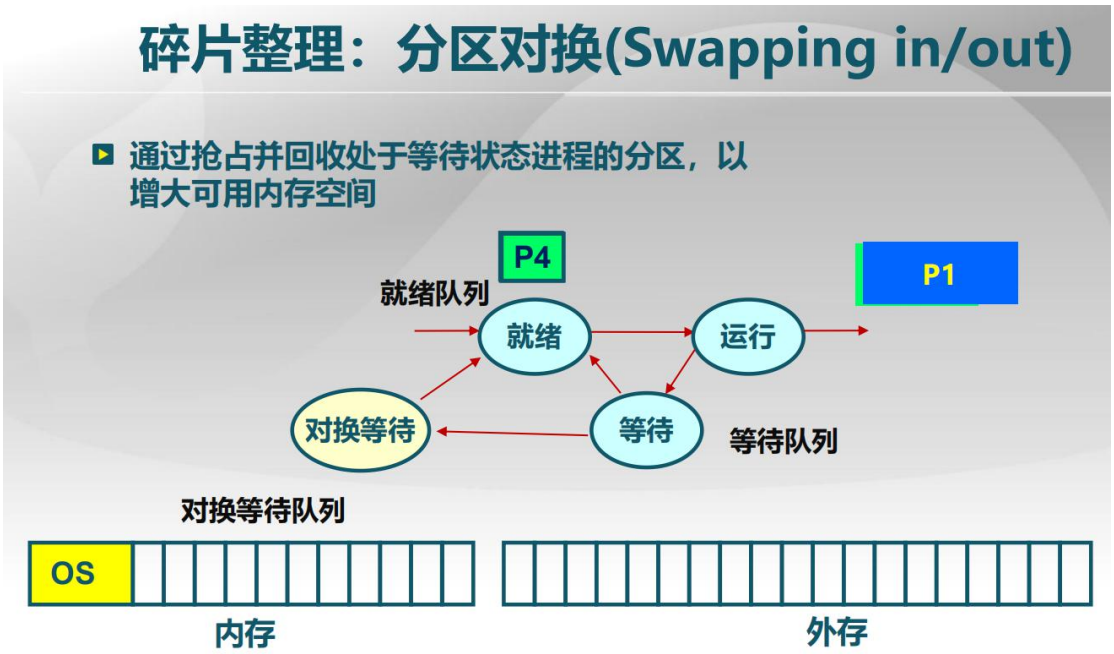
碎片整理方法一：



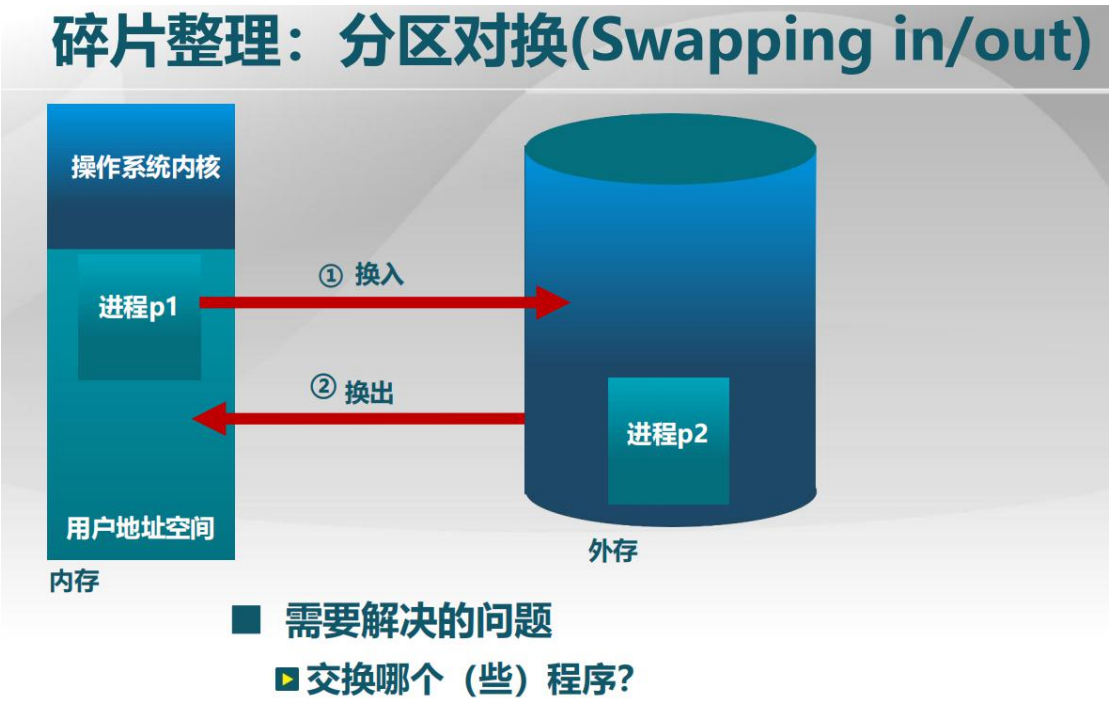
动态重定位：应用程序 A 中可能引用了某个程序 B 所在的地址，动态重定位保证 B 移位合并后，仍能正确定位到原来所需的内容。

