【译】内核是如何管理内存的[1]

秦新良

June 15, 2013

[1]Gustavo Duarte的原文在这里。

上篇文章中我们讲了进程的虚拟内存布局，在这篇文章中我们将关注内核管理用户空间内存的机制。这里还是以程序 gonzo 举例说明，如图1所示。

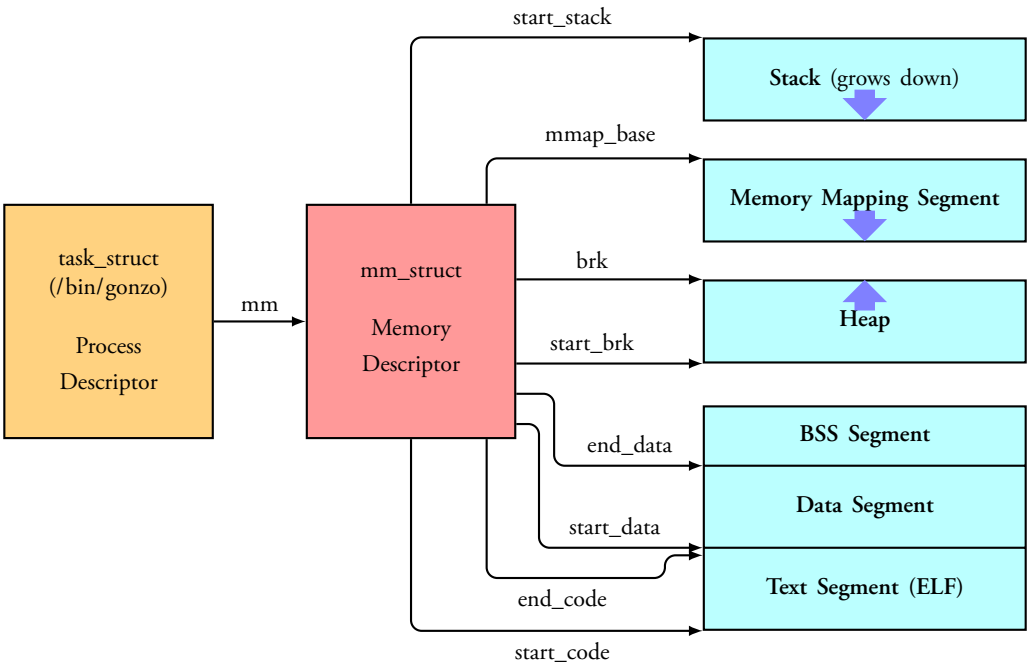

图 1: gonzo

Linux 中的进程是进程描述符task_struct的一个实例。task_struct 的成员mm指向内存描述符mm_struct，mm_struct 是进程内存的总述。内存描述符中有各个内存段的起始和结束地址，当前进程使用的物理内存页个数（**rss** 是 Resident Set Size 的缩写），以及当前进程使用的虚拟地址总数等。在内存描述符中有两个管理内存的驱动：虚拟内存区集合和页表。进程 gonzo 的内存区如图2所示。

每一个虚拟内存区（VMA）都是一段相互之间不会重叠的连续的虚拟内存地址。结构体vm_area_struct的一个实例完全描述一段虚拟内存区，包括该虚拟内存区的起始和结束地址，决定该内存区访问权限的标识，如果有文件映射的话，成员vm_file标识该区域映射的是哪个文件等。没有映射文件的虚拟内存区是匿名的。图2中除内存映射段外的每一个段（如堆、栈）都对应一个虚拟内存区。这样的对应关系并不是必须的，但在 X86 的机子上通常都是这样对应的。虚拟内存区并不关心它到底属于哪个段。

一个进程的虚拟内存区是通过内存描述符中的成员mmap和mm_rb来管理的。mmap 是以虚拟内存区的起始地址排序的一个链表，mm_rb 是以 mm_rb 为根的红黑树。给定一个虚拟地址，通过红黑树来搜索会更快些。当我们读文件/proc/pid_of_process/maps 时，内核只是遍历虚拟内存区链表，并打印每段虚拟内存区信息。

