# 这一块得看书，看视频很混乱，不写了

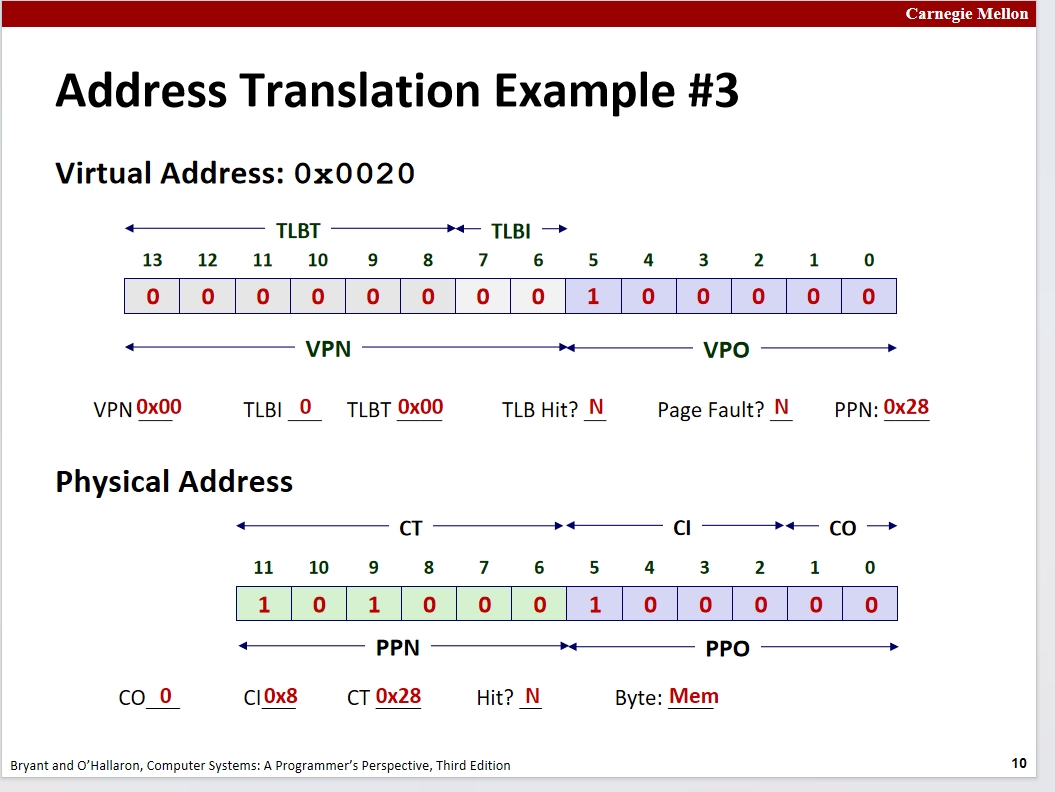


图 1

我们直接来分析虚拟地址往物理地址转换的工作原理。

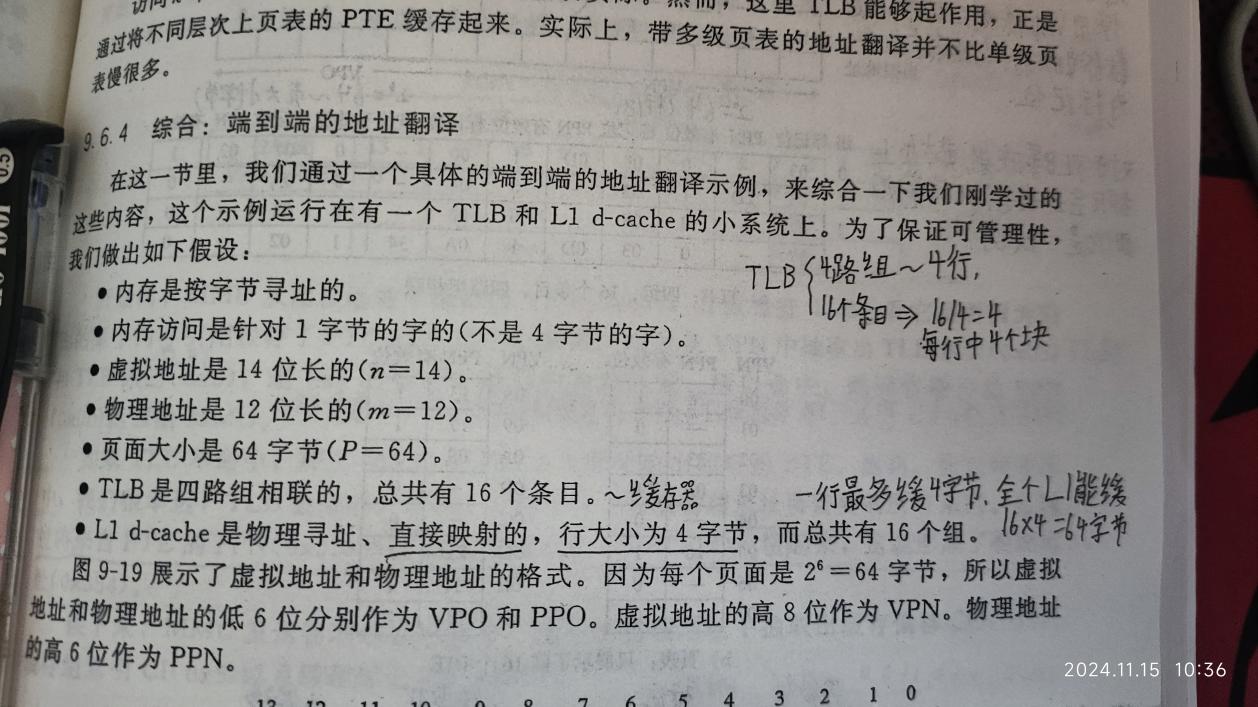
首先，我们使用上图中的内存系统设定

图 2

首先，先来分析下虚拟地址的情况。在上图1中的12为虚拟地址中。我们将把这几个划分为一系列的部分。

1~对于虚拟地址的VPO，这个与我们系统虚拟内存页的大小相关联。就比如，这里一个页的大小是64个字节，故我们这里的VPO需要6位来进行标识。

2~对于虚拟地址的VPN，这个是虚拟地址所要映射的区域(系统将通过这个和其他来进行一系列的物理内存页的映射)。这个的大小取决于整个虚拟地址的位长减去我们用来标识页偏移的VPO的位长。

接下来要进行虚拟地址往物理地址上的转换。

首先，我们需要对虚拟地址的VPN进行切割。请注意，在整个往物理地址的映射过程中，VPO只在最后一步用到。

对于VPN的切割需要考虑几个因素。我们看我们想要将VPN切割后得出的数据。这里我们会把这个VPN切割为一块映射TLB的数据。在这里对VPN的切割就需要考虑到TLB的属性。

