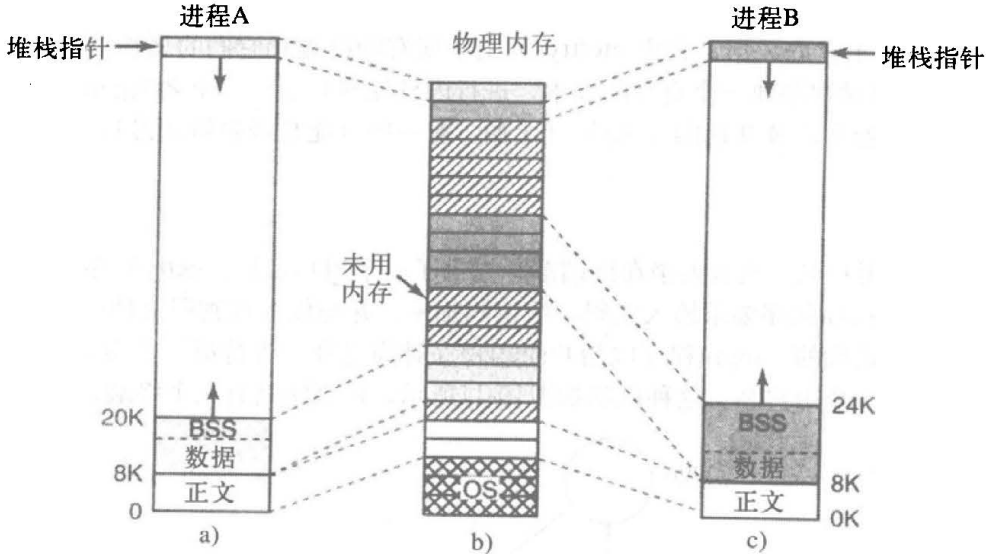

图10-12 a) 进程A的虚拟地址空间；b) 物理内存；c) 进程B的虚拟地址空间

*数据段*包含了所有程序变量、字符串、数字和其他数据的存储。它有两部分，初始化数据和未初始化数据。由于历史的原因，后者就是我们所知道的BSS（*历史上称作符号起始块*）。数据段的初始化部分包括编译器常量和那些在程序启动时就需要一个初始值的变量。所有BSS部分中的变量在加载后被初始化为0。

例如，在C语言中可以在声明一个字符串的同时初始化它。当程序启动的时候，字符串要拥有其初始值。为了实现这种构造，编译器在地址空间给字符串分配一个位置，同时保证在程序启动的时候该位置包含了合适的字符串。从操作系统的角度来看，初始化数据跟程序代码并没有什么不同——二者都包含了由编译器产出的位串，它们必须在程序启动的时候加载到内存。

未初始化数据的存在实际上仅仅是个优化。如果一个全局变量未显式地初始化，那么C语言的语义说明它的初始值是0。实际上，大部分全局变量并没有显式初始化，因此都是0。这些可以简单地通过设置可执行文件的一个段来实现，其大小刚好等于数据所需的字节数，同时初始化包括缺省值为零的所有量。

然而，为了节省可执行文件的空间，并没有这样做。取而代之的是，文件包含所有显式初始化的变量，跟随在程序代码之后。那些未初始化的变量都被收集在初始化数据之后，因此编译器要做的就是在文件头部放入一个字段说明要分配的字节数。

为了清楚地说明这一点，再考虑图10-12a。这里代码段的大小是8KB，初始化数据段的大小也是8KB。未初始化数据（BSS）是4KB。可执行文件仅有16KB（代码＋初始化数据），加上一个很短的头部来告诉系统在初始化数据后另外再分配4KB，同时在程序启动之前把它们初始化为0。这个技巧避免了在可执行文件中存储4KB的0。

为了避免分配一个全是0的物理页框，在初始化的时候，Linux就分配了一个静态零页面，即一个全0的写保护页面。当加载程序的时候，未初始化数据区域被设置为指向该零页面。当一个进程真正要写这个区域的时候，写时复制的机制就开始起作用，一个实际的页框被分配给该进程。

跟代码段不一样，数据段可以改变。程序总是修改它的变量。而且，许多程序需要在执行时动态分配空间。Linux允许数据段随着内存的分配和回收而增长和缩减，通过这种机制来解决动态分配的问题。有一个系统调用brk，允许程序设置其数据段的大小。那么，为了分配更多的内存，一个程序可以增加数据段的大小。C库函数malloc通常被用来分配内存，它就大量使用这个系统调用。进程地址空间描述符包含信息：进程动态分配的内存区域（通常叫做*堆*，heap）的范围。

第三段是*栈段*。在大多数机器里，它从虚拟地址空间的顶部或者附近开始，并且向下生长。例如，在32位x86平台上，栈的起始地址是0xC0000000，这是在用户态下对进程可见的3GB虚拟地址限制。如果栈生长到了栈段的底部以下，就会产出一个硬件错误同时操作系统把栈段的底部降低一个页面。程序并不显式地控制栈段的大小。