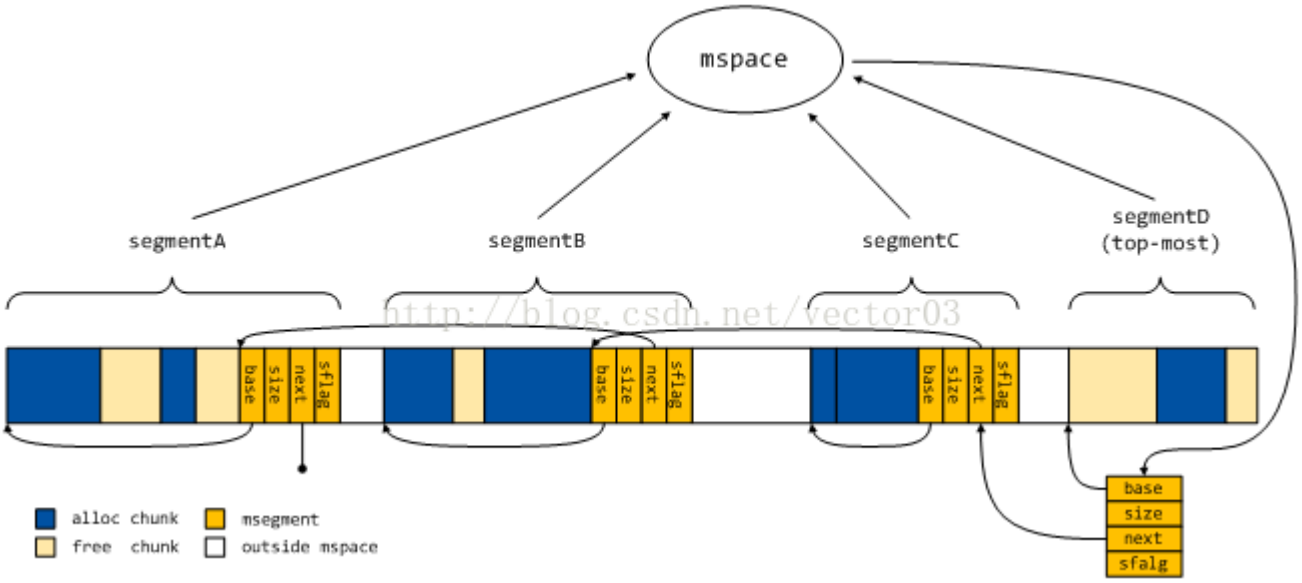
2.3 区段(segment)

在dlmalloc的内部结构中,除了基础的chunk外,还存在一种粒度更粗的结构,称为区段(segment).

之所以需要额外引入这种数据结构是为了提升对非连续内存的管理能力. dlmalloc将连续内存划分到一个segment中管理,而段与段之间则是不连续的,所有的段都由一个mspace统一去管理.如果用户程序创建了多个mspace,这些mspace内部包含的segment有可能在地址上连续,但实际归属于不同的空间.

大体上, segment在内存中可能是这样分布的,



根据分配器原理, allocator从系统获取一定量内存后按照用户程序需求返回不同大小的chunk.然而,当内部保留内存耗尽或无法满足需求时,就会重新向系统申请内存.还记得那张进程内存映射图吗,系统内存申请就是围绕着mmap区和system heap两者对向生长的.

在dlmalloc中,向系统申请内存有两种方式,分别为MORECORE和MMAP.前者用于申请连续内存,后者则可以返回非连续内存.多数时候, segment是通过MMAP产生出来的. dlmalloc在获取系统内存时会秉着先MORECORE后MMAP的顺序,且在申请成功后会检查是否与当前top-most段相毗邻,如果是就将新区域并入其中.由于MORECORE是连续开辟的,因此新区域总是可以append到当前top-most段后面.一旦MORECORE不能满足需求,转而调用MMAP,将可能产生非连续内存(MMAP也可能返回连续内存,且新区域既可能被append到当前段的后面,也可能会prepend到前面),此时dlmalloc就会建立新的segment.

