3.4 sys_alloc

sys_alloc是dlmalloc中向系统获取内存的主要接口. 由于涉及到mmap, top-most segment, top chunk的交互, 相对要更复杂. 我们同样先介绍主要分配算法, 再详细分析子函数.

3.4.1 核心算法

基本上sys_alloc分为四个步骤,

1. 首先检查请求大小nb是否超出mmap_threshold的 阈值 . 如果是, 则放弃由分配器管理, 直接在mmap区开辟, 原因前面说过, 不再赘述.

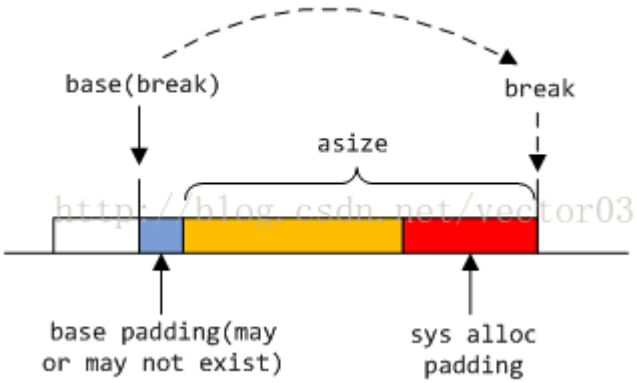
2. 根据mspace设定及当前top space的使用情况, 向系统申请一块适当的内存.

Dlmalloc按照下面的顺序由主到次开辟,

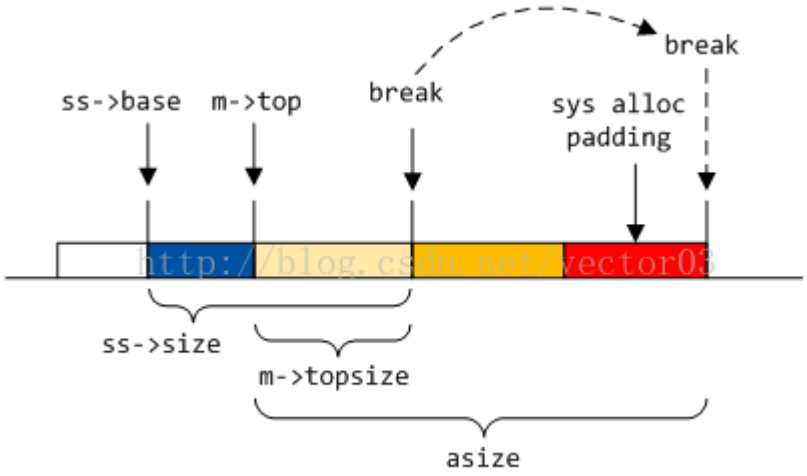
第一, 如果允许MORECORE, 则优先通过MORECORE开辟连续内存空间.

连续空间开辟又分为如下几种情况,

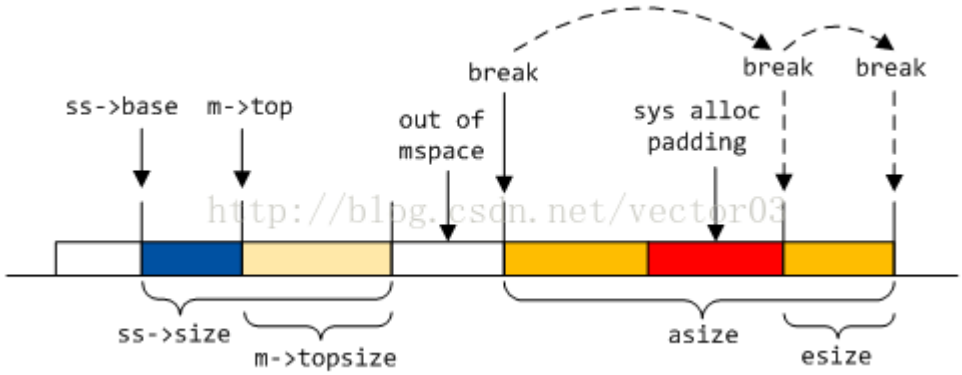
若当前mspace处于诞生阶段, 则直接开辟nb + SYS_ALLOC_PADDING大小的空间.



