4. 释放与实现

释放过程相对分配就简单多了, 基本着重在chunk合并, top裁剪, segment释放上. dlmalloc中合并是减少外部碎片最有效的方法了.

4.1 dlfree

释放的主要过程就是根据用户传入的payload, 找到chunk指针, 然后分别检查前一个和后一个chunk是否可以合并. 这里唯一需要注意的就是与dv和top这些特殊chunk的交互.

基本流程如下,

1. 通过用户传入的mem指针计算出chunk指针p. 如果FOOTERS打开, 则通过magic计算出其所属的mspace指针, 并进行校验.

2. 若p是通过direct mmap生成的, 则还原其头尾的fake chunk后直接munmap释放并结束. 详细内容请参考3.4.2小节的说明.

3. 若p的prev chunk也是free chunk则将p和prev合并. 若prev同时又是dv, 则还需要考虑p的next chunk. 假如next是inused chunk, 则直接将合并后的p替换为新的dv并返回, 否则进入下一步.

4. 若p的next chunk也是空闲的, 则又分为三种情况,

    a. next是普通的free chunk, 与p进行合并. 如果p同时是dv, 则更新dv.

    b. next同时是top, 则与p合并后更新top为p. 如果p同时又是dv, 则取消当前记录的dv(相当于dv被top吞并了). 若top已经超出trim阈值, 则执行sys_trim.

    c. next同时是dv, 则与p合并后更新dv为p.

5. 若p是经历前面步骤的普通chunk, 则将更新后的p重新插入分箱系统. 如果realse_check满足, 则检查并回收当前mspace下所有的free segment.