segment在源码中被定义为malloc_segment,简称msegment,

```
struct malloc_segment {
  char*        base;             /* base address */
  size_t       size;             /* allocated size */
  struct malloc_segment* next;   /* ptr to next segment */
  flag_t       sflags;           /* mmap and extern flag */
};

typedef struct malloc_segment  msegment;
typedef struct malloc_segment* msegmentptr;
```

在msegment中记录了当前段的基址(base),该段长度(size),下一个msegment指针(next)以及一个标志(sflags).不难看出segment使用单链表来管理.因此与chunk相比,遍历segment的时间复杂度要高的多.这也从侧面反应一个事实, dlmalloc是不欢迎过多数量的segment的.因为区段越多反映出当前mspace下管理的内存越不连续,在chunk检索时时间消耗会更大,倘若使用非线性检索,则又增加额外的空间消耗.另外,不连续的区段在释放时也会导致麻烦,一旦某个段内的chunk全部释放,就需要考虑整体释放该段,但dlmalloc查找空闲段是很慢的,这将拖慢释放的整体效率.基于上述原因, dlmalloc才采取优先MORECORE以及区段合并的策略,尽量减少segment产生.

另外, 如图所示,除了top-most段之外, msegment大多被放置在区段末尾.而top-most信息被单独记录在mspace中.这一点也不难理解,因为top-most段是可扩展的,如果放在最后就会很不灵活,所以在mspace中单独开辟一个区域存放它的信息.

最后, 在msegment中记录了一个flag信息,以标记该区段的来历.如果flag等于EXTERN_BIT,表示该段是由外界分配并交给dlmalloc管理的,因此dlmalloc无权对其进行任何增删操作.这多出现在用户程序调用函数create_mspace_with_base时.此调用会根据传入的基址建立一个新的mspace,同时在内部生成一个segment.如果flag等于USE_MMAP_BIT,代表该段是通过MMAP创建的,因此其毗邻的MMAP段将会与之合并,且当munmap时会释放.若flag等于0,则表明该段是通过MORECORE建立的(在多mspace的条件下, MORECORE有可能会产生新的segment),因此其毗邻的MORECORE段将与之合并,且可以被反向释放.

2.4 malloc_state

dlmalloc中用来描述mspace的结构体被称为malloc_state,它也是dlmalloc中的核心结构体,其内部记录了关于内存分配的所有关键数据.

malloc_state的定义如下,后面将简写为mstate,

```
struct malloc_state {
  binmap_t   smallmap;
  binmap_t   treemap;
  size_t     dvsize;
  size_t     topsize;
  char*      least_addr;
  mchunkptr  dv;
  mchunkptr  top;
  size_t     trim_check;
  size_t     release_checks;
  size_t     magic;
  mchunkptr  smallbins[(NSMALLBINS+1)*2];
  tbinptr    treebins[NTREEBINS];
  size_t     footprint;
  size_t     max_footprint;
  size_t     footprint_limit; /* zero means no limit */
  flag_t     mflags;
#if USE_LOCKS
  MLOCK_T    mutex;     /* locate lock among fields that rarely change */
#endif /* USE_LOCKS */
  msegment   seg;
  void*      extp;      /* Unused but available for extensions */
  size_t     exts;
} ? end malloc_state ? ;

typedef struct malloc_state*    mstate;
```


下面详细解释其内部成员变量,

2.4.1 Bin maps

smallmap和treemap是两张位图,它们缓存了small bins和tree bins的使用情况.这里使用的技术类似于操作系统中的就绪表(ready table),不同的是就绪表中每1bit代表对应的任务组是否有ready task,而bin map中每1bit代表对应的分箱是否存在free chunk.通过bin maps可以O(1)的时间复杂度找到可用分箱.

从数据类型看, binmap_t就是无符号整型,

```
typedef unsigned int binmap_t;          /* Described below */
```

因此, bin map的32bit位就恰好对应small bins和tree bins的各32个分箱.