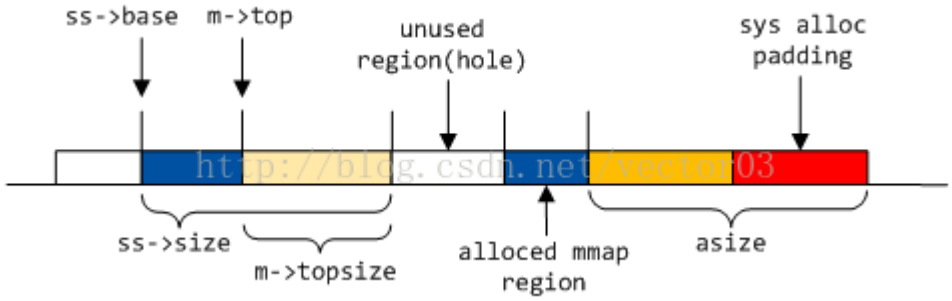
若当前mspace已存在top, 则部分空间可利用top, 剩余nb – m->topsize + SYS_ALLOC_PADDING大小的空间向系统申请.



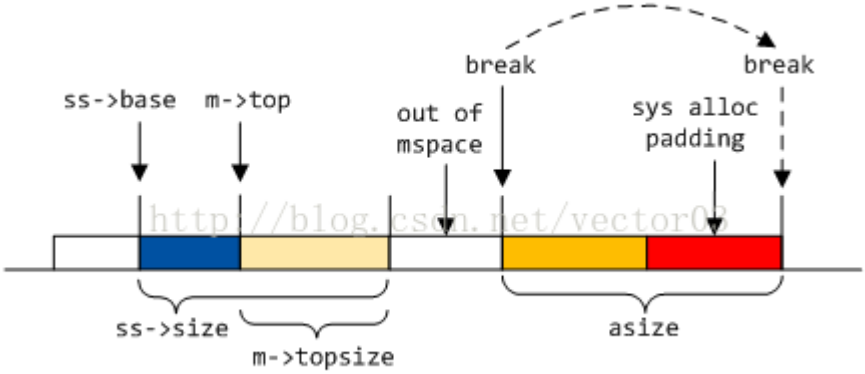
若MORECORE返回成功, 但空间不连续, 则会尝试扩展空间esize大小以满足分配需求. 如果空间扩展失败, 则反向MORECORE将之前申请的空间归还给系统.



第二, 如果上一步申请失败, 或不允许MORECORE, 则通过MMAP申请. 注意, 这里的MMAP同sys alloc步骤1的mmap是两码事. 这一步申请的结果是要归入mspace空间的.

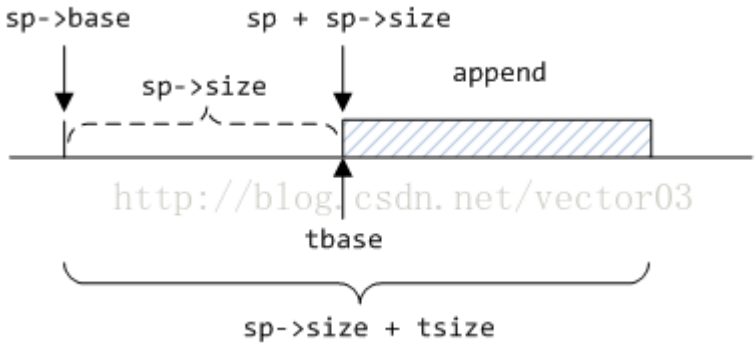


第三, 倘若前两步都失败, 且允许非连续(noncontiguous)MORECORE, 则尝试直接在system heap上分配非连续空间. 这有些类似第一步中的扩展空间, 区别是此时已经明确top space不连续, 直接申请目标大小.

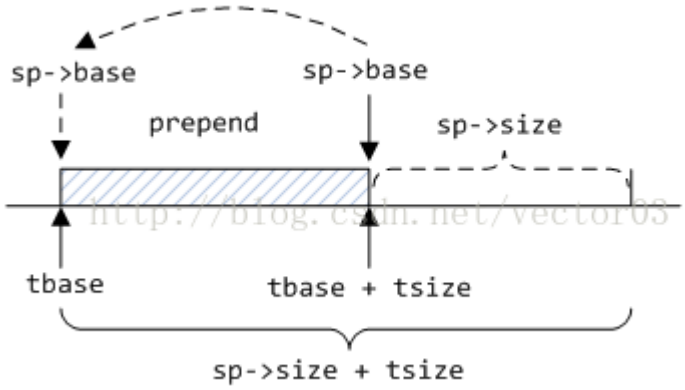


3. 根据申请成功的地址与原top space的关系, 对连续空间合并. 如果不能合并, 则新开区段. 申请的地址和大小保存在tbase和tsize临时变量里.

