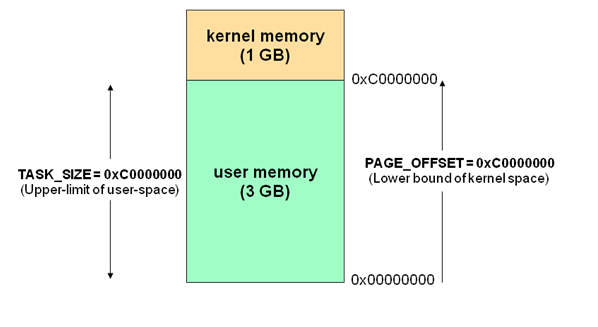
[用户空间与内核空间，进程上下文与中断上下文[总结]](http://www.cnblogs.com/Anker/p/3269106.html)

**1、前言**

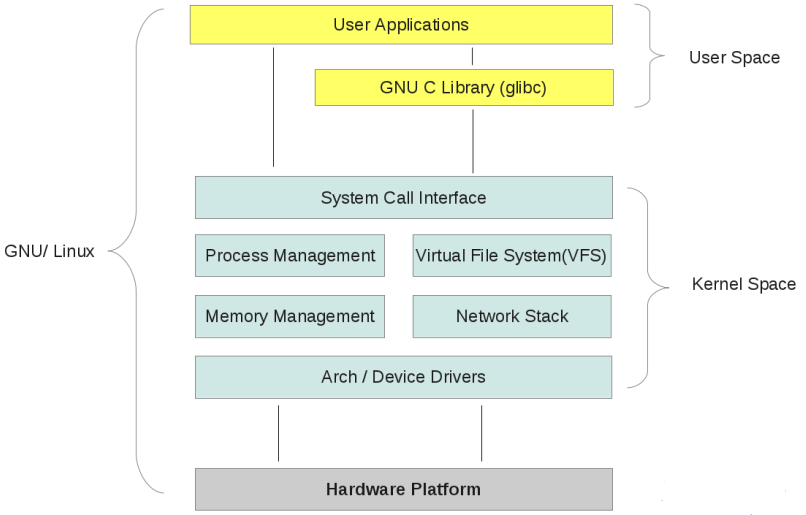
　　最近在学习linux内核方面的知识，经常会看到用户空间与内核空间及进程上下文与中断上下文。看着很熟悉，半天又说不出到底是怎么回事，有什么区别。看书过程经常被感觉欺骗，似懂非懂的感觉，很是不爽，今天好好结合书和网上的资料总结一下，加深理解。

**2、用户空间与内核空间**

　　我们知道现在操作系统都是采用虚拟存储器，那么对32位操作系统而言，它的寻址空间（虚拟存储空间）为4G（2的32次方）。操心系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核，保证内核的安全，操心系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。针对linux操作系统而言，将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为内核空间，而将较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF），供各个进程使用，称为用户空间。每个进程可以通过系统调用进入内核，因此，Linux内核由系统内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有4G字节的虚拟空间。空间分配如下图所示：



**有了用户空间和内核空间，整个linux内部结构可以分为三部分，从最底层到最上层依次是：硬件-->内核空间-->用户空间。**如下图所示：



**需要注意的细节问题：**

（1） 内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。

（2） Linux使用两级保护机制：0级供内核使用，3级供用户程序使用。

**内核态与用户态：**

（1）**当一个任务（进程）执行系统调用而陷入内核代码中执行时，称进程处于内核运行态（内核态）。**此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。

（2）**当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）**。此时处理器在特权级最低的（3级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。

**参考资料：**

<http://blog.csdn.net/f22jay/article/details/7925531> [链接](#linux用户空间与内核空间)

<http://blog.csdn.net/zhangskd/article/details/6956638>

<http://blog.chinaunix.net/uid-26838492-id-3162146.html>

**3、进程上下文与中断上下文**

　　我在看《linux内核设计与实现》这本书的第三章进程管理时候，看到进程上下文。书中说当一个程序执行了系统调用或者触发某个异常（软中断），此时就会陷入内核空间，内核此时代表进程执行，并处于进程上下文中。看后还是没有弄清楚，什么是进程上下文，如何上google上面狂搜一把，总结如下：

