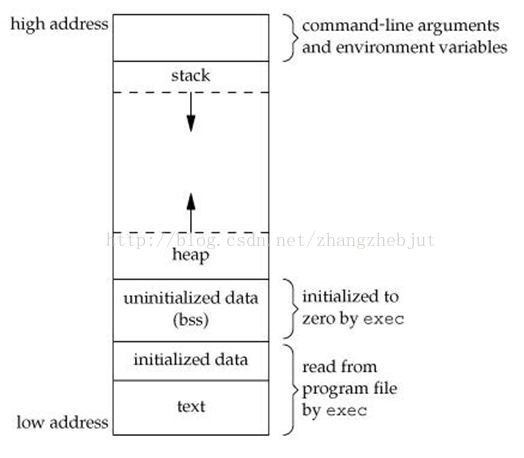
一 进程空间分布概述

    对于一个进程，其空间分布如下图所示：



**程序段(Text):**程序代码在内存中的映射，存放函数体的二进制代码。

**初始化过的数据(Data):**在程序运行初已经对变量进行初始化的数据。

**未初始化过的数据(BSS):**在程序运行初未对变量进行初始化的数据。

**栈 (Stack):**存储局部、临时变量，函数调用时，存储函数的返回指针，用于控制函数的调用和返回。在程序块开始时自动分配内存,结束时自动释放内存，其操作方式类似于[**数据结构**](http://lib.csdn.net/base/datastructure)中的栈。

**堆 (Heap):**存储动态内存分配,需要程序员手工分配,手工释放.注意它与数据结构中的堆是两回事，分配方式类似于链表。

**注：**1.Text, BSS, Data段在编译时已经决定了进程将占用多少VM

        可以通过size，知道这些信息：

          2. 正常情况下，Linux进程不能对用来存放程序代码的内存区域执行写操作，即程序代码是以只读的方式加载到内存中，但它可以被多个进程安全的共享。

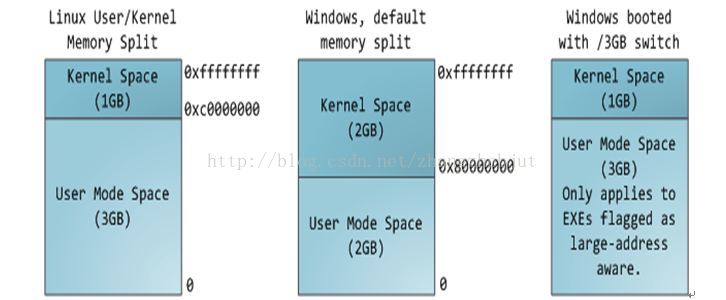
**二  内核空间和用户空间**

       Linux的虚拟地址空间范围为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，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。  
    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。 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），另外，使用虚拟地址可以很好的保护内核空间被用户空间破坏，虚拟地址到物理地址转换过程有操作系统和CPU共同完成(操作系统为CPU设置好页表，CPU通过MMU单元进行地址转换)。

**注**：多任务操作系统中的每一个进程都运行在一个属于它自己的内存沙盒中，这个沙盒就是虚拟地址空间（virtual address space），在32位模式下，它总是一个4GB的内存地址块。这些虚拟地址通过页表（page table）映射到物理内存，页表由操作系统维护并被处理器引用。每个进程都拥有一套属于它自己的页表。

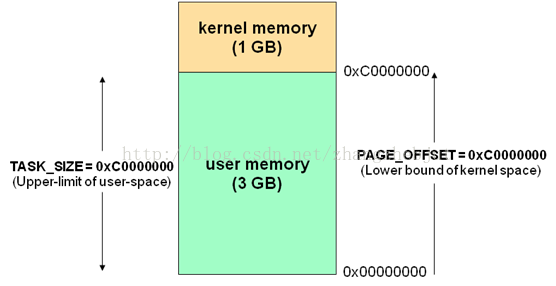
  进程内存空间分布如下图所示：



     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

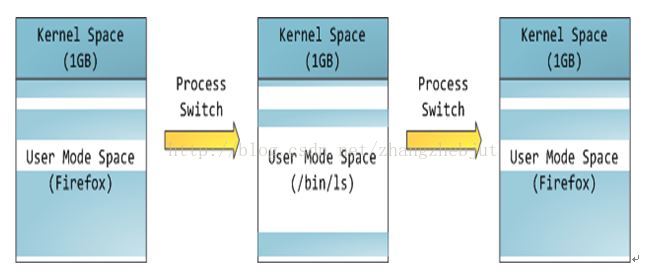
**注**: 1.这里是32位内核地址空间划分，64位内核地址空间划分是不同的

          2.现代的操作系统都处于32位保护模式下。每个进程一般都能寻址4G的物理空间。但是我们的物理内存一般都是几百M，进程怎么能获得4G 的物理空间呢？这就是使用了虚拟地址的好处，通常我们使用一种叫做虚拟内存的技术来实现，因为可以使用硬盘中的一部分来当作内存使用 。