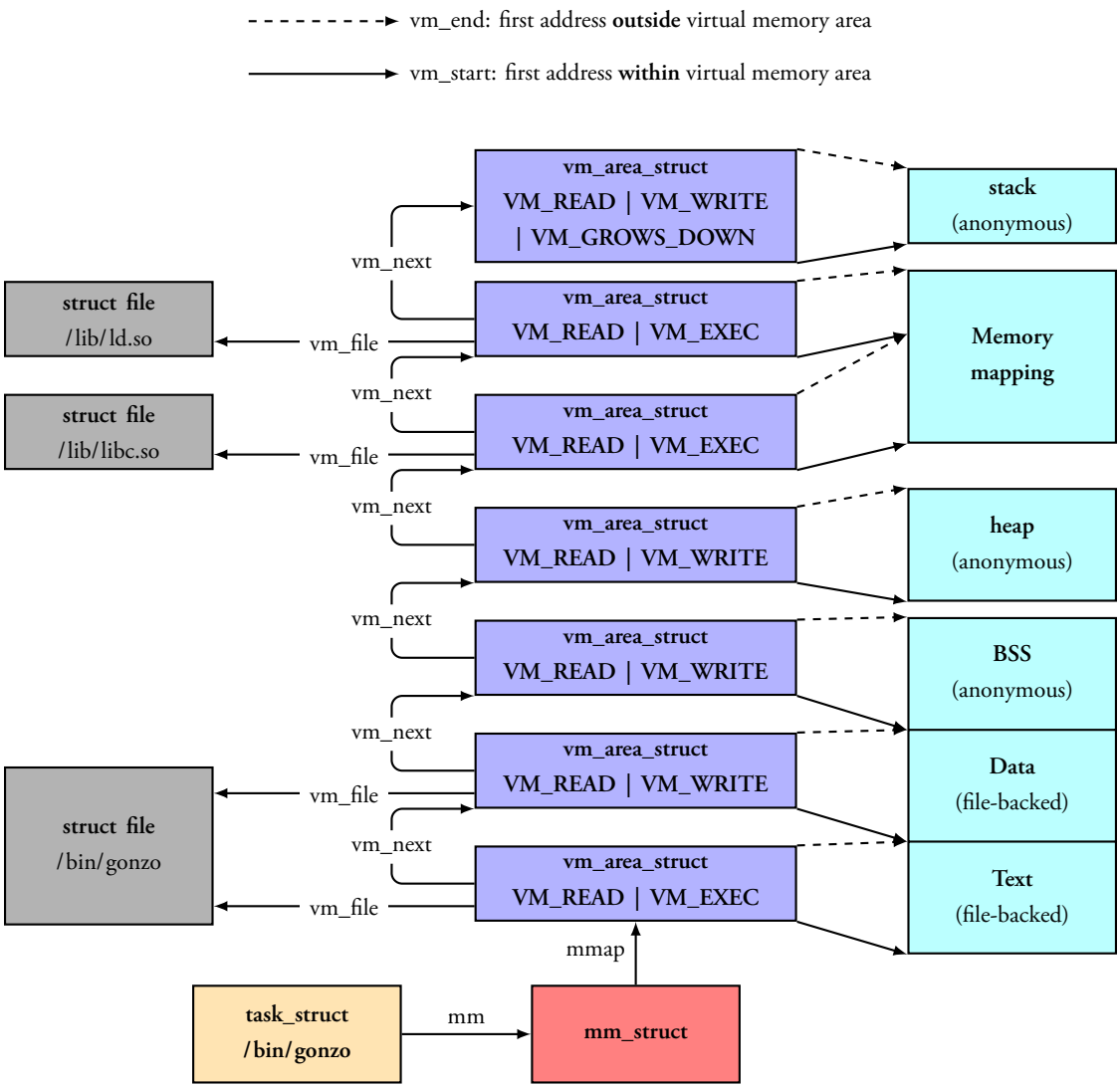

图 2: gonzo的内存区

在Windows中，每一个进程块总的来说就是task_struct和mm_struct的组合。Windows中和Linux的虚拟内存区功能相近的就是虚拟地址描述符（VAD），它们是以AVL树来组织管理的。你知道Linux和Windows之间的这些有趣的事情吗？就是这些细小的差别。

