**浅谈程序的内存布局**

herongwei

**前言**

1、什么是 User space 与 Kernel space？

2、什么是栈区？

3、什么是堆区？

4、malloc 算法是如何实现的？

5、Linux 系统下，有几种堆空间分配方式？

6、Linux 下一个进程地址空间布局是怎样的？

上面几个问题，你心里有答案吗？如果没有，跟我一起来探究一下吧。

**User space 与 Kernel space**

现代的应用程序都运行在一个内存空间里，在 32 位系统中，这个内存空间拥有 4GB （2 的 32 次方）的寻址能力。

尽管现在大部分计算机的内存空间配置越来越高，但实际上内存仍然在不同的地址区间有着不同的地位，例如，大多数操作系统都会将 4GB 的内存空间一部分挪给内核使用，应用程序无法直接访问这一段内存，这一部分内存地址被称为内核空间。

Windows 在默认的情况下会将高地址的 2GB 空间分配给内核（也可以配置 1GB）。

Linux 默认情况下将高地址的 1GB 空间分配给内核。

用户使用的剩下的 2GB 或 3GB 的内存空间称为用户空间。

**为什么要区分内核空间和用户空间？**

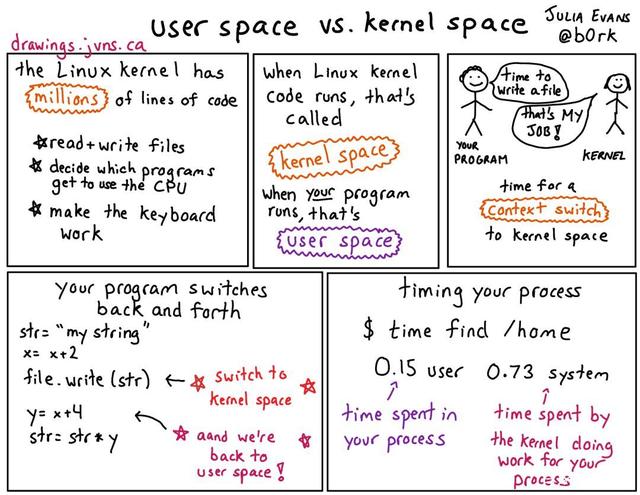
大致有三点因素：

第一点：操作系统的数据都是存放于系统空间的，用户进程的数据是存放于用户空间的；

第二点：分开来存放，就让系统的数据和用户的数据互不干扰，保证系统的稳定性，并且管理上很方便；

第三点：也是重要的一点，将用户的数据和系统的数据隔离开，就可以对两部分的数据的访问进行控制。这样就可以确保用户程序不能随便操作系统的数据，这样防止用户程序误操作或者是恶意破坏系统。

下面这一张图，比较形象的解释了 User space 与 Kernel space 的区别。



第一点：操作系统的数据都是存放于系统空间的，用户进程的数据是存放于用户空间的；

简单说，Kernel space 是 Linux 内核的运行空间，User space 是用户程序的运行空间。为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不受影响。

Kernel space 可以执行任意命令，调用系统的一切资源；

相对来说，User space 执行的是较为简单的运算，执行的运算不影响其他程序的执行，并且不能直接调用系统资源，必须通过系统接口（又称 system call），才能向内核发出指令。

