
默认基于Glibc2.34进行源码分析，但涉及多版本时，以具体指定为准

Malloc 算法

> malloc底层使用brk以及mmap实现，申请内存时先申请地址，只有真正访问时，发生缺页中断，内核才会真正分配物理页
>
> brk/mmap底层都是使用的vm_area_struct在管理地址空间

仅描述流程，细节方面需查看源码，下面的缩进表明了调用层次，函数入口__libc_malloc (size_t bytes)为第一层：

- bytes是否属于tcahce范围，且对应的tcachebin链表不为空，则取出tcache chunk并返回；否则，往下运行
    - 分配与释放都从头部，LIFO，后入先出，除了第一个chunk，链表后续chunk之间的指针加密，单链表，地址相邻的下一个chunk的prev_inuse总为1，每个线程都有自己的tcache
- arena_get获取arena av，调用_int_malloc (mstate av, size_t bytes)进行分配
    - 将bytes转换成能容纳bytes字节的最小chunk大小，后续都叫做nb(normal bytes)
    - 如果av==null，那么直接尝试通过sysmalloc进行分配，分配结束后，**返回**；否则，往下运行

```
//一般情况下av !=NULL;
//1.在arena数量已满但是当前线程thread_arena还没创建过，那么arena_get会返回null，并传递到这
//2.在arena数量未满但是地址空间不足以申请一个最小的heap(HEAP_MIN_SIZE)来创建arena，那么arena_get会返回null，并传递到这
//在上述情况下，会退化到直接通过sysmalloc来申请内存
if (__glibc_unlikely (av == NULL))
{
void *p = sysmalloc (nb, av);//sysmalloc handles malloc cases requiring more memory from the system
if (p != NULL)
    alloc_perturb (p, bytes);//如果perturb_byte存在，，对bytes字节的内容按特定字节进行memset，注意不包含nb中可能多出的字节
return p;
}
```

        - sysmalloc执行过程，细节参考源码

```
主要逻辑：（假设了topchunk不满足分配需要）
首先满足下面条件的进入try_mmap，尝试通过mmap进行分配：
1.在arena数量已满但是当前线程thread_arena还没创建过，那么arena_get会返回null，并传递到这
2.在arena数量未满但是地址空间不足以申请一个最小的heap(HEAP_MIN_SIZE)来创建arena，那么arena_get会返回null，并传递到这
3.av存在但是 nb字节书大于mmap_threshold且mmap region的数量没有达到最大值，尝试直接mmap映射，mmap_threshold 默认值：128K  最小值128k 最大值：32位512K 64位16\32M，通常32M
if(av == NULL || ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_threshold)&& (mp_.n_mmaps < mp_.n_mmaps_max))) //满足进行mmap分配，并返回申请区域；但是如果av==NULL且mmap失败，则直接返回NULL；其它则往下走

接着判断是否是非主分配区域
非主分配区：先尝试grow_heap，再尝试new_heap（会在oldheap topchunk处产生fencepost,来让一部分oldheap topchunk进行free），要是还是失败的话判断是否执行过try_mmap逻辑，若是没有则跳到try_mmap尝试直接映射
主分配区：正常情况下直接sbrk增长即可。少数情况下，会存在外部非glibc调用sbrk，造成区域不连续，oldtop需要加入fencepost来让oldtop部分尽量进行free；要么就无法sbrk，只能mmap补救，造成set_noncontiguous (main_arena)（默认是true的）；这部分非正常情况处理复杂，但很少见，暂不花费很多精力

最后分裂top,一部分作为chunk返回，一部分作为新的top
```

    - 如果nb属于fastbin范围，且fastbin对应的链表中存在chunk，则：
        - 将第一个fastbin chunk记录

        - 分配与释放都从头部，LIFO，后入先出，除了第一个chunk，链表后续chunk之间的指针加密，单链表，地址相邻的下一个chunk的prev_inuse总为1
    - 如果nb属于tcache范围，且nb大小对应的tcachebin链表未满（默认为7），且fastbin中chunk数量大于1，则尝试填充满对应tcachebin，直到fastbin为空或者tcachebin满
    - **返回**第一个记录的fastbin
- 如果nb属于smallbin范围，且smallbin对应的链表中存在chunk，则：
    - 将第一个smallchunk记录
        - 双链表，FIFO，从尾部分配，从头部插入，在链表中时地址相邻的下一个chunk的prev_inuse为0，分配后地址相邻的下一个chunk的prev_inuse为1，相同索引位置的双链表中的chunk大小相同
    - 如果nb属于tcache范围，且nb大小对应的tcachebin链表未满（默认为7），且smallbin中chunk数量大于1，则尝试填充满对应tcachebin，直到smallbin为空或者tcachebin满
    - **返回**第一个记录的smallbin
- 如果nb不属于fastbin/smallbin范围，那么将被视为“large”，按“large”更新idx索引，并且如果fastbin不为空，还会malloc_consolidate合并fastbin
    - malloc_consolidate逻辑

```
//从后往前，从大到小遍历所有fastbin，并对fastchunk的前后chunk尝试进行合并
//合并后的chunk如果不挨着topchunk则放入unsortedbin头部
//合并后的chunk如果挨着topchunk则并入topchunk
//合并过程中，前后chunk如果是freechunk，那么freechunk会从原先的bins链表（unsorted/smallbin/largebin）unlink
//为什么这里只需要尝试合并前后chunk就可以呢？可以参考前面推荐的B站视频，讲述了Allocated/Free chunk的递归分析
```

- ```
  //到这说明如果是smallbin,说明其对应的链表为空了，但没进行malloc_consolidate
  //或者如果是largebin，说明可能进行了malloc_consolidate(fastchunks存在合并，不存在则不合并)
  ```

- 处理unsortedbin，整个过程嵌套在两个循环中，具体细节可以参考源码，这里仅仅做一些主要部分总结

```
//此时初始化一些值
//如果后续循环填充过tcache，那么该变量就会设为1
int return_cached = 0;
//初始化为0，用于计数移出unsortedbin且不是填充tcache的chunk数
//当 tcache_unsorted_count > mp_.tcache_unsorted_limit 且 return_cached为1，那么就直接从tcache中取出chunk返回,防止处理unsortedbin过程占用太多时间
tcache_unsorted_count = 0;
for (;; )
{
      int iters = 0;
      //unsorted chunks bk不指向自己，说明列表非空;victim为列表尾部chunk;采用FIFO，头部放入，尾部取出
      //unsortedbin非空，进入while循环
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
      {
          ...
          size = chunksize (victim);//当前处理的victim的大小
          ...
```
```