4GB的虚拟地址空间被分成了多个页。X86的处理器在32位模式下支持的页大小有4KB，2M和4M。Linux和Windows在映射用户空间的虚拟内存时使用的都是4KB大小的页。0-4095字节落在第0页，4096-8191落在第1页，以此类推。虚拟内存区的大小一定是页大小的整数倍。图3是3GB的用户空间以4KB大小划分页的结果。

处理器通过页表将虚拟地址转换成物理地址。每一个进程都有自己的页表，当进程切换时，用户空间的页表也会随着切换。Linux将进程的页表指针存放在进程描述符的成员pgd中。每一个虚拟页在页表中都对应一个页表项（PTE），在X86平台中是4字节的记录，如图4所示。

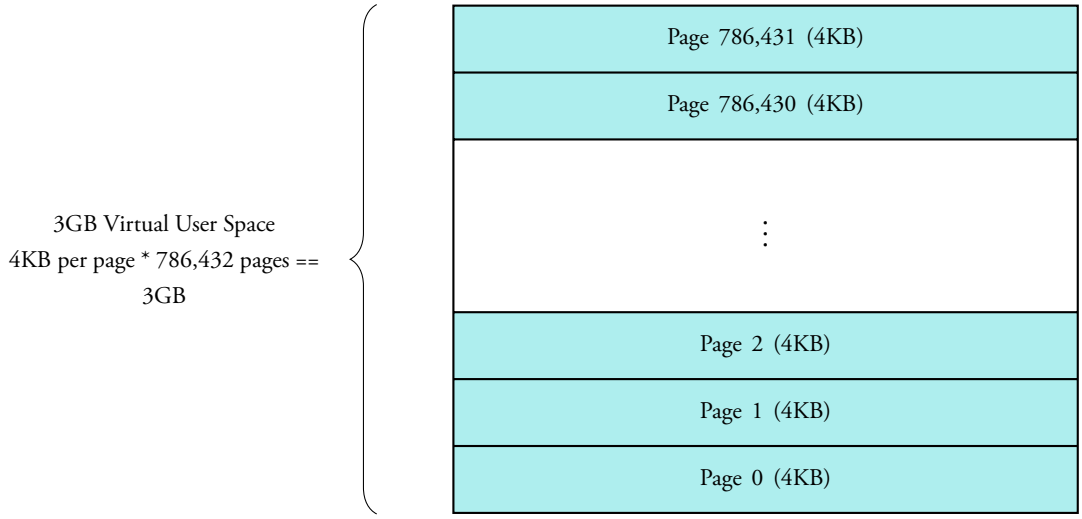

图 3: 用户空间页划分

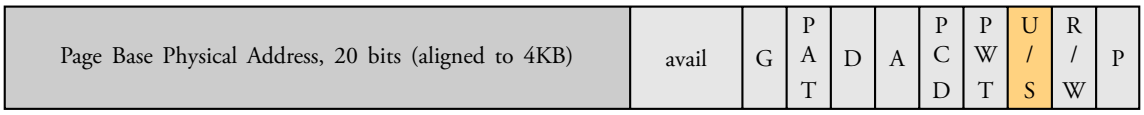

图 4: 页表项

Linux 中可通过函数来读写页表项里的标志位。位 P 标识该页是否在物理内存中。如果 P 位是清除的（等于 0），访问该页会触发缺页故障。这里强调一下，如果位 P 是置 0 状态，内核可以对剩余的位做任意的操作。R/W 标识读写状态，0 表示只读。U/S 标识用户权限，0 表示只有内核才有权限访问。这些标识位用来限制内存只读，以此保护内核空间。

D 和 A 表示该页是否被写或访问过。两个标识有一个共性就是：都被处理器访问，但必须被内核清除。最后，页表项保存了该页的起始物理地址，该地址是 4KB 对齐的。这样的设计也是一些痛苦之源，如限制了最大只能寻址 4GB 的物理内存。剩余还有一些标识这里暂时不作说明，后面还会继续深入讨论。