这里补充下知乎网友@风云评论：

其实，在用户空间，几乎所有内核资源在用户空间都是可以访问的（必须有相应的权限），即使是操作系统内核的大脑（调度程序）。

**Linux 进程地址空间布局**

在用户空间里，也有许多地址区间有特权的地位，一般来讲，应用程序使用的内存空间里有如下“默认”的区域。

**栈区：**栈用于维护函数调用的上下文，离开了栈，函数调用就无法实现，栈通常在用户空间的最高地址处分配，通常有数兆字节的大小。

**堆区：**堆是用来容纳应用程序动态分配的内存区域，当程序使用 malloc 或者 new 分配内存的时候，得到的内存会来自堆里。

堆通常存在栈的下方（低地址方向），在某些时候，堆也可能没有固定统一的存储区域。堆一般比栈大很多，可以有几十至数百兆字节的容量。

**可执行文件映像：**存储着可执行文件在内存里的映像，由装载器在装载时将可执行文件的内存读取或映射到这里。

**保留区：**保留区并不是一个单一的内存区域，而是对内存中受到保护而禁止访问的内存区域的总称：例如大多数操作系统中，极小的地址通常都是不允许访问的，如 NULL，C 语言将无效指针赋值为 0 也是这个考虑。

**动态链接库映射区：**这个区域用于映射装载的动态链接库。在 Linux 下，如果可执行文件依赖其它共享库，那么系统就会为它在从 0x40000000 开始的地址分配相应的空间，并将共享库载入该空间。

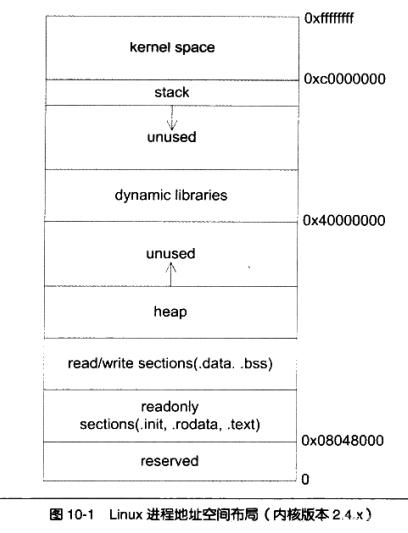
剩下的还有以下几部份组成：

（1）代码段

（2）初始化数据段（数据段）

（3）未初始化数据段（BSS 段）

下图是 Linux 下一个进程里典型的内存布局



图中的箭头，标明了几个大小可变的尺寸增长的方向，在这里，可以清晰地看出：

栈是由高地址向低地址增长。

堆是由低地址向高地址增长。

当栈或堆现有的大小不够用的时候，它将按照图中的增长方向扩大自身的尺寸，直到预留的空间被用完为止。

在讲堆和栈之前，我们先来看一下代码段，初始化数据段和未初始化数据段。

**代码段**

代码段中存放可执行的指令，在内存中，为了保证不会因为堆栈溢出被覆盖，将其放在了堆栈段下面（从上图可以看出）。

通常来讲代码段是共享的，这样多次反复执行的指令只需要在内存中驻留一个副本即可，比如 C 编译器，文本编辑器等。

代码段一般是只读的，这样程序执行时不能随意更改指令，也是为了进行隔离保护。

**初始化数据段**

初始化数据段有时就称之为数据段。数据段是一个程序虚拟地址空间的一部分，包括一全局变量和静态变量，这些变量在编程时就已经被初始化。数据段是可以修改的，不然程序运行时变量就无法改变了，这一点和代码段不同。

数据段可以细分为初始化只读区和初始化读写区。这一点和编程中的一些特殊变量吻合。比如全局变量 int global n = 1就被放在了初始化读写区，因为 global 是可以修改的。而 const int m = 2 就会被放在只读区，很明显，m 是不能修改的。

**栈**

栈 (stack) 是现代计算机程序里最为重要的概念之一，几乎每一个程序都使用了栈，没有栈就没有函数，没有局部变量，也就没有我们如今能够看见的所有的计算机语言。

在解释为什么栈会如此重要之前，让我们来先了解一下传统的栈的定义：

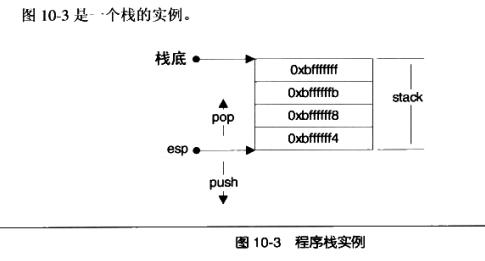
在经典的计算机科学中，栈被定义为一个特殊的容器，用户可以将数据压入栈中（入栈,push），也可以将已经压入栈中的数据弹出(出栈, pop)。

但栈这个容器必须遵守一条规则：**先入栈的数据后出栈(First In Last Out, FIFO)，多多少少像叠成一叠的书：先叠上去的书在最下面：因此要最后才能取出。**

在计算机系统中，栈则是一个具有以上属性的动态内存区域，程序可以将数据压入栈中，也可以将数据从栈顶弹出，压栈操作使得栈增大,而弹出操作使栈减小。

在经典的操作系统里，栈总是向下增长的。

在 i386 下，栈顶由称为 esp 的寄存器进行定位。压栈的操作使栈顶的地址减小，弹出的操作使栈顶地址增大。



这里栈底的地址是 0xbffff，而 esp 寄存器标明了栈顶，地址为 0xbifff4。

**在栈上压入数据会导致 esp 减小，弹出数据使得 esp 增大。**

栈在程序运行中具有举足轻重的地位。最重要的，栈保存了一个函数调用所需要的维护信息，这常常被称为堆栈帧(Stack Frame)或活动记录(Activate Record)，堆栈帧一般包括如下几方面内容：

