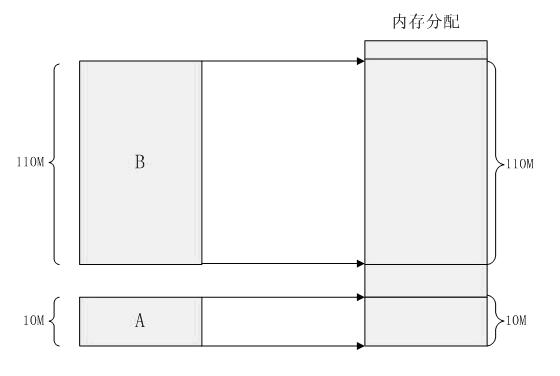
# 浅谈进程地址空间与虚拟存储空间

**早期的内存分配机制**

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。



早期的内存分配方法

**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。

**分段**

**为 了解决上述问题，人们想到了一种变通的方法，就是增加一个中间层，利用一种间接的地址访问方法访问物理内存。按照这种方法，程序中访问的内存地址不再是实际的物理内存地址，而是一个虚拟地址，然后由操作系统将这个虚拟地址映射到适当的物理内存地址上**。这样，只要操作系统处理好虚拟地址到物理内存地址的映射，就可以保证不同的程序最终访问的内存地址位于不同的区域，彼此没有重叠，就可以达到内存地址空间隔离的效果。

当创建一个进程时，操作系统会为该进程分配一个 4GB 大小的虚拟进程地址空间。之所以是 4GB ，是因为在 32 位的操作系统中，一个指针长度是 4 字节，而 4 字节指针的寻址能力是从 0x00000000~0xFFFFFFFF，最大值 0xFFFFFFFF 表示的即为 4GB 大小的容量。与虚拟地址空间相对的，还有一个物理地址空间，这个地址空间对应的是真实的物理内存。如果你的计算机上安装了 512M 大小的内存，那么这个物理地址空间表示的范围是 0x00000000~0x1FFFFFFF 。当操作系统做虚拟地址到物理地址映射时，只能映射到这一范围，操作系统也只会映射到这一范围。当进程创建时，每个进程都会有一个自己的 4GB 虚拟地址空间。要注意的是这个 4GB 的地址空间是“虚拟”的，并不是真实存在的，而且每个进程只能访问自己虚拟地址空间中的数据，无法访问别的进程中的数据，通过这种方法实现了进程间的地址隔离。那是不是这 4GB 的虚拟地址空间应用程序可以随意使用呢？很遗憾，在 Windows 系统下，这个虚拟地址空间被分成了 4 部分： NULL 指针区、用户区、 64KB 禁入区、内核区。

1）NULL指针区 （0x00000000~0x0000FFFF）: 如果进程中的一个线程试图操作这个分区中的数据，CPU就会引发非法访问。他的作用是，调用 malloc 等内存分配函数时，如果无法找到足够的内存空间，它将返回 NULL。而不进行安全性检查。它只是假设地址分配成功，并开始访问内存地址 0x00000000（NULL）。由于禁止访问内存的这个分区，因此会发生非法访问现象，并终止这个进程的运行。

2）用户模式分区 ( 0x00010000~0xBFFEFFFF)：这个分区中存放进程的私有地址空间。一个进程无法以任何方式访问另外一个进程驻留在这个分区中的数据 （相同 exe，通过 copy-on-write 来完成地址隔离）。（在windows中，所有 .exe 和动态链接库都载入到这一区域。系统同时会把该进程可以访问的所有内存映射文件映射到这一分区）。

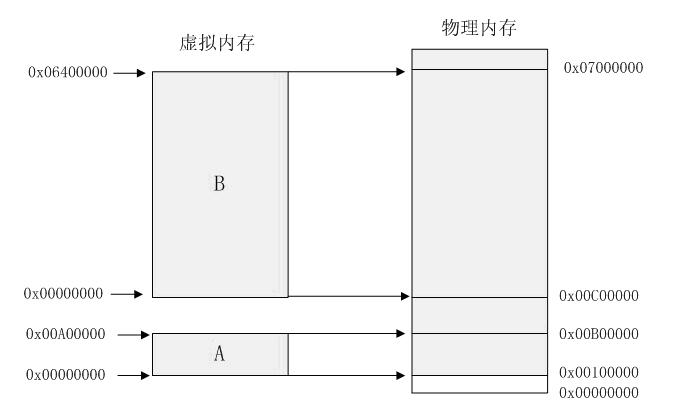
2）隔离区 (0xBFFF0000~0xBFFFFFFF)：这个分区禁止进入。任何试图访问这个内存分区的操作都是违规的。微软保留这块分区的目的是为了简化操作系统的现实。