页是内存保护机制的最小单位，因为该页的所有地址共用标识 U/S 和 R/W。但是相同的物理内存有可能被映射到不同的页，甚至可能是不同的保护机制。需注意的一点是，可执行权限并没有在页表项中体现。这也是经典的 X86 分页允许代码在栈上执行的原因，这也使得利用栈溢出攻击更加容易（利用其它方法来攻击不可执行的栈也是有可能的，如return-to-libc方法）。页表项没有不可执行的标志位说明了一个事实：虚拟内存区的权限标志可能也可能不会直接转化为硬件的保护机制。内核做自己力所能及的事，但最终是硬件决定内核能做什么。

虚拟内存不存储任何东西，它只是简单地将程序的内存空间映射到底层的物理内存，物理内存被处理器成块儿访问，成块儿的内存被称为物理内存空间。这里先不谈内存在总线上是怎么

操作的，假设内存从0开始，以一字节递增，一直到可用内存的最大值。物理地址空间被内核分成页帧。处理器并不关心页帧，但页帧对内核至关重要，因为页帧是内核管理物理内存的最小单位。在32位模式下，Windows和Linux都使用4KB大小的页帧。图5是一个2G物理内存的例子。

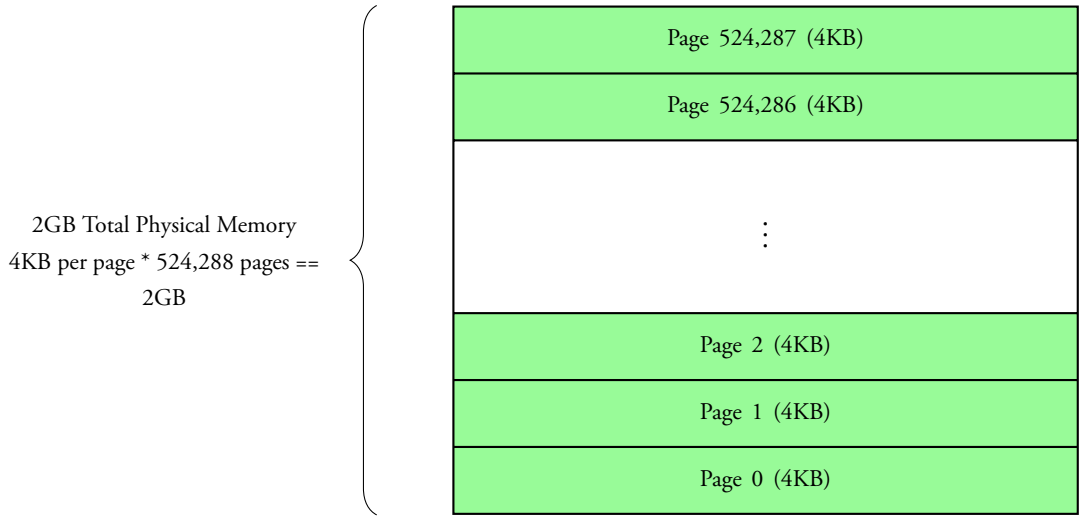

图 5: 用户空间

在Linux中，每一个页帧通过一个描述符和几个标志位来跟踪。所有的的描述符加起来就可以跟踪电脑的所有物理内存，每个页帧的当前状态都是可知的。物理内存通过伙伴内存分配技术来管理，因此一个页帧只要可通过伙伴系统分配就是空闲的。已分配的页帧可能是匿名的，存储程序数据；也可能在页缓存中，存储文件或块设备的数据。页帧还有其它用途，这里暂且不谈。Windows也有一个类似的页帧数数据库来跟踪物理内存。

