前述

Dlmalloc是一个著名的内存分配器,最早由Doug Lea在1980s年代编写.由于早期C库的内置分配器在某种程度上的缺陷,因此dlmalloc出现后立即获得了广泛应用,足见其出色的表现.尽管时至今日, dlmalloc中的技术在一些地方已然落后于时代,很多优秀的allocator如google的tcmalloc, freeBSD的jemalloc等在某些情况下性能可以达到dlmalloc的数十甚至上百倍.但前者的很多思想和基本算法对后来者产生了深远的影响.走进memory allocator的神秘世界, dlmalloc可说是最好的教材之一.

本篇文章试图以初学者的角度全面阐述dlmalloc的设计思路和基本实现手法,以及内存分配相关的一些基础知识.事实上,内存分配器的设计思路和我们日常生活中"批发-零售"的概念是一致的,很多看似复杂的算法的基本原理都比较好理解.但是从实现的角度说,如果真的完全看懂dlmalloc的代码,则需要极大的耐心.因为Doug Lea本人在这份代码中使用了大量的技巧,对于初次接触此类代码的人来说,很多写法非常的古怪.然而当你耐下心细细体会作者的思路和意图之后,会对之心悦诚服,感受到其匠心独到之处.这对于开拓自身的编程视野也是具有莫大帮助的.

文档中使用的代码基于android4.4上的malloc版本.源代码位于目录,

bionic\libc\upstream-dlmalloc\Malloc.c

最新版本的dlmalloc源码请参考该网址,

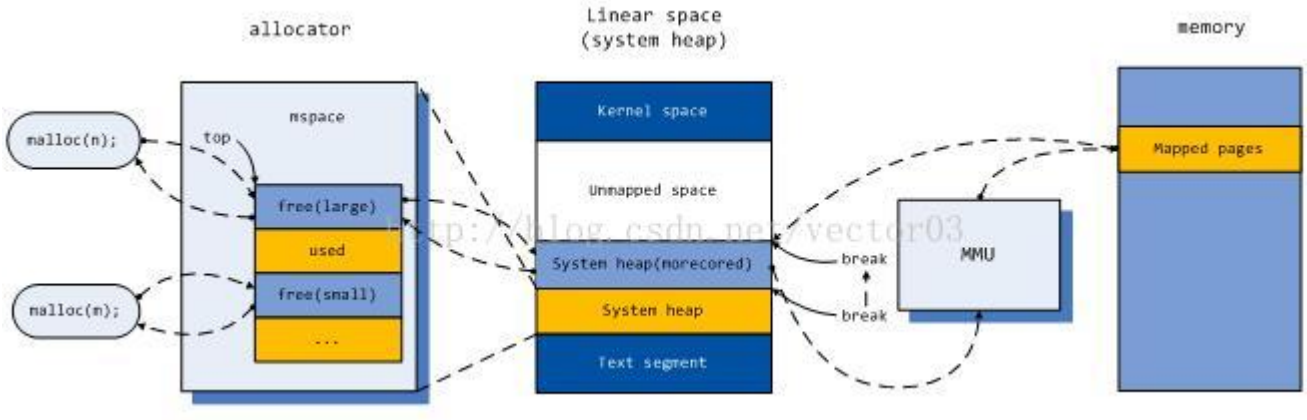
ftp://gee.cs.oswego.edu/pub/misc/malloc.c

1. 基础知识

在正式开始介绍dlmalloc之前,首先需要了解一些与之相关的基础知识,否则基本上很难看懂dlmalloc的代码和文档.

1.1 内存分配原理

最早接触C/C++这类语言时, malloc/free或者new/delete都是常常被提及的.这些函数或运算符无论参数还是返回值都非常简单.但具体来说,内存到底是如何从硬件上某个内存颗粒中的地址单元里,一路返回到程序员的手里呢?这绝非一个简单的过程,事实上整个过程相当之复杂.另外,我们需要探讨的内存分配器在其中又是扮演哪个角色呢?