什么时候移动？不可以当进程正处于运行状态时去移动，通常在进程处于等待状态时去移动。 考虑到开销，并不需要移动所有进程，而是有一定的策略。

碎片整理方法二：



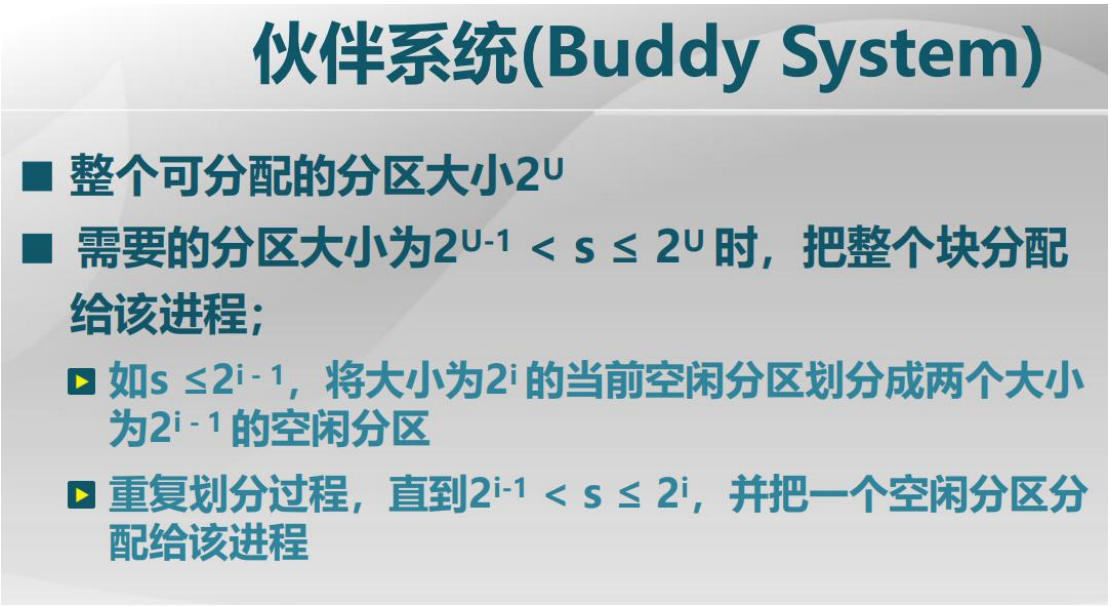
通过抢占并回收处于等待状态进程的分区，将其数据存到外存中去，以增大可用内存空间。(对换详细过程演示见 PPT)　　在 Linux 和 Unix 系统中，有一个分区叫做对换分区。这个对换区在早期的时候，就是一种充分利用内存的方法。