为了方便操作这两张位图, dlmalloc使用了一些宏,下面做简单介绍,

```
#define idx2bit(i)              ((binmap_t)(1) << (i))
```

该宏正如其名, 是将分箱编号转化成对应的位图掩码. 因为bitmap是从lsb到msb开始计算的,所以要左移i位.

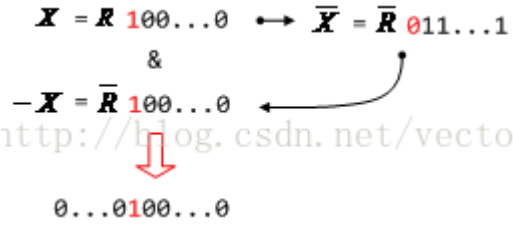
```
#define mark_smallmap(M,i)      ((M)->smallmap |=  idx2bit(i))
#define clear_smallmap(M,i)     ((M)->smallmap &= ~idx2bit(i))
#define smallmap_is_marked(M,i) ((M)->smallmap &   idx2bit(i))

#define mark_treemap(M,i)       ((M)->treemap  |=  idx2bit(i))
#define clear_treemap(M,i)      ((M)->treemap  &= ~idx2bit(i))
#define treemap_is_marked(M,i)  ((M)->treemap  &   idx2bit(i))
```

上面这两组宏比较好理解, 就是针对small bins和tree bins的设置,清除和测试掩码操作.

```
#define least_bit(x)         ((x) & -(x))
```

这个宏用来分离给定数x的最右侧1bit位(right-most)掩码,一般用来查找第一个不为空的分箱.

原理很简单, 我们可以把一个二进制数X以最右侧1bit分为左右两部分,右半部分肯定都是0,左侧剩余部分简称R.则-X为X取反加1,因此左侧部分变成R的取反,而右半部分保持不变.最后与原数位与就得到最右侧1bit掩码,如下图,




```
#define left_bits(x)         ((x<<1) | -(x<<1))
```

了解了least_bit, 这个left_bits就好理解了.它的作用是返回除了最右侧1bit之外,左侧所有剩余高位的掩码.即如果有X = 1000100b,则left_bits(X) = 1111 1000b.该宏一般用来快速剔除最小可用分箱,在更大分箱中查找.

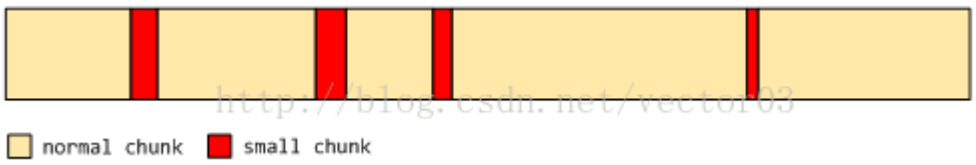
除了上述几个宏, 还有一个compute_bit2idx, 它是idx2bit的逆运算,即给定掩码返回相应分箱号.同2.2.4中的compute_tree_index一样,它同样存在四种实现,其实也就是利用BSF(正向比特位扫描)指令获得掩码中1bit所在位号.下面列出其中一种实现作为参考,

```
#define compute_bit2idx(X, I)\
{\
  unsigned int J;\
  J = _bit_scan_forward (X); \
  I = (bindex_t)J;\
}
```

2.4.2 牺牲块(dv)

dv的全称是Designated victim即指定牺牲品,这里我们叫它牺牲块.牺牲块是一种特殊的chunk,它即不被small bins也不被tree bins管理,而是直接记录在mstate的dv指针中,同时其大小缓存在dvsize里.

dv的设计基于局部性原理,即CPU对某一大小内存的分配和释放都趋向于在一段时间里连续执行,对小块内存尤其明显.事实上,小块内存如同人体血液中的坏胆固醇一样,不加以管控会导致内存分配机制严重受损,对于整个内存空间都是一场灾难.下图描述的就是不健康的内存空间,