```
          //第一部分：unsortedbin中唯一chunk就是lastremainder,申请的nb字节属于
smallbin范围，而且满足(size) > (nb + MINSIZE)，则拆分它，并返回chunk
          if (in_smallbin_range (nb) &&//说明申请nb属于small bin，nb是用户
req转换成chunk后的字节数
              bck == unsorted_chunks (av) &&//说明是unsortedbin中最后一个
chunk
              victim == av->last_remainder &&//说明这个chunk是
last_remainder :The remainder from the most recent split of a small
request 最近拆分的 small request 的剩余部分
              (unsigned long) (size) > (unsigned long) (nb + MINSIZE))//
说明可以拆分last remainder
          {.....  return p;}
          ...
          //第二部分：如果申请的nb字节刚好等于victim字节，且nb属于tcache范围，且
对应所属tcachebin未满，那么先填充tcache，return_cached置1;如果申请的nb字节刚好等
于victim字节，且（nb不属于tcache范围，或对应所属tcachebin已满),那么进行返回
           if (size == nb){
               #if USE_TCACHE
          if (tcache_nb && tcache->counts[tc_idx] < mp_.tcache_count)
        {//这一步说明申请的nb属于tcache范围，对应所属tcachebin未满，且当前
unsortedbin中处理的victim大小等同申请需要的大小
          tcache_put (victim, tc_idx);
          return_cached = 1;//标识填充过tcachebin,可以从tcachebin中返回
          continue;//while循环处理unsorted bin
        }
          else
        {
#endif      //到这说明申请的nb不属于tcache范围，且当前unsortedbin中处理的victim
大小等同申请需要的大小，则进行返回
            //或者说明申请的nb属于tcache范围，对应所属tcachebin已满，且当前
unsortedbin中处理的victim大小等同申请需要的大小，则进行返回
              check_malloced_chunk (av, victim, nb);//检测
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
#if USE_TCACHE
        }
#endif
        }

       //第三部分
       //如果size和nb不一样,会到此处,这里会将当前的victim分别放入对应的smallbin
和largebin中
          /* place chunk in bin */
          if (in_smallbin_range (size))//如果在small bin 范围内
            {//此处记录size对应的smallbin链表
              victim_index = smallbin_index (size);
              //后面最终会插入在bck fwd之间，位于smallbin的头部，这里先记录链表
相关信息
              bck = bin_at (av, victim_index);//待插入位置前一个地址;
              fwd = bck->fd;//带插入位置的后一个地址;
            }
          else
            {//large bin ; 每个索引位置的bin都包含了一个区间范围，其中chunk按大
小递减排序
              victim_index = largebin_index (size);
              bck = bin_at (av, victim_index);//待插入位置前一个地址 但只是初
步的值
```

```
                fwd = bck->fd;//带插入位置的后一个地址，但只是初步的值，默认这么设是是假定比头第一个chunk大来思考的，
                //后面代码会调整，直到位置满足largebin中chunk排序从大大小(对应从头到尾)
                ...
              }
              mark_bin (av, victim_index);//将bitmap置位,表示对应的bin存在chunk；bin为空时，不会改变置位，只有后面serach时发现bin为空才会清0
              ...
        //第四部分：当处理了足够多的unsorted_chunk后，从tcache中获取一个返回；主要用于提高分配速度，避免在unsorted chunk处理太长时间
       #if USE_TCACHE
       ++tcache_unsorted_count;
       if (return_cached
       && mp_.tcache_unsorted_limit > 0  //默认为0，无限制，不会提前通过tcahce返回；Maximum number of chunks to remove from the unsorted list, which aren't used to prefill the cache
       && tcache_unsorted_count > mp_.tcache_unsorted_limit)
     {
       return tcache_get (tc_idx);
     }
     #endif

#define MAX_ITERS       10000
        //超出10000次循环后，break while循环，停止unsortedbin处理，防止处理消耗太多时间
           if (++iters >= MAX_ITERS)
             break;
       }//while结尾
     #if USE_TCACHE
       /* If all the small chunks we found ended up cached, return one now.  */
       //如果上面所有unsortedbin chunk处理完，且tcahce放置过chunk
       //或者处理了10000次unsortedchunk，且tcache放置过chunk
       //则立马取出缓存中的一个进行返回
       if (return_cached)
     {
       return tcache_get (tc_idx);
     }
     #endif
     ...
     //binmap
     //topchunk处理
}
```

- 如果nb属于largebin范围，则尝试在nb对应的largebin中找到最小的满足要求的chunk;否则此处不执行，继续往下走
  - 如果找到了chunk，且chunksize>=nb+MINSIZE，则进行分裂，分裂后新的victim**返回**，而remainder加入到unsorted_chunks头部；否则不分裂，直接**返回**chunk
  - 如果没有找到chunk，则跳过此步骤，继续往下走
  - 注意：如果之前unsortedbin中不存在nb大小的chunk，那么可能走到这会仍然是smallrange范围内的bin，这种情况下此处不执行，往下走
- 到这，说明无论是smallbin还是largebin，nb字节对应bin都无法满足，则通过最小适配原则匹配
  - 利用binmap技术快速查找，另外必要时更新binmap信息

        - 如果找到了，就会依据chunk大小**决定是否直接返回，还是分裂后返回**（remainder会放入unsortedbin；如果nb还属于smallbin范围，那么remainder还会被设置为lastremainder）
    - 最小适配原则未匹配到，进入use_top区域（个人理解通常刚开始时，bin中不存在任何chunk，会进入此处通过topchunk进行分配）
        - 如果能topchunk能分配，则分裂top，一部分**返回**，一部分成为新的topchunk
        - top不够分配了，判断fastbin是否有chunk，如果有则尝试合并fastbin，则合并后再次进入for循环unosrtedbin处理
        - 上面两个条件都不满足，则直接尝试通过sysmalloc进行分配
            - sysmalloc参考前文描述
- 如果前一步骤获取到chunk，则成功返回；否则，调用arena_get_retry获取新的arena，再次调用_int_malloc进行分配，**返回**分配结果。注意：arena_get_retry执行时，有些情况只是更换arena尝试进行分配，但是thread_arena没有变化；有些情况，可能会更新thread_arena为新的arean，相当于切换了线程绑定的arena

> 算法中到处分布着chunk校验，由于过于零散，此处并没有进行描述，具体参考源码

Free算法

仅描述流程，细节方面需查看源码，下面的缩进表明了调用层次，函数入口__libc_free (void *mem)为第一层：

- 如果mem对应的chunk是通过mmap映射得到的，则按条件更新mmap_threshold以及trim_threshold，并对其进行munmap_chunk取消映射，**返回**；不然此处不执行，往下走