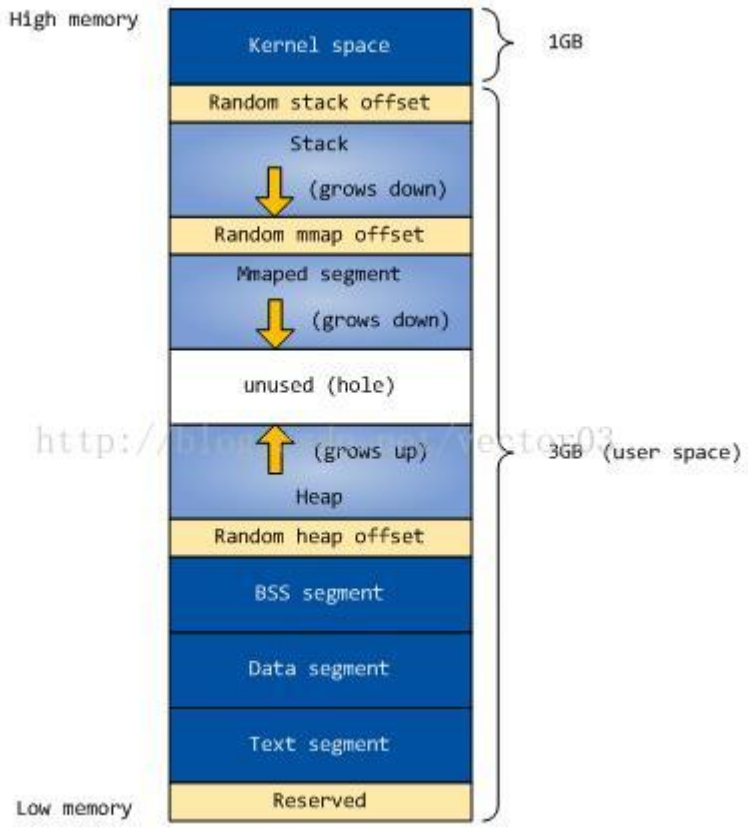


上面的图大致上描述了内存分配的过程,注意这只是一个示意图,它并不是很精确.首先我们需要知道的是allocator与普通程序一样都工作在用户态,并且其本身也是作为system heap的一部分。当用户通过malloc之类的库函数提交分配请求后,将首先由allocator查找,如果在其内部保留的空间中找到满足需求的内存块,就将之返回给用户(一般都是相对较小的块).如果找不到合适的块,则进一步向system发起请求,将划分system heap上的一部分给allocator,再由后者划拨给用户.另一方面,在划拨heap空间后,这部分新生成的空间其实还无法直接使用,系统会在合适的时机,通过名为MMU的硬件将实际物理内存上的某些区域(以page为单位),映射到需要使用的线性地址上.这样就完成了一块内存从申请到使用的全过程.

从这些描述上, 你可能会感受到一点, allocator和system之间的关系就如批发和零售的关系一样. System作为批发商,手中握有大量的待批商品——线性地址.而allocator作为零售商,会不定期的从批发商手里提货,将这些一次性批来的地址再进行二次管理.当用户程序作为顾客,向零售商购买商品时, allocator的整套算法都是为了保证尽可能的将货都卖出去而不砸在手里,也就是尽量避免在尚有存货的情况下向批发商购进新的货品.同时,它还要保证能在最快的时间将这些货卖出去. Allocator的初衷就是帮助用户程序又快又好的管理system heap中的内存.

1.2 Linux进程地址空间

说到system heap,需要了解一下进程的线性地址空间布局.如图所示,



关于此类内容, 介绍计算机体系架构的书中会有详述, 此章节做一个简单的介绍.当应用程序的二进制映像被加载到一个32bit机器的进程地址空间后,会被定位成如下的区域(这里与有些资料上介绍的略有不同,基本上该图会更加准确一些).从低地址到高地址分别为,

最开始的一段区域为保留区域, 这里并不存放有效的代码和数据.