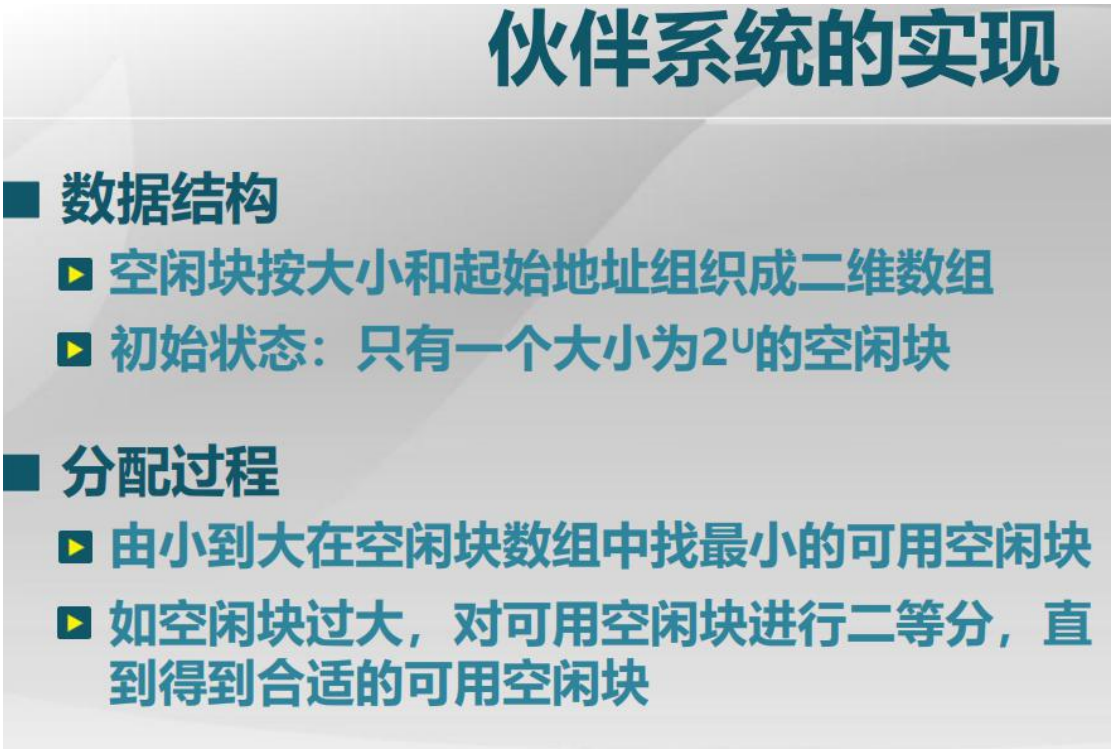
①换入：进程 P1 将换入外存(换出内存)　　②换出：进程 P2 将换出外存(换入内存)

第五节 伙伴系统

伙伴系统(Buddy System)是一个连续内存分配的实例。伙伴系统实际上是连续存储分配的一种办法，它比较好的折中了分配和回收过程当中这种合并和分配块的位置以及碎片的问题。

