**1. 虚拟地址和物理地址：**

虚拟地址关系到进程的用户空间和内核空间，而物理地址则用来寻址实际可用的内存。

linux用页表来为物理地址分配虚拟地址。

进程的虚拟地址空间，都被内核划分为很多等长的部分，这样的部分成为页。物理内存也划分为同样大小的页。

**2. 页表**

**页表**是一种数据结构，用来将虚拟地址空间映射到物理地址空间。

实现两个地址空间关联最容易的方法是使用数组，对虚拟地址空间的每一页都分配一个数组项，同时该数组项指向与之关联的物理地址页。

因为虚拟地址空间的大部分区域没有使用，所以可以使用内存用量少的**多级分页模型。**

**3. 多级分页模型：**

为减少页表的大小并忽略不需要的区域，linux把虚拟地址划分为多个部分。

举个例子：

我们将虚拟地址划分为4个部分，这样就需要一个三级的页表。

**a. 虚拟地址的第一个部分称为全局页目录(Page Global Directory, PGD)** --- PGD 用于索引进程中的一个数组，每个进程有且只有一个PGD，PGD的数组项指向另一些数组的起始地址，这些数组称为**中间页目录。**

**b. 虚拟地址的第二部分称为中间页目录(Page Middle Directory)** --- 在通过PGD中的数组找到对应的PMD之后，PMD的数组项也是一个指针，指向下一级数组，即**页表或页目录**。

**c. 虚拟地址的第三部分称为PTE(Page Table Entry, 页表数组) ---**PTE 用来索引页表，**页表的数组项指向物理内存页。**

**虚拟内存页和物理内存页之间的映射就此完成，**

**d. 虚拟地址的最后一部分称为偏移量，它指定了页内部的一个字节位置，每个地址都指向地址空间的某个字节。**

**页表的特色在于，对虚拟地址空间中不需要的区域，不必创建中间页目录或页表，所以多级页表节省了大量内存。**

**多级页表的缺点就是每次访问内存时，必须逐级访问多个数组后，才能将虚拟地址转换为物理地址。linux使用下面两种方法来加速该过程：**

**4. 内存访问加速方法：**

**a. 使用了MMU（Memory Management Unit，内存管理单元）。**

**b. 地址转换中出现最频繁的那些地址，保存到 地址转换后备缓冲器(Translation Lookaside Buffer, TLB)的高速缓存中。这些地址无需访问页表即可从高速缓存中直接获得地址数据。**

**5. 内存映射：**

**将任意来源的数据传输到进程的虚拟地址空间中。**

**用户可以像普通内存一样访问作为映射目标的地址空间区域，但任何修改都会自动传输到数据源。**

**例如：文件的内容可以映射到内存中，只需读取相应的内存即可访问文件内容，或者向内存写入数据来修改文件的内容。内核会保证任何修改都会自动同步到文件中。**

**内核在实现设备驱动程序时直接使用了内存映射，外设的I/O可以映射到虚拟地址空间中，对相应内存区域的读写会由系统重定向到设备，从而大大简化了驱动程序的实现。**

我们知道编译好的可执行程序都保存在硬盘上，那么它是怎么load到内存以及被用户访问的呢？

大概讲一下过程：当某个程序开始执行时，可执行映象必须装入到进程的虚拟地址空间。当程序执行时，被引用的部分会由操作系统装入到物理内存。映射到进程虚拟地址空间这一过程由do\_mmap()函数完成。它扫描当前进程地址空间的vm\_area\_struct结构所形成的红黑树，找到一个合适的区间，找到之后为新的虚拟区分配一个vm\_area\_struct结构，这是通过调用Slab分配函数kmem\_cache\_alloc()来实现的，然后就对这个结构的各个域进行了初始化。页面的映射到底在何时建立？实际上，generic\_file\_mmap( )就是真正进行映射的函数。因为这个函数的实现涉及很多文件系统的内容，了解了文件系统的有关内容后，再分析。注意，这里要说明的是，文件到虚存的映射仅仅是建立了一种映射关系，也就是说，虚存页面到物理页面之间的映射还没有建立。当某个可执行映象映射到进程虚拟内存中并开始执行时，因为只有很少一部分虚拟内存区间装入到了物理内存，可能会遇到所访问的数据不在物理内存。这时，处理器将向 Linux 报告一个页故障及其对应的故障原因，于是就用到了请页机制(即缺页异常)。虚拟地址映射到物理地址，只有在请页时才完成，这时要建立页表和更新页表（页表是动态建立的）。页表不可被换出，不记年龄，它们被内核中保留，只有在exit时清除。