```
if (chunk_is_mmapped (p))                       /* release mmapped memory.
*/
    {
      /* See if the dynamic brk/mmap threshold needs adjusting.
     Dumped fake mmapped chunks do not affect the threshold.  */
      if (!mp_.no_dyn_threshold//默认mmap_threshold是动态的，除非用户手动设置非动
态
          && chunksize_nomask (p) > mp_.mmap_threshold  //p的chunksize大于动态
的mmap_threshold 且小于默认的DEFAULT_MMAP_THRESHOLD_MAX（32:512K/64:16\32M）
          && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX)
        {
          //利用p的chunksize更新mmap_threshold，该阈值只增不降；默认(32/64
位)128K,32位最大512K,64位最大16\32M
          mp_.mmap_threshold = chunksize (p);
          //trim_threshold影响systrim,说明mmap chunk释放的越大，说明大内存越多，越
不需要进行收缩，该阈值只增长不降
          //更新trim_threshold收缩阈值，为2倍mmap_threshold即默认(32/64位)256K，
32位最大1M,64位最大32\64M
          mp_.trim_threshold = 2 * mp_.mmap_threshold;
          LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                      mp_.mmap_threshold, mp_.trim_threshold);
        }
      munmap_chunk (p);
    }
```

- 对于非mmap的chunk，通过_int_free(mstate av, mchunkptr p, int have_lock)进行释放；此处调用时have_lock=0，表示av没有进入方法时占用mutex锁

- 如果属于tcache范围，且对应tcahcebin链表未满，则放入tcahce，**返回**；否则，往下走
- 如果 (不属于tcahce范围或者tcachebin已满)&(chunk size <= get_max_fast () 即属于fastbin)时，则放入fastbin，**返回**；否则，往下走
- 如果chunksize > get_max_fast ()，且不是mmap得到的chunk；否则，往下走
  - 尝试合并前后chunk，free即可合并
    - 如果next chunk不是topchunk，则合并后的chunk会放入unsortedbin；否则，往下走
    - 如果next chunk是topchunk，则合并后的chunk会并入topchunk；否则，往下走
    - 如果size（相邻free合并后释放的大小）>= fastbin合并阈值（FASTBIN_CONSOLIDATION_THRESHOLD：64K）
      - 如果fastbin中存在数据，则触发fastbin的合并操作；否则，往下走
      - 如果是回收的chunk属于main_arena，且top chunk的size > trim_threshold，则调用systrim收缩main_arena的topchunk;否则，往下走
        - systrim简单来说就是依据topchunksize向下页对齐的值为缩小量对topchunk进行缩减
      - 如果是回收的chunk不属于main_arena，则尝试调用heap_trim进行收缩
        - 从后往前遍历arena的所有heap，依次判断整个heap能否delete，能删除就delete_heap (heap)，遇到第一个不能删除就停止
        - 接下来的处理就类似main arena中topchunk收缩过程，不过主要收缩过程使用shrink_heap
    - **返回**
- 如果chunksize > get_max_fast ()，且是mmap得到的chunk，则munmap_chunk取消映射，**返回**；此处看似和Free算法中第一段分析出现了重复，实际上是因为_int_free方法不仅仅是__libc_free方法调用，还存在其它方法调用，因此需要此流程；但是对于__libc_free来说，该流程是不会经过的

> 算法中到处分布着chunk校验，由于过于零散，此处并没有进行描述，具体参考源码

Realloc算法

仅描述流程，细节方面需查看源码，下面的缩进表明了调用层次，函数入口__libc_realloc (void *oldmem, size_t bytes)为第一层：

- 如果申请字节bytes为0，oldmem不为NULL，则调用__libc_free (oldmem)释放，**返回**；否则，往下走
- 如果oldmem为NULL，则相当于__libc_malloc (bytes)申请内存，**返回**；否则，往下走
- 如果待回收chunk属于mmap chunk
  - 如果启用了mremap机制，则尝试使用mremap来重新分配内存，成功则**返回**；否则，往下走
  - 如果没有启用mremap机制
    - 如果oldchunk能够容纳申请的内存，那么什么也不做，直接利用oldchunk，**返回**；否则，往下走
    - 如果oldchunk无法容纳新申请的内存，那么执行申请__libc_malloc/拷贝memcpy/释放munmap_chunk达到目标效果，**返回**
- 如果待回收chunk不属于mmap chunk
  - 尝试调用_int_realloc处理

- 成功则**返回**
- 失败则尝试__libc_malloc申请内存，成功的话接着则执行拷贝memcpy与释放_int_free达到目标效果，**返回**

> 算法中到处分布着chunk校验，由于过于零散，此处并没有进行描述，具体参考源码

Calloc算法

仅描述流程，细节方面需查看源码，下面的缩进表明了调用层次，函数入口__libc_calloc (size_t n, size_t elem_size)为第一层：

- n*elem_size 结果转换存放bytes
- 利用bytes作为请求，通过_int_malloc申请内存，申请成功，往下走
    - 如果申请失败，通过arena_get_retry再次尝试获取并切换arena，然后再通过_int_malloc获取内存，若申请失败则返回
- 接下来通过memset以及一些特殊方法进行清0

> 算法中到处分布着chunk校验，由于过于零散，此处并没有进行描述，具体参考源码

数据结构

源码分析

Arena

定义

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);
  /* Flags (formerly in max_fast).  */
  int flags;
  /* Set if the fastbin chunks contain recently inserted free blocks.  */
  /* Note this is a bool but not all targets support atomics on booleans.  */
  int have_fastchunks;
  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];
  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;
  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;
  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];
  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];
  /* Linked list */
  struct malloc_state *next;
  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;//next_free仅构成freelist；而next既包含freelist数据，也包含非freelist数据
```

```
  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */
  INTERNAL_SIZE_T attached_threads;
  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  I NTERNAL_SIZE_T max_system_mem;
};
```

初始化

```
//无论是main_arean还是非main_arena，在刚创建的时候，结构体中默认值为0，除非进一步被初始化了；比如fastbin/last_remainder一开始元素都为NULL;
static struct malloc_state main_arena =//主arena定义
{
  .mutex = _LIBC_LOCK_INITIALIZER,
  .next = &main_arena,
  .attached_threads = 1
};
/* Thread specific data.  每个线程所绑定的arena;__thread表示每个线程都有一份独立的数据*/
static __thread mstate thread_arena attribute_tls_model_ie;

/*
   Initialize a malloc_state struct.
   This is called from ptmalloc_init () //第一次malloc时调用，会进一步初始化main arena
   or from _int_new_arena () when creating a new arena.
 */
static void malloc_init_state (mstate av)
{
  int i;
  mbinptr bin;

  /* Establish circular links for normal bins */
  //初始化bins，bin的头结点指向所在chunk自己，相当于初始化为空链表
  //chunk实际不存在，只是为了方便计算地址，而且用于表示链表头
  for (i = 1; i < NBINS; ++i)
    {
      bin = bin_at (av, i);
      bin->fd = bin->bk = bin;
    }
//MORECORE->sbrk
//MORECORE_CONTIGUOUS为1,就是说定义了sbrk（sbrk区域是连续的）
//如果定义了sbrk，那么非主分区默认就是非连续的，主分区默认是连续的；
//主分区第一次因为brk不足而mmap时，就会变成非连续了（但是如果是因为mmap_threshold而mmap则还是连续的）
#if MORECORE_CONTIGUOUS
  if (av != &main_arena)
#endif
  set_noncontiguous (av);//设置非连续标志；
  if (av == &main_arena)//av为主分区仅仅在第一次时会走此路径，设置global_max_fast
     //Maximum size of memory handled in fastbins.
     //在SIZE_SZ=4时，global_max_fast=64；在SIZE_SZ=8时，global_max_fast=128；通常为128
    set_max_fast (DEFAULT_MXFAST);
  atomic_store_relaxed (&av->have_fastchunks, false);//设置av->have_fastchunks=0
  //av->top = &(av->bins[0]) - offsetof (struct malloc_chunk, fd))
```

```
  //初始化top值为unsortedbin对应的链表头
  av->top = initial_top (av);
}

/* There is only one instance of the malloc parameters.  */
//这里是一些默认参数的值
static struct malloc_par mp_ =
{
  .top_pad = DEFAULT_TOP_PAD,//0  简单理解就是申请内存时，会比申请的多申请一些pad空间
  .n_mmaps_max = DEFAULT_MMAP_MAX,//65536 the maximum number of requests to
service using mmap
  .mmap_threshold = DEFAULT_MMAP_THRESHOLD,//128 * 1024
  .trim_threshold = DEFAULT_TRIM_THRESHOLD,//128 * 1024
#define NARENAS_FROM_NCORES(n) ((n) * (sizeof (long) == 4 ? 2 : 8))
  .arena_test = NARENAS_FROM_NCORES (1)
#if USE_TCACHE
  ,
  .tcache_count = TCACHE_FILL_COUNT,//7：一个tcahce bin中最多有7个chunk
  .tcache_bins = TCACHE_MAX_BINS,//64
  .tcache_max_bytes = tidx2usize (TCACHE_MAX_BINS-1),
  .tcache_unsorted_limit = 0 /* No limit.  */
#endif
};
```