　　程序在执行过程中通常有用户态和内核态两种状态，CPU对处于内核态根据上下文环境进一步细分，因此有了下面三种状态：

（1）内核态，运行于进程上下文，内核代表进程运行于内核空间。  
（2）内核态，运行于中断上下文，内核代表硬件运行于内核空间。  
（3）用户态，运行于用户空间。

**上下文context： 上下文简单说来就是一个环境。**

　　用户空间的应用程序，通过系统调用，进入内核空间。这个时候用户空间的进程要传递 很多变量、参数的值给内核，内核态运行的时候也要保存用户进程的一些寄存 器值、变量等。**所谓的“进程上下文”，可以看作是用户进程传递给内核的这些参数以及内核要保存的那一整套的变量和寄存器值和当时的环境等。**

　　相对于进程而言，就是进程执行时的环境。具体来说就是各个变量和数据，包括所有的寄存器变量、进程打开的文件、内存信息等。一个进程的上下文可以分为三个部分:用户级上下文、寄存器上下文以及系统级上下文。

（1）用户级上下文: 正文、数据、用户堆栈以及共享存储区；  
（2）寄存器上下文: 通用寄存器、程序寄存器(IP)、处理器状态寄存器(EFLAGS)、栈指针(ESP)；  
（3）系统级上下文: 进程控制块task\_struct、内存管理信息(mm\_struct、vm\_area\_struct、pgd、pte)、内核栈。

**当发生进程调度时，进行进程切换就是上下文切换(context switch).操作系统必须对上面提到的全部信息进行切换，新调度的进程才能运行。而系统调用进行的模式切换(mode switch)。模式切换与进程切换比较起来，容易很多，而且节省时间，因为模式切换最主要的任务只是切换进程寄存器上下文的切换。**

　　硬件通过触发信号，导致内核调用中断处理程序，进入内核空间。这个过程中，硬件的 一些变量和参数也要传递给内核，内核通过这些参数进行中断处理。**所谓的“ 中断上下文”，其实也可以看作就是硬件传递过来的这些参数和内核需要保存的一些其他环境（主要是当前被打断执行的进程环境）。**中断时，内核不代表任何进程运行，它一般只访问系统空间，而不会访问进程空间，内核在中断上下文中执行时一般不会阻塞。

**摘录Linux注释的内容如下：**

**Process Context**  
**-------------------------------------------**  
**One of the most important parts of a process is the executing program code. This code is read in from an executable file and executed within the program's address space. Normal program execution occurs in user-space. When a program executes a system call or triggers an exception, it enters kernel-space. At this point, the kernel is said to be "executing on behalf of the process" and is in process context. When in process context, the current macro is valid[7]. Upon exiting the kernel, the process resumes execution in user-space, unless a higher-priority process has become runnable in the interim(过渡期), in which case the scheduler is invoked to select the higher priority process.**  
  
**Other than process context there is interrupt context, In interrupt context, the system is not running on behalf of a process, but is executing an interrupt handler. There is no process tied to interrupt handlers and consequently no process context.**  
  
**System calls and exception handlers are well-defined interfaces into the kernel. A process can begin executing in kernel-space only through one of these interfaces -- all access to the kernel is through these interfaces.**  
  
**-------------------------------------------**

**Interrupt Context**  
-------------------------------------------  
   **When executing an interrupt handler or bottom half, the kernel is in interrupt context. Recall that process context is the mode of operation the kernel is in while it is executing on behalf of a process -- for example, executing a system call or running a kernel thread. In process context, the current macro points to the associated task. Furthermore, because a process is coupled to the kernel in process context(因为进程是以进程上文的形式连接到内核中的), process context can sleep or otherwise invoke the scheduler.**  
  
**Interrupt context, on the other hand, is not associated with a process. The current macro is not relevant (although it points to the interrupted process). Without a backing process(由于没有进程的背景),interrupt context cannot sleep -- how would it ever reschedule?(否则怎么再对它重新调度?) Therefore, you cannot call certain functions from interrupt context. If a function sleeps, you cannot use it from your interrupt handler -- this limits the functions that one can call from an interrupt handler.(这是对什么样的函数可以在中断处理程序中使用的限制)**  
  