1、函数的返回地址和参数。

2、临时变量:包括函数的非静态局部变量以及编译器自动生成的其他临时变量。

3、保存的上下文:包括在函数调用前后需要保持不变的寄存器。

**堆**

相对于栈，堆这片内存面临着一个稍微复杂的行为模式：在任意时刻，程序可能发出请求，要么申请一段内存，要么释放一段已经申请过的内存，而且申请的大小从几个字节到数 GB 都是有可能的。

我们不能假设程序会一次申请多少堆空间，因此，这时候堆的作用就凸显出来了，同样，相比较与栈，堆的管理显得较为复杂。

光有栈，对于面向过程的程序设计还远远不够，因为栈上的数据在函数返回的时候就会被释放掉，所以无法将数据传递至函数外部。而全局变量没有办法动态地产生，只能在编译的时候定义，有很多情况下缺乏表现力，在这种情况下，堆（Heap）是一种唯一的选择。

堆是一块巨大的内存空间，常常占据整个虚拟空间的绝大部分，在这片空间里，程序可以请求一块连续的内存，并自由地使用，这块内存在程序主动放弃之前都活一直保持有效，下面是一个申请堆空间最简单的例子：

int main(){char\* p = (char\*) malloc(233); free(p); return 0;}

在代码中，第 3 行用 malloc 申请了 233 个字节的空间之后，程序可以自由地使用这 233个字节，直到程序用 free 函数释放它。

**那么 malloc 到底是怎么实现的呢？**

有一种做法是，把进程的内存管理交给操作系统内核去做，既然内核管理着进程的地址空间，那么如果它提供一个系统调用，可以让程序使用这个系统调用申请内存，不就可以了吗？

当然这是一种理论上可行的做法，但实际上这样做的性能比较差，原因在于每次程序申请或者释放堆空间都需要进行系统调用。

我们知道系统调用的性能开销是很大的，当程序对堆的操作比较频繁时，这样做的结果是会严重影响程序的性能的。

比较好的做法就是：**程序向操作系统申请一块适当大小的堆空间，然后由程序自己管理这块空间，而具体来讲，管理着堆空间分配的往往是程序的运行**库。

运行库相当于是向操作系统 “批发” 了一块较大的堆空间，然后 “零售” 给程序用。

当全部“售完”或程序有大量的内存需求时，再根据实际需求向操作系统“进货”。

当然运行库在向程序零售堆空间时，必须管理它批发来的堆空间，不能把同一块地址出售两次，导致地址的冲突。

**Linux 进程堆管理**

由第一节可知，进程的地址空间中，除了可执行文件，共享库和栈之外，剩余的未分配的空间都可以用来作为堆空间。

Linux 系统下，提供两种堆空间分配方式：

**brk() 统调用和 mmap() 系统调用。**

这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。

在标准 C 库中，提供了malloc/free 函数分配释放内存，这两个函数底层是由 brk，mmap，munmap 这些系统调用实现的。

**brk() 系统调用**

C 语言形式声明：int brk() {void\* end\_data\_segment;}

brk() 的作用实际上就是设置进程数据段的结束地址，即它可以扩大或者缩小数据段（Linux 下数据段和 BBS 合并在一起统称数据段）。

如果我们将数据段的结束地址向高地址移动，那么扩大的那部分空间就可以被我们使用，把这块空间拿过来使用作为堆空间是最常见的做法。

**mmap() 系统调用**

和 Windows 系统下的 VirtualAlloc 很相似，它的作用就是向操作系统申请一段虚拟地址空间，（堆和栈中间，称为文件映射区域的地方）这块虚拟地址空间可以映射到某个文件。

glibc 的 malloc 函数是这样处理用户的空间请求的：对于小于 128KB 的请求来说，它会在现有的堆空间里面，按照堆分配算法为它分配一块空间并返回；对于大于128KB 的请求来说，它会使用 mmap() 函数为它分配一块匿名空间，然后在这个匿名空间中为用户分配空间。