在这里，TLB是一个四路组，总共有16个条目。也就是说这个TLB总共有16行，每一行中有着4个小块。我们的VPN需要切割出这里的行数索引和块数索引来取出对应块的数据。

看这里，一行有4个块，所以这里只需要俩个位就可以映射到每一行中的对应块了。接着，再看我们剩下的6个位，这里的6个位将会用于标志位。也就是说，这里的VPN就需要分割出需要的TLB的组索引即可，剩下的位都会用于标志

接下来，我们会先通过我们分割出来的VPN的组索引去查找我们TLB中对应的组，然后匹配标志位并查看对应的有效位是否为1。如果上面的几个条件都满足，就意味着这个TLB缓存了对应的虚拟地址对应的物理地址的PPN。这里就可以进入下一步。

但是，如果上面的几个条件中有一个未满足，就代表着在TLB中并没有对应的缓存，此时我们就需要直接去主存中的页表中进行查找。页表中的查找不需要细分VPN位，直接使用整个VPN作为索引去找到对应的PPN。但是如果此时有效位还为0，那么如前面所说还会触发异常，这里不进行重复了。

当无论是在TLB或者页表中查找到对应的虚拟地址的对应值时。系统会将这个值返回给我们的MMU。对于这个返回的值，我们需要知道的就只是这个值是物理地址的PPN。MMU会将这个PPN和虚拟地址的VPO进行合并，进而生成一个物理地址。这个物理地址就是虚拟地址的映射地址。如前面所说，VPO在当前整个过程中的使用情况就只有这最后的一次合并。

