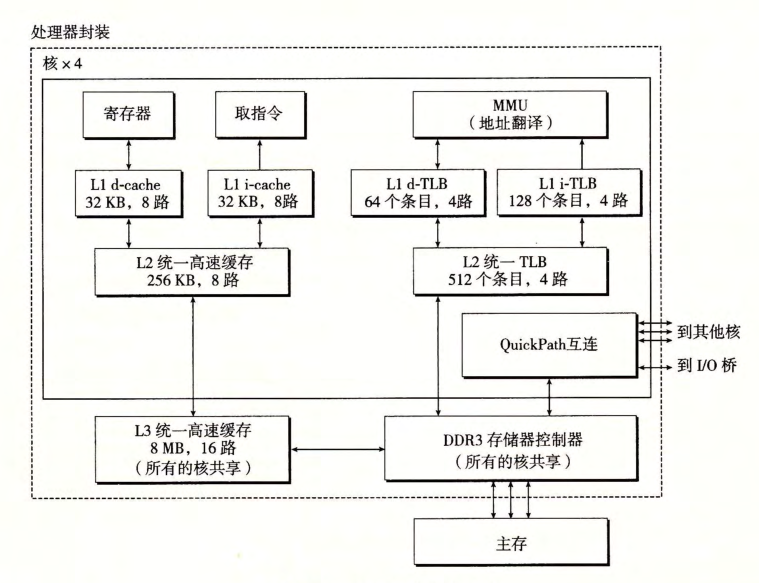
案例研究：Intel Core i7 /Linux 内存系统

虽然底层的 Haswell 微体系结构允许完全的 64 位虚拟和物理地址空间，而

现在的（以及可预见的未来的） Core i7 实现支持 **48 位 (256TB) 虚拟地址空间**和 **52 位 (4PB)物理地址空间**，还有一个兼容模式，支持 32 位 (4GB) 虚拟和物理地址空间。