其中, 区段合并又分为两种. 若tbase与top-most区段末尾相毗邻, 则从后面合并. 这种情况适用于大部分MORECORE以及小部分MMAP申请到的空间.



若tbase与top-most区段开始相毗邻, 则从前面合并. 这种情况出现的比较少, 多在MMAP时产生, MORECORE虽然也可能出现该情况, 相对就更少.



4. 最后从已扩展的top space中划分chunk返回给用户. 这样就完成了sys_alloc的全部流程.

详细代码注释如下,

```
04050: static void* sys_alloc(mstate m, size_t nb) {
04051:   char* tbase = CMFAIL; /* 从系统申请的内存指针 */
04052:   size_t tsize = 0;     /* 从系统申请的空间大小 */
04053:   flag_t mmap_flag = 0;
04054:   size_t asize;         /* 根据nb计算的理论分配大小 */
04055:   /* 确保全局参数已初始化 */
04056:   ensure_initialization();
04057:   /* 若nb > mmap_threshold, 直接从mmap分配并返回 */
04058:   if (use_mmap(m) && nb >= mparams.mmap_threshold && m->topsize != 0) {
04059:     void* mem = mmap_alloc(m, nb);
04060:     if (mem != 0)
04061:       return mem;
04062:   }
04063:   /* 计算asize大小, 增加padding区域并对齐到粒度上 */
04064:   asize = granularity_align(nb + SYS_ALLOC_PADDING);
04065:   if (asize <= nb) {    /* 确保asize不发生溢出 */
04066:     MALLOC_FAILURE_ACTION;
04067:     return 0;
04068:   }
04069:   if (m->footprint_limit != 0) {    /* 若限制footprint且asize超出fp的限制, 分配失败 */
04070:     size_t fp = m->footprint + asize;
04071:     if (fp <= m->footprint || fp > m->footprint_limit) {
04072:       MALLOC_FAILURE_ACTION;
04073:       return 0;
04074:     }
04075:   }
04076:   /* 开始向系统申请空间 */
04077:   /* 若允许连续MORECORE且不强制使用非连续MORECORE */
04078:   if (MORECORE_CONTIGUOUS && !use_noncontiguous(m)) {
04079:     char* br = CMFAIL;      /* br保存MORECORE结果 */
04080:     size_t ssize = asize;   /* ssize保存MORECORE大小 */
04081:     /* 获得top-most区段指针, 若mspace未初始化, 则返回0 */
04082:     msegmentptr ss = (m->top == 0)? 0 : segment_holding(m, (char*)m->top);

04083:     /* 全局加锁 */
04084:     ACQUIRE_MALLOC_GLOBAL_LOCK();
04085:     if (ss == 0) {  /* 若ss为空, 说明mspace诞生阶段 */
04086:       /* 获取当前break指针, 保存在base中 */
04087:       char* base = (char*)CALL_MORECORE(0);
04088:       if (base != CMFAIL) {
04089:         size_t fp;
04090:         /* 若base未对齐, 将其对齐到页面上 */
04091:         if (!is_page_aligned(base))
04092:           ssize += (page_align((size_t)base) - (size_t)base);
04093:         fp = m->footprint + ssize;
04094:         if (ssize > nb && ssize < HALF_MAX_SIZE_T &&  /* 重新检查ssize溢出 */
04095:             (m->footprint_limit == 0 ||               /* 若不限制fp */
04096:              (fp > m->footprint && fp <= m->footprint_limit)) &&  /* 或者fp在限制范围内 */
04097:             (br = (char*)(CALL_MORECORE(ssize))) == base) {  /* MORECORE及连续性检查 */
04098:           /* 连续地址, 保存tbase及tsize */
04099:           tbase = base;
04100:           tsize = ssize;
04101:         }
04102:       }
04103:     }
04104:     else {  /* 若ss不为空, 可利用当前top */
04105:       /* 减去topsize, 并对其到粒度上 */
04106:       ssize = granularity_align(nb - m->topsize + SYS_ALLOC_PADDING);
04107:       /* MORECORE及连续检查 */
04108:       if (ssize < HALF_MAX_SIZE_T &&
04109:           (br = (char*)(CALL_MORECORE(ssize))) == ss->base+ss->size) {
04110:         tbase = br;
04111:         tsize = ssize;
04112:       }
04113:     }
04114:     if (tbase == CMFAIL) {    /* 若tbase失败 */
04115:       if (br != CMFAIL) {     /* 但MORECORE成功, 尝试扩展已获得的空间 */
```

```
04116:         if (ssize < HALF_MAX_SIZE_T &&  /* 例行检查 */
04117:             ssize < nb + SYS_ALLOC_PADDING) {
04118:           /* 减去已获得的空间ssize并对齐粒度，获得esize */
04119:           size_t esize = granularity_align(nb + SYS_ALLOC_PADDING - ssize);
04120:           if (esize < HALF_MAX_SIZE_T) {
04121:             /* MORECORE大小为esize的扩展空间 */
04122:             char* end = (char*)CALL_MORECORE(esize);
04123:             /* 成功扩展，在ssize基础上增加esize */
04124:             if (end != CMFAIL)
04125:               ssize += esize;
04126:             else {     /* 否则，已获得的空间不可用，反向MORECORE归还系统 */
04127:               (void) CALL_MORECORE(-ssize);
04128:               br = CMFAIL;
04129:             }
04130:           }
04131:         }
04132:       }
04133:       /* br不为空代表成功扩展，保存tbase和tsize */
04134:       if (br != CMFAIL) {
04135:         tbase = br;
04136:         tsize = ssize;
04137:       }
04138:       else
04139:         disable_contiguous(m); /* 扩展失败，禁止使用连续MORECORE */
04140:     } ? end if tbase==CMFAIL ?
04141:     /* 释放全局锁 */
04142:     RELEASE_MALLOC_GLOBAL_LOCK();
04143:   } ? end if MORECORE_CONTIGUOUS&&... ?
04144:   /* 若前面失败，且允许MMAP，则尝试MMAP获取 */
04145:   if (HAVE_MMAP && tbase == CMFAIL) {