代码注释如下,

```
04704: void dlfree(void* mem) {
04705:   if (mem != 0) {
04706:     /* 计算mem所在chunkptr */
04707:     mchunkptr p  = mem2chunk(mem);
04708: #if FOOTERS
04709:     /* 获取p所在mspace指针, 并校验 */
04710:     mstate fm = get_mstate_for(p);
04711:     if (!ok_magic(fm)) {
04712:       USAGE_ERROR_ACTION(fm, p);
04713:       return;
04714:     }
04715: #else /* FOOTERS */
04716: #define fm gm
04717: #endif /* FOOTERS */
04718:     if (!PREACTION(fm)) {
04719:       check_inuse_chunk(fm, p);
04720:       if (RTCHECK(ok_address(fm, p) && ok_inuse(p))){
04721:         /* 当前chunksize */
04722:         size_t psize = chunksize(p);
04723:         /* next chunk指针 */
04724:         mchunkptr next = chunk_plus_offset(p, psize);
04725:         if (!pinuse(p)) {   /* 如果prev chunk是空闲的 */
04726:           /* 从当前chunk的prev_foot获取prev size */
04727:           size_t prevsize = p->prev_foot;
04728:           if (is_mmapped(p)) {  /* 如果p是直接mmap出来的 */
04729:             /* 获得当初mmap时的size */
04730:             psize += prevsize + MMAP_FOOT_PAD;
04731:             /* 直接munmap, 返回 */
04732:             if (CALL_MUNMAP((char*)p - prevsize, psize) == 0)
04733:               fm->footprint -= psize;
04734:             goto ↓postaction;


04735:           }
04736:           else {    /* p是普通free chunk的情况 */
04737:             /* 根据prevsize, 得到prev chunk指针 */
04738:             mchunkptr prev = chunk_minus_offset(p, prevsize);
04739:             /* 合并p和prev */
04740:             psize += prevsize;
04741:             p = prev;
04742:             if (RTCHECK(ok_address(fm, prev))) {
04743:               if (p != fm->dv) { /* 如果prev不是dv, 将prev从分箱中摘除 */
04744:                 unlink_chunk(fm, p, prevsize);
04745:               }
04746:               /* prev是dv, 且next不为free */
04747:               else if ((next->head & INUSE_BITS) == INUSE_BITS) {
04748:                 /* 将dv更新为p, 返回 */
04749:                 fm->dvsize = psize;
04750:                 set_free_with_pinuse(p, psize, next);
04751:                 goto ↓postaction;
04752:               }
04753:             }
04754:             else
04755:               goto ↓erroraction;
04756:           } ? end else ?
04757:         } ? end if !pinuse(p) ?
04758:         /* 检查next的情况 */
04759:         if (RTCHECK(ok_next(p, next) && ok_pinuse(next))) {
04760:           if (!cinuse(next)) {  /* 如果next为free */
04761:             if (next == fm->top) {  /* next同时又是top */
04762:               /* 将p和next合并, 且更新top为合并后的p */
04763:               size_t tsize = fm->topsize += psize;
04764:               fm->top = p;
```

```
04765:              p->head = tsize | PINUSE_BIT;
04766:              /* 如果此时p同时为dv, 则取消dv */
04767:              if (p == fm->dv) {
04768:                fm->dv = 0;
04769:                fm->dvsize = 0;
04770:              }
04771:              /* 如果top足够大, 则执行sys_trim */
04772:              if (should_trim(fm, tsize))
04773:                sys_trim(fm, 0);
04774:              goto ↓postaction;
04775:            }
04776:            else if (next == fm->dv) {  /* 如果next为dv */
04777:              /* 将p与next合并, 且更新dv为合并后的p */
04778:              size_t dsize = fm->dvsize += psize;
04779:              fm->dv = p;
04780:              set_size_and_pinuse_of_free_chunk(p, dsize);
04781:              goto ↓postaction;
04782:            }
04783:            else {  /* 如果next是普通的free chunk */
04784:              /* 将next与p合并 */
04785:              size_t nsize = chunksize(next);
04786:              psize += nsize;
04787:              /* 将next从分箱系统中摘除 */
04788:              unlink_chunk(fm, next, nsize);
04789:              set_size_and_pinuse_of_free_chunk(p, psize);
04790:              if (p == fm->dv) {    /* 如果p同时是dv, 则更新dv */
04791:                fm->dvsize = psize;
04792:                goto ↓postaction;
04793:              }
04794:            }

04795:          } ? end if !cinuse(next) ?
04796:          else
04797:            set_free_with_pinuse(p, psize, next);
04798:          /* 如果p为普通free chunk, 将合并后的p重新插回分箱系统 */
04799:          if (is_small(psize)) {
04800:            insert_small_chunk(fm, p, psize);
04801:            check_free_chunk(fm, p);
04802:          }
04803:          else {
04804:            tchunkptr tp = (tchunkptr)p;
04805:            insert_large_chunk(fm, tp, psize);
04806:            check_free_chunk(fm, p);
04807:            /* 如果release_check计数足够, 则进行mspace中的free segment检查 */
04808:            if (--fm->release_checks == 0)
04809:              release_unused_segments(fm);
04810:          }
04811:          goto ↓postaction;
04812:        } ? end if RTCHECK(ok_next(p,nex... ?
04813:      } ? end if RTCHECK(ok_address(fm... ?
04814:    erroraction:
04815:      USAGE_ERROR_ACTION(fm, p);
04816:    postaction:
04817:      POSTACTION(fm);
04818:    } ? end if !PREACTION(fm) ?
04819:  } ? end if mem!=0 ?
04820: #if !FOOTERS
04821: #undef fm
04822: #endif /* FOOTERS */
04823: } ? end dlfree ?
```


4.2 sys_trim

当dlmalloc在执行free请求时, 会检测当前top剩余空间是否超出trim_check规定的阈值. 如果是就会尝试收缩当前的top空间. 默认情况下, dlmalloc会保留一个粒度(granularity)大小的空间, 剩余的都将归还给系统, 可以传入参数pad指定额外的剩余空间(多数情况下是0). 另外, 由于top所在区段有可能位于heap区或mmap区, 因此也会有不同的收缩方式. 对于heap

区的top空间, 采取反向MORECORE的方式, 而对于mmap区的, 则先尝试用mremap进行收缩, 如果失败则使用mumap释放掉. 假设遇到trim失败的情况, dlmalloc就会自动关闭auto-trimming功能.