处理器封装 (processor package) 包括**四个核**、一个大的**所有核共享的L3 高速缓存**，以及一个 **DDR3 内存控制器**。

每个核包含一个**层次结构的 TLB** 、一个**层次结构的数据&指令高速缓存**，以及一组快速的点到点链路，

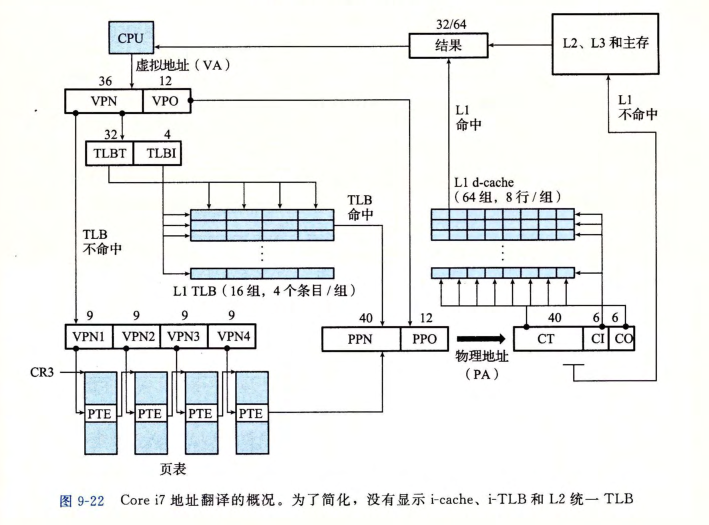
L1 、L2 和 L3 高速缓存是物理寻址的，块大小为 64 字节。

页大小可以在启动时被配置为4KB 或 4MB 。**Linux 使用的是 4KB 的页。**

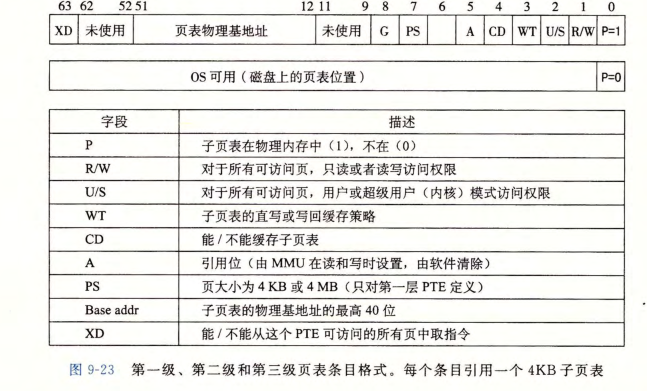
下图总结了完整的 Corei7 地址翻译过程。

**Corei7 采用四级页表层次结构。**每个进程有它自己私有的页表层次结构。当一个 Linux 进程在运行时，虽然 Corei7 体系结构允许页表换进换出，但是与已分配了的页相关联的页表都是驻留在内存中的。

**CR3 控制寄存器指向第一级页表(L1) 的起始位置。**CR3 的值是**每个进程上下文的一部分**，每次上下文切换时，CR3 的值都会被恢复。



*这里采用4级页表结构，页大小为4KB=2^12Byte，因此需要12位的VPO，而虚拟地址为48位，所以VPN为36位，每一级的VPN为9位。*



上图给出了第一级、第二级或第三级页表中条目的格式。（第四级页表的条目多了一个Dirty位）

当 P=1时，地址字段包含一个 **40 位**物理页号 (PPN)，它指向适当的页表的开始处。这强加了一个要求，**要求物理页表 4KB 对齐。**

*（每一级的VPN为9bit，因此子页表有2^9个PTE。而每个PTE为8字节，*

*子页表大小为 2^9\*8Byte=4KB，与页的大小一样。*

*52位物理地址空间可以划分为2^40个4KB的页，因此物理页号是40位的。）*

PTE 有三个权限位，控制对页的访问。

R /W 位确定页的内容是可以读写的还是只读的。

U /S 位确定是否能够在用户模式中访问该页，从而保护操作系统内核中的代码和数据不被用户程序访问。

XD( 禁止执行）位是在 64 位系统中引入的，可以用来禁止从某些内存页取指令。通过限制只能执行只读代码段，使得操作系统内核降低了缓冲区溢出攻击的风险。

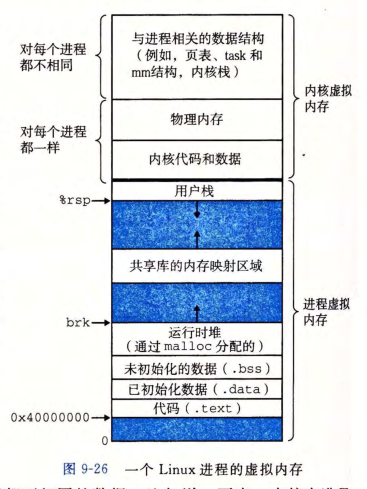
当 MMU 翻译每一个虚拟地址时，它还会更新另外两个内核缺页处理程序会用到的位。

**每次访问一个页时，MMU 都会设置 A 位，**称为引用位 (reference bit) 。内核可以用这个引用位来实现它的**页替换算法**。

**每次对一个页进行了写之后，MMU 都会设置 D 位**，又称修改位或脏位 (dirty bit）。修改位告诉内核在复制替换页之前是否必须写回牺牲页。

这里可以发现一个特点：高速缓存的CI+CO=12位，而VPO也是12位。这不是巧合，而是故意这样设计来加速地址翻译。我们知道，VPO=PPO，而PPN需要通过地址翻译获得，则一开始输入虚拟地址时，就能一下等到PPO，然后等待检索PPN。此时我们就能直接将PPO输入到高速缓存中，因为PPO确定了对应的高速缓存组和块偏移量，就能**先通过PPO获得对应的高速缓存组**，然后只要等检索到PPN时，就能直接和高速缓存组中每一行的标志位进行比较，极大加速了地址翻译过程。