之后从某一个地址开始, 为程序的入口点(C语言在该地方会有一句跳转,进入__cinit()).往上为代码段,存放编译好的二进制映像,基本上是一些指令,常量和字符串.

代码段之上是数据段, 存放所有全局和静态变量.

数据段再往上是bss段,存放所有未初始化的全局和静态变量.在程序一开始这段区域是一大片0.数据段和bss段也被合称为静态区.

数据段之上也就是我们平时常称作heap的堆区.注意heap是从低到高向上生长的,另外在heap一开始会有一段随机长度的offset区,加入这个主要是为了防止恶意代码的溢出型攻击.因为只要保证堆的基址每次都是随机的,就无法通过将恶意跳转指令插入到某个固定地址而导致因缓冲区溢出造成系统控制权丢失.

再往上是一大片未开垦的荒芜之地,虽然图上画的很小,但实际却很大.在通过系统调用申请之前,这片地址是既不可写也不可读的,否则会导致segmentation fault.

在中间的空洞之上是另一片比较大的区域,为mmap区.这片区域主要是保存文件映射,包括程序中使用的动态链接库so的映射,以及匿名文件映射,或者程序中共享数据用的手动映射.这部分区域是从高地址向低地址扩展.同样,在mmap区之上,

也存在一片用于防止溢出的随机offset区.

接着向上, 就是被称作stack的栈区.图中有些误导的是栈区其实本来是无法扩展的,最大容量是一早在内核中设定好的.这里标示的stack向低地址grows down指的是栈顶sp指针的生长方向.因此一旦sp超过这段区域的设定下限,就意味着爆栈了.同样stack之上也存在防止溢出的随机offset区.

前面说的所有区域, 在32bit机器上加起来为3GB大小.这3GB被称作进程的用户空间,在不同进程之间是不可见的(fork的父子进程某些值在写入前虽然是一样的,但写入后也是不一样的).之所以可以做到这一点在于每个进程控制块中保存了各自的页表目录和页表,这样通过MMU查询到的物理页面其实根本是两个地址.具体信息请查阅linux下mm的相关部分.

最后, 最高地址的1GB空间为系统所有的内核空间,所有内核代码包括内核驱动都执行在这部分中.与用户空间不同的是,内核空间的映射是固定的,因此不同进程虽然有自己的用户空间,却有着相同的内核空间.显然这部分地址也是我们用户无法直接读写的(可以通过驱动将内核空间的地址和用户空间映射在一起以实现访问).

通过上面的介绍, 可以看到应用程序的内存分配事实上是通过在中间空洞两边的heap区和mmap区相对生长来完成的.因此我们常说的动态内存在堆区分配其实是不准确的.

1.3 术语

介绍dlmalloc之前,还需要了解一些相关术语及其含义,这些到后面就不再做过多解释了.

Payload: 有效负载.指的是实际交给应用程序使用的内存大小.

Overhead: 负载,开销.本意是为了满足分配需求所消耗的内存量,实际在代码注释中多指除了payload之外的额外开销(有些书中也称之为cookie).

Chunk: 区块.是内存分配的基本单位,类似物质世界中的原子不可再分. dlmalloc对内存的管理基本上都是以chunk为单位.一个典型的chunk是由用户程序使用的部分(payload)以及额外的标记信息(overhead)组成.

Bin: 分箱.用来管理相同或同一区间大小的chunk.在dlmalloc中分为sbin和tbin两种.

Mspace: 分配空间.说白了就是dlmalloc中内存池的叫法.在dlmalloc中可以管理多个mspace.如果不显式声明,将会使用一个全局的匿名空间,或者用户可以自行划分空间交给dlmalloc管理.

Segment: 区段.一般情况下,内存分配都是在一片连续区间内开采(exploit).但也会遇到不连续的情况,这就需要分成若干个区段记录.多个区段可以同属一个mspace.