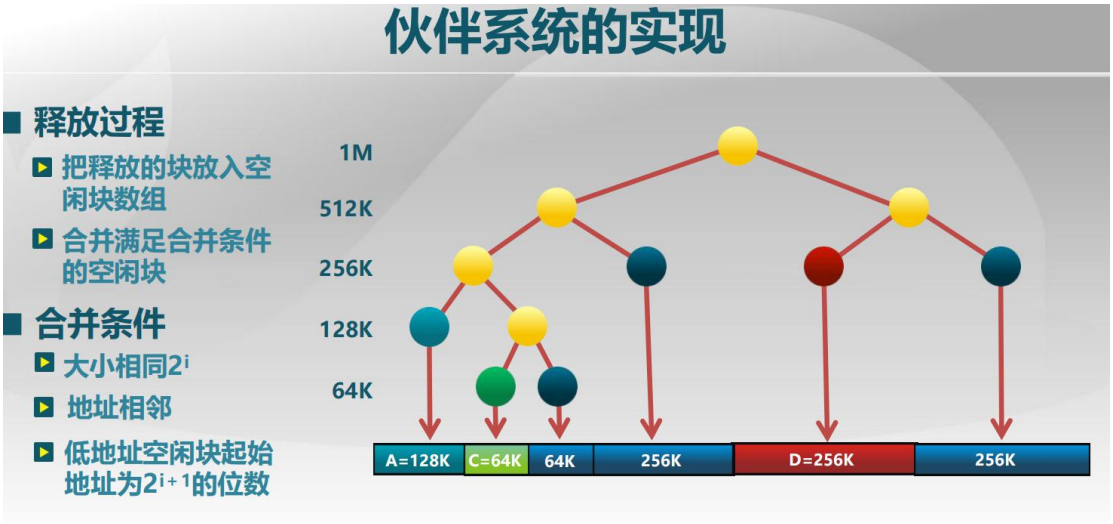


整个可分配的分区大小必须为 2 的幂。即将当前空闲分区不断切半分配或整个分配。



二维数组第一维是空闲块的大小，由小到大拍成第一维，在相同大小的空闲块中，按照地址排序排成第二维。

上图是伙伴系统中内存分配流程的实例.Release B 后，无法和前面的 64K 进行合并，因为合并之后分区大小不是 2 的幂。Relaese C 后，可以和后面的 64K 进行合并，得到 128K，诸如此类。



低地址空闲块起始地址为 2^(i+1)的倍数(PPT 上打错了，改成倍数)，从直观上来说，就是两个合并的空闲块要有相同的父亲。因此红色 256K 的块释放后应该和右边的块合并。

Linux 及 Unix 中都有 Buddy System 的实现，它是用来做内核里的存储分配的。

ucore 中的物理内存管理(ucore 中物理内存管理的标准接口)：

```
struct pmm_manager {
        const char *name; //管理算法的名字
        void (*init)(void); //初始化，即数据结构的初始化
        void (*init_memmap)(struct Page *base, size_t n);
        struct Page *(*alloc_pages)(size_t order); //分配
        void (*free_pages)(struct Page *base, size_t n); //回收
        size_t (*nr_free_pages)(void); //空闲分区中还有多大空间
        void (*check)(void);//检查，测试以上函数是否好使
};
```

在分配和回收分给一个进程之后，在用的时候还得把它映射到进程的地址空间中，因此有 void (*init_memmap)(struct Page *base, size_t n);以及 size_t (*nr_free_pages)(void);函数用于映射到地址空间。

ucore 中的伙伴系统实现：

```
const struct pmm_manager buddy_pmm_manager = {
    .name = "buddy_pmm_manager",
    .init = buddy_init,
    .init_memmap = buddy_init_memmap,
    .alloc_pages = buddy_alloc_pages, //要填的分配函数
    .free_pages = buddy_free_pages,   //要填的回收函数
    .nr_free_pages = buddy_nr_free_pages,
    .check = buddy_check,
};
```

练习题：

1.在启动页机制的情况下，在 CPU 运行的用户进程访问的地址空间是(逻辑地址空间)

2.在使能分页机制的情况下，(不会)出现外碎片，对于(连续内存分配)，外碎片的整理方法有：(紧凑，分区对换)

3.操作系统中可采用的内存管理方式包括：(重定位 relocation、分段 segmentation、分页 paging、段页式 segmentation+paging)

4.连续内存分配的算法中，会产生外碎片的有：(最先匹配算法、最差匹配算法、最佳匹配算法)

5.伙伴系统(buddy system)的特征：(多个小空闲空间可合并为大的空闲空间、会产生外碎片、会产生内碎片)(参见该节举的例子即可知)