**Interrupt context is time critical because the interrupt handler interrupts other code. Code should be quick and simple. Busy looping is discouraged. This is a very important point; always keep in mind that your interrupt handler has interrupted other code (possibly even another interrupt handler on a different line!). Because of this asynchronous nature, it is imperative(必须) that all interrupt handlers be as quick and as simple as possible. As much as possible, work should be pushed out from the interrupt handler and performed in a bottom half, which runs at a more convenient time.**  
  
**The setup of an interrupt handler's stacks is a configuration option. Historically, interrupt handlers did not receive(拥有) their own stacks. Instead, they would share the stack of the process that they interrupted[1]. The kernel stack is two pages in size; typically, that is 8KB on 32-bit architectures and 16KB on 64-bit architectures. Because in this setup interrupt handlers share the stack, they must be exceptionally frugal(必须非常节省) with what data they allocate there. Of course, the kernel stack is limited to begin with, so all kernel code should be cautious.**  
  
**A process is always running. When nothing else is schedulable, the idle task runs.**

**-------------------------------------------**

**LINUX完全注释中的一段话：**

**当一个进程在执行时,CPU的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文。当内核需要切换到另一个进程时，它需要保存当前进程的所有状态，即保存当前进程的上下文，以便在再次执行该进程时，能够必得到切换时的状态执行下去。在LINUX中，当前进程上下文均保存在进程的任务数据结构中。在发生中断时,内核就在被中断进程的上下文中，在内核态下执行中断服务例程。但同时会保留所有需要用到的资源，以便中继服务结束时能恢复被中断进程的执行。**

**参考资料**：

<http://www.cnblogs.com/hustcat/articles/1505618.html>

<http://mprc.pku.edu.cn/~zhengyansong/blog/?p=199>

<http://blog.chinaunix.net/uid-26980210-id-3235544.html>

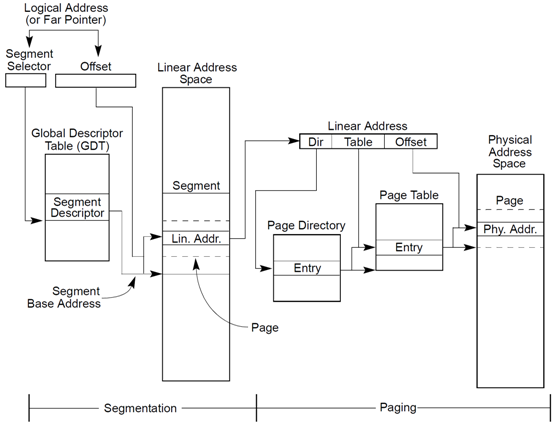
Linux用户空间与内核空间

Linux 操作系统和驱动程序运行在内核空间，应用程序运行在用户空间，两者不能简单地使用指针传递数据，因为Linux使用的虚拟内存机制，用户空间的数据可能被换出，当内核空间使用用户空间指针时，对应的数据可能不在内存中。

**Linux内核地址映射模型**

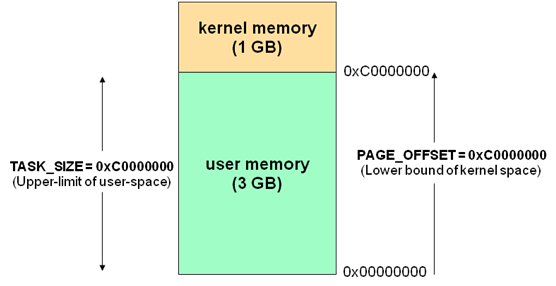
**x86 CPU**采用了段页式地址映射模型。**进程代码**中的地址为逻辑地址，经过段页式地址映射后，才真正访问物理内存。

