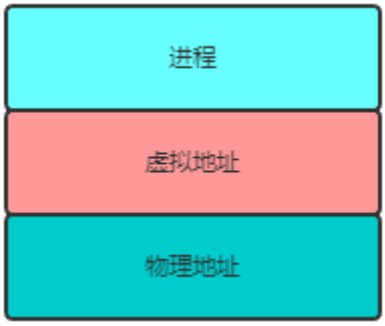
内存管理

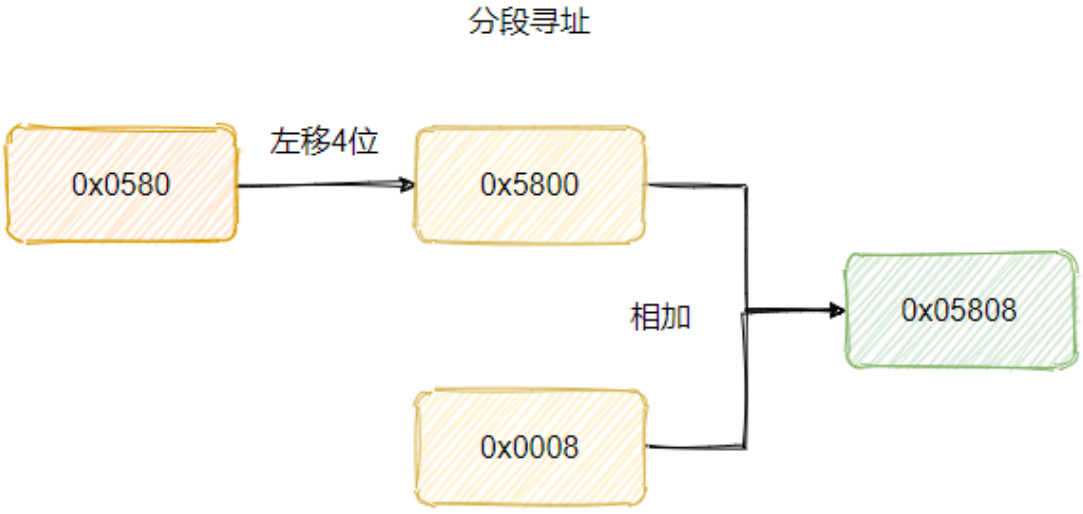
你的电脑是32位操作系统，那可支持的最大内存就是4G，你有没有好奇为什么可以同时运行2个以上的2G内存的程序。应用程序不是直接使用的物理地址，操作系统为每个运行的进程分配了一套**虚拟地址**，每个进程都有自己的**虚拟内存地址**，进程是无法直接进行**物理内存地址**的访问的。至于虚拟地址跟物理地址的映射，进程是感知不到的！**操作系统自身会提供一套机制将不同进程的虚拟地址和不同内存的物理地址进行映射。**



Intel与内存管理发展史

段基地址+段内偏移

1971 年 11 月 15 日，Intel 推出世界第一块个人微型处理器 4004（4位处理器）。随后又推出了 8080（8 位处理器）。那时候访问内存就只有直白自然的想法，用具体物理地址。所有的内存访问就是通过绝对物理地址去访问的，那时候还没有段的概念。
段的概念是起源于 8086，这个 16 位处理器。限于当时的技术背景和经济，寄存器只有 16 位，而地址总线是 20 位。那 16 的位的寄存器如何能访问 20 位的地址？2 的16 次方如果直着来如何能访问到 2 的 20 次方所表达的数？直着来是不可能的，因此就需要操作一下。也就是引入段的概念，让 CPU 通过「**段基地址+段内偏移**」来访问内存。
所以再具体一点的计算规则其实是：段基地址左移 4 位（就是乘16）再加上段内偏移，这样得到的就是 20 位的地址。

