【译】进程内存布局剖析[1]

秦新良

June 7, 2013

[1]Gustavo Duarte的原文在这里。

内存管理是操作系统的核心，对编程和系统管理都至关重要。在接下来的几篇博文中，我将从实际使用的角度来阐述内存，当然也会对其内部原理作一些介绍。内存管理在不同平台和操作系统下都很类似，这里主要以32位x86平台下的Linux和Windows系统举例说明。这篇（也是第一篇）文章主要讲述程序在内存中的布局。

多任务操作系统中，每个进程都运行在自己的内存空间。这个空间就是虚拟地址空间，在32位模式下，是一段4GB的内存地址。虚拟地址由页表映射到物理内存，页表由操作系统内核维护，被处理器访问[1]。每个进程都有自己的页表集[2]。处理器的虚地址模式一旦被打开，对所有运行在其上的程序都生效，包括系统内核。因此部分虚拟地址空间必须为内核保留，如图1所示。

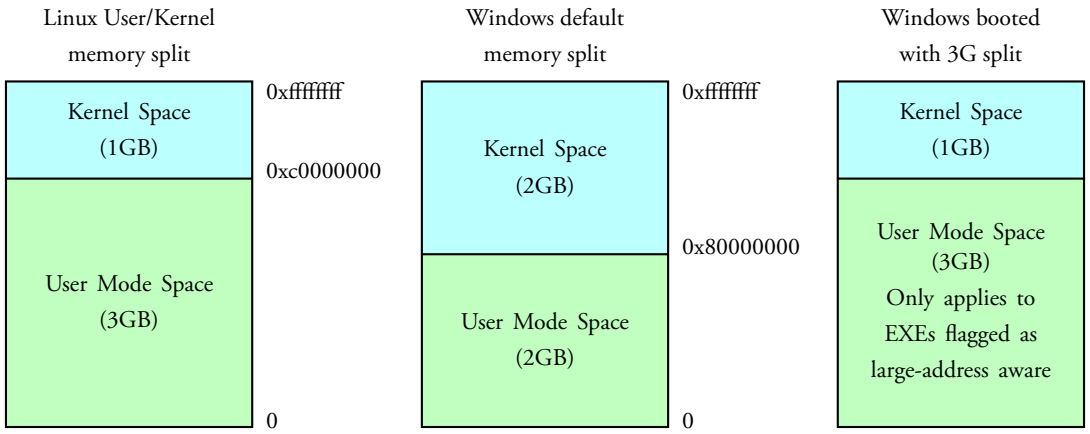

图 1: 内核、用户地址空间划分

但这并不是指内核会使用那么多[3]的物理内存，而是指当内核需要物理内存时，为内核保留的地址空间随时可用来作地址映射。在页表中，内核空间地址被标识为特权级代码（小于等于2[4]），用户态的程序如果读或写这些地址，将引起缺页故障。在Linux中，内核空间常驻在每个进程的地址空间中，并且被映射到相同的物理地址[5]。内核的代码段和数据段总是可寻址的，准备随时处理中断或系统调用。相反的，用户态进程的地址映射随着进程的切换而改变，如图2所示。

图2中绿色区域表示虚拟地址已被映射到物理地址，白色表示未映射。在图2的例子中，Firefox是出了名的吃货，其大部分地址空间都已使用。不同段的地址空间对应着不同的内存段，如有的在堆上、有的在栈上等等。记住，这里的段只是内存地址的范围，跟Intel处理器的段没有

[1]原文：These virtual addresses are mapped to physical memory by page tables, which are maintained by the operating system kernel and consulted by the processo.

[2]原文：Each process has its own set of page tables, but there is a catch.