段页式机制如下图。



**Linux内核地址空间划分**

通常**32位Linux内核**地址空间划分0~3G为用户空间，3~4G为内核空间。注意这里是32位内核地址空间划分，64位内核地址空间划分是不同的。



**Linux内核高端内存的由来**

当内核模块代码或线程访问内存时，代码中的内存地址都为逻辑地址，而对应到真正的物理内存地址，需要地址**一对一**的映射，如逻辑地址0xc0000003对应的物理地址为0×3，0xc0000004对应的物理地址为0×4，… …，逻辑地址与物理地址对应的关系为

物理地址 = 逻辑地址 – 0xC0000000

|  |  |
| --- | --- |
| **逻辑地址** | **物理内存地址** |
| 0xc0000000 | 0×0 |
| 0xc0000001 | 0×1 |
| 0xc0000002 | 0×2 |
| 0xc0000003 | 0×3 |
| … | … |
| 0xe0000000 | 0×20000000 |
| … | … |
| **0xffffffff** | **0×40000000 ??** |

假 设按照上述简单的地址映射关系，那么内核逻辑地址空间访问为0xc0000000 ~ 0xffffffff，那么对应的物理内存范围就为0×0 ~ 0×40000000，即只能访问1G物理内存。若机器中安装8G物理内存，那么内核就只能访问前1G物理内存，后面7G物理内存将会无法访问，因为内核 的地址空间已经全部映射到物理内存地址范围0×0 ~ 0×40000000。即使安装了8G物理内存，那么物理地址为0×40000001的内存，内核该怎么去访问呢？代码中必须要有内存逻辑地址 的，0xc0000000 ~ 0xffffffff的地址空间已经被用完了，所以无法访问物理地址0×40000000以后的内存。

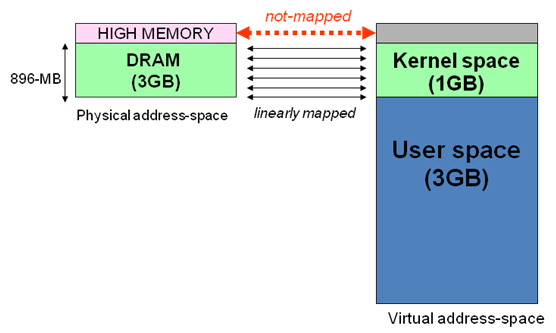
显 然不能将内核地址空间0xc0000000 ~ 0xfffffff全部用来简单的地址映射。因此x86架构中将内核地址空间划分三部分：ZONE\_DMA、ZONE\_NORMAL和 ZONE\_HIGHMEM。ZONE\_HIGHMEM即为高端内存，这就是内存高端内存概念的由来。

在x86结构中，三种类型的区域如下：

**ZONE\_DMA**      内存开始的16MB

**ZONE\_NORMAL**     16MB~896MB

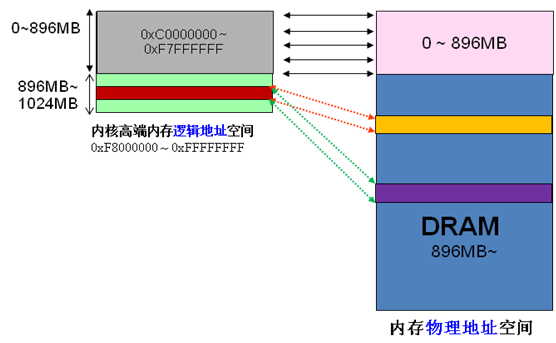
**ZONE\_HIGHMEM**       896MB ~ 结束



**Linux内核高端内存的理解**

前 面我们解释了高端内存的由来。 Linux将内核地址空间划分为三部分ZONE\_DMA、ZONE\_NORMAL和ZONE\_HIGHMEM，高端内存HIGH\_MEM地址空间范围为 0xF8000000 ~ 0xFFFFFFFF（896MB～1024MB）。那么如内核是**如何借助128MB高端内存地址空间是如何实现访问可以所有物理内存**？

当内核想访问高于896MB物理地址内存时，从0xF8000000 ~ 0xFFFFFFFF地址空间范围内找一段相应大小空闲的逻辑地址空间，借用一会。借用这段逻辑地址空间，建立映射到想访问的那段物理内存（即填充内核PTE页面表），**临时用一会，用完后归还**。这样别人也可以借用这段地址空间访问其他物理内存，实现了使用有限的地址空间，访问所有所有物理内存。如下图。