接下来进入到物理地址的解析。首先，我们需要知道的是，物理地址要做到的是去通知取到对应内存的数据。而在计算机架构中，构建完这个物理地址就只是CPU发送地址到MMU中。还没有实现任何取存操作。接下来MMU会发送这个地址到最近的储存器，这里的最近的储存器就是L1的高速储存。

我们这里就只分析这个物理地址送到了L1高速缓存的情况，更下层的情况与之类似，不进行赘述。

当物理地址送到L1缓存中时，缓存会对这个地址进行解析。首先我们需要明确我们的切割位置，对这个地址的切割只局限于我们的PPN。我们如果需要通过这个物理地址映射到缓存中的具体位置。就需要这个物理地址所对应的缓存组和缓存块。

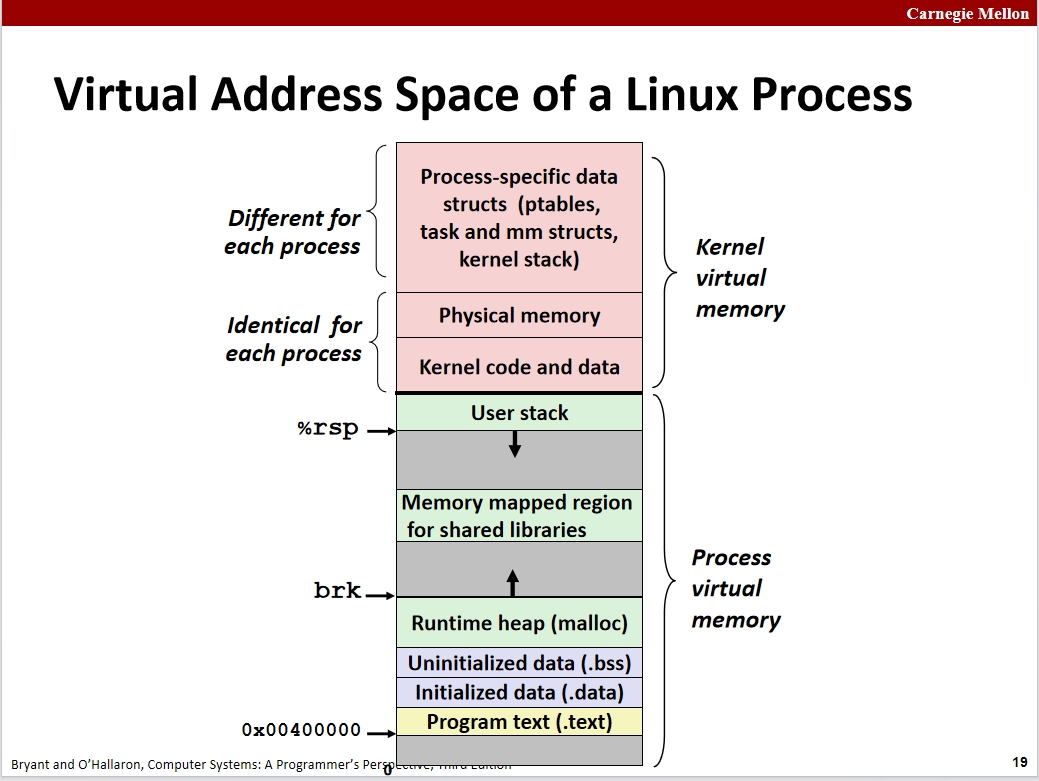
这里就需要考虑缓存的架构了。在这次的缓存中，总共有16组，需要4个位来进行组索引的映射。每个行有4个字节，而内存是按字节寻址，所以这里每个行中就有4个块，就需要俩个位进行块的映射。这里与前面的虚拟地址的映射不同了。不能通过标志位来进行索引了，毕竟这里每个缓存行只有一个标志位。

故这里我们对PPO的划分如下：划分出4个位来进行组的索引，划分出2个位来进行块的索引，这里总共需要6个位，而PPO就刚好就有6位，所以这里就刚刚好。这里的划分会是从右往左划分俩位作为块索引，剩下的4为作为组索引。对于整个物理地址，除开PPO的其他位都将用于接下来的标志位识别

