Virtual Memory

In [computing](https://en.wikipedia.org/wiki/Computing" \o "Computing), **virtual memory** (also **virtual storage**) is a [memory management](https://en.wikipedia.org/wiki/Memory_management_(operating_systems)" \o "Memory management (operating systems)) technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine"[[1]](https://en.wikipedia.org/wiki/Virtual_memory" \l "cite_note-2) which "creates the illusion to users of a very large (main) memory".[[2]](https://en.wikipedia.org/wiki/Virtual_memory" \l "cite_note-3)

The computer's [operating system](https://en.wikipedia.org/wiki/Operating_system" \o "Operating system), using a combination of hardware and software, maps [memory addresses](https://en.wikipedia.org/wiki/Memory_address" \o "Memory address) used by a program, called *[virtual addresses](https://en.wikipedia.org/wiki/Virtual_address_space" \o "Virtual address space)*, into *physical addresses* in [computer memory](https://en.wikipedia.org/wiki/Computer_memory" \o "Computer memory). [Main storage](https://en.wikipedia.org/wiki/Main_storage" \l "Primary_storage" \o "Main storage), as seen by a process or task, appears as a contiguous [address space](https://en.wikipedia.org/wiki/Address_space" \o "Address space) or collection of contiguous [segments](https://en.wikipedia.org/wiki/Memory_segmentation" \o "Memory segmentation). The operating system manages [virtual address spaces](https://en.wikipedia.org/wiki/Virtual_address_space" \o "Virtual address space) and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a [memory management unit](https://en.wikipedia.org/wiki/Memory_management_unit" \o "Memory management unit) (MMU), automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of [paging](https://en.wikipedia.org/wiki/Paging" \o "Paging).

Virtual Memory - Example

作者：匿名用户  
链接：https://www.zhihu.com/question/50796850/answer/522734117  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

要理解分段和分页，那么得理解为什么会出现分段和分页的技术

首先，这两个技术都是为了利用和管理好计算机的资源--内存。

在分段这个技术还没有出现之前，程序运行是需要从内存中分配出足够多的连续的内存，然后把整个程序装载进去。举个例子，某个程序大小是10M，然后，就需要有连续的10M内存空间才能把这个程序装载到内存里面。如果无法找到连续的10M内存，就无法把这个程序装载进内存里面，程序也就无法得到运行。

上面这种直接把整个程序装载进内存的方式是有一定的问题的。例如：

1、地址空间不隔离

如何理解地址空间不隔离？

举个例子，假设我有两个程序，一个是程序A，一个是程序B。程序A在内存中的地址假设是0x00000000~0x00000099，程序B在内存中的地址假设是0x00000100~x00000199。那么假设你在程序A中，本来想操作地址0x00000050，不小心手残操作了地址0x00000150，那么，不好的事情或许会发生。你影响了程序A也就罢了，你把程序B也搞了一顿。

2、程序运行时候的地址不确定

如何理解程序运行时候的地址不确定？

因为我们程序每次要运行的时候，都是需要装载到内存中的，假设你在程序中写死了要操作某个地址的内存，例如你要地址0x00000010。但是问题来了，你能够保证你操作的地址0x00000010真的就是你原来想操作的那个位置吗？很可能程序第一次装载进内存的位置是0x00000000~0x00000099，而程序第二次运行的时候，这个程序装载进内存的位置变成了0x00000200~0x00000299，而你操作的0x00000010地址压根就不是属于这个程序所占有的内存。

3、内存使用率低下

如何理解内存使用率低下呢？

举个例子，假设你写了3个程序，其中程序A大小为10M，程序B为70M，程序C的大小为30M你的计算机的内存总共有100M。

这三个程序加起来有110M，显然这三个程序是无法同时存在于内存中的。

并且最多只能够同时运行两个程序。可能是这样的，程序A占有的内存空间是0x00000000～0x00000009，程序B占有的内存空间是0x00000010～0x00000079。假设这个时候程序C要运行该怎么做？可以把其中的一个程序换出到磁盘上，然后再把程序C装载到内存中。假设是把程序A换出，那么程序C还是无法装载进内存中，因为内存中空闲的连续区域有两块，一块是原来程序A占有的那10M，还有就是从0x00000080～0x00000099这20M，所以，30M的程序C无法装载进内存中。那么，唯一的办法就是把程序B换出，保留程序A，但是，此时会有60M的内存无法利用起来，很浪费对吧。