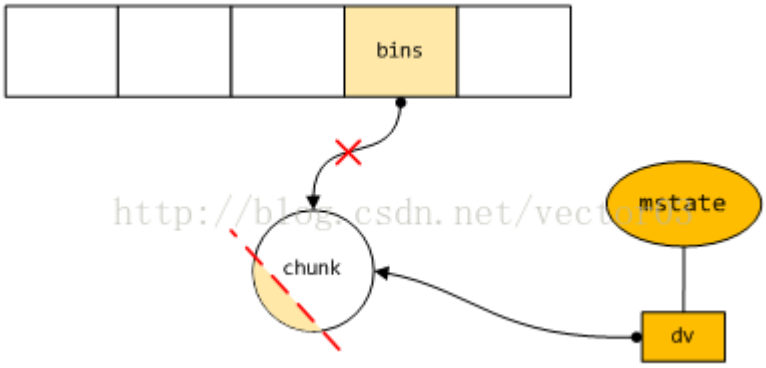
我们看到, 小块内存将整个空间切割成若干部分.如果这些小chunk不释放,将很难组织起更大规模的分配.不仅如此,这些小chunk还有可能阻止我们向system heap归还内存,尽管它们可能仅仅只有十几个字节,却导致大量空闲内存滞留在当前空间中.

相对来说, 健康的内存空间应该类似于下面的排布,

上图中, 小chunk都被约束在一个连续区间内,这样它们就不会跑出来捣蛋了.内存空间的整体分配能力和分配速度也获得了明显提升.

从分配器的角度看, 获取小型chunk要么从small bins中筛选,要么就只有切割大型chunk(准确地说,所有小型chunk都是从大型chunk中获得的).为了达到理想的分配效果, dlmalloc引入了dv作为牺牲块,专门用于满足那些在small bins中无法找到合适chunk的小块内存分配请求.

dv在被指定前可以是分箱中任意一个空闲块,只要根据分配算法,该空闲块遭到切割, dlmalloc就指定它作为dv,并将其与所在分箱脱离开来,直接交由mstate管理.也就是选定一个牺牲品,单切它一个,以保证小型chunk维持在一个可控范围内.一旦某个dv被切割殆尽, dlmalloc就会寻找下一个目标来替代前任空出的位子.



事实上, dv的算法原形在日常生活中司空见惯.举例来说,假如我们需要到烟草店买一包烟.我们知道,烟草的软包装都是10包合一条的.如果单买的话,老板一般都会拆开一条卖.这样其他人再来买同牌子的烟就继续从拆开的那条里取,这是显而易见的事.我想任谁也不会傻到再去库房里拆开一条来卖吧,那样的话不亏死才怪呢. dv在这里就相当于被拆封的那条烟,事情就是那么简单.

2.4.3 Top

top在dlmalloc中也是一种特殊的chunk,它指代top-most的意思.同dv一样, top并不记录在任何分箱中,而是由mstate的top指针保存,同时其大小缓存在topsize里.

top的作用与dv有相似之处,即它们都不被最优先选择,而是当普通分箱中暂时找不到合适chunk时才被列入考察对象.所不同的是, dv仅服务于小型分配请求,而top则要广泛的多,除了特别巨大的,多数分配请求都可以通过它满足. top如此全能的原因在于它位于内存空间的最顶端(比如可能毗邻break指针),这样其size就可以随时改变,非常灵活.

事实上, 在mspace的诞生阶段,整个内存空间中就只有top这一个可用chunk,像small bins和tree bins此时还是空空如也的状态.因此,也可以将top看作所有chunk的始祖.正因为如此, top的另一个重要作用是被用来辨别mspace是否初始化.一旦内存空间成功初始化top就必然被赋值,