获取

```
//获取arena并锁住它，通常用于多线程环境下
//首先尝试通过__thread mstate thread_arena 获取；
//如果没有获取到则通过arena_get2获取
//补充：在SINGLE_THREAD_P为真的单线程环境下，将只存在main_arena，将直接通过main_arena进行
_int_malloc，不需要arena_get
#define arena_get(ptr, size) do { \
      ptr = thread_arena;                                \
      arena_lock (ptr, size);                                \
  } while (0)

#define arena_lock(ptr, size) do {                           \
      if (ptr)                                         \
        __libc_lock_lock (ptr->mutex);                       \
      else                                             \
        ptr = arena_get2 ((size), NULL);                     \
  } while (0)

//首先从next_free字段构成的freelist中获取，如果没有获取到且没达到上限narenas_limit则尝试依
据size hint创建新的arena
//如果到达上限narenas_limit(32: n*2 64:n*8 n=cores)还没有可用的话，则reused_arena
(avoid_arena)，从arena_get调用arena_get2时avoid_arena为NULL，从arena_get_retry调用
时，avoid_arena为main_arena
//reused_arena：就是遍历arena,获取next链表中第一个能使用的arena，如果遇到的第一个是
avoid_arena，则返回avoid_arena.next
//补充：
//在多线程环境中某个父线程fork后，父子线程刚开始时共享内存空间，但只有被fork的线程子线程对应的
arena会被使用，其它的在子线程中不会被使用了，所以将他们放入了freelist
//类似的还有线程栈，只有当前线程栈还会被使用，其它的线程栈就不再会使用了，这些线程栈的空间也要被
回收
//这些操作发生在glibc的fork方法（不是系统调用）
static mstate
```

```
arena_get2 (size_t size, mstate avoid_arena)
{
  mstate a;

  static size_t narenas_limit;

  a = get_free_list ();
  if (a == NULL)
    {
      /* Nothing immediately available, so generate a new arena.  */
      if (narenas_limit == 0)
        {
          // There is only one instance of the malloc parameters.  static struct
malloc_par mp_  其中的一些参数收到mallopt影响
          if (mp_.arena_max != 0)
            narenas_limit = mp_.arena_max;
          else if (narenas > mp_.arena_test)
            {
              int n = __get_nprocs ();//获取核心数

              if (n >= 1)
                narenas_limit = NARENAS_FROM_NCORES (n);//32：n*2   64:n*8
              else
                /* We have no information about the system.  Assume two
                   cores.  */
                narenas_limit = NARENAS_FROM_NCORES (2);//32：  4    64:16
            }
        }
    repeat:;
      size_t n = narenas;
      /* NB: the following depends on the fact that (size_t)0 - 1 is a
         very large number and that the underflow is OK.  If arena_max
         is set the value of arena_test is irrelevant.  If arena_test
         is set but narenas is not yet larger or equal to arena_test
         narenas_limit is 0.  There is no possibility for narenas to
         be too big for the test to always fail since there is not
         enough address space to create that many arenas.  */
      if (__glibc_unlikely (n <= narenas_limit - 1))
        {
          if (catomic_compare_and_exchange_bool_acq (&narenas, n + 1, n))//CAS 如
果计数成功，则返回false，进入_int_new_arena
            goto repeat;
          a = _int_new_arena (size);//这里新建arena
      if (__glibc_unlikely (a == NULL))
            catomic_decrement (&narenas);//申请失败，则退回计数
        }
      else
      /* Lock and return an arena that can be reused for memory allocation.
         Avoid AVOID_ARENA as we have already failed to allocate memory in  it
and it is currently locked.  */
        a = reused_arena (avoid_arena);//简单理解就是遍历arena,获取第一个next能使用的
arena，如果遇到的第一个是avoid_arena，则返回avoid_arena.next
    }
  return a;
}

//这里新建的arena都是非main_arena,申请成功后会加入next链表
//size是一个hint告诉新建的arena的空间要能容下size
```

```
//但不是绝对，如果一个heap装不下，那么会新建一个最小的能容下元数据并满足相关对齐要求的arena及其附属的heap
static mstate
_int_new_arena (size_t size)
{
  mstate a;
  heap_info *h;
  char *ptr;
  unsigned long misalign;
  //sizeof(*h)等同于sizeof(heap_info)获取heap_info结构的大小
  //Create a new heap.这里很明显将mmap出来 的start地址作为heap_info的收地址
  //new heap区域被mmap匿名映射出来时区域中的值都初始化为0，因而arena中fastbin等都为0，其他值需要后续进一步设置
  h = new_heap (size + (sizeof (*h) + sizeof (*a) + MALLOC_ALIGNMENT),
                mp_.top_pad);//top_pad属于malloc Tunable parameters,DEFAULT_TOP_PAD
  if (!h)
    {//申请大小可能太大了，不能再一个heap放下，那么就考虑让后面_int_malloc来处理，先申请一个不包含申请内存的最小的heap
      /* Maybe size is too large to fit in a single heap.  So, just try
         to create a minimally-sized arena and let _int_malloc() attempt
         to deal with the large request via mmap_chunk().  */
      h = new_heap (sizeof (*h) + sizeof (*a) + MALLOC_ALIGNMENT, mp_.top_pad);
      if (!h)
        return 0;
    }
    //将arena设置为紧挨着heap_info位置的地方
  a = h->ar_ptr = (mstate) (h + 1);
  malloc_init_state (a);//初始化arena
  a->attached_threads = 1;
  /*a->next = NULL;*/
  a->system_mem = a->max_system_mem = h->size;/* mmap且读写区域的大小，h->size页对齐 */

  /* Set up the top chunk, with proper alignment. */
  ptr = (char *) (a + 1);
  misalign = (unsigned long) chunk2mem (ptr) & MALLOC_ALIGN_MASK;
  if (misalign > 0)
    ptr += MALLOC_ALIGNMENT - misalign;//修正ptr，让ptr对齐MALLOC_ALIGNMENT
  top (a) = (mchunkptr) ptr;//设置arena的top，top末端页对齐
  set_head (top (a), (((char *) h + h->size) - ptr) | PREV_INUSE);//初始化top_chunk的mchunk_size并且设置PREV_INUSE

  LIBC_PROBE (memory_arena_new, 2, a, size);
  mstate replaced_arena = thread_arena;//从这开始后面的操作基本上就是将新创建的arena加入next链表
  thread_arena = a;
  __libc_lock_init (a->mutex);
  __libc_lock_lock (list_lock);

  /* Add the new arena to the global list.  */
  a->next = main_arena.next;
  /* FIXME: The barrier is an attempt to synchronize with read access
     in reused_arena, which does not acquire list_lock while
     traversing the list.  */
  atomic_write_barrier ();
  main_arena.next = a;
```

```
  __libc_lock_unlock (list_lock);
  __libc_lock_lock (free_list_lock);
  detach_arena (replaced_arena);
  __libc_lock_unlock (free_list_lock);
  /* Lock this arena.  NB: Another thread may have been attached to
     this arena because the arena is now accessible from the
     main_arena.next list and could have been picked by reused_arena.
     This can only happen for the last arena created (before the arena
     limit is reached).  At this point, some arena has to be attached
     to two threads.  We could acquire the arena lock before list_lock
     to make it less likely that reused_arena picks this new arena,
     but this could result in a deadlock with
     __malloc_fork_lock_parent.  */
  __libc_lock_lock (a->mutex);
  return a;
}

 /* If we don't have the main arena, then maybe the failure is due to running
out of mmapped areas, so we can try allocating on the main arena.Otherwise, it is
likely that sbrk() has failed and there is still a chance  to mmap(), so try one
of the other arenas.*/
//arena_get (ar_ptr, bytes); victim = _int_malloc (ar_ptr, bytes);初次申请失败时进行如下补救
//基本等同于如果入参ar_ptr不是main_arena，则返回main_arena;
//否则，arena_get2的avoid_arena=main_arena来获取一个arena
static mstate arena_get_retry (mstate ar_ptr, size_t bytes)
{
  LIBC_PROBE (memory_arena_retry, 2, bytes, ar_ptr);
  if (ar_ptr != &main_arena)
    {
      __libc_lock_unlock (ar_ptr->mutex);
      ar_ptr = &main_arena;
      __libc_lock_lock (ar_ptr->mutex);
    }
  else
    {
      __libc_lock_unlock (ar_ptr->mutex);
      ar_ptr = arena_get2 (bytes, ar_ptr);
    }

  return ar_ptr;
}
```