现在我们把虚拟内存区、页表项和页帧联系起来看一下它们是如何工作的。图6是堆内存映射的一个例子。

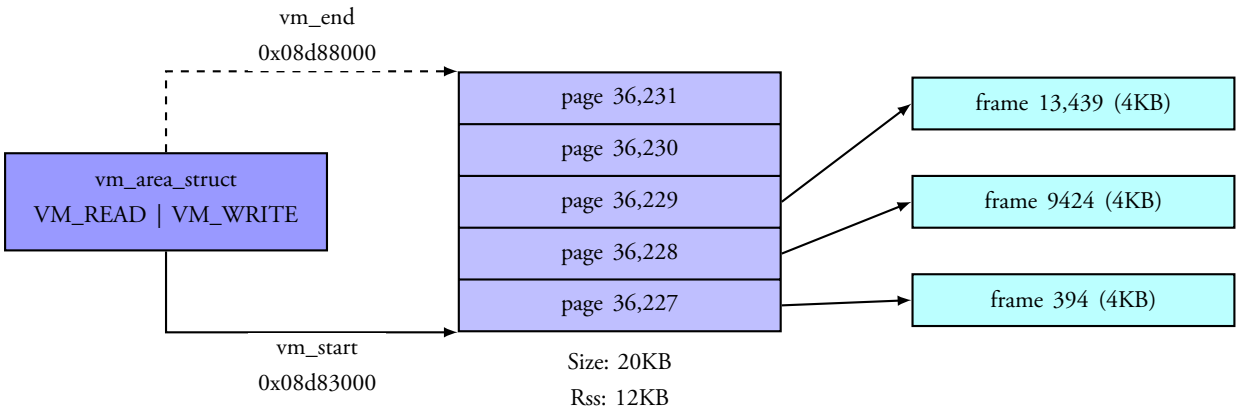

图 6: 堆内存映射

蓝色的长方形表示虚拟内存区中的页面，箭头表示通过页表项将页面映射到了页帧。有一

些虚拟页面没有箭头，意味着这些页面对应的页表项的标志位 P 是置 0 的。这可能是页面从来就没有被使用过，或者已经被交换出去了。不管是哪种情况，如果访问这些页面将会触发缺页中断，即使这个页面在这个虚拟内存区。虚拟内存区和页表会有不同，这看起来很奇怪，但却会经常发生。

虚拟内存区就像是进程和内核之间的一座桥梁。进程向内核发出请求（内存分配，文件映射等），内核说可以，然后就创建或更新了对应的虚拟内存区。但内核不是真正的响应了这次请求，而是要等到缺页中断时才真正映射内存。内核是有惰性的，这也是虚拟内存的基本原则。大多数情况都是这样的，有的熟悉，有的陌生，但总的规则就是虚拟内存区记录已经响应的操作，而页表项反映内核实际完成的操作。这两个数据结构结合起来管理程序的内存，都扮演着响应缺页中断、释放内存、将内存换出等请求。图7是一个内存分配的例子。