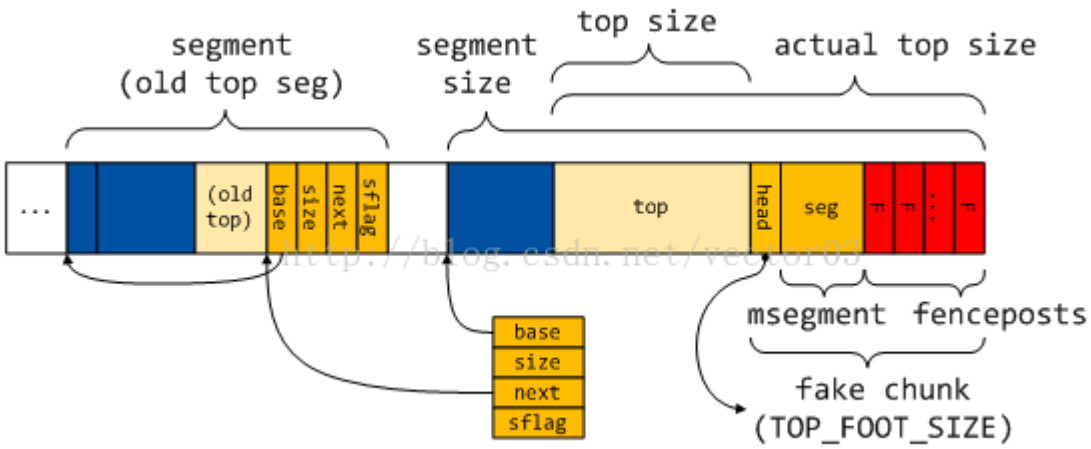
```
#define is_initialized(M)  ((M)->top != 0)
```

除top和topsize之外,在mstate中还有两个字段与top有关.

首先一个是msegment seg,注意这里它不是指针.在2.3节中曾提及,这个seg记录的是top-most段的信息.尽管我们说top chunk不受任何分箱管理,但从范围上它隶属于top-most段.

特别要说明的是, 虽然top-most段信息是单独保存的,但在它的末尾还是隐藏了一段预留空间. dlmalloc将它伪装成一个chunk放置在top chunk后面.其目的是一旦从系统获得了新的区段,可以将mstate中当前段信息写回到它的末尾(正如其他段那样),再将新的区段信息写入mstate.而如果没有预留空间,上述操作有可能因为得不到连续空间而失败.

因此, 更细化的top空间大概是这个样子的,



如图所示, 在mstate中记录的top size实际上不包含隐藏空间的大小, dlmalloc将这部分伪装成大小为TOP_FOOT_SIZE的fake chunk, 该chunk包括将来可能用于转存top-most段信息的msegment部分,以及后面的fenceposts部分.因此真实的top空间相当于topsize + TOP_FOOT_SIZE.

TOP_FOOT_SIZE定义如下,

```
#define TOP_FOOT_SIZE\
  (align_offset(chunk2mem(0))+pad_request(sizeof(struct malloc_segment))+MIN_CHUNK_SIZE)
```

而fenceposts的作用其实在dlmalloc中并不明显.在dlmalloc中fenceposts被定义成大小为SIZE_T_SIZE的fake chunk.事实上我们知道这么小的chunk根本就不存在,其本意就是连续写入FENCEPOST_HEAD作为交叉检查的预留字段,以检查segment间

发生溢出.

FENCEPOST_HEAD的定义,

```
#define FENCEPOST_HEAD      (INUSE_BITS|SIZE_T_SIZE)
```

一旦dlmalloc生成了新的segment,当前的top-most段就会引退.这时top空间中的隐藏区段就会被查找出来,并向其中写入mstate中的段记录信息.同时, top chunk也会从mstate中脱离,并重新作为普通chunk插入分箱系统中.接着, dlmalloc会为新的segment划分隐藏空间,并初始化top chunk.

另一个与top相关的字段是trim_check.它作为dlmalloc返还系统内存的阈值.由于top可以自由伸缩,因此,如果topsize大于trim_check设定的水线,就会触发trimming. Dlmalloc会试图将多余内存返还给系统.这部分内容在后面介绍free算法时会详细说明.

2.4.4 其他

除了上面几节介绍的之外, mstate中还记录了一些信息,下面集中加以说明.

least_addr:这个字段记录的是当前mspace中最低地址.它多用于安全检查,任何低于该地址的free或realloc等操作都将触发安全保护.增加该字段实际上是为了增强分配器的自我保护功能.因为一般的,低于least_addr的地址可能与mstate有关,为了保证核心数据结构的完整,需要对可操作的内存地址加以保护.