Heap Info

定义

```
/* A heap is a single contiguous memory region holding (coalesceable)
   malloc_chunks.  It is allocated with mmap() and always starts at an
   address aligned to HEAP_MAX_SIZE.  */
typedef struct _heap_info
{
  mstate ar_ptr; /* Arena for this heap. */
  struct _heap_info *prev; /* Previous heap. */
  size_t size;   /* Current size in bytes. */
```

```
  size_t mprotect_size; /* Size in bytes that has been mprotected
                           PROT_READ|PROT_WRITE.  */
  /* Make sure the following data is properly aligned, particularly
     that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
     MALLOC_ALIGNMENT. */
  char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

新建

```
/* Create a new heap.  size is automatically rounded up to a multiple
   of the page size. */
//top_pad作用是在顶部多预申请一些内存
static heap_info* new_heap (size_t size, size_t top_pad)
{
  size_t pagesize = GLRO (dl_pagesize);
  char *p1, *p2;
  unsigned long ul;
  heap_info *h;

  if (size + top_pad < HEAP_MIN_SIZE)//32K
    size = HEAP_MIN_SIZE;
  else if (size + top_pad <= HEAP_MAX_SIZE)//32:1M   64:32\64M 通常是64M
    size += top_pad;
  else if (size > HEAP_MAX_SIZE)//超过允许值，直接返回NULL
    return 0;
  else//多预申请的内存不满足要求，size满足要求，直接申请最大值
    size = HEAP_MAX_SIZE;
  size = ALIGN_UP (size, pagesize);//对size进行页对齐

  /* A memory region aligned to a multiple of HEAP_MAX_SIZE is needed.
     No swap space needs to be reserved for the following large
     mapping (on Linux, this is the case for all non-writable mappings
     anyway). */
  p2 = MAP_FAILED;
  if (aligned_heap_area)//这个变量主要用于减少虚拟地址空间碎片，表明偏向于两个连续的
HEAP_MAX_SIZE挨着
    {
      //将aligned_heap_area作为hint，申请虚拟地址空间;
      //__mmap((addr), (size), (prot), (flags)|MAP_ANONYMOUS|MAP_PRIVATE, -1, 0)
      //MAP_ANONYMOUS：The mapping is not backed by any file; its contents are
initialized to zero. 匿名映射会被初始化为0
      p2 = (char *) MMAP (aligned_heap_area, HEAP_MAX_SIZE, PROT_NONE,
                          MAP_NORESERVE);
      aligned_heap_area = NULL;
      if (p2 != MAP_FAILED && ((unsigned long) p2 & (HEAP_MAX_SIZE - 1)))//p2如果
没有HEAP_MAX_SIZE对齐，则放弃这次映射
        {
          __munmap (p2, HEAP_MAX_SIZE);
          p2 = MAP_FAILED;
        }
    }
  if (p2 == MAP_FAILED)
    {
      //有内核随机分配，之后进行HEAP_MAX_SIZE对齐，为此将首尾多余的虚拟地址空间进行unmap
      p1 = (char *) MMAP (0, HEAP_MAX_SIZE << 1, PROT_NONE, MAP_NORESERVE);
```

```
      if (p1 != MAP_FAILED)
        {
          p2 = (char *) (((unsigned long) p1 + (HEAP_MAX_SIZE - 1))
                         & ~(HEAP_MAX_SIZE - 1));//heap区域必定HEAP_MAX_SIZE
          ul = p2 - p1;
          if (ul)
            __munmap (p1, ul);
          else
            aligned_heap_area = p2 + HEAP_MAX_SIZE;//紧挨着的这个p2的下一个
HEAP_MAX_SIZE地址用于下次尝试分配
          __munmap (p2 + HEAP_MAX_SIZE, HEAP_MAX_SIZE - ul);
        }
      else
        {
          /* Try to take the chance that an allocation of only HEAP_MAX_SIZE
             is already aligned. */
          p2 = (char *) MMAP (0, HEAP_MAX_SIZE, PROT_NONE, MAP_NORESERVE);
          if (p2 == MAP_FAILED)
            return 0;

          if ((unsigned long) p2 & (HEAP_MAX_SIZE - 1))
            {
              __munmap (p2, HEAP_MAX_SIZE);
              return 0;
            }
        }
    }
    //mprotect改变指定地址空间范围内的权限，如果有必要的话，一个VMA 可能因此分裂成多个VMA ，
不同VMA 设置不同权限
    //虽然申请了HEAP_MAX_SIZE的空间，但是只有size范围内的heap允许读写，其它部分无法使用
  if (__mprotect (p2, size, mtag_mmap_flags | PROT_READ | PROT_WRITE) != 0)
    {
      __munmap (p2, HEAP_MAX_SIZE);
      return 0;
    }
  h = (heap_info *) p2;
  h->size = size;//从这来看,size/mprotect_size都是真正最终可读写的内存区域，不包括分配了
虚拟地址但没有权限的区域
  h->mprotect_size = size;//h->size尾部对齐
  LIBC_PROBE (memory_heap_new, 2, h, h->size);
  return h;
}
```

增长

```
/* Grow a heap.  size is automatically rounded up to a multiple of the page size.
*/
//没有mmap新的region，只是将之前不可读写的区域变成可读写，没有mmap新的region
//如果增长会超过HEAP_MAX_SIZE，则放弃增长返回-1;通常增长发生在topchunk空间不够使用
//grow时，没有top_pad的需要
static int grow_heap (heap_info *h, long diff)
{
  size_t pagesize = GLRO (dl_pagesize);
  long new_size;
  diff = ALIGN_UP (diff, pagesize);//diff是还需的内存大小，这里直接按页向上对齐
  new_size = (long) h->size + diff;//即使之前存在多个heap，h->size只是这个heap的szie
  if ((unsigned long) new_size > (unsigned long) HEAP_MAX_SIZE)
```