04146:     char* mp = (char*)(CALL_MMAP(asize));
04147:     if (mp != CMFAIL) {
04148:       tbase = mp;

04149:       tsize = asize;
04150:       mmap_flag = USE_MMAP_BIT;
04151:     }
04152:   }
04153:   /* 若前面失败，且允许MORECORE，尝试非连续MORECORE */
04154:   if (HAVE_MORECORE && tbase == CMFAIL) {
04155:     if (asize < HALF_MAX_SIZE_T) {
04156:       char* br = CMFAIL;
04157:       char* end = CMFAIL;
04158:       ACQUIRE_MALLOC_GLOBAL_LOCK();
04159:       /* 直接MORECORE大小asize的空间 */
04160:       br = (char*)(CALL_MORECORE(asize));
04161:       /* 获得结尾指针 */
04162:       end = (char*)(CALL_MORECORE(0));
04163:       RELEASE_MALLOC_GLOBAL_LOCK();
04164:       if (br != CMFAIL && end != CMFAIL && br < end) {  /* 有效性检查 */
04165:         size_t ssize = end - br;
04166:         if (ssize > nb + TOP_FOOT_SIZE) {/* 非连续MORECORE成功，保存tbase和tsize */
04167:           tbase = br;
04168:           tsize = ssize;
04169:         }
04170:       }
04171:     }
04172:   }
04173:   /* 对申请到的空间进行处理，尝试合并或创建 */
04174:   if (tbase != CMFAIL) {    /* 若前面获得的tbase不为空 */
04175:     /* 更新max footprint记录 */
04176:     if ((m->footprint += tsize) > m->max_footprint)
04177:       m->max_footprint = m->footprint;
04178:     if (!is_initialized(m)) {   /* mspace初始化阶段 */
04179:       /* 若least_addr需要更新，则更新least_addr */
04180:       if (m->least_addr == 0 || tbase < m->least_addr)
04181:         m->least_addr = tbase;
```

```
04182:       /* 初始化top-most segment */
04183:       m->seg.base = tbase;
04184:       m->seg.size = tsize;
04185:       m->seg.sflags = mmap_flag;
04186:       m->magic = mparams.magic;
04187:       m->release_checks = MAX_RELEASE_CHECK_RATE;
04188:       /* 初始化分箱系统 */
04189:       init_bins(m);
04190:       /* 初始化top */
04191: #if !ONLY_MSPACES
04192:       if (is_global(m))
04193:         init_top(m, (mchunkptr)tbase, tsize - TOP_FOOT_SIZE);
04194:       else
04195: #endif
04196:       {
04197:         /* Offset top by embedded malloc_state */
04198:         mchunkptr mn = next_chunk(mem2chunk(m));
04199:         init_top(m, mn, (size_t)((tbase + tsize) - (char*)mn) -TOP_FOOT_SIZE);
04200:       }
04201:     } ? end if !is_initialized(m) ?
04202:     else {  /* 若已存在top, 尝试合并tbase和top-most segment */
04203:       msegmentptr sp = &m->seg;
04204:       /* 查找与tbase起始地址毗邻的segment, 保存到sp */
04205:       while (sp != 0 && tbase != sp->base + sp->size)
04206:         sp = (NO_SEGMENT_TRAVERSAL) ? 0 : sp->next;
04207:       if (sp != 0 &&    /* 找到对应的sp */
04208:           !is_extern_segment(sp) &&     /* 该sp不是外部分配的(外部的无法合并) */
04209:           (sp->sflags & USE_MMAP_BIT) == mmap_flag &&  /* 允许使用mmap */
04210:           segment_holds(sp, m->top)) {  /* 当前top属于该sp */
04211:         /* append在该sp后面, 并更新top */
04212:         sp->size += tsize;
04213:         init_top(m, m->top, m->topsize + tsize);
04214:       }

04215:       else {  /* 若不满足, 查找可以prepend的segment */
04216:         if (tbase < m->least_addr)
04217:           m->least_addr = tbase;
04218:         sp = &m->seg;
04219:         /* 查找与tbase结尾地址毗邻的segment, 保存到sp */
04220:         while (sp != 0 && sp->base != tbase + tsize)
04221:           sp = (NO_SEGMENT_TRAVERSAL) ? 0 : sp->next;
04222:         if (sp != 0 &&
04223:             !is_extern_segment(sp) &&
04224:             (sp->sflags & USE_MMAP_BIT) == mmap_flag) {
04225:           /* prepend到该sp的前面 */
04226:           char* oldbase = sp->base;
04227:           sp->base = tbase;
04228:           sp->size += tsize;
04229:           return prepend_alloc(m, tbase, oldbase, nb);
04230:         }
04231:         else  /* 找不到可以append及prepend的segment, 只能建立新的segment */
04232:           add_segment(m, tbase, tsize, mmap_flag);
04233:       }
04234:     } ? end else ?
04235:     /* 从申请到的空间中划分chunk, 并返回用户 */
04236:     if (nb < m->topsize) {
04237:       size_t rsize = m->topsize -= nb;
04238:       mchunkptr p = m->top;
04239:       mchunkptr r = m->top = chunk_plus_offset(p, nb);
04240:       r->head = rsize | PINUSE_BIT;
04241:       set_size_and_pinuse_of_inuse_chunk(m, p, nb);
04242:       check_top_chunk(m, m->top);
04243:       check_malloced_chunk(m, chunk2mem(p), nb);
04244:       return chunk2mem(p);
04245:     }
04246:   } ? end if tbase!=CMFAIL ?
04247:   /* 分配失败 */
04248:   MALLOC_FAILURE_ACTION;
04249:   return 0;
04250: } ? end sys_alloc ?
```