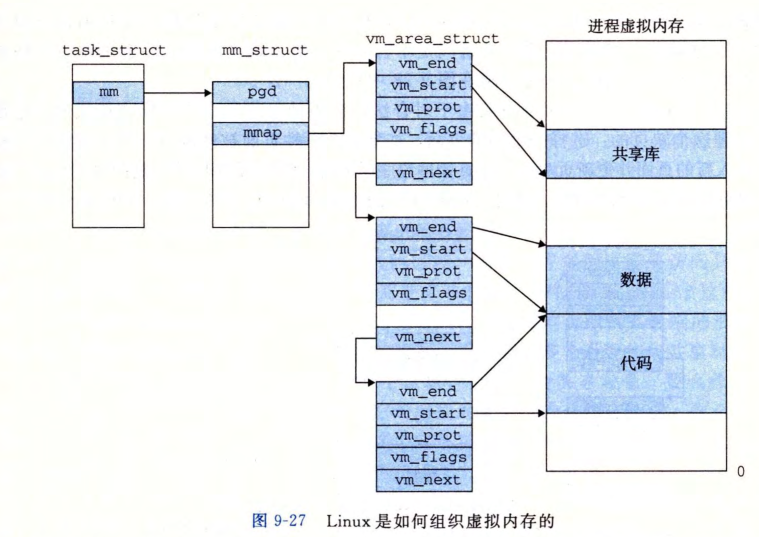
Linux为每个进程都维护了一个单独的虚拟地址空间



内核虚拟内存的某些区域被映射到所有进程共享的物理页面。例如，每个进程共享内核的代码和全局数据结构。

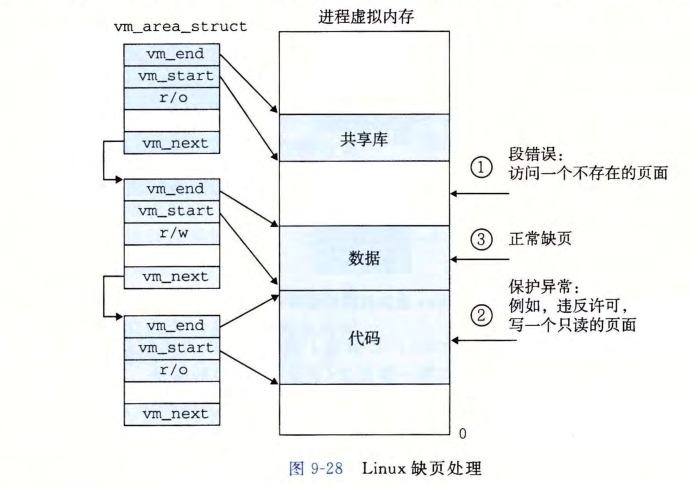
内核还将一组连续的虚拟页（大小等于DRAM总量）映射到一组相应的连续物理页上，这样内核可以在这个虚拟内存上进行读写，实际上就是对物理内存进行读写，这为内核提供一种便利的方法来访问物理内存。

此外， 与进程相关的一些信息被保存在内核虚拟内存中，并且**Linux将虚拟内存组织成段集合的形式**，比如代码段、数据段、共享库段以及用户栈，这些段就是已分配的连续虚拟页。每个已分配的虚拟页都保存在某个段中。通过这种组织成段的形式，可以使得虚拟地址存在间隙。