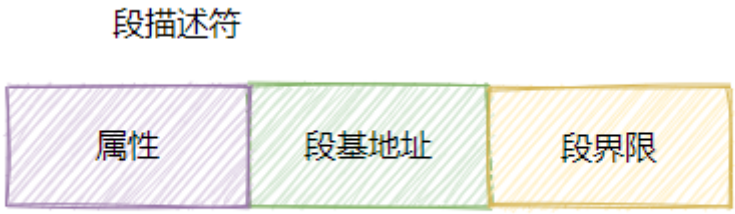


比如现在的要访问的内存地址是0x05808，那么段基地址可以是 0x0580，偏移量就是 0x0008。图片这样内存的寻址空间就扩大到 20 位了。至于为什么称之为段，其实就是因为寄存器只有 16 位一段只能

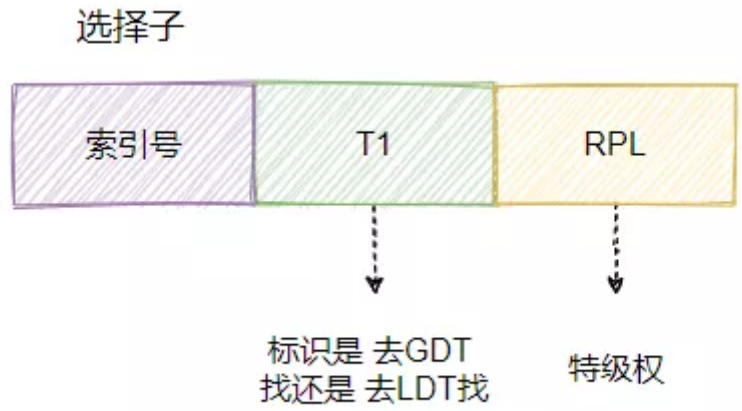
访问 64 KB，所以需要移动基地址，一段一段的去访问所有的内存空间。专门为分段而生的寄存器为段寄存器，当时里面直接存放段基地址。不过渐渐地人们就考虑到安全问题，**因为在这个时候程序之间的地址没有隔离，我的程序可以访问你的程序地址，这就很不安全。**

保护模式、实模式

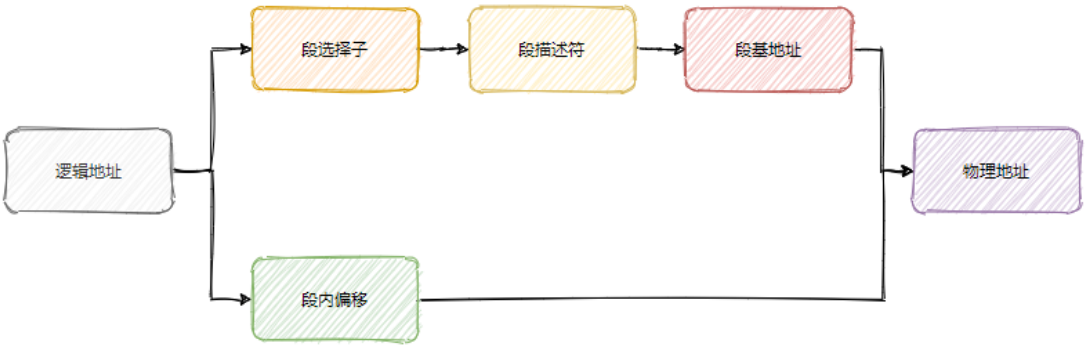
于是在 1982 年 80286 推出时，就有了保护模式。其实就是 CPU 在访问地址的时候做了约束，会判断地址是否在允许的范围内，会判断当前的程序对目的地址是否有访问权限。搞了个 GDT **（全局描述符表）**存放所有段描述符。



段寄存器里面也不是直接放段基地址了，而是放了一个叫选择子的东西。大致可以认为就是段描述符的索引，也就是通过这个索引去找到段描述符，所以叫选择子。这个选择子里面还有一点属性。