这个代码基本上就是本小节开始时介绍的流程, 相信看懂了前面的算法说明, 这里自然没有什么难度. 需要说明的仅有两点,

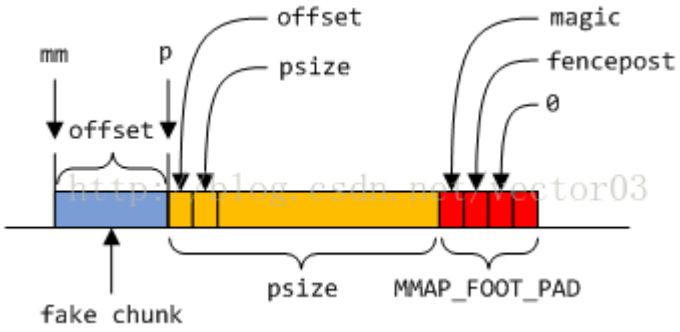
一个就是在Line4064和Line4065出现的判断, 先对nb做padding计算得到asize, 再判断其是否小于等于nb. 这里就是溢出检测, 对于两个无符号整数的加法, 这是比较便捷的检验方法. 类似的代码在dlmalloc中到处都是.

另一个参考Line4108, 这里同样是溢出检测. 因为MORECORE的入参在dlmalloc中被认为是有符号的. 而HALF_MAX_SIZE_T是size_t的一半, 以此来判断ssize是否溢出.

3.4.2 mmap_alloc

当nb大于mmap_threshold时, 会调用该函数直接进行mmap分配. 与sys_alloc通过其他途径申请的区别在于, dlmalloc对这类空间倾向于不长期持有, 也不纳入任何分箱或区段中. 可以认为它们是脱离dlmalloc管理的孤立内存区域.