        Linux系统对自身进行了划分，一部分核心软件独立于普通应用程序，运行在较高的特权级别上，它们驻留在被保护的内存空间上，拥有访问硬件设备的所有权限，Linux将此称为内核空间。  
        相对地，应用程序则是在“用户空间”中运行。运行在用户空间的应用程序只能看到允许它们使用的部分系统资源，并且不能使用某些特定的系统功能，也不能直接访问内核空间和硬件设备，以及其他一些具体的使用限制。  
        将用户空间和内核空间置于这种非对称访问机制下有很好的安全性，能有效抵御恶意用户的窥探，也能防止质量低劣的用户程序的侵害，从而使系统运行得更稳定可靠。

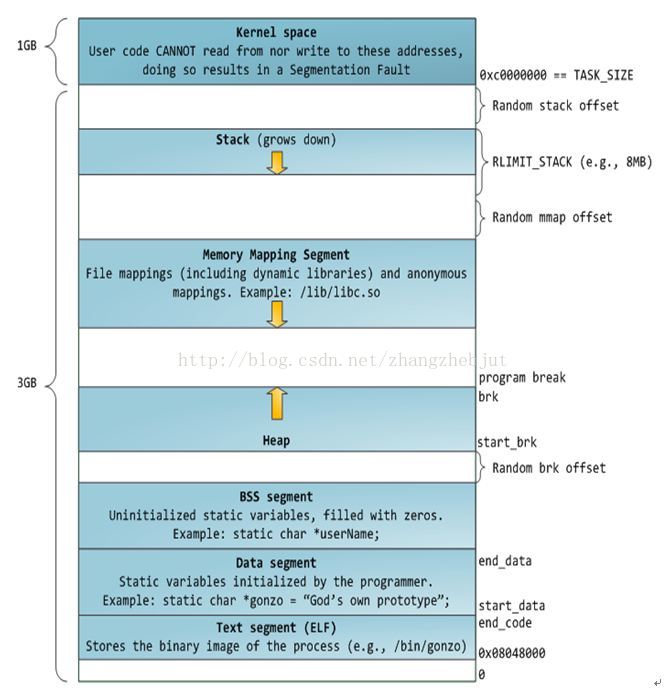
            内核空间在页表中拥有较高的特权级（ring2或以下），因此只要用户态的程序试图访问这些页，就会导致一个页错误（page fault）。在Linux中，内核空间是持续存在的，并且在所有进程中都映射到同样的物理内存，内核代码和数据总是可寻址的，随时准备处理中断和系统调用。与之相反，用户模式地址空间的映射随着进程切换的发生而不断的变化，如下图所示：

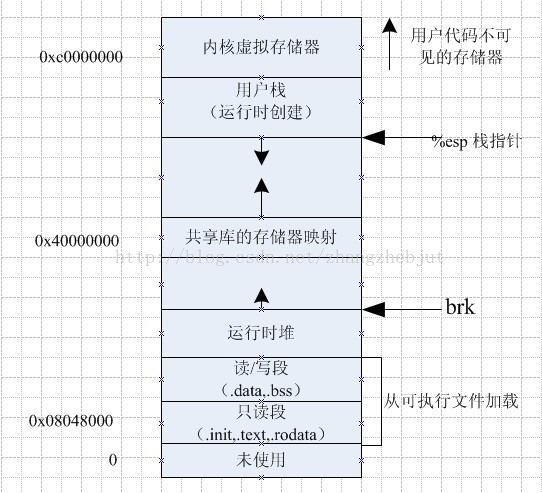


      上图中蓝色区域表示映射到物理内存的虚拟地址，而白色区域表示未映射的部分。可以看出，Firefox使用了相当多的虚拟地址空间，因为它占用内存较多。

**三  进程内存布局**

       Linux进程标准的内存段布局，如下图所示，地址空间中的各个条带对应于不同的内存段（memory segment），如：堆、栈之类的。



                                                  q

**注**：这些段只是简单的虚拟内存地址空间范围，与Intel处理器的段没有任何关系。

      几乎每个进程的虚拟地址空间中各段的分布都与上图完全一致，这就给远程发掘程序漏洞的人打开了方便之门。一个发掘过程往往需要引用绝对内存地址：栈地址，库函数地址等。远程攻击者必须依赖地址空间分布的一致性，来探索出这些地址。如果让他们猜个正着，那么有人就会被整了。因此，地址空间的随机排布方式便逐渐流行起来，Linux通过对栈、内存映射段、堆的起始地址加上随机的偏移量来打乱布局。但不幸的是，32位地址空间相当紧凑，这给随机化所留下的空间不大，削弱了这种技巧的效果。