这个 T1 就是标明要去哪个表找，而 RPL 就是特权级了，一共分为四层，0 为最高特权级，3 为最低特权级。当地址访问时，如果 RPL 的权限低于目标特权级（DPL）时，就会拒绝访问，于是就起到了保护的作用。所以称之为**保护模式**，之前的那种没有判断权限的称之为**实模式。**



当时 80286 的地址总线已经是 24 位，但是用于寻址的通用寄存器还是 16 位，虽然段基地址的位数已经足够访问到 24 位（因为已经放到 GDT 中，且有 24位）。但是因每次一段只有 64 KB，这样访问就很不方便，需要不断的更换段基地址，于是 80286 很快就被淘汰，换上了 80386。这是 Intel 第一代 32 位处理器。

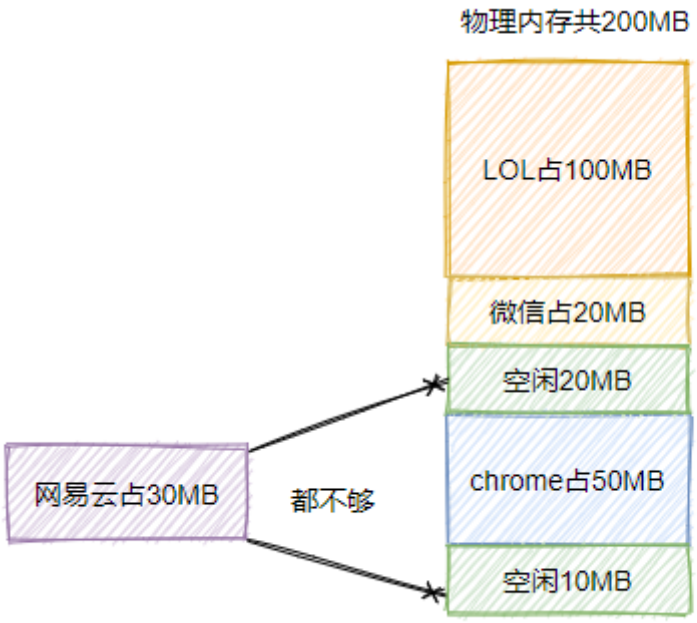
除了段寄存器还是 16 位之外，地址总线和寄存器都是 32 位，这就意味着以前为了寻址搞的段机制其实没用了。因为单单段内偏移就可以访问到 4GB 空间，但是为了向前兼容段机制还是保留了下来，段寄存器还是 16 位是因为够用了，所以没必要扩充。
不过上有政策，下有对策。虽说段机制保留了，但是咱可以“忽悠”着用，把段基值都设置为 0 ，就用段内偏移地址来访问内存空间就好了。这其实就意味着每个段的起始地址都是一样的，那就等于不分段了，这就叫**平坦模式**。Linux 就是这样实现的。