例 如内核想访问2G开始的一段大小为1MB的物理内存，即物理地址范围为0×80000000 ~ 0x800FFFFF。访问之前先找到一段1MB大小的空闲地址空间，假设找到的空闲地址空间为0xF8700000 ~ 0xF87FFFFF，用这1MB的逻辑地址空间映射到物理地址空间0×80000000 ~ 0x800FFFFF的内存。映射关系如下：

|  |  |
| --- | --- |
| **逻辑地址** | **物理内存地址** |
| 0xF8700000 | 0×80000000 |
| 0xF8700001 | 0×80000001 |
| 0xF8700002 | 0×80000002 |
| … | … |
| 0xF87FFFFF | 0x800FFFFF |

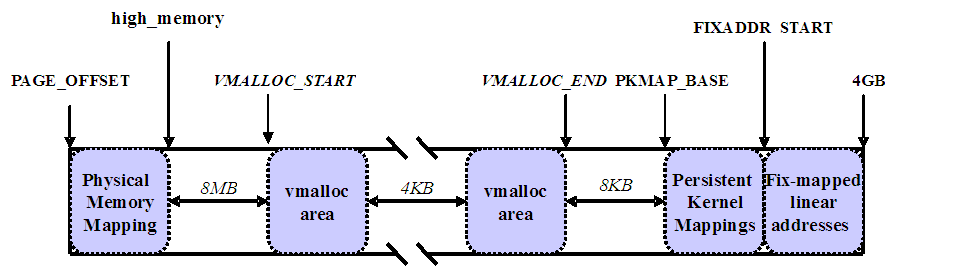
当内核访问完0×80000000 ~ 0x800FFFFF物理内存后，就将0xF8700000 ~ 0xF87FFFFF内核线性空间释放。这样其他进程或代码也可以使用0xF8700000 ~ 0xF87FFFFF这段地址访问其他物理内存。

从上面的描述，我们可以知道**高端内存的最基本思想**：借一段地址空间，建立临时地址映射，用完后释放，达到这段地址空间可以循环使用，访问所有物理内存。

看到这里，不禁有人会问：万一有内核进程或模块一直占用某段逻辑地址空间不释放，怎么办？若真的出现的这种情况，则内核的高端内存地址空间越来越紧张，若都被占用不释放，则没有建立映射到物理内存都无法访问了。

*在 香港尖沙咀有些写字楼，洗手间很少且有门锁的。客户要去洗手间的话，可以向前台拿钥匙，方便完后，把钥匙归还到前台。这样虽然只有一个洗 手间，但可以满足所有客户去洗手间的需求。要是某个客户一直占用洗手间、钥匙不归还，那么其他客户都无法上洗手间了。Linux内核高端内存管理的思想类 似。*

**Linux内核高端内存的划分**  
内核将高端内存划分为3部分：VMALLOC\_START~VMALLOC\_END、KMAP\_BASE~FIXADDR\_START和FIXADDR\_START~4G。



对 于高端内存，可以通过 alloc\_page() 或者其它函数获得对应的 page，但是要想访问实际物理内存，还得把 page 转为线性地址才行（为什么？想想 MMU 是如何访问物理内存的），也就是说，我们需要为高端内存对应的 page 找一个线性空间，这个过程称为高端内存映射。

对应高端内存的3部分，高端内存映射有三种方式：  
**映射到”内核动态映射空间”（noncontiguous memory allocation）**  
这种方式很简单，因为通过 vmalloc() ，在”内核动态映射空间”申请内存的时候，就可能从高端内存获得页面（参看 vmalloc 的实现），因此说高端内存有可能映射到”内核动态映射空间”中。