这是通过内核为系统中每个进程维护一个单独的task\_struct实现的（在最上面那块中），该结构中包含或指向内核运行该进程所需的**所有信息**。其中有一条指向**描述虚拟内存当前状态的结构**mm\_struct，其中包含的**pgd就是指向一级页表的起始地址**，当进程被调度时，就会将pgd拷贝到CR3中，通过这种形式来改变虚拟地址空间；

mmap指向一个链表结构的头结点，其中每个链表元素vm\_area\_struct就描述了虚拟内存中的一个段。



Linux缺页异常处理：

1) 虚拟地址 A 是合法的吗？换句话说，A 在某个区域结构定义的区域内吗？为了回答这个问题，缺页处理程序搜索区域结构的链表，把 A 和每个区域结构中的 vm\_start 和vm\_end 做比较。如果这个指令是不合法的，那么缺页处理程序就触发一个段错误，从而终止这个进程。

2) 试图进行的内存访问是否合法？换句话说，进程是否有读、写或者执行这个区域内页面的权限？例如，这个缺页是不是对一个代码段里的只读页进行写操作造成的？这个缺页是不是因为一个运行在用户模式中的进程试图从内核虚拟内存中读取字造成的？如果试图进行的访问是不合法的，那么缺页处理程序会触发一个保护异常，从而终止这个进程。

3) 正常缺页。处理程序会选择一个物理牺牲页，如果牺牲页被修改了，则进行写回，然后将虚拟地址A对应的虚拟页写入对应的物理页中，修改对应的页表，然后从处理程序返回。

**内存映射**

如果虚拟内存系统可以集成到传统的文件系统中，就能提供一种简单而高效的方法把程序和数据加载到内存中。这里通过内存映射（Memory Mapping）将一个**虚拟内存段**与一个**磁盘上的对象**（Object）关联起来，来初始化虚拟内存段的内容，所以当我们第一次引用某个段时，它的初始值来自于磁盘上的普通文件，然后通过按需页面调度，将需要的虚拟内存段加载到物理内存中。

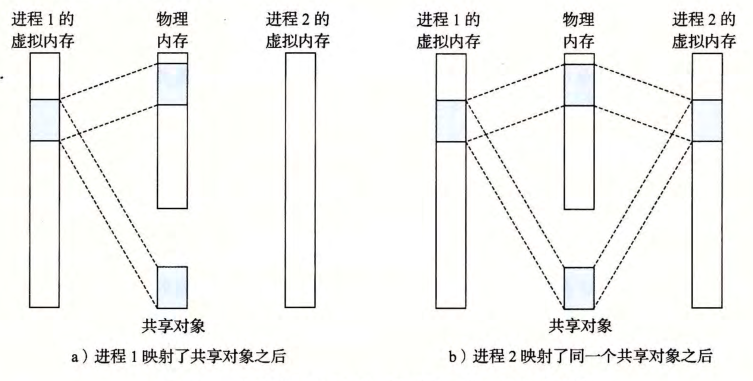
内存映射可以用两种磁盘上的对象来初始化虚拟内存段：

* Linux文件系统中的普通文件：比如一个可执行目标文件，将其分成页大小的数据块，用来初始化对应的虚拟内存段，如果段比文件大，则用0来填充剩下的内容。**因为按需进行页面调度，所以这些虚拟页面没有实际交换进入物理内存，直到 CPU 第一次引用到页面。**
* 匿名文件：内核可以创建大小任意的全是二进制0的匿名文件，该文件其实并不存在。当我们想要创建一个全为0的页，可以将虚拟页和匿名文件关联起来，当内核引用该虚拟页时，会直接覆盖牺牲页的内容，得到全为0的页，然后修改对应的页表内容。这个过程磁盘和内存之间没有实际的数据传送。

无论在哪种情况中，一旦一个虚拟页面被初始化了，它就在一个由内核维护的专门的交换文件 (swap file) 之间换来换去。交换文件也叫做交换空间 (swap space) 或者交换区域。(swap area) 。需要意识到的很重要的一点是，在任何时刻，交换空间都限制着当前运行着的进程能够分配的虚拟页面的总数。