接下来系统通过这个信息去查找对应缓存中的数据，系统会先查看有效位是否为1，为0就是缓存未命中的情况喽，需要继续往下去搜索并缓存。同时也会查看对应的标志位是否匹配。不匹配也是一个缓存未命中的情况。

当系统检测到缓存标志位与物理地址的前面几位标志位匹配且缓存的有效位为1时。就会根据接下来的块偏移为来进行组内块的查找，接着缓存就会将这个块中的数据返回给MMU了，此时就完成了对虚拟地址的使用。

接下来看一个系统优化缓存访问的技巧，如前面所说，虚拟地址的VPO与物理地址的PPO是一模一样的。当系统收到一个虚拟地址时，系统会先取出对应的VPO并将其送到缓存中去查找。当查找到后这个操作会进入一种类似阻塞的状态，知道，物理地址的标志位被构造出来。此时系统会将这个标志位与缓存等的标志位进行比较。也就是说，这种查找与物理地址的构造是同时的。



接下来分析下虚拟内存系统在一个Linux进程上的实例，一个架构图如上(见过很多遍了不是吗)

由于虚拟内存系统的原因，每个进程的代码段(.text)基本都是从0x0040000开始的。在这之上是可执行文件的初始化数据(.date段)，再上是未初始化数据(.bss)，再接下来是内存堆区，这个区在未初始化数据上进行向上生长。由一个进程的全局变量brk指针指向堆顶。

在堆区之上，是一块用来存放共享库的内存映射区域。接着在一个进程，或者说用户能够管理的内存区域顶部，是一个向下生长的栈区。这个区有一个寄存器储存了它的栈顶位置(%rsp)。

内核代码和数据在这个进程的内存架构的最上层。事实上，在这段内核代码与用户能够访问的栈区域之后，会有一段用于填充的内存区域。当然，这里的填充是指虚拟内存上的填充，并不意味着这段代码是会被使用的或者实际分配内存的。

在Intel的虚拟内存架构中，虚拟地址是48位的。对于这种地址，如果其的最高为为0，那么接下来除开后面的32位，这前面的16位都将被置为0，如果最高位为1，那么这些位都将被置为1。

也就是说这种处理会使得内核所在的虚拟地址空间的最高16全部为1。所以通常我们可以认为内核代码位于虚拟地址空间的高位，前16位总是1，而用户的内存区域的前16位总是0.

你可以预想到，这一段空白内存是相当大的。事实上，这个内存区域有将近100+T。但系统显然是不可能真的分配这么多给进程的。这里就需要回到之前的虚拟地址在未被使用时并不会被分配物理内存，这只是一种软件上的隔离手段。这一段内存有相当多的用处，就比如如果当前已有的内存空间已经满了并且还需要一段内存的时候，整个用户的进程空间就会向上生长，逐步的使用到这一大块的内存。