Fenceposts: 栅栏.大多数分配器中, fencepost起到非连续内存间的隔离作用.一般这种隔离被用做安全检查.分配器会在fenceposts所在位置写上特殊标记,一旦非连续内存间发生写入溢出(overwrite)就可以通过异常的fenceposts值发出警告.

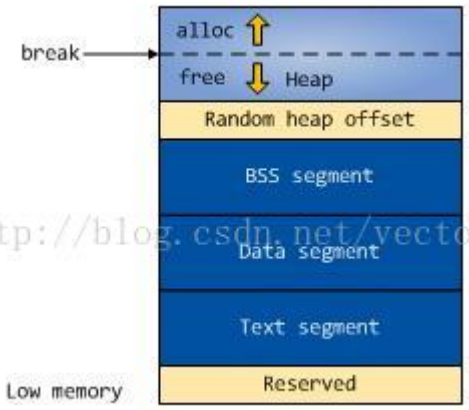
Bookkeeping: 记录信息.不同于每个chunk中的overhead,这里指的是整个mspace控制块的记录信息.往往这部分信息都固定在mspace开始的一段空间,或者干脆就放在地址空间的静态区中.

Granularity: 粒度.这个粒度指的是从system heap上获取内存的最小单位.一般来说该值至少为一个page size, 且必须以2为底.

Mmap: 本意是类unix系统的文件映射调用.但在dlmalloc中表示的更宽泛,这里指代可以在进程地址空间中开辟非连续内存空间的系统调用.

Morecore: 指可以在进程地址空间中开辟连续内存空间的系统调用.在类unix系统下morecore指的是sbrk调用.

Program break: 前面提到的sbrk()实际也是一个库函数,真正起作用的是brk()系统调用.这个函数其实就是break的缩写.所谓的break是一个代表进程heap区top-most位置的指针.当我们通过sbrk/brk向系统请求内存时,系统做的仅仅是移动break指针,内存就这样被划拨到heap中了.而当释放内存时,就反方向移动该指针,内存就返回给系统.



Footprint: 从系统获得的内存量.指的是当前dlmalloc从system heap获取的内存总和.设立footprint一方面是为了方便统计,另一方面也可以限制dlmalloc从系统获取的最大内存量.

Trimming: 裁剪.被dlmalloc管理的内存被free后,并不直接返还给系统,而是当积累到一定程度会通过一些算法判断system heap是否收缩(shrink),这个过程在dlmalloc中称作auto-trimming.

1.4 位运算

Allocator在计算内存大小时为了加快处理速度,会使用大量位运算.基本上很多计算技巧已经成为通用化的公式了.熟悉这些计算方法会对理解dlmalloc的代码有巨大帮助.

这里提一下Doug Lea推荐的一本书《Hacker's Delight》.该书是一本详细介绍各种位运算奇技淫巧的著作,读之前需要做好被虐的准备.

1.4.1 地址对齐

我们知道, 在计算机硬件体系结构中,内存的读写必须是对齐到某一个数值上的,比如常见的32bit机器对齐在4字节上.那么诸如0x08000003这样的地址就不能被直接访问.

判断一个地址或一段长度是否对齐到某个边界上其实就是判断该值能否被对齐边界整除.我们以对齐到8上为例, 24/8 = 3余0,说明是对齐的.而27/8 = 3余3,就不是对齐的.从代码角度考虑,很容易得到这样的结果,

```
#define is_align(A, n) ((A) % (n) == 0)
```

显然, 取模运算的性能代价相对于allocator来说是巨大的,有必要考虑其他的方法.前面说到,所有硬件体系都需要内存读写对齐到某一数值上,但还有一点没有提及,就是该数值必须是以2为底的指数.这一次,我们将所有的除数和被除数用二进制来表示,

24 / 8 ⇨ 11 000b / 1 000b = 11b mod 0

35 / 8 ⇨ 100011b / 1 000b = 100b mod 11b

50 / 8 ⇨ 110101b / 1 000b = 110b mod 101b

126 / 8 ⇨ 1111110b / 1 000b = 1 111b mod 110b

我想你可能发现了一个有趣的规律, 8相当于2^3,而所有被8除的结果中,在二进制下余数都是被除数的最后3位.是否说如果n相当于2^m次幂,则A除以n的余数相当于二进制下A的最后m位呢?答案是肯定的.换句话说,上面的判断中,如果我们能知道n是2的几次幂就可以通过取出A的最后m位来判断是否对齐.