既然是孤立内存, 首尾就不会有毗邻的chunk, 但直接mmap出来的payload地址未必是对齐的, 因此在对齐后会产生内部碎片. dlmalloc就将这些碎片伪装成一个chunk. 这样, 当用户释放这片内存时, 可以根据记录在prev_foot中的size信息找到当初mmap出来的首地址.



上图中, mmap分配的原始地址是mm, 经过对齐后的地址是p. dlmalloc将前面的对齐部分伪装成一个free chunk, 长度记录在p->prev_foot中. 当释放时, 就可以根据payload指针重新计算出mm的地址. 在结尾, 有长度为MMAP_FOOT_PAD的一段区域, 用来放置fake next chunk. 也就是保存magic以及fencepost.

代码注释如下,

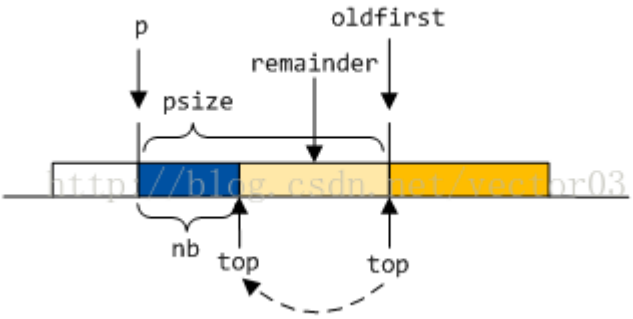
```
03840: static void* mmap_alloc(mstate m, size_t nb) {
03841:   /* 将nb填充并对齐到页面边界上, 得到mmsize */
03842:   size_t mmsize = mmap_align(nb + SIX_SIZE_T_SIZES + CHUNK_ALIGN_MASK);
03843:   /* 检测分配大小是否超出系统footprint限制 */
03844:   if (m->footprint_limit != 0) {
03845:     size_t fp = m->footprint + mmsize;
03846:     if (fp <= m->footprint || fp > m->footprint_limit)
03847:       return 0;
03848:   }
03849:   if (mmsize > nb) {     /* 溢出检查 */
03850:     /* 调用mmap直接申请内存 */
03851:     char* mm = (char*)(CALL_DIRECT_MMAP(mmsize));
03852:     if (mm != CMFAIL) {
03853:       /* 将payload指针对齐到边界, 计算出offset */
03854:       size_t offset = align_offset(chunk2mem(mm));
03855:       /* 去掉前面的offset和后面的padding */
03856:       size_t psize = mmsize - offset - MMAP_FOOT_PAD;
03857:       /* 得到对齐后的chunk */
03858:       mchunkptr p = (mchunkptr)(mm + offset);
03859:       /* 将offset区域伪装成free chunk */
03860:       p->prev_foot = offset;
03861:       p->head = psize;
03862:       mark_inuse_foot(m, p, psize);
03863:       /* 将padding区域写入fencepost */
03864:       chunk_plus_offset(p, psize)->head = FENCEPOST_HEAD;
03865:       chunk_plus_offset(p, psize+SIZE_T_SIZE)->head = 0;
03866:       /* 根据申请的内存地址和大小, 调整least_addr和fp */
03867:       if (m->least_addr == 0 || mm < m->least_addr)
03868:         m->least_addr = mm;
03869:       if ((m->footprint += mmsize) > m->max_footprint)
03870:         m->max_footprint = m->footprint;
03871:       /* 对最终结果检查并返回payload指针 */
03872:       assert(is_aligned(chunk2mem(p)));
03873:       check_mmapped_chunk(m, p);
03874:       return chunk2mem(p);
03875:     } ? end if mm!=CMFAIL ?
03876:   } ? end if mmsize>nb ?
03877:   return 0;
03878: } ? end mmap_alloc ?
```