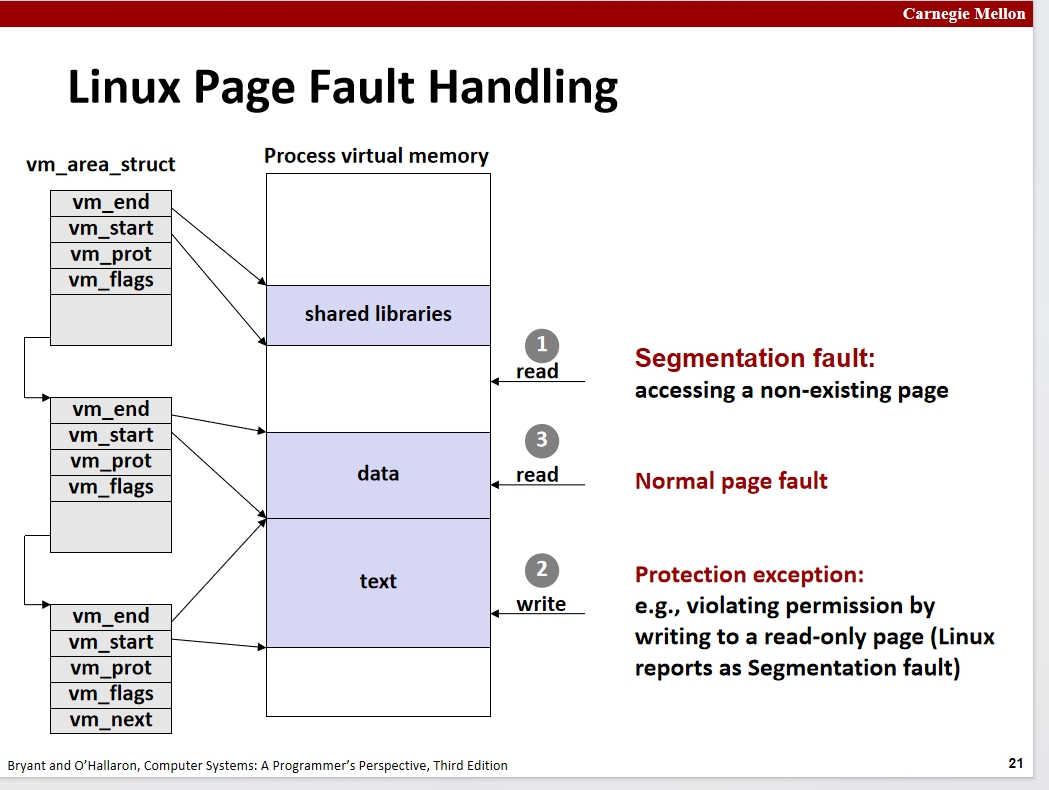
对于内核，我们对其进行一些分析。内核本质上也是一些代码块和对应的数据。其在系统中也会映射去一些虚拟内存页，对应的又会映射出一些物理内存页。对于这些内存页，其的大小可能是很大的。虽然说一般不会像视频里说的跟DRAM一样大那么夸张，不过相对于常规的内存页，它算的上大页。对于这些内存页，系统还是使用了懒加载的特性，在需要时才加载对应的内核代码分区。总的来说，这种架构为内核提供了一个便利的方法来访问物理内存。

对于内核访存的情况，我们可以看到在一个进程的内核代码的那块中的下层中，包含着一些Physical memory和Kernel code and data，这俩块是相对于所有的进程的啮合代码都Identical，即完全相同的。也就是说，对于所有的进程，其的内核代码所用的都会是一个物理内存区和代码区。

除此之外，我们还可以看到在内核的内存区域中，还存在着一段内存区域储存着对应于这个进程的数据结构，包括当前整个系统的进程表，任务结构和内存管理结构，内核栈等。这些进程相关的数据结构被称为**进程的上下文**。对于每个进程来说，上下文是不同的。

接下来我们分析下内核的**缺页异常处理**

对于这种处理，先得区分出对应的异常到底是什么产生的，首先对于这种异常的基本触发条件，就是当MMU翻译一个虚拟地址时，发现这个地址对应的页表根本就不在这个进程的内存中，这时就会触发缺页异常。



1~当我们访问内存时，发现地址是不合法的。我们想要访问一个未被创建的页。此时会触发一个段错误。至于内核对这种的识别，是基于内核代码中的进程上下文中的vm\_area\_struct结构体。这个结构体储存了这个进程中各个块的地址，事实上是通过一系列的指针来标识的，实际上，这每一个数组中还包含了对应进程块的属性。当内核遍历这个结构体链表后仍然不能找到对应的地址所在区间时，只能说明这个地址并不在这次的进程中。

2~当程序想要对一个只读页面进行写入时，此时也会触发一个错误。在地址翻译时，这个地址确实是合法的，但是MMU在这个翻译的同时一般也会检查对应地址的权限位，如果发现了尝试对一个只读页进行写操作，那么就会抛出一个异常。但是事实上，对于处于MMU的翻译阶段时，它是访问不到页表条目的，毕竟权限位我们前面说了是位于条目项中的。但是此时内核就又发挥作用了，内核可以检查这个区域的保护位，如果是代码段这种只读的区域，那么就会触发一个保护错误。在linux中，这个错误被识别为一个段错误。