3）内核区 (0xC0000000~0xFFFFFFFF)：这个分区存放操作系统驻留的代码。线程调度、内存管理、文件系统支持、网络支持和所有设备驱动程序代码都在这个分区加载。这个分区被所有进程共享。

应用程序能使用的只是用户区而已，大约 2GB 左右 ( 最大可以调整到 3GB) 。内核区为 2GB ，内核区保存的是系统线程调度、内存管理、设备驱动等数据，这部分数据供所有的进程共享，但应用程序是不能直接访问的。

**人们之所以要创建一个虚拟地址空间，目的是为了解决进程地址空间隔离的问题**。但程序要想执行，必须运行在真实的内存上，所以，必须在虚拟地址与物理地址间建立一种映射关系。这样，通过映射机制，当程序访问虚拟地址空间上的某个地址值时，就相当于访问了物理地址空间中的另一个值。人们想到了一种分段(Sagmentation) 的方法，它的思想是在虚拟地址空间和物理地址空间之间做一一映射。比如说虚拟地址空间中某个 10M 大小的空间映射到物理地址空间中某个 10M 大小的空间。这种思想理解起来并不难，操作系统保证不同进程的地址空间被映射到物理地址空间中不同的区域上，这样每个进程最终访问到的。

物理地址空间都是彼此分开的。通过这种方式，就实现了进程间的地址隔离。还是以实例说明，假设有两个进程 A 和 B ，进程 A 所需内存大小为 10M ，其虚拟地址空间分布在 0x00000000 到 0x00A00000 ，进程 B 所需内存为 100M ，其虚拟地址空间分布为 0x00000000 到 0x06400000 。那么按照分段的映射方法，进程 A 在物理内存上映射区域为 0x00100000 到 0x00B00000 ，，进程 B 在物理内存上映射区域为0x00C00000 到 0x07000000 。于是进程 A 和进程 B 分别被映射到了不同的内存区间，彼此互不重叠，实现了地址隔离。从应用程序的角度看来，进程 A 的地址空间就是分布在 0x00000000 到 0x00A00000 ，在做开发时，开发人员只需访问这段区间上的地址即可。应用程序并不关心进程 A 究竟被映射到物理内存的那块区域上了，所以程序的运行地址也就是相当于说是确定的了。 下图显示的是分段方式的内存映射方法：



分段方式的内存映射方法

这种分段的映射方法虽然解决了上述中的问题一和问题三，但并没能解决问题二，即内存的使用效率问题。在分段的映射方法中，每次换入换出内存的都是整个程序， 这样会造成大量的磁盘访问操作，导致效率低下。所以这种映射方法还是稍显粗糙，粒度比较大。实际上，程序的运行有局部性特点，在某个时间段内，程序只是访问程序的一小部分数据，也就是说，程序的大部分数据在一个时间段内都不会被用到。基于这种情况，人们想到了粒度更小的内存分割和映射方法，这种方法就是分页 (Paging) 。

**分页**

分页的基本方法是，将地址空间分成许多的页。每页的大小由 CPU 决定，然后由操作系统选择页的大小。目前 Inter 系列的 CPU 支持 4KB 或 4MB 的页大小，而 PC上目前都选择使用 4KB 。按这种选择， 4GB 虚拟地址空间共可以分成 1048576 页， 512M 的物理内存可以分为 131072 个页。显然虚拟空间的页数要比物理空间的页数多得多。

在分段的方法中，每次程序运行时总是把程序全部装入内存，而分页的方法则有所不同。分页的思想是程序运行时用到哪页就为哪页分配内存，没用到的页暂时保留在硬盘上。当用到这些页时再在物理地址空间中为这些页分配内存，然后建立虚拟地址空间中的页和刚分配的物理内存页间的映射。

下面通过介绍一个可执行文件的装载过程来说明分页机制的实现方法。一个可执行文件 (PE 文件 ) 其实就是一些编译链接好的数据和指令的集合，它也会被分成很多页，在 PE 文件执行的过程中，它往内存中装载的单位就是页。当一个 PE 文件被执行时，操作系统会先为该程序创建一个 4GB 的进程虚拟地址空间。前面介绍过，虚拟地址空间只是一个中间层而已，它的功能是利用一种映射机制将虚拟地址空间映射到物理地址空间，所以，创建 4GB 虚拟地址空间其实并不是要真的创建空间，只是要创建那种映射机制所需要的数据结构而已，这种数据结构就是页目和页表。