声明如下：

void\* mmap{void\* start; size\_t length; int prot; int flags; int fd; off\_t offset;}

mmap 前两个参数分别用于指定需要申请的空间的起始地址和长度，如果起始地址设置 0，那么 Linux 系统会自动挑选合适的起始地址。

prot/flags 参数：用于设置申请的空间的权限（可读，可写，可执行）以及映射类型（文件映射，匿名空间等）。

最后两个参数用于文件映射时指定的文件描述符和文件偏移的。

了解了 Linux 系统对于堆的管理之后，我们可以思考这么一个问题：

**malloc 到底一次能够申请的最大空间是多少？**

为了回答这个问题，就不得不再回头仔细研究一下之前的图一。我们可以看到在有共享库的情况下，留给堆可以用的空间还有两处。

第一处就是从 BSS 段结束到 0x40 000 000 即大约 1GB 不到的空间；

第二处是从共享库到栈的这块空间，大约是 2GB 不到。这两块空间大小都取决于栈、共享库的大小和数量。

于是可以估算到 malloc 最大的申请空间大约是 2GB 不到。（Linux 内核 2.4 版本）。

当然还有其它诸多因素会影响 malloc 的最大空间大小，比如系统的资源限制（ulimit），物理内存和交换空间的总和等。

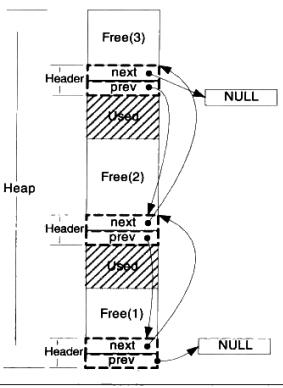
mmap 申请匿名空间时，系统会为它在内存或交换空间中预留地址，但是申请的空间大小不能超过空闲内存+空闲交换空间的总和。

**堆分配算法**

**1、空闲链表法（即调用 malloc 分配）**

就是把堆中各个空闲的块按照链表的方式连接起来，当用户请求一块空间的时候，可以遍历整个列表，直到找到合适大小的块并且将它拆分；当用户释放空间的时候将它合并到空闲链表中。

空闲链表是这样一种结构，在堆里的每一个空闲空间的开头(或结尾)有一个头 (header)，头结构里记录了上一个 (prev) 和下一个 (next) 空闲块的地址，也就是说，所有的空闲块形成了一个链表。如图所示。



具体实现方案：

malloc 函数的实质是它有一个将可用的内存块连接为一个长长的列表的所谓空闲链表。调用 malloc函数时，它沿着连接表寻找一个大到足以满足用户请求所需要的内存块。然后，将该内存块一分为二（一块的大小与用户申请的大小相等，另一块的大小就是剩下来的字节）。接下来，将分配给用户的那块内存存储区域传给用户，并将剩下的那块（如果有的话）返回到连接表上。调用 free 函数时，它将用户释放的内存块连接到空闲链表上。到最后，空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段， 那么空闲链表上可能没有可以满足用户要求的片段了。于是，malloc() 函数请求延时，并开始在空闲链表上检查各内存片段，对它们进行内存整理，将相邻的小空闲块合并成较大的内存块。**2、位图法**

针对空闲链表的弊端，另一种分配方式显得更加稳健。这种方式称为位围(Bitmap)，其核心思想是将整个堆划分为大量的块(block)，每个块的大小相同。

当用户请求内存的时候，总是分配整数个块的空间给用户，第一个块我们称为已分配区域的头(Head)，其余的称为己分配区域的主体(Body)，而我们可以使用一个整数数组来记录块的使用情况，由于每个块只有头/主体/空闲三种状态，因此仅仅需要两位即可表示一个块，因此称为位图。

**3、对象池**

还有一种方法是对象池，也是把堆空间分成了大小相等的一些块，它是认为某些场合每次分配的空间都相等，所以每次就直接返回一个块的大小，它的管理方法可以是链表也可以是位图。因为不用每次查找合适的大小的内存返回，所以效率很高。

实际上很多现实应用中，堆的分配算法往往是采取多种算法复合而成的。

比如对于glibc来说，它对于小于64字节的空间申请是采用类似于对象池的方法。

而对于大于512字节的空间申请采用的是最佳适配算法；

对于大于64字节而小于512字节的，它会根据情况采取上述方法中的最佳折中策略；

对于大于128KB的申请，它会使用mmap机制直接向操作系统申请空间。

**参考资料：**

《程序员的自我修养》http://www.ruanyifeng.com/blog/2016/12/user\_space\_vs\_kernel\_space.htm