[3]译者注：图1中的蓝色部分。

[4]译者注：x86的CPU分0–3三个特权级，其中0最高，3最低，Linux使用的是0级和3级。

[5]译者注：这样进程切换时，内核地址空间就不必切来切去。

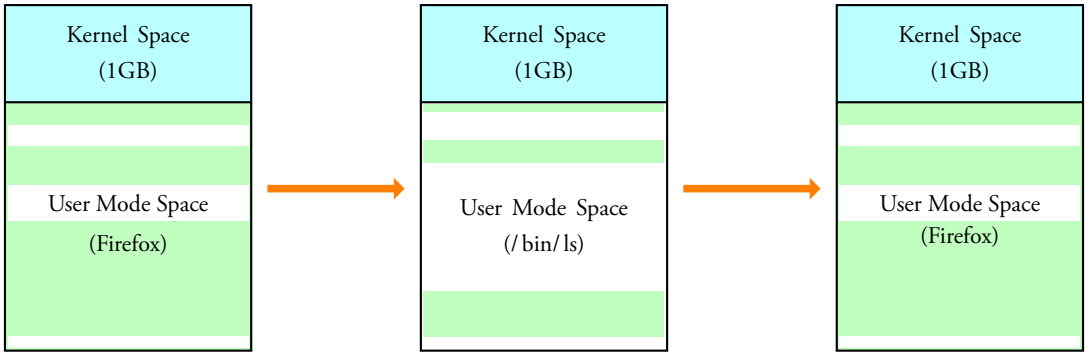

图 2: 进程切换

任何关系。图3是 Linux 中进程的各个内存段的布局。

当系统在欢快地运行着时，几乎所有运行在其上的进程的各个段的起时虚拟地址都相同。这样就很容易产生被远程利用的安全漏洞。攻击需要引用绝对的内存地址，如栈地址、某个库函数的地址等。远程攻击者相信所有的地址空间都是相同的，进而试着找到这样的内存地址。一旦他们得逞了，就会造成破坏。正因如此，随机的地址空间就变得流行起来了。Linux 通过将栈、内存映射段和堆的起始地址加一个偏移而使其随机化。但不幸的是，32 位的地址空间非常有限，随机范围小，影响随机的效果。

进程地址空间最上面的一个段是栈，大部分的编程语言都在其上存储局部变量和函数入参。每次函数调用都会有一个新的栈帧压栈，该栈帧随着函数的返回而销毁。这种简约的设计可能是因为所有的数据都遵从后进先出的原则，这也意味着不需要使用复杂的数据结构来跟踪栈的内容，只需要一个指向栈顶的指针就可以搞定。因此进栈、出栈那是相当的快。同时，持续的栈重用有利于将活动的栈内存保持在 CPU 的 cache 中，以加快数据的访问。进程中的每个线程都有自己独立的栈空间。

往栈里压的数据大于当前栈的内存映射大小时，会导致已映射的内存耗尽，进而触发缺页中断。Linux 通过调用expand_stack()处理缺页中断；expand_stack()调用acct_stack_growth()检测增加栈空间是否合适；如果当前栈的大小小于RLIMIT_STACK（通常是 8M），则栈会增长，程序继续执行。这种情况下，程序完全感知不到还发生过缺页这回事。这种是正常的机制，栈大小按需调整。但如果当前栈的大小达到了栈的最大值，则会栈溢出，程序会产生段错误。当映射后的栈空间按需扩展后，即使栈变小了，该空间也不会收缩。这就像联邦的预算，从来都是只增不减。

动态的栈增长是唯一的一种访问未映射内存区可能合法的情况。任何其它访问未映射内存区的方式都会导致段错误。有些映射的内存区是只读的，如果对这些区域进行写操作，也会导致段错误。