**持久内核映射（permanent kernel mapping）**  
如果是通过 alloc\_page() 获得了高端内存对应的 page，如何给它找个线性空间？  
内核专门为此留出一块线性空间，从 PKMAP\_BASE 到 FIXADDR\_START ，用于映射高端内存。在 2.6内核上，这个地址范围是 4G-8M 到 4G-4M 之间。这个空间起叫”内核永久映射空间”或者”永久内核映射空间”。这个空间和其它空间使用同样的页目录表，对于内核来说，就是 swapper\_pg\_dir，对普通进程来说，通过 CR3 寄存器指向。通常情况下，这个空间是 4M 大小，因此仅仅需要一个页表即可，内核通过来 pkmap\_page\_table 寻找这个页表。通过 kmap()，可以把一个 page 映射到这个空间来。由于这个空间是 4M 大小，最多能同时映射 1024 个 page。因此，对于不使用的的 page，及应该时从这个空间释放掉（也就是解除映射关系），通过 kunmap() ，可以把一个 page 对应的线性地址从这个空间释放出来。

**临时映射（temporary kernel mapping）**  
内核在 FIXADDR\_START 到 FIXADDR\_TOP 之间保留了一些线性空间用于特殊需求。这个空间称为”固定映射空间”在这个空间中，有一部分用于高端内存的临时映射。

这块空间具有如下特点：  
（1）每个 CPU 占用一块空间  
（2）在每个 CPU 占用的那块空间中，又分为多个小空间，每个小空间大小是 1 个 page，每个小空间用于一个目的，这些目的定义在 kmap\_types.h 中的 km\_type 中。

当要进行一次临时映射的时候，需要指定映射的目的，根据映射目的，可以找到对应的小空间，然后把这个空间的地址作为映射地址。这意味着一次临时映射会导致以前的映射被覆盖。通过 kmap\_atomic() 可实现临时映射。

**常见问题：**

1、用户空间（进程）是否有高端内存概念？

用户进程没有高端内存概念。只有在内核空间才存在高端内存。用户进程最多只可以访问3G物理内存，而内核进程可以访问所有物理内存。

2、64位内核中有高端内存吗？

目前现实中，64位Linux内核不存在高端内存，因为64位内核可以支持超过512GB内存。若机器安装的物理内存超过内核地址空间范围，就会存在高端内存。

3、用户进程能访问多少物理内存？内核代码能访问多少物理内存？

32位系统用户进程最大可以访问3GB，内核代码可以访问所有物理内存。

64位系统用户进程最大可以访问超过512GB，内核代码可以访问所有物理内存。

4、高端内存和物理地址、逻辑地址、线性地址的关系？

高端内存只和逻辑地址有关系，和逻辑地址、物理地址没有直接关系。

5、为什么不把所有的地址空间都分配给内核？

若把所有地址空间都给内存，那么用户进程怎么使用内存？怎么保证内核使用内存和用户进程不起冲突？

（1）让我们忽略Linux对段式内存映射的支持。 在保护模式下，我们知道无论CPU运行于用户态还是核心态，CPU执行程序所访问的地址都是虚拟地址，MMU 必须通过读取控制寄存器CR3中的值作为当前页面目录的指针，进而根据分页内存映射机制（参看相关文档）将该虚拟地址转换为真正的物理地址才能让CPU真 正的访问到物理地址。  
  
（2）对于32位的Linux，其每一个进程都有4G的寻址空间，但当一个进程访问其虚拟内存空间中的某个地址时又是怎样实现不与其它进程的虚拟空间混淆 的呢？每个进程都有其自身的页面目录PGD，Linux将该目录的指针存放在与进程对应的内存结构task\_struct.(struct mm\_struct)mm->pgd中。每当一个进程被调度（schedule()）即将进入运行态时，Linux内核都要用该进程的PGD指针设 置CR3（switch\_mm()）。  
  