分页

因为分段粒度太粗了，导致内存碎片大，不利于管理。

当时加载到内存等于一个段都得搞到内存中，而段的范围过大，举个例子。

假设此时你有 200M 内存，此时有 3 个应用在运行，分别是 LOL、chrome、微信。



此时内存中明明有 30MB 的空闲，但是网易云加载不进来，这内存碎片就有点大了。

然后就得把 chrome 先换到磁盘中，然后再让 chrome 加载进来到微信的后面，这样空闲的 30MB 就连续了，于是网易云就能加载到内存中了。

但是这样等于要把 50MB 的内存来个反复横跳，磁盘的访问太慢了，所以效率就很低。

总体而言可以认为分段内存的管理粒度太粗了，所以随着 80386 就出来了个分页管理，一个更加精细化的内存管理方式。

简单地说就是把内存等分成一页一页，每页 4KB 大小，按页为单位来管理内存。

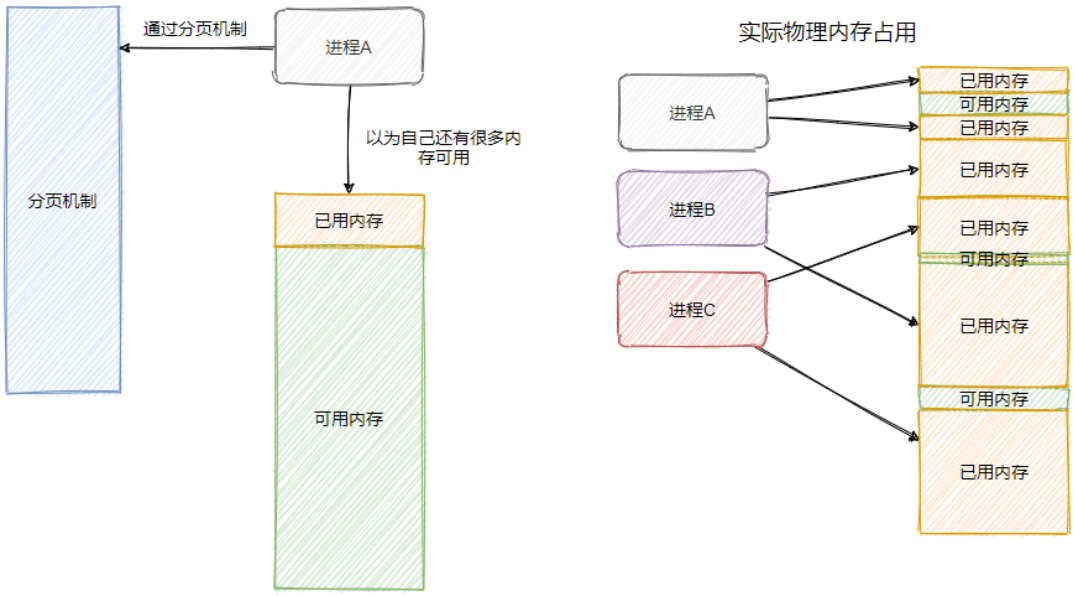
你看按一页一页来管理这样就不用把一段程序都加载进内存，只需要将用到的页加载进内存。