[**Linux 虚拟内存和物理内存的理解**](http://blog.csdn.net/dlutbrucezhang/article/details/9058583)

分类： [Linux System Programming](http://blog.csdn.net/DLUTBruceZhang/article/category/1442457)2013-06-12 09:483902人阅读[评论](http://blog.csdn.net/dlutbrucezhang/article/details/9058583#comments)(1)[收藏](javascript:void(0);)[举报](http://blog.csdn.net/dlutbrucezhang/article/details/9058583#report)

[内存](http://www.csdn.net/tag/%e5%86%85%e5%ad%98)[Linux](http://www.csdn.net/tag/Linux)[内核](http://www.csdn.net/tag/%e5%86%85%e6%a0%b8)[磁盘](http://www.csdn.net/tag/%e7%a3%81%e7%9b%98)[物理虚拟内存](http://www.csdn.net/tag/%e7%89%a9%e7%90%86%e8%99%9a%e6%8b%9f%e5%86%85%e5%ad%98)

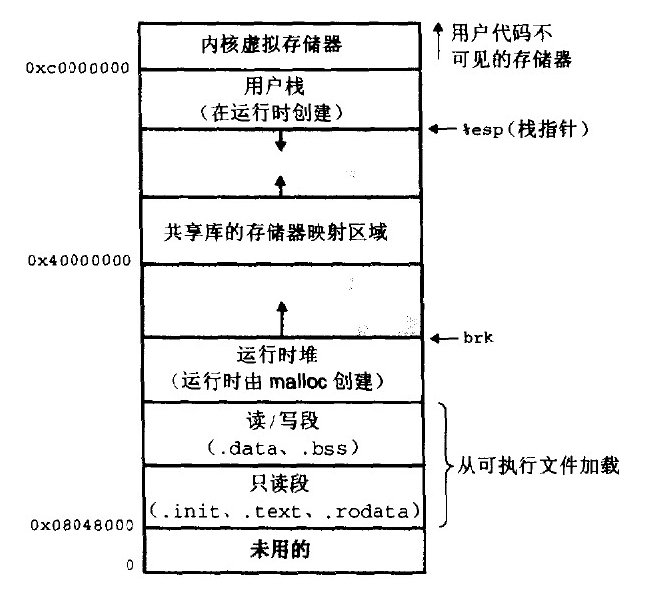
**微笑首先，让我们看下虚拟内存：**

**第一层理解**

**1. 每个进程都有自己独立的4G内存空间，各个进程的内存空间具有类似的结构**

**2. 一个新进程建立的时候，将会建立起自己的内存空间，此进程的数据，代码等从磁盘拷贝到自己的进程空间，哪些数据在哪里，都由进程控制表中的task\_struct记录，task\_struct中记录中一条链表，记录中内存空间的分配情况，哪些地址有数据，哪些地址无数据，哪些可读，哪些可写，都可以通过这个链表记录**

**3. 每个进程已经分配的内存空间，都与对应的磁盘空间映射**

****

**问题：**

**计算机明明没有那么多内存（n个进程的话就需要n\*4G）内存**

**建立一个进程，就要把磁盘上的程序文件拷贝到进程对应的内存中去，对于一个程序对应的多个进程这种情况，浪费内存！**

**第二层理解**

**1. 每个进程的4G内存空间只是虚拟内存空间，每次访问内存空间的某个地址，都需要把地址翻译为实际物理内存地址**

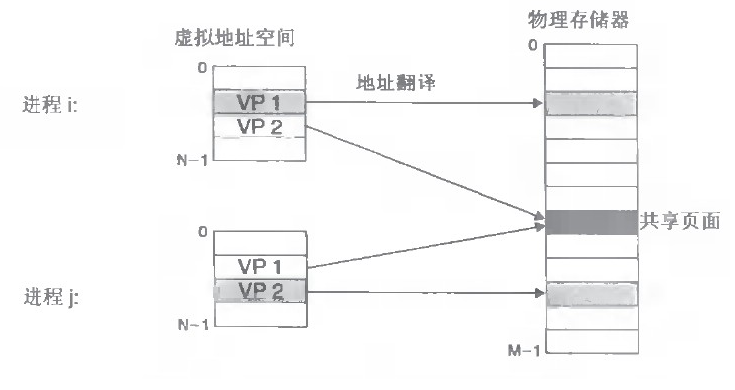
**2. 所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。**