```
    return -1;
  if ((unsigned long) new_size > h->mprotect_size)
    {
      if (__mprotect ((char *) h + h->mprotect_size,
                      (unsigned long) new_size - h->mprotect_size,
                      mtag_mmap_flags | PROT_READ | PROT_WRITE) != 0)
        return -2;

      h->mprotect_size = new_size;
    }
  h->size = new_size;
  LIBC_PROBE (memory_heap_more, 2, h, h->size);
  return 0;
}
```

Chunk

> Chunk的管理是有两种视角的
> 一种是所有的chunk在一段虚拟地址连续的区域内都是紧挨着的，不管是Alloc/Free，这种属于真实布局视角
> 一种是free chunk list视角，它们按照chunk的大小存放在各自的free bin链表中，通过fd/bk或者fd_nextsize/bk_nextsize连接起来
>
> 堆管理的4条（递归）规则,达到外部碎片最低的效果
> Definition: A->Allocated Chunk F->Free Chunk
> rule1：[A] -在Free一个[A]的时候，[A]前后都有可能是A/F -> [A/F][A][A/F]
> rule2: [F] -在Malloc一个[F]的时候，[F]前后都必须是A -> [A][F][A] 用来保证外部碎片最低，Max free of possibility，相邻的F会被Merge,递归这样操作，那么只会出现这种情形
> rule3: bottom底部边界[A/F]都可能
> rule4: top顶部边界[A/F]都可能
> 但是Glibc为提升速度，对于tcache chunk与fast bin chunk这种特殊chunk有不同于上述思想的处理过程，即使free也被视为Allocated chunk，对应地址上相邻的下一个chunk的prev_inuse也会仍然保持1，用于防止合并以及加速分配
> 该四条规则思想学习自B站视频：yaaangmin 第28期malloc第一种实现(先看)，可以结合27一起看：
> 28期：https://www.bilibili.com/video/BV1tL4y1Y72y?share_source=copy_web
> 27期：https://www.bilibili.com/video/BV1g64y1e7LW?share_source=copy_web

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).
*/
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead.
*/
  //fd:free chunk存在意义;   bk:仅为双向链表中存在意义;
  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
  //下面两者仅为large chunk存在意义
  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

```
/*
   malloc_chunk details:

    (The following includes lightly edited explanations by Colin Plumb.)

    Chunks of memory are maintained using a `boundary tag' method as
    described in e.g., Knuth or Standish.  (See the paper by Paul
    Wilson ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps for a
    survey of such techniques.)  Sizes of free chunks are stored both
    in the front of each chunk and at the end.  This makes
    consolidating fragmented chunks into bigger chunks very fast.  The
    size fields also hold bits representing whether chunks are free or
    in use.

    An allocated chunk looks like this:


    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of previous chunk, if unallocated (P clear)  |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             User data starts here...                          .
        .                                                               .
        .             (malloc_usable_size() bytes)                      .
        .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             (size of chunk, but used for application data)    |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of next chunk, in bytes                |A|0|1|
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    Where "chunk" is the front of the chunk for the purpose of most of
    the malloc code, but "mem" is the pointer that is returned to the
    user.  "Nextchunk" is the beginning of the next contiguous chunk.

    Chunks always begin on even word boundaries, so the mem portion
    (which is returned to the user) is also on an even word boundary, and
    thus at least double-word aligned.

    Free chunks are stored in circular doubly-linked lists, and look like this:

    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of previous chunk, if unallocated (P clear)  |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                     |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Forward pointer to next chunk in list             |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Back pointer to previous chunk in list            |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Unused space (may be 0 bytes long)                .
        .                                                               .
        .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of next chunk, in bytes                |A|0|0|
```

```
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    The P (PREV_INUSE) bit, stored in the unused low-order bit of the
    chunk size (which is always a multiple of two words), is an in-use
    bit for the *previous* chunk.  If that bit is *clear*, then the
    word before the current chunk size contains the previous chunk
    size, and can be used to find the front of the previous chunk.
    The very first chunk allocated always has this bit set,
    preventing access to non-existent (or non-owned) memory. If
    prev_inuse is set for any given chunk, then you CANNOT determine
    the size of the previous chunk, and might even get a memory
    addressing fault when trying to do so.

    The A (NON_MAIN_ARENA) bit is cleared for chunks on the initial,
    main arena, described by the main_arena variable.  When additional
    threads are spawned, each thread receives its own arena (up to a
    configurable limit, after which arenas are reused for multiple
    threads), and the chunks in these arenas have the A bit set.  To
    find the arena for a chunk on such a non-main arena, heap_for_ptr
    performs a bit mask operation and indirection through the ar_ptr
    member of the per-heap header heap_info (see arena.c).

    Note that the `foot' of the current chunk is actually represented
    as the prev_size of the NEXT chunk. This makes it easier to
    deal with alignments etc but can be very confusing when trying
    to extend or adapt this code.

    The three exceptions to all this are:

     1. The special chunk `top' doesn't bother using the
    trailing size field since there is no next contiguous chunk
    that would have to index off it. After initialization, `top'
    is forced to always exist.  If it would become less than
    MINSIZE bytes long, it is replenished.

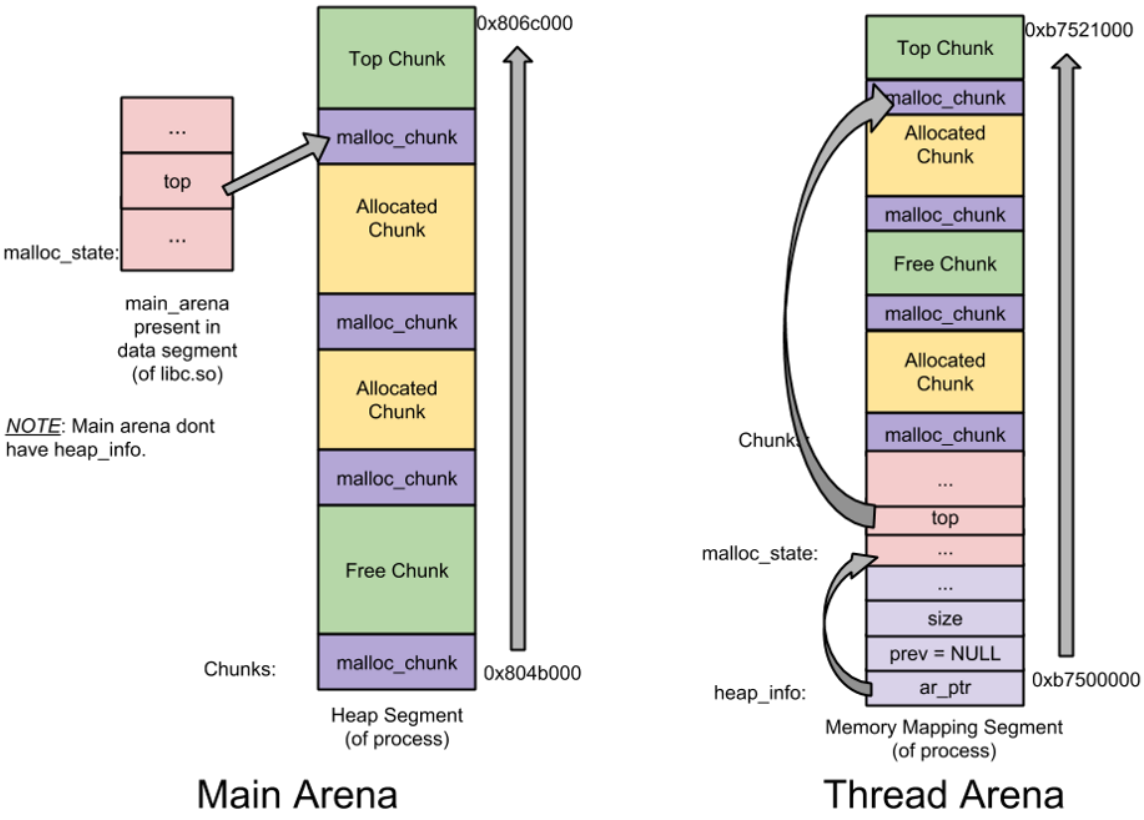
     2. Chunks allocated via mmap, which have the second-lowest-order
    bit M (IS_MMAPPED) set in their size fields.  Because they are
    allocated one-by-one, each must contain its own trailing size
    field.  If the M bit is set, the other bits are ignored
    (because mmapped chunks are neither in an arena, nor adjacent
    to a freed chunk).  The M bit is also used for chunks which
    originally came from a dumped heap via malloc_set_state in
    hooks.c.