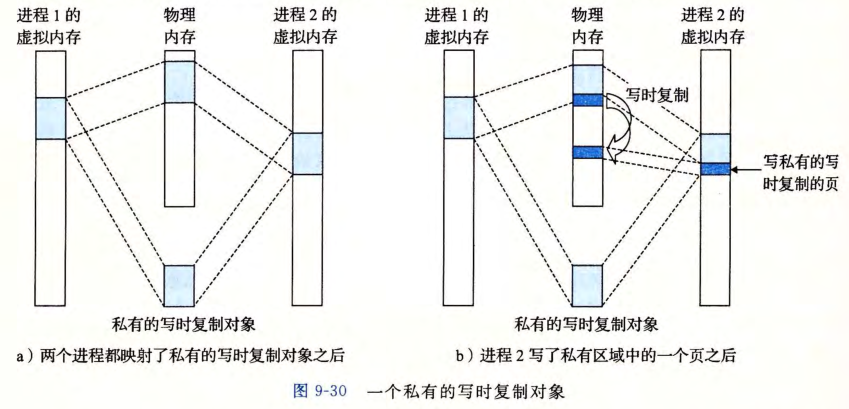
**再看共享对象**

虚拟内存段中通过vm\_prot确定该虚拟内存段中的虚拟页是否在进程之间共享，如果共享，则称该段为共享段，则各个进程在该段中的操作是互相可见的，且变化也会反映在磁盘上；如果不共享，则称该段为私有段，则各个进程在该段中的操作是互相不可见的，也不会反映到磁盘上。



在进程1中，当我们将磁盘上的一个对象通过内存映射与该进程的一个共享段关联起来时，就会使得虚拟页对应的PTE指向该对象，当引用该对象时，就会将对应的虚拟页加载到物理页中；

此时，进程2也将该对象与自己的一个共享段关联起来。当引用该对象时，由于每个对象都有一个唯一的文件名，所以内核可以发现进程1将该对象加载的物理页，就直接在进程2中将对应的PTE指向相同的物理页就行了。



私有对象使用一种叫做**写时复制**的技术被映射到虚拟内存中。

简略地说，如果两个进程映射同一个私有对象，只要没有写操作，则物理内存中只放置一份该对象。

当任何一个进程试图进行写操作，则一般保护程序把要写的那部分拷贝，在拷贝出来的那一份进行写操作。

**再看Fork函数**

当 fork 函数被当前进程调用时，内核为新进程创建各种数据结构，并分配给它一个唯一的PID。

1、为了具有和父进程相同的虚拟内存状态，内核会复制父进程的**mm\_struct**

2、为了具有和父进程相同的虚拟内存段分配，内核复制父进程的**vm\_area\_struct**

3、为了子进程和父进程具有相同的虚拟内存内容，内核会复制父进程的**页表**，就能将相同的磁盘内容映射到相同的虚拟页中，并将虚拟页缓存在相同的物理页中。

4、为了子进程和父进程的虚拟地址空间能相互独立，两个进程的页都设置为只读的，且段都标记为私有的**写回复制**。当父子进程都没有对页进行修改时，父子进程是共享相同的物理内存的，当其中一个进程对页进行修改时，就会对该页进行写回复制，并为该页赋予写权限，并更新进程对应的页表。

**再看 execve 函数**

假设执行了：execve("a.out", NULL, NULL);

• 删除已存在的用户段。

• 映射私有段：

所有这些新的段都是**私有的、写时复制的**。

代码和数据段被映射为 a.out 文件中的 .text和 .data 区。

bss 段初始为0（映射到匿名文件），其大小包含在 a.out 中。

栈和堆段初始为0，初始长度也为零。

• 映射共享段。如果 a.out 程序与共享对象链接，那么这些对象都是动态链接到这个程序的，然后再映射到用户虚拟地址空间中的共享区域内。

• 设置程序计数器 (PC) 。execve 做的最后一件事情就是设置当前进程上下文中的程序计数器，使之指向代码区域的入口点。下一次调度这个进程时，它将从这个入口点开始执行。

Linux 将根据需要换入代码和数据页面。