（3）当创建一个新的进程时，都要为新进程创建一个新的页面目录PGD，并从内核的页面目录swapper\_pg\_dir中复制内核区间页面目录项至新建进程页面目录PGD的相应位置，具体过程如下：  
do\_fork() --> copy\_mm() --> mm\_init() --> pgd\_alloc() --> set\_pgd\_fast() --> get\_pgd\_slow() --> memcpy(&PGD + USER\_PTRS\_PER\_PGD, swapper\_pg\_dir + USER\_PTRS\_PER\_PGD, (PTRS\_PER\_PGD - USER\_PTRS\_PER\_PGD) \* sizeof(pgd\_t))  
这样一来，每个进程的页面目录就分成了两部分，第一部分为“用户空间”，用来映射其整个进程空间（0x0000 0000－0xBFFF FFFF）即3G字节的虚拟地址；第二部分为“系统空间”，用来映射（0xC000 0000－0xFFFF FFFF）1G字节的虚拟地址。可以看出Linux系统中每个进程的页面目录的第二部分是相同的，所以从进程的角度来看，每个进程有4G字节的虚拟空间， 较低的3G字节是自己的用户空间，最高的1G字节则为与所有进程以及内核共享的系统空间。  
  
（4）现在假设我们有如下一个情景：  
在进程A中通过系统调用sethostname(const char \*name,seze\_t len)设置计算机在网络中的“主机名”.  
在该情景中我们势必涉及到从用户空间向内核空间传递数据的问题，name是用户空间中的地址，它要通过系统调用设置到内核中的某个地址中。让我们看看这个 过程中的一些细节问题：系统调用的具体实现是将系统调用的参数依次存入寄存器ebx,ecx,edx,esi,edi（最多5个参数，该情景有两个 name和len），接着将系统调用号存入寄存器eax，然后通过中断指令“int 80”使进程A进入系统空间。由于进程的CPU运行级别小于等于为系统调用设置的陷阱门的准入级别3，所以可以畅通无阻的进入系统空间去执行为int 80设置的函数指针system\_call()。由于system\_call()属于内核空间，其运行级别DPL为0，CPU要将堆栈切换到内核堆栈，即 进程A的系统空间堆栈。我们知道内核为新建进程创建task\_struct结构时，共分配了两个连续的页面，即8K的大小，并将底部约1k的大小用于 task\_struct（如#define alloc\_task\_struct() ((struct task\_struct \*) \_\_get\_free\_pages(GFP\_KERNEL,1))）,而其余部分内存用于系统空间的堆栈空间，即当从用户空间转入系统空间时，堆栈指针 esp变成了（alloc\_task\_struct()+8192），这也是为什么系统空间通常用宏定义current（参看其实现）获取当前进程的 task\_struct地址的原因。每次在进程从用户空间进入系统空间之初，系统堆栈就已经被依次压入用户堆栈SS、用户堆栈指针ESP、EFLAGS、 用户空间CS、EIP，接着system\_call()将eax压入，再接着调用SAVE\_ALL依次压入ES、DS、EAX、EBP、EDI、ESI、 EDX、ECX、EBX，然后调用sys\_call\_table+4\*%EAX，本情景为sys\_sethostname()。  
  
（5）在sys\_sethostname()中，经过一些保护考虑后，调用copy\_from\_user(to,from,n），其中to指向内核空间 system\_utsname.nodename，譬如0xE625A000，from指向用户空间譬如0x8010FE00。现在进程A进入了内核，在 系统空间中运行，MMU根据其PGD将虚拟地址完成到物理地址的映射，最终完成从用户空间到系统空间数据的复制。准备复制之前内核先要确定用户空间地址和 长度的合法性，至于从该用户空间地址开始的某个长度的整个区间是否已经映射并不去检查，如果区间内某个地址未映射或读写权限等问题出现时，则视为坏地址， 就产生一个页面异常，让页面异常服务程序处理。过程如 下：copy\_from\_user()->generic\_copy\_from\_user()->access\_ok()+\_\_copy\_user\_zeroing().  
  