     3. Chunks in fastbins are treated as allocated chunks from the
    point of view of the chunk allocator.  They are consolidated
    with their neighbors only in bulk, in malloc_consolidate.
*/
```

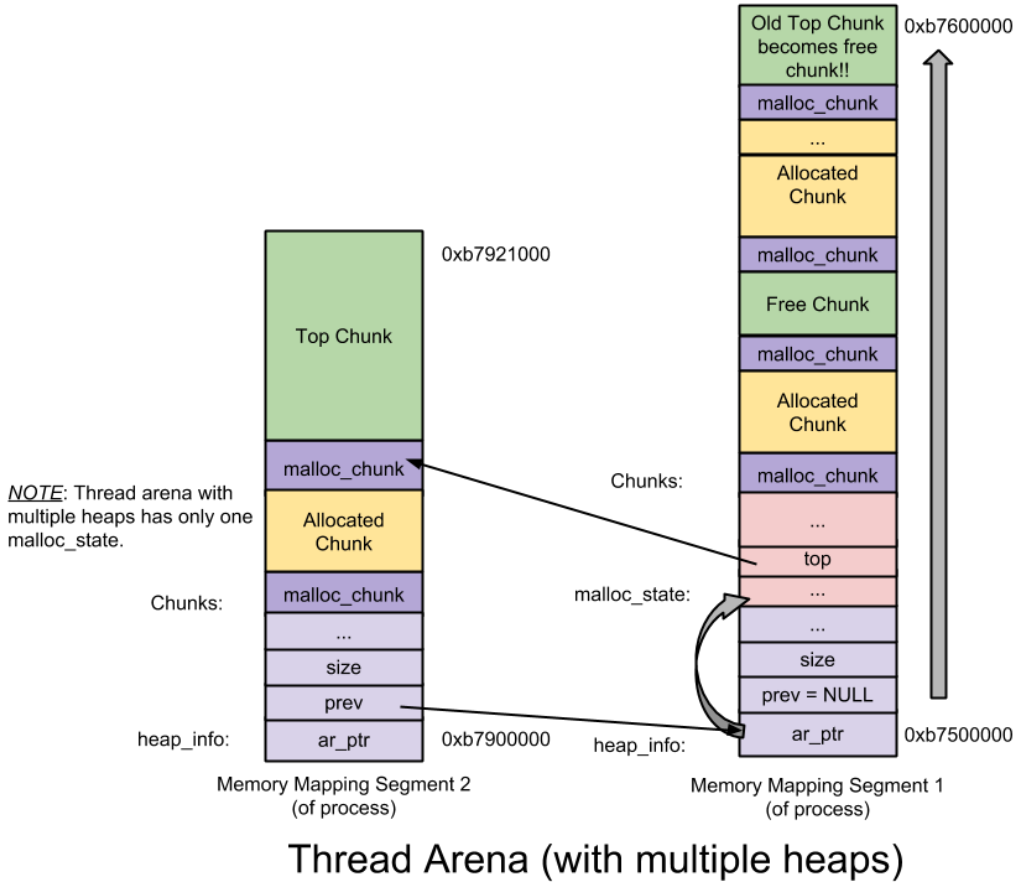
Arena&Heap Info&Chunk Relation

> Main arena 没有多个 heap 段，也因此没有 heap_info 结构。当 main arena 中堆空间耗尽了，就会调用 sbrk 来扩展堆空间（连续区域），直到顶端达到了内存映射段(即mmap段，实际上mmap的生长方向有不同的布局)。不同于线程 arena，main arena存储位置不属于 sbrk 的 heap 段。它是一个全局变量， 可以在 libc.so 的数据段中看到它的身影。
>
> Linux 2.6.7以前，经典布局：mmap向上/stack向下/heap(brk)向上 ，这样在x32位中brk增长会很快接触到mmap底部（一般mmap起始于0x4000000（即1GB））
>
> Linux 2.6.7以后，引入默认布局：mmap向下/stack向下/heap(brk)向上，这样在x32中brk就不会受到经典布局中1G的限制（mmap起始地址不再是0x4000000,而是靠近用户空间顶部的区域）
>
> x64中由于地址空间足够，默认采用的是经典布局：mmap向上/stack向下/heap(brk)向上

Main Arena & Thread Arena 单堆段



Thread Arena多堆段(所有ar_ptr都指向相同的arena，只是图上没有画出来)



> 上述两幅图引用自
>
> 英文版：Understanding glibc malloc – sploitF-U-N (wordpress.com)
>
> 翻译版：理解 glibc malloc：主流用户态内存分配器实现原理_猫科龙的博客-CSDN博客
>
> 结合一开始堆管理的4条（递归）规则来分析，其中的malloc_chunk个人理解是一些特殊的chunk，比如tcache/fastbin，这种chunk被free后，其对应prev_inuse仍然为1，不符合4条（递归）规则，因此才能出现在Free Chunk边上。（四条（递归）规则中明确要求了Free Chunk周围必须是Allocated Chunk，Glibc中打破这一思想的是tcache和fastbin，换种角度想，把它们单独先分配（视为Allocated），然后再独立做为一种快速缓存管理，实际也没有什么问题）

x32&x64 Variable

	32 bit value	64 bit value
INTERNAL_SIZE_T	32 bit typedef	32/64 bit typedef，通常64 bit
SIZE_SZ	4	4/8，通常是8
MALLOC_ALIGMENT 体系结构相关，但全篇都视为8/16进行分析	amd64-gcc-m32验证8	amd64-gcc验证16
MALLOC_ALIGN_MASK	7（0...0111）	15（0...01111）
MIN_CHUNK_SIZE	16	24/32，通常为32
MINSIZE	16	32
MAX_FAST_SIZE	80	80/160，通常是160
NFASTBINS	10	11/10，通常是10
NBINS	128	128
NSMALLBINS	64	64
CHUNK_HDR_SZ	8	8/16，通常是16
SMALLBIN_WIDTH	8	16
SMALLBIN_CORRECTION	0	1/0，通常是0
MIN_LARGE_SIZE	512	1008/1024，通常是1024
BINMAPSIZE	4	4
global_max_fast	64	64/128，通常是128
TCACHE_MAX_BINS	64	64
MAX_TCACHE_SIZE	516	1036/1032
DEFAULT_TOP_PAD	0	0
DEFAULT_MMAP_MAX	65536	65536
DEFAULT_MMAP_THRESHOLD	128 * 1024	128 * 1024
DEFAULT_MMAP_THRESHOLD_MIN	128 * 1024	128 * 1024
DEFAULT_MMAP_THRESHOLD_MAX	512 * 1024	16\32M，通常是32M
DEFAULT_TRIM_THRESHOLD	128 * 1024	128 * 1024
TCACHE_FILL_COUNT	7	7
HEAP_MIN_SIZE	32 * 1024	32 * 1024
HEAP_MAX_SIZE	1M	32\64M，通常是64M
MMAP_AS_MORECORE_SIZE	1024*1024	1024*1024

	32 bit value	64 bit value
tcache_unsorted_limit	默认0，代表无限制	默认0，代表无限制
no_dyn_threshold	默认0，代表动态mmap_threshold	默认0，代表动态mmap_threshold

> 64 bit value列中，当INTERNAL_SIZE_T为32 bit时，对应列中所有 '/' 左侧值，64bit时对应列中所有 '/' 右侧值，其它字段的变化都产生于INTERNAL_SIZE_T的区别；64bit列中'\'代表可能存在不同的值，但不是INTERNAL_SIZE_T的区别造成的，而是别的原因。

Source Code Analysis

- **INTERNAL_SIZE_T** Chunk Header元数据大小