**栈**

     进程地址空间中最顶部的段是栈，大多数编程语言将之用于存储函数参数和局部变量。调用一个方法或函数会将一个新的栈帧（stack frame）压入到栈中，这个栈帧会在函数返回时被清理掉。由于栈中数据严格的遵守FIFO的顺序，这个简单的设计意味着不必使用复杂的数据结构来追踪栈中的内容，只需要一个简单的指针指向栈的顶端即可，因此压栈（pushing）和退栈（popping）过程非常迅速、准确。进程中的每一个线程都有属于自己的栈。

      通过不断向栈中压入数据，超出其容量就会耗尽栈所对应的内存区域，这将触发一个页故障（page fault），而被**[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "_blank)**的expand\_stack()处理，它会调用acct\_stack\_growth()来检查是否还有合适的地方用于栈的增长。如果栈的大小低于RLIMIT\_STACK（通常为8MB），那么一般情况下栈会被加长，程序继续执行，感觉不到发生了什么事情。这是一种将栈扩展到所需大小的常规机制。然而，如果达到了最大栈空间的大小，就会栈溢出（stack overflow），程序收到一个段错误（segmentation fault）。

**注**:动态栈增长是唯一一种访问未映射内存区域而被允许的情形，其他任何对未映射内存区域的访问都会触发页错误，从而导致段错误。一些被映射的区域是只读的，因此企图写这些区域也会导致段错误。

**内存映射段**

     在栈的下方是内存映射段，内核将文件的内容直接映射到内存。任何应用程序都可以通过Linux的mmap()系统调用或者Windows的CreateFileMapping()/MapViewOfFile()请求这种映射。内存映射是一种方便高效的文件I/O方式，所以它被用来加载动态库。创建一个不对应于任何文件的匿名内存映射也是可能的，此方法用于存放程序的数据。在Linux中，如果你通过malloc()请求一大块内存，C运行库将会创建这样一个匿名映射而不是使用堆内存。“大块”意味着比MMAP\_THRESHOLD还大，缺省128KB，可以通过mallocp()调整。

**堆**

      与栈一样，堆用于运行时内存分配；但不同的是，堆用于存储那些生存期与函数调用无关的数据。大部分语言都提供了堆管理功能。在C语言中，堆分配的接口是malloc()函数。如果堆中有足够的空间来满足内存请求，它就可以被语言运行时库处理而不需要内核参与，否则，堆会被扩大，通过brk()系统调用来分配请求所需的内存块。堆管理是很复杂的，需要精细的算法来应付我们程序中杂乱的分配模式，优化速度和内存使用效率。处理一个堆请求所需的时间会大幅度的变动。实时系统通过特殊目的分配器来解决这个问题。堆在分配过程中可能会变得零零碎碎，如下图所示：



       一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。注意它与数据结构中的堆是两回事，分配方式类似于链表。

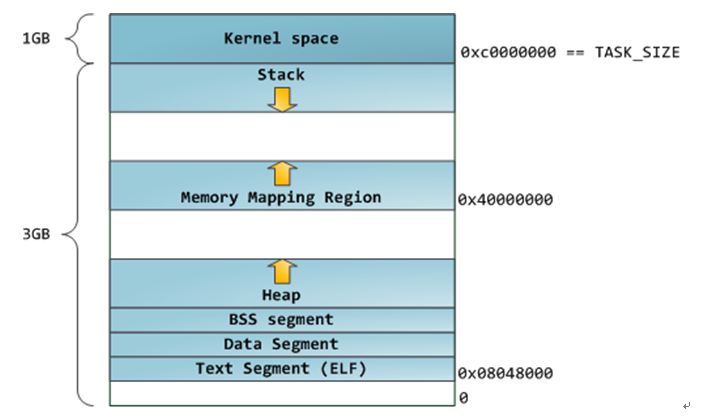
**BBS和数据段**

      在C语言中，BSS和数据段保存的都是静态（全局）变量的内容。区别在于BSS保存的是未被初始化的静态变量内容，他们的值不是直接在程序的源码中设定的。BSS内存区域是匿名的，它不映射到任何文件。如果你写static intcntActiveUsers，则cntActiveUsers的内容就会保存到BSS中去。