以2为底的指数在二进制下还有一个有用的特性,

比如, 8的二进制表示为1000b,而 8 – 1表示为111b.

16的二进制表示为 10000b,而 16 – 1表示为1111b.

而32的二进制表示为100000b, 32 – 1表示为11111b.

这样我们又发现了第二个规律, 以2为底的m次幂的二进制数减去1时,由于连续借位,最终出现除了MSB之外,低位全部被1填充的情况.因此我们有,

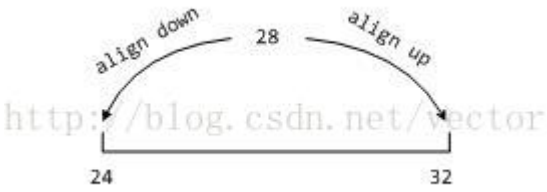
如果n为以2为底的m (m > 0, m = 1, 2, 3, ...)次幂指数,当以n为对齐边界时,称n – 1为该对齐边界的对齐掩码(align mask).

了解上面的两个特性后, 判断某地址是否对齐到以2为底的指数时就变得简单了,只要将原数值与对齐掩码作与,判断结果是否等于0即可,

```
#define is_align(A, n) ((A) & ((n)-1) == 0)
```

1.4.2 上对齐与下对齐

那么对于一个非对齐的数值进行处理实际上存在两种方式,上对齐(align up)和下对齐(align_down).还是以8为对齐边界来讨论,



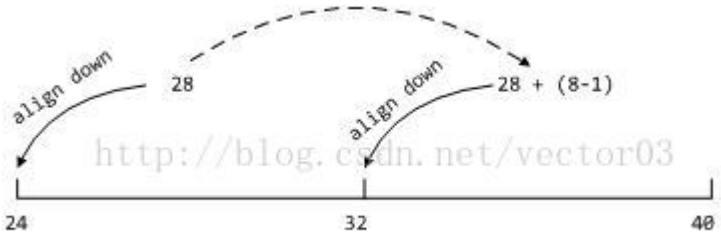
图中, 28如果采用下对齐的方式,对齐到24,则只需要简单舍去其余数即可.操作的方法有两种,一种是先右移再左移的方式,

```
#define align_down(A) (((A) >> 3) << 3)
```

或者, 直接使用对齐掩码的方式,

```
#define align_down(A, n) ((A) & ~((n)-1))
```

如果选择上对齐, 则需要另外的方法,



如上图所示, 首先将28加上一个偏移,使其跳跃到下一个对齐区间里去,然后再对其使用下对齐方式,就间接实现了在当前区间中上对齐.这个偏移值为对齐边界减1.为什么这里会减1呢?这里与对齐掩码什么的无关了,因为若被对齐数本身已经处于对齐状态,再加上一个对齐长度并与对齐掩码运算后就会留在下一个对齐区间中,这个结果显然是不正确的了.因此得到的代码如下,

```
#define align_up(A, n) (((A) + ((n)-1)) & ~((n)-1))
```

这个宏就是可以被作为公式的内存对齐宏.

1.4.3 计算对齐偏移量

有时候不需要直接获取对齐后的地址,而是计算出一个对齐偏移量,

```
#define align_offset(A, n) (((A) & (n-1) == 0) ? 0 : \
                            ((n) - (A) & ((n)-1)) & ((n)-1))
```

这个宏看起来复杂, 其实就是前面的组合, 首先判断是否已经对齐,如果是则偏移量为0.没有对齐则用n – A & (n-1)获取偏移量,再与上对齐掩码保证位数正确.事实上,大多数机器上后面的与操作其实是多余的.