源码注释如下,

```
04317: static int sys_trim(mstate m, size_t pad) {
04318:   size_t released = 0;
04319:   ensure_initialization();
04320:   if (pad < MAX_REQUEST && is_initialized(m)) {
04321:     /* 保证top结尾的隐藏chunk */
04322:     pad += TOP_FOOT_SIZE;
04323:
04324:     if (m->topsize > pad) {
04325:       size_t unit = mparams.granularity;
04326:       /* 计算可收缩size, 减去pad区域, 并对齐unit上, 至少保留一个unit */
04327:       size_t extra = ((m->topsize - pad + (unit - SIZE_T_ONE)) / unit -
04328:                       SIZE_T_ONE) * unit;
04329:       /* 查找当前top所在segment */
04330:       msegmentptr sp = segment_holding(m, (char*)m->top);
04331:
04332:       if (!is_extern_segment(sp)) {     /* 若该seg不是外部分配的(外部分配的无法收缩) */
04333:         if (is_mmapped_segment(sp)) {   /* 若该seg位于mmap区 */
04334:           if (HAVE_MMAP &&              /* 允许MMAP */
04335:               sp->size >= extra &&      /* 当前段大小大于计算收缩大小 */
04336:               !has_segment_link(m, sp)) { /* 验证该seg是否为top-most */
04337:             size_t newsize = sp->size - extra;  /* trim后剩余segment size */
04338:             (void)newsize; /* 消除编译器warning */
04339:             /* 首先尝试mremap, 若失败则使用munmap */
04340:             if ((CALL_MREMAP(sp->base, sp->size, newsize, 0) != MFAIL) ||
04341:                 (CALL_MUNMAP(sp->base + newsize, extra) == 0)) {
04342:               /* 成功则记录下释放的空间 */
04343:               released = extra;
04344:             }
04345:           }
04346:         }

04347:         else if (HAVE_MORECORE) {   /* 若该seg位于heap区, 且允许MORECORE */
04348:           if (extra >= HALF_MAX_SIZE_T) /* 避免extra过大产生负值 */
04349:             extra = (HALF_MAX_SIZE_T) + SIZE_T_ONE - unit;
04350:           ACQUIRE_MALLOC_GLOBAL_LOCK();
04351:           {
04352:             /* 记录当前break位置 */
04353:             char* old_br = (char*)(CALL_MORECORE(0));
04354:             if (old_br == sp->base + sp->size) {    /* 保证top结尾在当前break的地方 */
04355:               char* rel_br = (char*)(CALL_MORECORE(-extra));    /* 释放内存 */
04356:               char* new_br = (char*)(CALL_MORECORE(0));     /* 记录新的break位置 */
04357:               if (rel_br != CMFAIL && new_br < old_br)
04358:                 released = old_br - new_br;                 /* 成功则记录释放的空间 */
04359:             }
04360:           }
04361:           RELEASE_MALLOC_GLOBAL_LOCK();
04362:         }
04363:       } ? end if !is_extern_segment(sp) ?
04364:       if (released != 0) {  /* 若前面成功释放 */
04365:         /* 更新当前段size */
04366:         sp->size -= released;
04367:         /* 更新footprint */
04368:         m->footprint -= released;
04369:         /* 更新top */
04370:         init_top(m, m->top, m->topsize - released);
04371:         check_top_chunk(m, m->top);
04372:       }
04373:     } ? end if m->topsize>pad ?
04374:     /* 同时尝试释放所有空闲区段 */
04375:     if (HAVE_MMAP)
04376:       released += release_unused_segments(m);
04377:
04378:     /* 若释放失败, 则自动关闭auto-trimming */
04379:     if (released == 0 && m->topsize > m->trim_check)
04380:       m->trim_check = MAX_SIZE_T;
04381:   } ? end if pad<MAX_REQUEST&&is_i... ?
04382:   return (released != 0)? 1 : 0;
04383: } ? end sys_trim ?
```