当创建完虚拟地址空间所需要的数据结构后，进程开始读取 PE 文件的第一页。在PE 文件的第一页包含了 PE 文件头和段表等信息，进程根据文件头和段表等信息，将 PE 文件中所有的段一一映射到虚拟地址空间中相应的页 (PE 文件中的段的长度都是页长的整数倍 ) 。这时 PE 文件的真正指令和数据还没有被装入内存中，操作系统只是据 PE 文件的头部等信息建立了 PE 文件和进程虚拟地址空间中页的映射关系而已。当 CPU 要访问程序中用到的某个虚拟地址时，当 CPU 发现该地址并没有相相关联的物理地址时， CPU 认为该虚拟地址所在的页面是个空页面， CPU 会认为这是个页错误 (Page Fault) ， CPU 也就知道了操作系统还未给该 PE 页面分配内存，CPU 会将控制权交还给操作系统。操作系统于是为该 PE 页面在物理空间中分配一个页面，然后再将这个物理页面与虚拟空间中的虚拟页面映射起来，然后将控制权再还给进程，进程从刚才发生页错误的位置重新开始执行。由于此时已为 PE 文件的那个页面分配了内存，所以就不会发生页错误了。随着程序的执行，页错误会不断地产生，操作系统也会为进程分配相应的物理页面来满足进程执行的需求。

分页方法的核心思想就是当可执行文件执行到第 x 页时，就为第 x 页分配一个内存页 y ，然后再将这个内存页添加到进程虚拟地址空间的映射表中 , 这个映射表就相当于一个 y=f(x) 函数。应用程序通过这个映射表就可以访问到 x 页关联的 y 页了。

**逻辑地址、线性地址、物理地址和虚拟地址的区别**

**逻辑地址（Logical Address）** 是指由程式产生的和段相关的偏移地址部分。例如，你在进行 C 语言指针编程中，能读取指针变量本身值( &操作 )，实际上这个值就是逻辑地址，他是相对于你当前进程数据段的地址，不和绝对物理地址相干。只有在 Intel 实模式下，逻辑地址才和物理地址相等（因为实模式没有分段或分页机制，cpu不进行自动地址转换）；逻辑也就是在Intel保护模式下程式执行代码段限长内的偏移地址（假定代码段、数据段如果完全相同）。应用程式员仅需和逻辑地址打交道，而分段和分页机制对你来说是完全透明的，仅由系统编程人员涉及。应用程式员虽然自己能直接操作内存，那也只能在操作系统给你分配的内存段操作。

**线性地址（Linear Address）**是逻辑地址到物理地址变换之间的中间层。程式代码会产生逻辑地址，或说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址能再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386 的线性地址空间容量为 4G（2的32次方即32根地址总线寻址）。

**物理地址（Physical Address）**是指出目前 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

**虚拟内存（Virtual Memory）**是指计算机呈现出要比实际拥有的内存大得多的内存量。因此他允许程式员编制并运行比实际系统拥有的内存大得多的程式。这使得许多大型项目也能够在具有有限内存资源的系统上实现。一个非常恰当的比喻是：你不必非常长的轨道就能让一列火车从上海开到北京。你只需要足够长的铁轨（比如说3公里）就能完成这个任务。采取的方法是把后面的铁轨即时铺到火车的前面，只要你的操作足够快并能满足需求，列车就能象在一条完整的轨道上运行。这也就是虚拟内存管理需要完成的任务。在 Linux0.11 内核中，给每个程式（进程）都划分了总容量为 64MB 的虚拟内存空间。因此程式的逻辑地址范围是 0x0000000 到 0x4000000。**有时我们也把逻辑地址称为 虚拟地址**。因为和虚拟内存空间的概念类似，逻辑地址也是和实际物理内存容量无关的。逻辑地址和物理地址的“差距”是 0xC0000000，是由于虚拟地址->线性地址->物理地址映射正好差这个值。这个值是由操作系统指定的。机理逻辑地址（或称为虚拟地址）到线性地址是由CPU的段机制自动转换的。如果没有开启分页管理，则线性地址就是物理地址。如果开启了分页管理，那么系统程式需要参和线性地址到物理地址的转换过程。具体是通过设置页目录表和页表项进行的。