**3. 进程要知道哪些内存地址上的数据在物理内存上，哪些不在，还有在物理内存上的哪里，需要用页表来记录**

**4. 页表的每一个表项分两部分，第一部分记录此页是否在物理内存上，第二部分记录物理内存页的地址（如果在的话）**

**5. 当进程访问某个虚拟地址，去看页表，如果发现对应的数据不在物理内存中，则缺页异常**

**6. 缺页异常的处理过程，就是把进程需要的数据从磁盘上拷贝到物理内存中，如果内存已经满了，没有空地方了，那就找一个页覆盖，当然如果被覆盖的页曾经被修改过，需要将此页写回磁盘**

****

**总结：**

**优点：**

**1.既然每个进程的内存空间都是一致而且固定的，所以链接器在链接可执行文件时，可以设定内存地址，而不用去管这些数据最终实际的内存地址，这是有独立内存空间的好处**

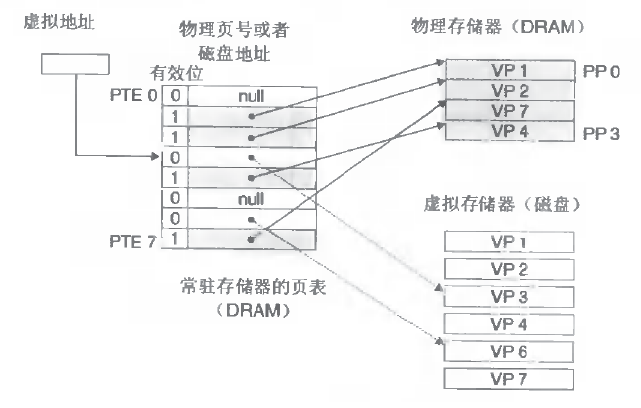
**2.当不同的进程使用同样的代码时，比如库文件中的代码，物理内存中可以只存储一份这样的代码，不同的进程只需要把自己的虚拟内存映射过去就可以了，节省内存**

**3.在程序需要分配连续的内存空间的时候，只需要在虚拟内存空间分配连续空间，而不需要实际物理内存的连续空间，可以利用碎片。**

**另外，事实上，在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，具体就是初始化进程控制表中内存相关的链表，实际上并不立即就把虚拟内存对应位置的程序数据和代码（比如.text .data段）拷贝到物理内存中，只是建立好虚拟内存和磁盘文件之间的映射就好（叫做存储器映射），等到运行到对应的程序时，才会通过缺页异常，来拷贝数据。还有进程运行过程中，要动态分配内存，比如malloc时，也只是分配了虚拟内存，即为这块虚拟内存对应的页表项做相应设置，当进程真正访问到此数据时，才引发缺页异常。**

**补充理解：**

**虚拟存储器涉及三个概念： 虚拟存储空间，磁盘空间，内存空间**

****

**可以认为虚拟空间都被映射到了磁盘空间中，（事实上也是按需要映射到磁盘空间上，通过mmap），并且由页表记录映射位置，当访问到某个地址的时候，通过页表中的有效位，可以得知此数据是否在内存中，如果不是，则通过缺页异常，将磁盘对应的数据拷贝到内存中，如果没有空闲内存，则选择牺牲页面，替换其他页面。**

**mmap是用来建立从虚拟空间到磁盘空间的映射的，可以将一个虚拟空间地址映射到一个磁盘文件上，当不设置这个地址时，则由系统自动设置，函数返回对应的内存地址（虚拟地址），当访问这个地址的时候，就需要把磁盘上的内容拷贝到内存了，然后就可以读或者写，最后通过manmap可以将内存上的数据换回到磁盘，也就是解除虚拟空间和内存空间的映射，这也是一种读写磁盘文件的方法，也是一种进程共享数据的方法 共享内存**

**微笑接下来我们来讨论下物理内存：**

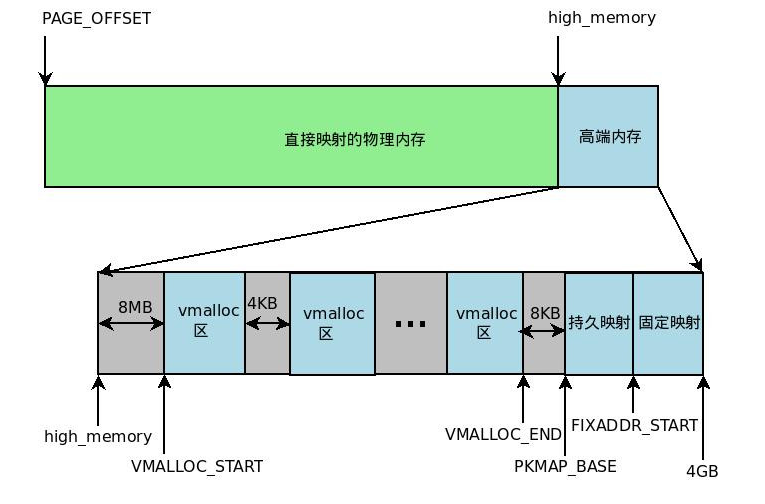
**在内核态申请内存比在用户态申请内存要更为直接，它没有采用用户态那种延迟分配内存技术。内核认为一旦有内核函数申请内存，那么就必须立刻满足该申请内存的请求，并且这个请求一定是正确合理的。相反，对于用户态申请内存的请求，内核总是尽量延后分配物理内存，用户进程总是先获得一个虚拟内存区的使用权，最终通过缺页异常获得一块真正的物理内存。**