4.3 release_unused_segments

尽管有auto-trimming压缩top空间, 但多数情况下, 只依靠这种方法是无法满足内存释放需求的. 尤其是当外部碎片导致top不连续的情况下, auto-trimming可能相当一段时间无法触发. 此时, dlmalloc就转而寻找内部可回收的空闲段. 由于查找空闲段是一个耗时操作, 且出现的频率较低, 所以实际上按照一个周期来进行此操作. 当周期计数为0时, 就调用release_unused_segments.

判断一个区段是否空闲也比较简单, 因为空闲chunk合并的原因, 若当前段的第一个chunk为空闲, 且其大小覆盖整个区段除隐藏区域的全部范围, 就可以判定该区段为空闲段. 接下来只要unlink空闲chunk, 且munmap该区段即可. 源码注释如下,

```
04270: static size_t release_unused_segments(mstate m) {
04271:   size_t released = 0;
04272:   int nsegs = 0;
04273:   /* pred记录前一个段指针 */
04274:   msegmentptr pred = &m->seg;
04275:   /* sp记录当前段指针 */
04276:   msegmentptr sp = pred->next;
04277:   while (sp != 0) {
04278:     /* 保存当前段的base, size和next */
04279:     char* base = sp->base;
04280:     size_t size = sp->size;
04281:     msegmentptr next = sp->next;
04282:     ++nsegs;
04283:     /* 若当前段位于mmap区且不为外部区段 */
04284:     if (is_mmapped_segment(sp) && !is_extern_segment(sp)) {
04285:       /* 获取当前段第一个chunk, 及其大小 */
04286:       mchunkptr p = align_as_chunk(base);
04287:       size_t psize = chunksize(p);
04288:       /* 若p为空闲chunk, 且p覆盖了当前段除隐藏区域外的范围, 则该段空闲 */
04289:       if (!is_inuse(p) && (char*)p + psize >= base + size - TOP_FOOT_SIZE) {
04290:         tchunkptr tp = (tchunkptr)p;
04291:         assert(segment_holds(sp, (char*)sp));
04292:         /* 若p同时为dv, 撤销dv */
04293:         if (p == m->dv) {
04294:           m->dv = 0;
04295:           m->dvsize = 0;
04296:         }
04297:         else {  /* 否则, 将p从treebins中摘除 */
04298:           unlink_large_chunk(m, tp);
04299:         }
```

```
04300:         /* munmap该段 */
04301:         if (CALL_MUNMAP(base, size) == 0) {
04302:           /* 更新released, 以及footprint */
04303:           released += size;
04304:           m->footprint -= size;
04305:           /* 将当前段从segment链表中摘除 */
04306:           sp = pred;
04307:           sp->next = next;
04308:         }
04309:         else { /* munmap失败, 重新插回p */
04310:           insert_large_chunk(m, tp, psize);
04311:         }
04312:       } ? end if !is_inuse(p)&&(char*)... ?
04313:     } ? end if is_mmapped_segment(sp.h ?
04314:     /* 若不允许segment traversal, 则只检测第一个区段 */
04315:     if (NO_SEGMENT_TRAVERSAL)
04316:       break;
04317:     /* sp指向下一个区段 */
04318:     pred = sp;
04319:     sp = next;
04320:   } ? end while sp!=0 ?
04321:   /* 重置计数器, 若当前segment链长度大于默认释放周期, 则使用链表长为初始值 */
04322:   m->release_checks = (((size_t) nsegs > (size_t) MAX_RELEASE_CHECK_RATE)?
04323:                        (size_t) nsegs : (size_t) MAX_RELEASE_CHECK_RATE);
04324:   return released;
04325: } ? end release unused segments ?
```