（6）小结：  
\*进程寻址空间0~4G    
\*进程在用户态只能访问0~3G，只有进入内核态才能访问3G~4G    
\*进程通过系统调用进入内核态  
\*每个进程虚拟空间的3G~4G部分是相同的    
\*进程从用户态进入内核态不会引起CR3的改变但会引起堆栈的改变  
  
Linux 简化了分段机制，使得虚拟地址与线性地址总是一致，因此，Linux的虚拟地址空间也为0～4G。Linux内核将这4G字节的空间分为两部分。将最高的 1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为“内核空间”。而将较低的3G字节（从虚拟地址 0x00000000到0xBFFFFFFF），供各个进程使用，称为“用户空间）。因为每个进程可以通过系统调用进入内核，因此，Linux内核由系统 内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有4G字节的虚拟空间。  
    Linux使用两级保护机制：0级供内核使用，3级供用户程序使用。从图中可以看出（这里无法表示图），每个进程有各自的私有用户空间（0～3G），这个空间对系统中的其他进程是不可见的。最高的1GB字节虚拟内核空间则为所有进程以及内核所共享。  
1．虚拟内核空间到物理空间的映射  
  内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。读者会问，系 统启动时，内核的代码和数据不是被装入到物理内存吗？它们为什么也处于虚拟内存中呢？这和编译程序有关，后面我们通过具体讨论就会明白这一点。  
虽 然内核空间占据了每个虚拟空间中的最高1GB字节，但映射到物理内存却总是从最低地址（0x00000000）开始。对内核空间来说，其地址映射是很简单 的线性映射，0xC0000000就是物理地址与线性地址之间的位移量，在Linux代码中就叫做PAGE\_OFFSET。  
  
  
我们来看一下在include/asm/i386/page.h中对内核空间中地址映射的说明及定义：  
/\*  
\* This handles the memory map.. We could make this a config  
\* option, but too many people screw it up, and too few need  
\* it.  
\*  
\* A \_\_PAGE\_OFFSET of 0xC0000000 means that the kernel has  
\* a virtual address space of one gigabyte, which limits the  
\* amount of physical memory you can use to about 950MB.   
\*  
\* If you want more physical memory than this then see the CONFIG\_HIGHMEM4G  
\* and CONFIG\_HIGHMEM64G options in the kernel configuration.  
\*/  
  
#define \_\_PAGE\_OFFSET           (0xC0000000)  
……  
#define PAGE\_OFFSET             ((unsigned long)\_\_PAGE\_OFFSET)  
#define \_\_pa(x)                 ((unsigned long)(x)-PAGE\_OFFSET)  
#define \_\_va(x)                 ((void \*)((unsigned long)(x)+PAGE\_OFFSET))  
源 代码的注释中说明，如果你的物理内存大于950MB，那么在编译内核时就需要加CONFIG\_HIGHMEM4G和CONFIG\_HIGHMEM64G选 项，这种情况我们暂不考虑。如果物理内存小于950MB，则对于内核空间而言，给定一个虚地址x，其物理地址为“x- PAGE\_OFFSET”，给定一个物理地址x，其虚地址为“x+ PAGE\_OFFSET”。  
这里再次说明，宏\_\_pa()仅仅把一个内核空间的虚地址映射到物理地址，而决不适用于用户空间，用户空间的地址映射要复杂得多。  
2．内核映像  
  在下面的描述中，我们把内核的代码和数据就叫内核映像（kernel image）。当系统启动时，Linux内核映像被安装在物理地址0x00100000开始的地方，即1MB开始的区间(第1M留作它用)。然而，在正常 运行时， 整个内核映像应该在虚拟内核空间中，因此，连接程序在连接内核映像时，在所有的符号地址上加一个偏移量PAGE\_OFFSET，这样，内核映像在内核空间 的起始地址就为0xC0100000。  
例如，进程的页目录PGD（属于内核数据结构）就处于内核空间中。在进程切换时，要将寄存器CR3设置成指 向新进程的页目录PGD，而该目录的起始地址在内核空间中是虚地址，但CR3所需要的是物理地址，这时候就要用\_\_pa()进行地址转换。在 mm\_context.h中就有这么一行语句：  
asm volatile(“movl %0,%%cr3”: :”r” (\_\_pa(next-&gt;pgd));  
这是一行嵌入式汇编代码，其含义是将下一个进程的页目录起始地址next\_pgd，通过\_\_pa()转换成物理地址，存放在某个寄存器中，然后用mov指令将其写入CR3寄存器中。经过这行语句的处理，CR3就指向新进程next的页目录表PGD了。

1、在elf格式的可执行代码中,ld总是从0x08000000开始安排代码段,对每个程序都这样.