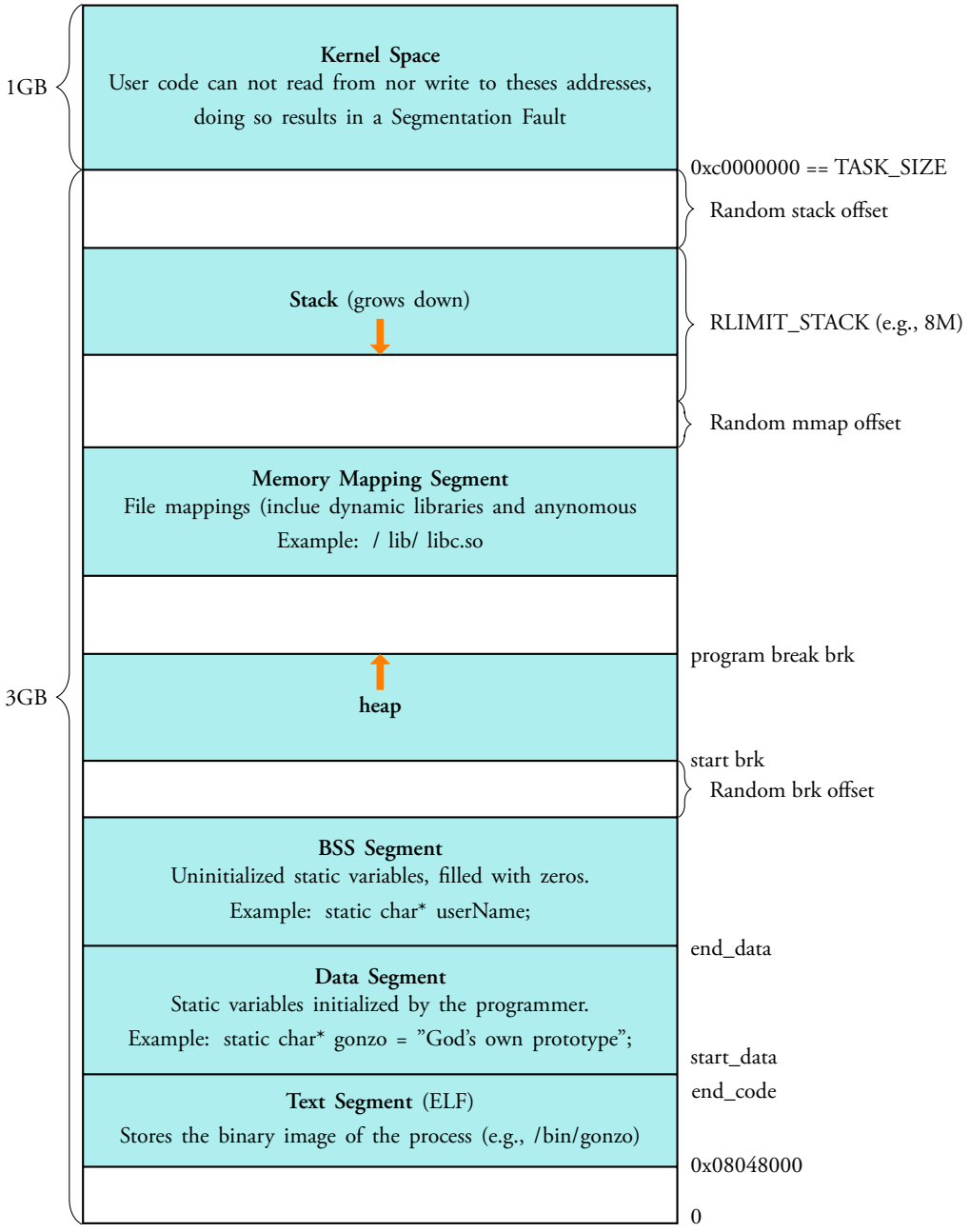

图 3: Linux 中进程的内存布局

紧接着栈下面的段是内存映射段。在该段，内核直接将文件映射到内存。在 Linux 中，任何程序都可通过mmap系统调用请求这样的内存映射；在 Windows 下有CreateFileMapping()或MapViewOfFile()。内存映射是一种方便和高效的文件 I/O 方式，所以也被用来加载动态库。创建不和任何文件对应的匿名内存映射也是可以的，这种方式映射的内存，用于程序的数据。在 Linux 中，如果你通过malloc()申请一大块内存，C 库会自动创建一块匿名内存映射，而不是使用堆上的内存。"大"意味着大于MMAP_THRESHOLD个字节，默认是 128kB，可通过mallopt()调整。