**1.物理内存的内核映射**

**IA32架构中内核虚拟地址空间只有1GB大小（从3GB到4GB），因此可以直接将1GB大小的物理内存（即常规内存）映射到内核地址空间，但超出1GB大小的物理内存（即高端内存）就不能映射到内核空间。为此，内核采取了下面的方法使得内核可以使用所有的物理内存。**

**1).高端内存不能全部映射到内核空间，也就是说这些物理内存没有对应的线性地址。不过，内核为每个物理页框都分配了对应的页框描述符，所有的页框描述符都保存在mem\_map数组中，因此每个页框描述符的线性地址都是固定存在的。内核此时可以使用alloc\_pages()和alloc\_page()来分配高端内存，因为这些函数返回页框描述符的线性地址。**

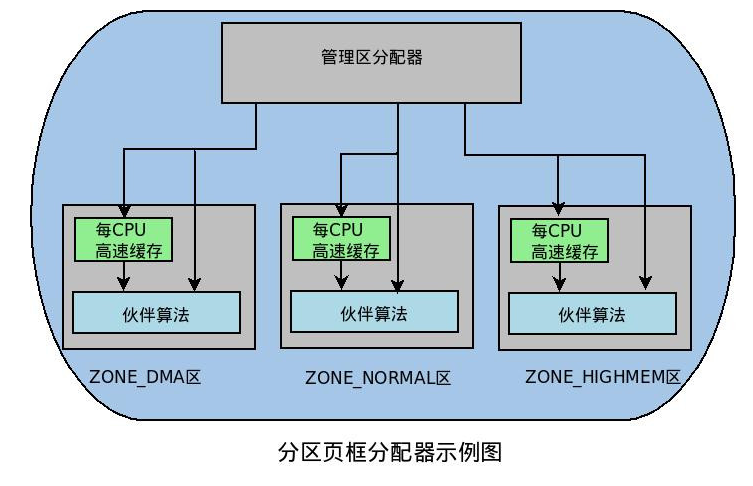
**2).内核地址空间的后128MB专门用于映射高端内存，否则，没有线性地址的高端内存不能被内核所访问。这些高端内存的内核映射显然是暂时映射的，否则也只能映射128MB的高端内存。当内核需要访问高端内存时就临时在这个区域进行地址映射，使用完毕之后再用来进行其他高端内存的映射。  
由于要进行高端内存的内核映射，因此直接能够映射的物理内存大小只有896MB，该值保存在high\_memory中。内核地址空间的线性地址区间如下图所示：**



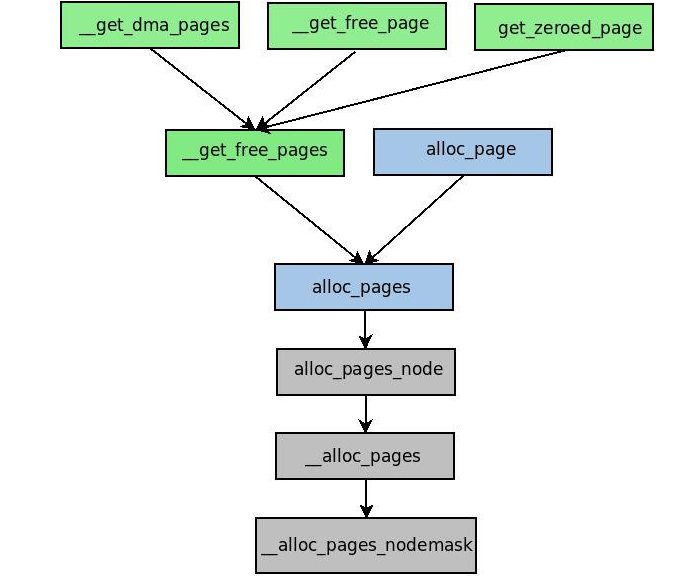
**从图中可以看出，内核采用了三种机制将高端内存映射到内核空间：永久内核映射，固定映射和vmalloc机制。**