这样内存的利用率就更高了，能同时运行的程序就更多了。

并且由于一页就 4KB， 所以内存交换的性能问题得以缓解，毕竟只要换一定的页，而不需要整个段都换到磁盘中。

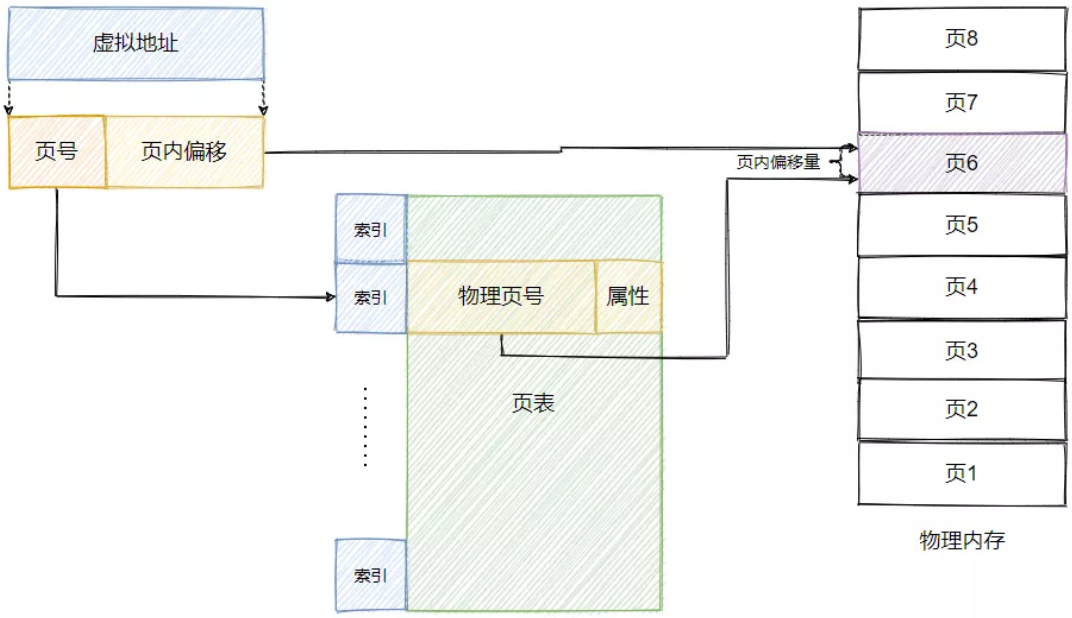
对应的还有个虚拟内存的概念。

分页机制构造了一个虚拟内存空间，让每个进程误以为自己掌控所有的内存。



再具体一点就是每个进程都有一个页表，页表中有物理页号和属性，这样寻址的时候通过页表就能利用虚拟地址找到对应的物理地址。

属性用来做权限的一些管理。



就理解为进程想要内存中的任意一个地址都行，没问题，反正背地里偷偷的会换成可以用的物理内存地址。

如果物理内存满了也没事，把不常用的内存页先换到磁盘中，即 swap，腾出空间来就好了，到时候要用再换到内存中。

上面提到的虚拟地址也叫线性地址，简单地说就是通过绕不开的段机制得到线性地址，然后再通过分页机制转化得到物理地址。

总结

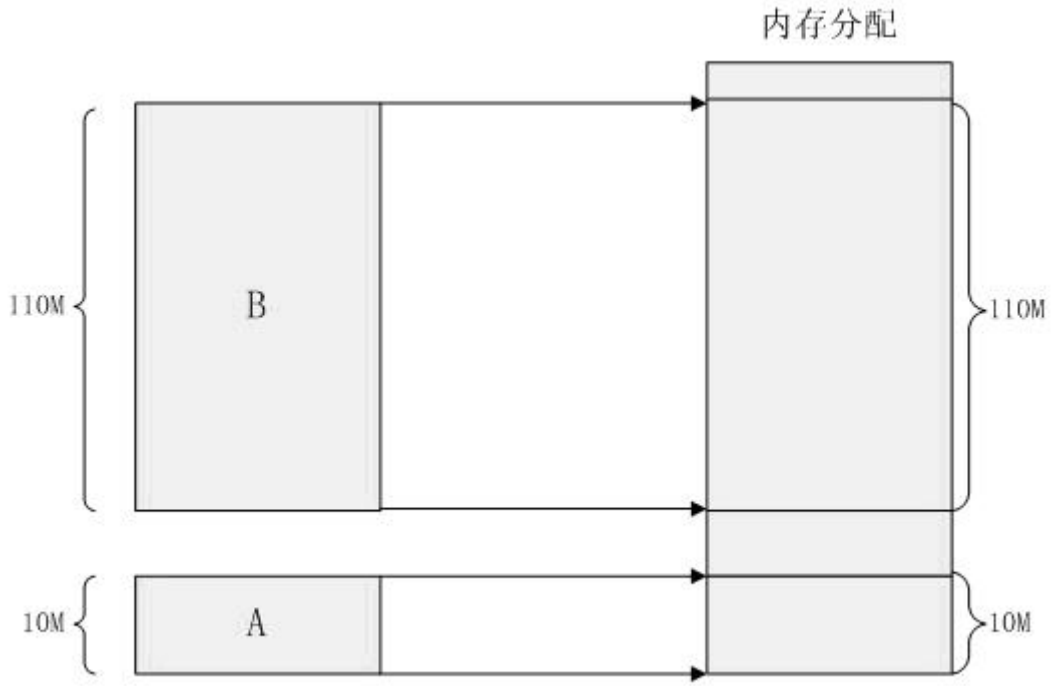
一开始限于技术和成本所以寄存器的位数不够，因此为了扩大寻址范围搞了个分段访问内存。而随后技术起来了，位数都扩充了，寄存器其实已经可以访问全部内存空间了，所以分段已经没用了。但是为了向前兼容还是保留着分段访问的形式，并且随着软件的发展，同时运行各种进程的需求越发强烈。为了更好的管理内存，提高内存的利用率和内存交互性能引入了分页管理。所以就变成了先分段，然后再分页的段页式。当然也可以和 Linux 那样让每一段的基地址都设为 0 ，这样就等于“绕开”了段机制。