然后，人们就去寻求一种办法来解决这些问题。

有一句话说的好：计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决。

（这种思想在现在也用的很广泛，例如很多优秀的中间层：Nginx、Redis等等）

所以，分段这种技术就出现了。

为了实现分段的这个技术，需要引入虚拟地址空间的概念。那么什么是地址空间呢？简单的说就是可以寻址的一片空间。如果这个空间是虚拟的，我们就叫做虚拟地址空间；如果这个空间是真实存在的，我们就叫做物理地址空间。虚拟地址空间是可以任意的大的，因为是虚拟的。而物理地址空间是真实存在的，所以是有限的。

然后，分段这个技术做了一件什么事情呢？

它把虚拟地址空间映射到了物理地址空间，并且你写的程序操作的是虚拟地址。假设，程序A的虚拟地址空间是0x00000100～0x00000200。此时，不仅需要一块连续的物理内存来存放程序A，还需要把程序A的虚拟地址空间映射到（转换为）物理地址空间。可能，程序A的虚拟地址空间从0x00000100～0x00000200映射到了物理地址空间0x00000000～0x00000100。

那么分段的技术可以解决什么问题呢？可以解决上面1、2两个问题。

在问题1中，假设程序A的虚拟地址空间是0x00000000~0x00000099，映射到的物理地址空间是0x00000600~0x00000699，程序B的虚拟地址空间是0x00000100~0x00000199，映射到的物理地址空间是0x00000300~0x00000399。假设你还是手残，在程序A中操作了地址0x00000150，但是英文此时的地址0x00000150是虚拟的，而虚拟化的操作是在操作系统的掌控中的，所以，操作系统有能力判断，这个虚拟地址0x00000150是有问题的，然后阻止后续的操作。所以，体现出了隔离性。（另一种体现隔离性的方式就是，操作同一个虚拟地址，实际上可能操作的是不同的物理地址）

（注意，实际上，很可能程序A和程序B的虚拟地址都是0x00000000~0x00000099。这里的举例只是为了方便理解。）

问题2也很好的解决了。正是因为这种映射，使得程序无需关注物理地址是多少，只要虚拟地址没有改变，那么，程序就不会操作地址不当。

但是问题3仍然没有解决。

因为第三个问题是换入换出的问题，这个问题的关键是能不能在换出一个完整的程序之后，把另一个完整的程序换进来。而这种分段机制，映射的是一片连续的物理内存，所以问题3得不到解决。

而问题出在哪呢？就是完整和连续。

而分页技术的出现就是为了解决这个问题的。分页这个技术仍然是一种虚拟地址空间到物理地址空间映射的机制。但是，粒度更加的小了。单位不是整个程序，而是某个“页”，一段虚拟地址空间组成的某一页映射到一段物理地址空间组成的某一页。（如何理解这个“页”的概念，这个问题下的其他同学回答过）

分页这个技术，它的虚拟地址空间仍然是连续的，但是，每一页映射后的物理地址就不一定是连续的了。正是因为有了分页的概念，程序的换入换出就可以以页为单位了。那么，为什么就可以只换出某一页呢？实际上，不是为什么可以换出某一页，而是可以换出CPU还用不到的那些程序代码、数据。但是，把这些都换出到磁盘，万一下次CPU就要使用这些代码和数据怎么办？又得把这些代码、数据装载进内存。性能有影响对吧。所以，我们把换入换出的单位变小，变成了“页”。（实际上，这利用了空间局部性）

所以，同学们想想，问题3是不是就解决了呢？

所以，分段和分页的区别在于：粒度。