**2.物理内存管理机制  
基于物理内存在内核空间中的映射原理，物理内存的管理方式也有所不同。内核中物理内存的管理机制主要有伙伴算法，slab高速缓存和vmalloc机制。其中伙伴算法和slab高速缓存都在物理内存映射区分配物理内存，而vmalloc机制则在高端内存映射区分配物理内存。  
伙伴算法  
伙伴算法负责大块连续物理内存的分配和释放，以页框为基本单位。该机制可以避免外部碎片。  
per-CPU页框高速缓存  
内核经常请求和释放单个页框，该缓存包含预先分配的页框，用于满足本地CPU发出的单一页框请求。  
slab缓存  
slab缓存负责小块物理内存的分配，并且它也作为高速缓存，主要针对内核中经常分配并释放的对象。  
vmalloc机制  
vmalloc机制使得内核通过连续的线性地址来访问非连续的物理页框，这样可以最大限度的使用高端物理内存。**

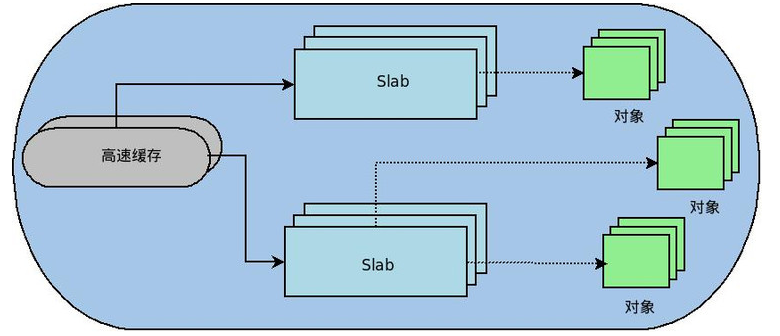
**3.物理内存的分配  
内核发出内存申请的请求时，根据内核函数调用接口将启用不同的内存分配器。  
3.1 分区页框分配器  
分区页框分配器 (zoned page frame allocator) ,处理对连续页框的内存分配请求。分区页框管理器分为两大部分:前端的管理区分配器和伙伴系统，如下图：**



**管理区分配器负责搜索一个能满足请求页框块大小的管理区。在每个管理区中,具体的页框分配工作由伙伴系统负责。为了达到更好的系统性能,单个页框的申请工作直接通过per-CPU页框高速缓存完成。  
该分配器通过几个函数和宏来请求页框,它们之间的封装关系如下图所示。**



**这些函数和宏将核心的分配函数\_\_alloc\_pages\_nodemask()封装,形成满足不同分配需求的分配函数。其中，alloc\_pages()系列函数返回物理内存首页框描述符，\_\_get\_free\_pages()系列函数返回内存的线性地址。  
3.2 slab分配器  
slab 分配器最初是为了解决物理内存的内部碎片而提出的，它将内核中常用的数据结构看做对象。slab分配器为每一种对象建立高速缓存。内核对该对象的分配和释放均是在这块高速缓存中操作。一种对象的slab分配器结构图如下：**



**可以看到每种对象的高速缓存是由若干个slab组成，每个slab是由若干个页框组成的。虽然slab分配器可以分配比单个页框更小的内存块，但它所需的所有内存都是通过伙伴算法分配的。  
slab高速缓存分专用缓存和通用缓存。专用缓存是对特定的对象，比如为内存描述符创建高速缓存。通用缓存则是针对一般情况，适合分配任意大小的物理内存，其接口即为kmalloc()。  
3.3 非连续内存区内存的分配  
内核通过vmalloc()来申请非连续的物理内存，若申请成功，该函数返回连续内存区的起始地址，否则，返回NULL。vmalloc()和kmalloc()申请的内存有所不同，kmalloc()所申请内存的线性地址与物理地址都是连续的，而vmalloc()所申请的内存线性地址连续而物理地址则是离散的，两个地址之间通过内核页表进行映射。  
vmalloc()的工作方式理解起来很简单：  
1).寻找一个新的连续线性地址空间；  
2).依次分配一组非连续的页框；  
3).为线性地址空间和非连续页框建立映射关系，即修改内核页表；  
vmalloc()的内存分配原理与用户态的内存分配相似，都是通过连续的虚拟内存来访问离散的物理内存，并且虚拟地址和物理地址之间是通过页表进行连接的，通过这种方式可以有效的使用物理内存。但是应该注意的是，vmalloc()申请物理内存时是立即分配的，因为内核认为这种内存分配请求是正当而且紧急的；相反，用户态有内存请求时，内核总是尽可能的延后，毕竟用户态跟内核态不在一个特权级。**