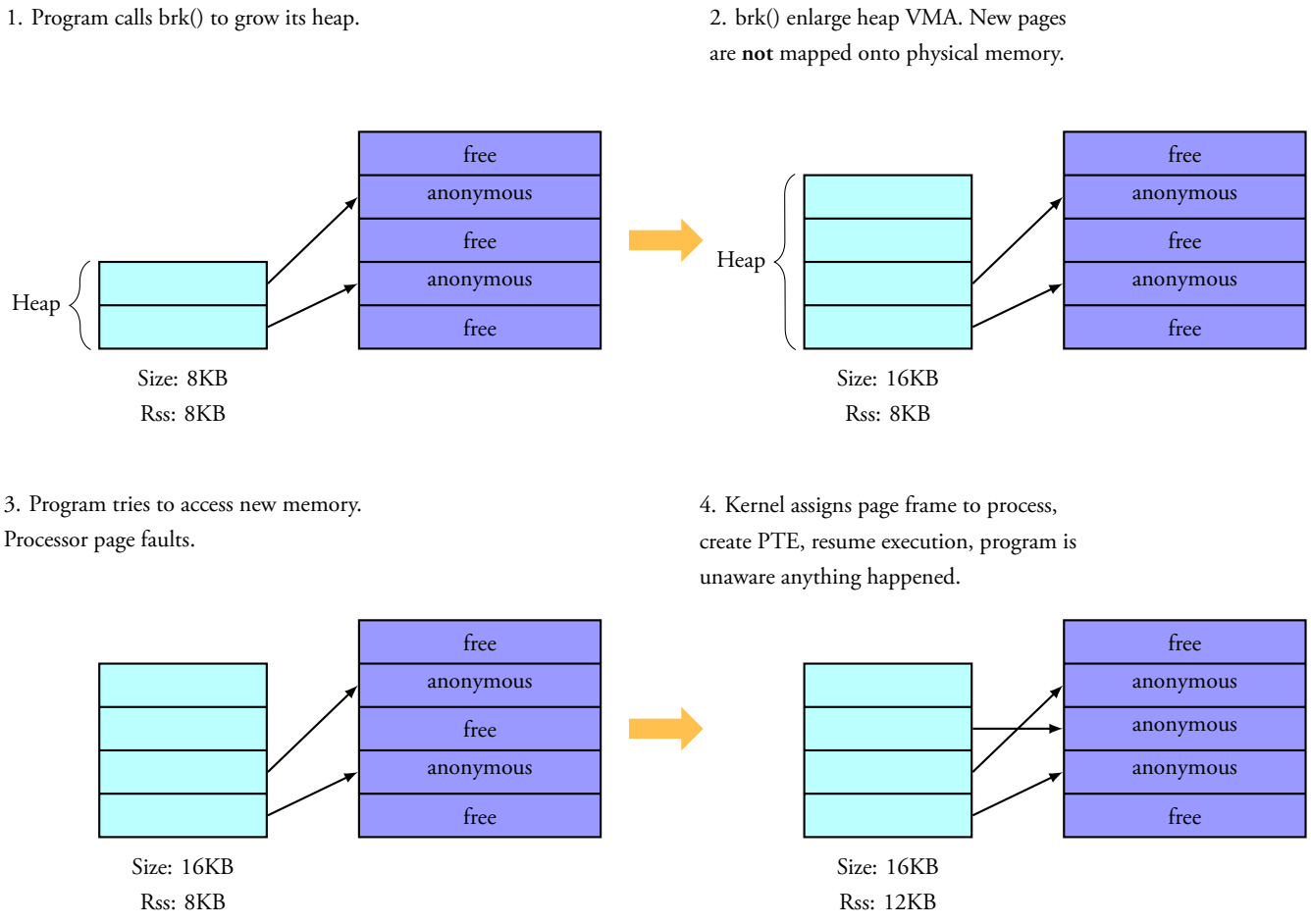

图 7: gonzo

当进程通过系统调用brk()申请更多的内存时，内核只是更新一下堆对应的虚拟内存区，这时内核并不会真正分配页帧，而且新的页面也不会在物理内存中。一旦页面被访问，处理器会产生缺页中断，然后do_page_fault()被调用。do_page_fault() 通过调用find_vma()搜索虚拟内存区来找到

引起缺页的虚拟内存地址。如果找到，内核会检测虚拟内存区的权限和本次访问的权限（读或写）。如果没有搜索到合适的虚拟内存区，内核将产生段错误。

当虚拟内存区找到时，内核必须通过页表项的内容和虚拟内存区的类型来处理中断。在我们的例子中，页面不存在。实际上，我们的页表项是完全空白的（全 0），在 Linux 中这意味着该页面从来没被映射过。因为这是匿名的虚拟内存区，所以本次操作是单纯的内存请求，这个请求必须通过do_anonymous_page()来处理。do_anonymous_page() 分配页帧和页表项，建立引起缺页中断的虚拟内存页面和刚分配的页帧的映射关系。

实际的情况可能会不同。例如，被交换出去的页面对应的页表项的标志位 P 为 0，但该页表项不为空。相反，页表项记录了存放页面内容的交换地址，页面的内容通过 do_swap_page() 从硬盘读取后加载到页帧。

这就是内核管理用户空间内存的前半部份。在下一篇博文中，我们将加入对文件的阐述来对内存作一个完整的分析，包括一些性能指标。