3~还有一种情况就是我们准备从一个有效区域中去读取数据。此时内核就会陷入异常，然后去读取对应地址上的数据，并将读取的数据返回给CPU。

额外一说，对于上面的VM链表在现实中的具体实现，实际上会是一个某种树，就比如红黑树等。

接下来我们进入一个对之前常提到的**写时复制**的解析

我们之前已经知道，当我们fork一个进程时，系统不会立即为其复制所有的父进程的数据。只有当发生修改时才会复制对应的内存页。但是在这之前，系统还是会复制一些数据的，就比如我们上面提到的每个进程都应该有的内核数据结构块。这一块是会在外面进行fork操作时就进行复制的。这个是无法避免的，但也是不应该避免的，而且他们大小也不大。

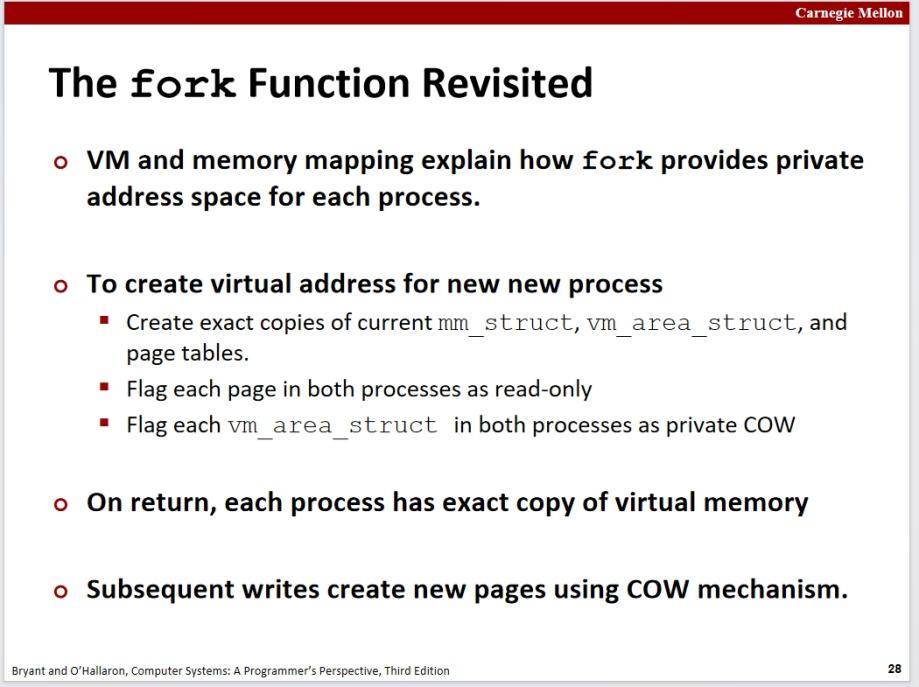
然后内核会将父进程所使用的页都标记为只读页。然后把每一个vm\_area\_struct都标记为私有写时复制，然后此时fork返回，此时父子进程都有一个相同的地址空间。毕竟他们拥有一个相同的页表和其他的进程上下文。