3.4.3 prepend_alloc

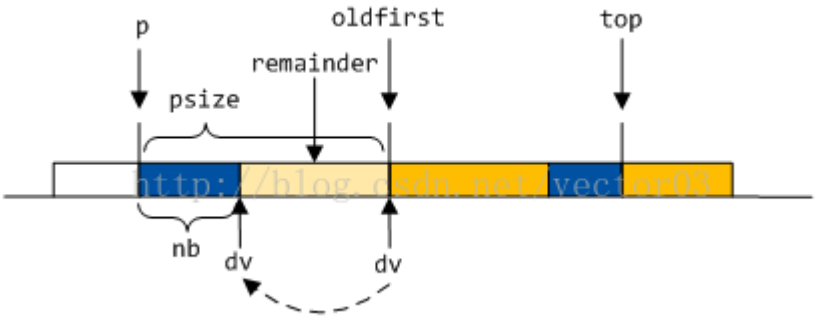
在3.4.1小节中介绍了从系统申请的扩展内存会根据其首地址和旧区段之间的位置关系做合并. 倘若append到区段后面, 申请内存是比较简单的, 因为扩展地址会直接补充到top中, 只需切割top即可. 但如果prepend到前面情况就相对复杂了, 因为从原区段base到top之间的情况不明, 所以必须分情况讨论. 而prepend_alloc函数就是为此而写的.

该函数会在一开始将分配请求从扩展空间中切割出来, 剩余工作就是根据不同情况对remainder做相应处理,

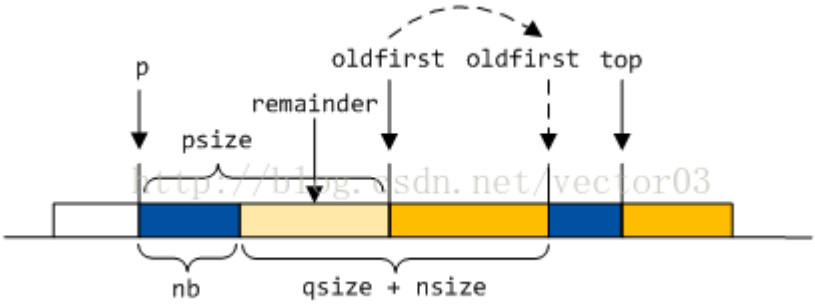
1. 如果旧区段base与top是同一个地址, 直接移动top指针, 将remainder吸收到top中.



2. 如果旧区段base与dv是同一地址, 则扩充dv的范围.



3. 若旧区段开始是普通的free chunk, 则移动oldfirst指针, 将remainder和free chunk合并.



4. 若旧区段开始是inused chunk, 则将remainder插入回分箱.

代码注释如下,

```
03954: static void* prepend_alloc(mstate m, char* newbase, char* oldbase,
03955:                            size_t nb) {
03956:   /* 获得扩展内存基址p以及旧区段基址oldfirst */
03957:   mchunkptr p = align_as_chunk(newbase);
03958:   mchunkptr oldfirst = align_as_chunk(oldbase);
03959:   size_t psize = (char*)oldfirst - (char*)p;
03960:   /* 切割, 获得remainder chunk指针q */
03961:   mchunkptr q = chunk_plus_offset(p, nb);
03962:   /* remainder chunk大小*/
03963:   size_t qsize = psize - nb;
03964:   set_size_and_pinuse_of_inuse_chunk(m, p, nb);
03965:   /* 检查地址合法 */
03966:   assert((char*)oldfirst > (char*)q);
03967:   assert(pinuse(oldfirst));
03968:   assert(qsize >= MIN_CHUNK_SIZE);
```

```
03969:   /* 若与top毗邻, 则吸收到top中 */
03970:   if (oldfirst == m->top) {
03971:     size_t tsize = m->topsize += qsize;
03972:     m->top = q;
03973:     q->head = tsize | PINUSE_BIT;
03974:     check_top_chunk(m, q);
03975:   }
03976:   /* 与dv毗邻, 扩展dv */
03977:   else if (oldfirst == m->dv) {
03978:     size_t dsize = m->dvsize += qsize;
03979:     m->dv = q;
03980:     set_size_and_pinuse_of_free_chunk(q, dsize);
03981:   }
03982:   else {
03983:     /* 普通free chunk, 与remainder合并 */
03984:     if (!is_inuse(oldfirst)) {
03985:       size_t nsize = chunksize(oldfirst);
03986:       /* 将free chunk从分箱中取出 */
03987:       unlink_chunk(m, oldfirst, nsize);
03988:       /* 移动oldfirst, 合并remainder */
03989:       oldfirst = chunk_plus_offset(oldfirst, nsize);
03990:       qsize += nsize;
03991:     }
03992:     /* 将remainder或合并后的remainder插入分箱系统 */
03993:     set_free_with_pinuse(q, qsize, oldfirst);
03994:     insert_chunk(m, q, qsize);
03995:     check_free_chunk(m, q);
03996:   }
03997:
03998:   check_malloced_chunk(m, chunk2mem(p), nb);
03999:   return chunk2mem(p);
04000: } ? end prepend_alloc ?
```


3.4.4 add_segment

对于无法合并的扩展内存区域, dlmalloc最后会将它们作为新的segment插入.