**数据段保存在源代码中已经初始化的静态变量的内容。数据段不是匿名的，它映射了一部分的程序二进制镜像，也就是源代码中指定了初始值的静态变量。**所以，如果你写static int cntActiveUsers=10，则cntActiveUsers的内容就保存在了数据段中，而且初始值是10。尽管数据段映射了一个文件，但它是一个私有内存映射，这意味着更改此处的内存不会影响被映射的文件。

      你可以通过阅读文件/proc/pid\_of\_process/maps来检验一个**[linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "_blank)**进程中的内存区域。记住：一个段可能包含许多区域。比如，每个内存映射文件在mmap段中都有属于自己的区域，动态库拥有类似BSS和数据段的额外区域。有时人们提到“数据段”，指的是全部的数据段+BSS+堆。

     你还可以通过nm和objdump命令来察看二进制镜像，打印其中的符号，它们的地址，段等信息。最后需要指出的是，前文描述的虚拟地址布局在linux中是一种“灵活布局”，而且作为默认方式已经有些年头了，它假设我们有值RLIMT\_STACK。但是，当没有该值得限制时，Linux退回到“经典布局”，如下图所示：



[**C语言**](http://lib.csdn.net/base/c)程序实例分析如下所示：

**[cpp]** [view plain](http://blog.csdn.net/yusiguyuan/article/details/45155035) [copy](http://blog.csdn.net/yusiguyuan/article/details/45155035)

1. #include<stdio.h>
2. #include <malloc.h>
4. **void** print(**char** \*,**int**);
5. **int** main()
6. {
7. **char** \*s1 = "abcde";  //"abcde"作为字符串常量存储在常量区 s1、s2、s5拥有相同的地址
8. **char** \*s2 = "abcde";
9. **char** s3[] = "abcd";
10. **long** **int** \*s4[100];
11. **char** \*s5 = "abcde";
12. **int** a = 5;
13. **int** b =6;//a,b在栈上，&a>&b地址反向增长
15. printf("variables address in main function: s1=%p  s2=%p s3=%p s4=%p s5=%p a=%p b=%p \n",
16. s1,s2,s3,s4,s5,&a,&b);
17. printf("variables address in processcall:n");
18. print("ddddddddd",5);//参数入栈从右至左进行,p先进栈,str后进 &p>&str
19. printf("main=%p print=%p \n",main,print);
20. //打印代码段中主函数和子函数的地址，编译时先编译的地址低，后编译的地址高main<print
21. }
23. **void** print(**char** \*str,**int** p)
24. {
25. **char** \*s1 = "abcde";  //abcde在常量区，s1在栈上
26. **char** \*s2 = "abcde";  //abcde在常量区，s2在栈上 s2-s1=6可能等于0，编译器优化了相同的常量，只在内存保存一份
27. //而&s1>&s2
28. **char** s3[] = "abcdeee";//abcdeee在常量区，s3在栈上，数组保存的内容为abcdeee的一份拷贝
29. **long** **int** \*s4[100];
30. **char** \*s5 = "abcde";
31. **int** a = 5;
32. **int** b =6;
33. **int** c;
34. **int** d;           //a,b,c,d均在栈上，&a>&b>&c>&d地址反向增长
35. **char** \*q=str;
36. **int** m=p;
37. **char** \*r=(**char** \*)malloc(1);
38. **char** \*w=(**char** \*)malloc(1) ;  // r<w 堆正向增长
40. printf("s1=%p s2=%p s3=%p s4=%p s5=%p a=%p b=%p c=%p d=%p str=%p q=%p p=%p m=%p r=%p w=%p \n",
41. s1,s2,s3,s4,s5,&a,&b,&c,&d,&str,q,&p,&m,r,w);
42. /\* 栈和堆是在程序运行时候动态分配的，局部变量均在栈上分配。
43. 栈是反向增长的，地址递减；malloc等分配的内存空间在堆空间。堆是正向增长的，地址递增。
44. r,w变量在栈上(则&r>&w)，r,w所指内容在堆中(即r<w)。\*/
45. }

**附录：**

栈与堆的区别



转载：http://blog.csdn.net/zhangzhebjut/article/details/39060253