无存储器抽象

无存储器抽象的概念

在 早期的计算机中，要运行一个程序，会把这些程序全都装入内存，程序都是直接运行在内存上的，也就是说程序中访问的内存地址都是实际的物理内存地址。当计算 机同时运行多个程序时，必须保证这些程序用到的内存总量要小于计算机实际物理内存的大小。那当程序同时运行多个程序时，操作系统是如何为这些程序分配内存 的呢？下面通过实例来说明当时的内存分配方法：

某台计算机总的内存大小是 128M ，现在同时运行两个程序 A 和 B ， A 需占用内存 10M ， B 需占用内存 110 。计算机在给程序分配内存时会采取这样的方法：先将内存中的前 10M 分配给程序 A ，接着再从内存中剩余的 118M 中划分出 110M分配给程序 B 。这种分配方法可以保证程序 A 和程序 B 都能运行，但是这种简单的内存分配策略问题很多。



无存储器抽象的问题

问题 1 ： 进程地址空间不隔离。由于程序都是直接访问物理内存，所以恶意程序可以随意修改别的进程的内存数据，以达到破坏的目的。有些非恶意的，但是有bug 的程序也可能不小心修改了其它程序的内存数据，就会导致其它程序的运行出现异常。这种情况对用户来说是无法容忍的，因为用户希望使用计算机的时候，其中一个任务失败了，至少不能影响其它的任务。

问题 2 ： 内存使用效率低。在 A 和 B 都运行的情况下，如果用户又运行了程序C ，而程序 C 需要 20M 大小的内存才能运行，而此时系统只剩下 8M 的空间可供使用，所以此时系统必须在已运行的程序中选择一个将该程序的数据暂时拷贝到硬盘上，释放出部分空间来供程序 C 使用，然后再将程序 C 的数据全部装入内存中运行。可以想象得到，在这个过程中，有大量的数据在装入装出，导致效率十分低下。

问题 3 ：程序运行的地址不确定。当内存中的剩余空间可以满足程序 C 的要求后，操作系统会在剩余空间中随机分配一段连续的 20M 大小的空间给程序 C 使用，因为是随机分配的，所以程序运行的地址是不确定的。

基址寄存器和变址寄存器

比较难的是给每个程序一个自己独有的地址空间，使得一个程序中的地址28所对应的物理地址与另一个程序中的地址28所对应的物理地址不同。下面我们将讨论一个简单的方撞，这个方法曾经很常见，但是在有能力把更复杂（而且更好）的机制运用在现代CPU芯片上之后，这个方法就不再使用了。

概念

这个简单的解决办法是使用**动态重定位(dynamic relocation)**，简单地把每个进程的地址空间映射到物理内存的不同部分。从CDC 6600 （世界上最早的超级计算机）到Intel 8088 （原始IBM PC的心脏），所使用的经典办法是给每个CPU配置两个特殊硬件寄存器，通常叫作基址寄存器和界限寄存器。当使用基址寄存器和界限寄存器时，程序装载到内存中连续的空闲位置且装载期间无须重定位，

- **基址寄存器**：存储数据内存的起始位置
- **变址寄存器**：存储应用程序的长度。

运行过程

每当进程引用内存以获取指令或读取、写入数据时，CPU 都会自动将**基址值**添加到进程生成的地址中，然后再将其发送到内存总线上。同时，它检查程序提供的地址是否大于或等于**变址寄存器** 中的值。如果程序提供的地址要超过变址寄存器的范围，那么会产生错误并中止访问。

缺点

使用基址寄存器和界限寄存器重定位的缺点是，每次访问内存都需要进行加陆和比较运算。比较运算可以做得很快，但是加法运算由于进位传递时间的问题，在没有使用特殊电路的情况下会显得很慢。