接下来，我们将讨论堆地址空间。堆和栈一样，提供运行时的内存分配，但跟栈不同的是，

堆上申请的内存的生存期可以大于申请其的函数。绝大多数的语言都支持堆内存管理。因此，满足语言运行时的内存请求就成了内核和应用程序的一个连接点。C语言中通过malloc()接口来申请堆内存，C# 中的接口是new关键字。

如果有足够的空间满足一次内存请求的话，这样的内存申请就不需要内核的参与。否则，堆就会通过系统调用brk()被扩展，以此为内存请求申请空间。堆内存的管理非常复杂，需要有复杂的算法来满足千奇百怪的内存使用场景，同时还得快速和高效。为一次内存请求提供服务所需的时间，不同系统可能会有天壤之别。实时的操作系统会有专门的分配器来处理这种问题。堆也会碎片化，如下图所示：

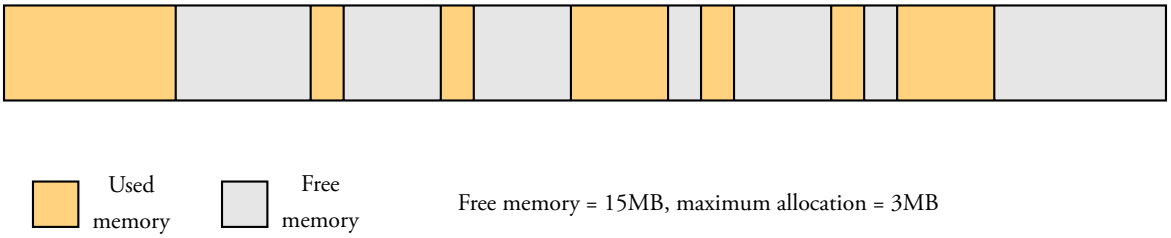

图 4: 内存碎片

最后，我们来了解下最下面的几个段：BSS，数据和代码段。在C语言中，BSS和数据段都存放静态和全局变量。不同的是，BSS存放未初始化的静态变量，这些变量在源代码中没有被赋值过。BSS段的内存区是匿名的，未映射任何文件。如果申明一个静态变量static int cntActiveUsers，那么cntActiveUsers就在BSS段。

相反地，数据段存放在源码中初始化过的静态变量。这个内存区是非匿名的。其映射了程序部分的二进制文件，这部分包含了源码中的初始静态值。如果定义了一个变量static int cntWorkerBees = 10，那么变量cntWorkerBees就在数据段，其初值为10。尽管数据段映射了一个文件，但该映射是私有内存映射，意味着对内存数据的更新不会体现在映射的文件中。否则，对全局数据的修改就会修改磁盘上的二进制文件。

图4中数据段中举例使用的是指针，理解起来有一些复杂。在这个例子中，指针gonzo（4字节的内存地址）的值是在数据段，而该指针指向的字符串是在代码段的。代码段是只读的，在代码段存放着代码和诸如字符串之类的数据。代码段也会映射到内存中，但如果对该段做写操作的话会引起段错误。这样做可以防止由于指针使用不当引入bug。图5是各个段和例子中各个变量的对应关系。

Linux中各个进程的内存信息可通过读文件/proc/pid_of_process/maps来查看。一定要牢记：一个内存段可能会包含多个内存区。例如，每个内存映射文件在内存映射段都有自己的内存区；