创建新segment按照如下步骤进行,

1. 首先, 根据top, 查找到当前top-most区段, 并且定位出在其结尾的隐藏chunk.

2. 将top重新初始化为新的segment的基址.

3. 将mstate中保存的旧top-most区段信息push到旧区段的隐藏chunk里. 并将新区段信息记录在mstate中.

4. 旧区段末尾写入一连串fenceposts.

5. 若旧区段剩余的top可用, 则将旧top重新插入分箱系统中.

源码注释如下,

```
03996: static void add_segment(mstate m, char* tbase, size_t tsize, flag_t mmapped) {
03997:   /* 保存旧top地址 */
03998:   char* old_top = (char*)m->top;
03999:   /* 找到旧top-most指针 */
04000:   msegmentptr oldsp = segment_holding(m, old_top);
04001:   /* 下面的计算就是定位旧top-most段末尾隐藏的chunk, 用来记录segment info */
04002:   char* old_end = oldsp->base + oldsp->size;
04003:   size_t ssize = pad_request(sizeof(struct malloc_segment));
04004:   char* rawsp = old_end - (ssize + FOUR_SIZE_T_SIZES + CHUNK_ALIGN_MASK);
04005:   size_t offset = align_offset(chunk2mem(rawsp));
04006:   char* asp = rawsp + offset;
04007:   char* csp = (asp < (old_top + MIN_CHUNK_SIZE))? old_top : asp;
04008:   /* sp即隐藏chunk的指针, ss为预先设计好的msegment指针 */
04009:   mchunkptr sp = (mchunkptr)csp;
04010:   msegmentptr ss = (msegmentptr)(chunk2mem(sp));
04011:   mchunkptr tnext = chunk_plus_offset(sp, ssize);
04012:   mchunkptr p = tnext;
04013:   int nfences = 0;
04014:
04015:   /* 将top重新指向新区段 */
04016:   init_top(m, (mchunkptr)tbase, tsize - TOP_FOOT_SIZE);
04017:
04018:   assert(is_aligned(ss));

04019:   set_size_and_pinuse_of_inuse_chunk(m, sp, ssize);
04020:   /* 将mstate中的旧区段信息push到旧区段末尾 */
04021:   *ss = m->seg;
04022:   /* 重新记录新的segment info */
04023:   m->seg.base = tbase;
04024:   m->seg.size = tsize;
04025:   m->seg.sflags = mmapped;
04026:   m->seg.next = ss;
04027:   /* 在旧区段已经不是top-most了, 在其末尾写入fenceposts */
04028:   for (;;) {
04029:     mchunkptr nextp = chunk_plus_offset(p, SIZE_T_SIZE);
04030:     p->head = FENCEPOST_HEAD;
04031:     ++nfences;
04032:     if ((char*)(&(nextp->head)) < old_end)
04033:       p = nextp;
04034:     else
04035:       break;
04036:   }
04037:   assert(nfences >= 2);
04038:   /* 将旧top重新插回分箱系统 */
04039:   if (csp != old_top) {
04040:     mchunkptr q = (mchunkptr)old_top;
04041:     size_t psize = csp - old_top;
04042:     mchunkptr tn = chunk_plus_offset(q, psize);
04043:     set_free_with_pinuse(q, psize, tn);
04044:     insert_chunk(m, q, psize);
04045:   }
04046:
04047:   check_top_chunk(m, m->top);
04048: } ? end add_segment ?
```

Line4016是top初始化函数, 该函数基本只是简单的信息记录, 并在末尾伪装隐藏chunk, 代码如下,

```
03910: static void init_top(mstate m, mchunkptr p, size_t psize) {
03911:   /* 确保对齐 */
03912:   size_t offset = align_offset(chunk2mem(p));
03913:   p = (mchunkptr)((char*)p + offset);
03914:   psize -= offset;
03915:   /* 在mstate中记录top信息 */
03916:   m->top = p;
03917:   m->topsize = psize;
03918:   p->head = psize | PINUSE_BIT;
03919:   /* 在末尾提前写入隐藏chunk的长度 */
03920:   chunk_plus_offset(p, psize)->head = TOP_FOOT_SIZE;
03921:   /* 重置trim计数 */
03922:   m->trim_check = mparams.trim_threshold;
03923: }
```