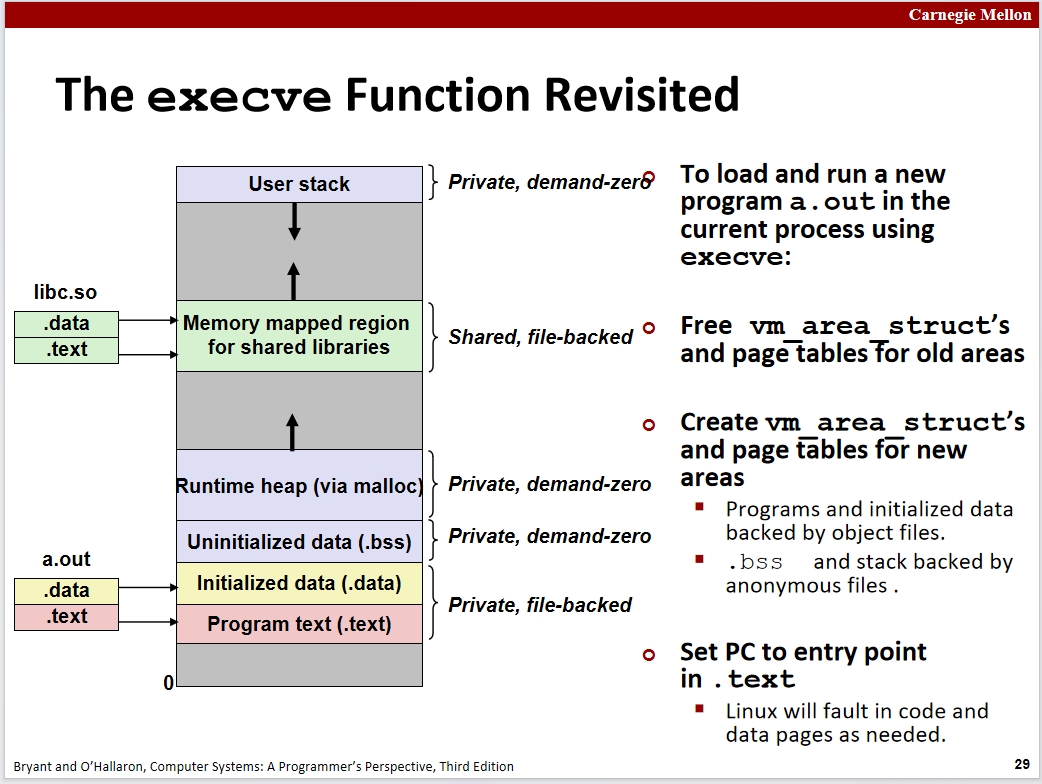
接下来，当任何一个进程想要写的时候，内核会查看对应地址的属性，如果属性是一个私有写时复制，那么内核将会对这个页进行拷贝，并将这个它映射到物理内存上的一个新区域。

这种操作允许在再次进行这个写操作时，也就是异常处理返回的时候，重新处理当前指令。此时操作的对象就是拷贝的页面了。

所以说，这种技巧是一种优秀的技巧，其尽可能的拖延了对虚拟内存页的赋值，只有在需要进行数据修改时才会进行拷贝。

基于这种特性，对于虚拟内存来说，标记为只读的区域永远不会被拷贝。这也就意味着我们的父子进程将会共享一套执行代码，共享一套常量数据，这种特性大大节省了空间和时间。





接下来就开始我们execve系统调用的分析了

对于execve函数，其在当前进程中加载并执行另外一个可执行文件。当调用这个函数时，系统会删除当前进程的所有的area\_struct结构和页表等。也就是说，execve并不创建进程，它只是在当前进程中，在已有但是新的虚拟地址空间上运行一个程序。

在删除原进程的页表等上下文数据后，程序初始化数据。这些区域被映射位一个文件，也就是我们的可执行文件。程序的代码段被映射为程序的.text段，数据段被映射为文件的.date段，这俩者都是私有的，没有被任何进程所共享。未初始化的数据的定义位于.bss段中。这些也是私有的。.bss中的数据，系统会全部初始化为0.这也就是为什么我们未初始化的全局数据和静态数据会被全部初始化为0的位置。接下来内存系统会将这次的进程内存中的共享内存空间映射到对应的共享库中。所有的进程都共享这个共享内存中的libc库。也就是说，这个共享区域是对应着一系列的内存页，而我们的链接就是为了确定到底我们需要映射哪些具体的页。这一段将会被标记为进程间共享的属性。

总的来说，execve的系统调用总的来说并不会加载任何数据，其只是利用已有的进程的虚拟空间，对进程的上下文信息进行重置，并进行一些虚拟地址映射的调整，将原本指向我们可执行文件的各个区域地址指向我们新的可执行文件的地址。同时会将各个区域的指针进行一波初始化。

接下来当程序继续执行时，系统会将指向.text文本的指针值赋给%rip,然后这个值又会被送给PC。此时程序想要去取到对应地址的数据，但是注意到当前进程中保存的内存页其实还是原先进程的内存页，此时会触发一个缺页错误，内核会去处理相应的错误后重新执行该条语句。