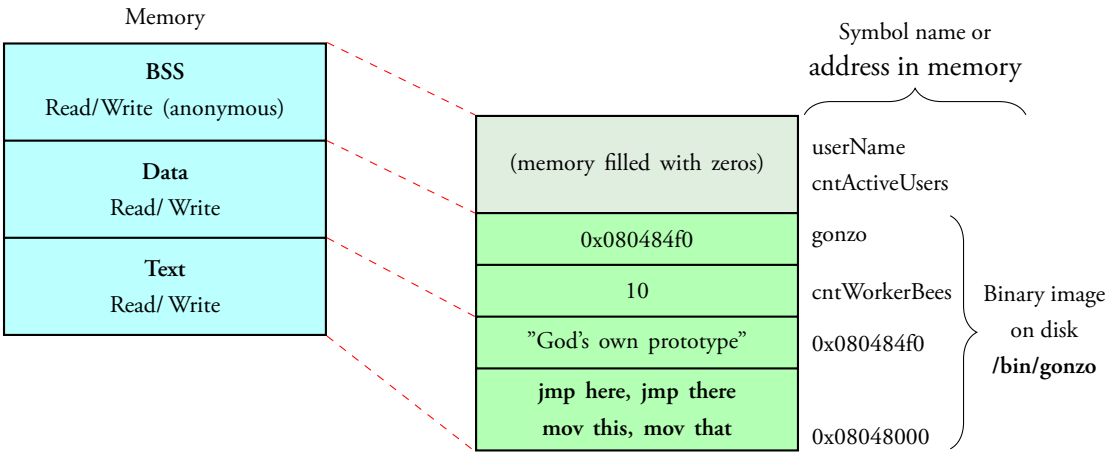

图 5: 段映射

动态库也有类似于 BSS 和数据段的内存区。后面的博文会进一步说明"内存区"到底是什么。有时候人们也把 data + bss + heap 统称为数据段。

通过nm和objdump命令可以查看二进制文件的符号、地址、段等信息。上面描述的内存布局在 Linux 中是一种"灵活的"布局方式，这种布局方式作为 Linux 中默认的内存布局已经有几年时间了。只要定义了RLIMIT_STACK，Linux 就使用这种布局方式，否则 Linux 会切换回"经典的"内存布局，如图6。

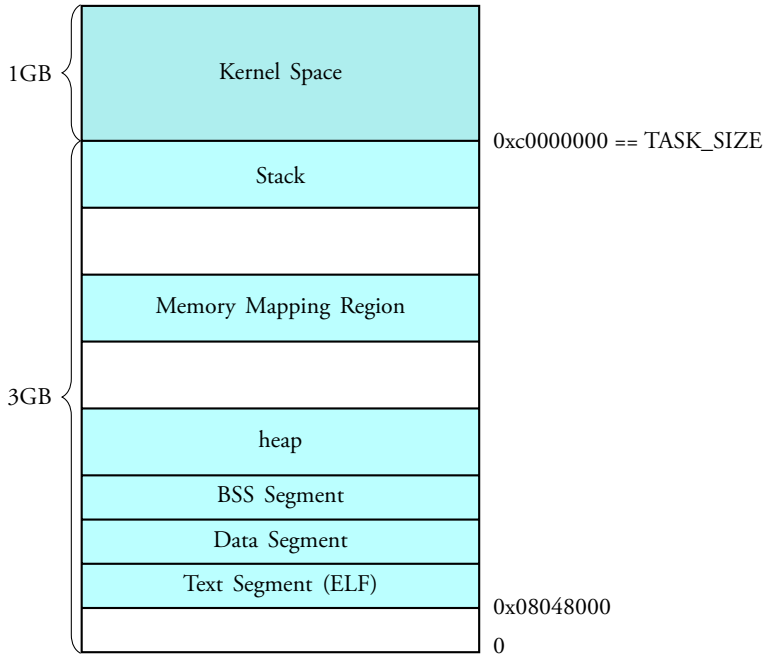

图 6: "经典"内存布局

这就是所有虚拟地址空间布局。下一篇文章将讲述内核是怎么跟踪这些内存区的，紧接着是内存映射、文件读写和内存使用到底是什么。