相关的宏如下,

```
#define ok_address(M, a) ((char*)(a) >= (M)->least_addr)
```

release_checks:这个字段用来记录free时内存合并的次数.一般情况下,尤其是当前mspace下存在多个区段时,仅仅依靠trimming释放系统内存是不够的.因为随着内存碎片的增加, top-most段有可能没有足够的空闲chunk以触发trim check.另外,非连续segment中的空闲 chunk大多也没有办法直接返还给系统.为了弥补这个缺陷, dlmalloc会检查是否存在完全空闲的区段.如果有,则通过munmap将整个区段释放.但显然,从整个mspace中查找空闲段是一个非常耗时的过程.作为折衷, dlmalloc通过记录free chunk合并的次数,以MAX_RELEASE_CHECK_RATE为周期进行空闲段检查.这个周期默认为4095,应该是一个经验值.

```
#define MAX_RELEASE_CHECK_RATE 4095
```

magic: 该字段保存一个随机数, 用于chunk的交叉检查.具体来说magic会与当前mstate指针做xor运算,如果FOOTERS宏打开,在chunk的prev_foot中就会保存该信息.当进行chunk检查时,将该值与magic再做一个xor还原mstate指针,以检查该chunk是否属于当前mspace下的合法区块.

与magic校验相关的宏有,

```
#define mark_inuse_foot(M,p,s)\
  (((mchunkptr)((char*)(p) + (s)))->prev_foot = ((size_t)(M) ^ mparams.magic))
```

向当前chunk的foot写入校验信息.

```
#define get_mstate_for(p)\
  ((mstate)(((mchunkptr)((char*)(p) +\
    (chunksize(p))))->prev_foot ^ mparams.magic))
```

反向提取mstate指针.

```
#define ok_magic(M)      ((M)->magic == mparams.magic)
```

该宏校验提取的mstate是否合法.

footprint, max_footprint, footprint_limit:这三个字段都与footprint有关.前面已经介绍footprint记录从系统获取的内存量.第一个代表当前申请量,第二个代表历史最高值,最后一个是限制阈值.

mflags: 记录当前mspace的设定信息,包括是否使用locks,是否允许mmap,以及是否使用非连续morecore.对应相关宏如下,

```
#define use_lock(M)           ((M)->mflags &   USE_LOCK_BIT)
#define enable_lock(M)        ((M)->mflags |=  USE_LOCK_BIT)
#if USE_LOCKS
#define disable_lock(M)       ((M)->mflags &= ~USE_LOCK_BIT)
#else
#define disable_lock(M)
#endif
```

是否使用全局lock锁的set, reset, test宏.

```
#define use_mmap(M)           ((M)->mflags &   USE_MMAP_BIT)
#define enable_mmap(M)        ((M)->mflags |=  USE_MMAP_BIT)
#if HAVE_MMAP
#define disable_mmap(M)       ((M)->mflags &= ~USE_MMAP_BIT)
#else
#define disable_mmap(M)
#endif
```

是否使用mmap的set, reset, test宏.

```
#define use_noncontiguous(M)  ((M)->mflags &   USE_NONCONTIGUOUS_BIT)
#define disable_contiguous(M) ((M)->mflags |=  USE_NONCONTIGUOUS_BIT)
```

是否使用非连续MORECORE的reset, test宏.

需要说明的是, mspace的初始状态与编译器的相关功能宏开关有关,详情请参考dlmalloc quick start手册。

mutex: 如果使用全局锁, 则会初始化mutex.注意mutex的类型MLOCK_T是一个系统相关类型,因此被设计成宏,在不同系统平台下代表的意义不同.尽管dlmalloc的单线程分配能力不错,但在多线程,尤其是SMP平台上其性能下降的非常厉害,主要就是由于对多线程支持的过于简单以致.

extp, exts:是两个未使用保留字段,用于扩展dlmalloc的功能.