```
# define INTERNAL_SIZE_T size_t
INTERNAL_SIZE_T might be signed or unsigned, might be 32 or 64 bits
On a 64-bit machine, you may be able to reduce malloc overhead by defining
INTERNAL_SIZE_T to be a 32 bit unsigned int at the expense of not being able
to handle more than 2^32 of malloced space.If this limitation is acceptable,
you are encouraged to set this unless you are on a platform requiring 16byte
alignments.
size_t等同于一种32/64bit的类型，INTERNAL_SIZE_T等效size_t也是类型定义
在32位中是32bit定义；
在64位中既可以是32bit定义,也可以是64bit定义，不过理论上64位一般而言不会进行限制，在amd64
linux环境下动态调试64位程序，默认就是64，也就是大部分都是64bit定义
```

- **SIZE_SZ** 等同于Chunk Header元数据大小

```
#define SIZE_SZ (sizeof (INTERNAL_SIZE_T))
等同于INTERNAL_SIZE_T的大小
```

- **MALLOC_ALIGNMENT** 对齐字节

```
#define MALLOC_ALIGNMENT (2 * SIZE_SZ < __alignof__ (long double) \
              ? __alignof__ (long double) : 2 * SIZE_SZ)
ANSI C标准规定了double变量存储为 IEEE 64 位（8 个字节）浮点数值，但并未规定long
double的确切精度
所以对于不同平台可能有不同的实现，有的是8字节，有的是10字节，有的是12字节或16字节
__alignof__类型通常需要的最小对齐，不同体系结构下即使相同的定义但结果也不一样
__alignof__ (long double) amd64体系结构下 gcc编译测试，32位程序为4，64位程序为16
因此，可在通常使用中认为MALLOC_ALIGNMENT在32程序是8，64位程序是16
```

- **MALLOC_ALIGN_MASK** 为了MALLOC_ALIGNMENT 对齐使用的掩码

```
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
32位为7 0111    64位为15 01111
```

- **MIN_CHUNK_SIZE** The smallest possible chunk

```
#define MIN_CHUNK_SIZE  (offsetof(struct malloc_chunk, fd_nextsize))
等同于 2*sizeof(INTERNAL_SIZE_T)+2*sizeof(malloc_chunk*)
在32位下：2*4 + 2*4 = 16
在64位下：2*（4 or 8） + 2*8 = 24 or 32， 由于sizeof(INTERNAL_SIZE_T)通常为8，所以结果通常为32
```

- **MIN_SIZE** The smallest size we can malloc is an aligned minimal chunk，MIN_CHUNK_SIZE 对齐的结果

```
#define MINSIZE  (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) &
~MALLOC_ALIGN_MASK))
32:  MIN_CHUNK_SIZE=16        MALLOC_ALIGN_MASK=7 （0...0111）
~MALLOC_ALIGN_MASK=(1...1000)
8字节对齐后结果 MINSIZE:16
64:  MIN_CHUNK_SIZE=24/32     MALLOC_ALIGN_MASK=15（0..01111）
~MALLOC_ALIGN_MASK=(1..10000)
16字节对齐后结果MINSIZE:32
```

- **MAX_FAST_SIZE**

```
#define MAX_FAST_SIZE   (80 * SIZE_SZ / 4)
```

- **NFASTBINS**

```
#define NFASTBINS  (fastbin_index (request2size (MAX_FAST_SIZE)) + 1)

32:MAX_FAST_SIZE=80
64:MAX_FAST_SIZE=80/160,通常160

//应该是转换成能容下请求req的合理的CHUNKSIZE大小
#define request2size(req)                                         \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)
32:req=80       SIZE_SZ=4     MALLOC_ALIGN_MASK=7（0...0111）      MINSIZE=16
request2size(req) = 88

64:
A    req=80   SIZE_SZ=4  MALLOC_ALIGN_MASK=15（0...01111）    MINSIZE=32
request2size(req) = 96
B    req=160  SIZE_SZ=8  MALLOC_ALIGN_MASK=15（0...01111）    MINSIZE=32
request2size(req) = 176

//sz 是Chunk的大小，通常会先用request2size转换成Chunk大小再计算索引
#define fastbin_index(sz)  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 :
3)) - 2)
32:sz=88    SIZE_SZ=4   fastbin_index(sz)=88/8-2=9
NFASTBINS=fastbin_index(sz)+1=10
64A:sz=96   SIZE_SZ=4   fastbin_index(sz)=96/8-2=10
NFASTBINS=fastbin_index(sz)+1=11
64B:sz=176  SIZE_SZ=8   fastbin_index(sz)=176/16-2=9
NFASTBINS=fastbin_index(sz)+1=10
```

- **NBINS**

```
#define NBINS    128
```

- **NSMALLBINS**

```
#define NSMALLBINS   64
```

- **CHUNK_HDR_SZ** Chunk Header包含头部两个元数据

```
#define CHUNK_HDR_SZ (2 * SIZE_SZ)
32:8   64:8/16
```

- **SMALLBIN_WIDTH**

```
#define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
32:8 64:16
```

- **SMALLBIN_CORRECTION**

```
#define SMALLBIN_CORRECTION (MALLOC_ALIGNMENT > CHUNK_HDR_SZ)
32: MALLOC_ALIGNMENT=8   CHUNK_HDR_SZ=8    SMALLBIN_CORRECTION=0
64: MALLOC_ALIGNMENT=16  CHUNK_HDR_SZ=8/16 SMALLBIN_CORRECTION=1/0，通常为0
```

- **MIN_LARGE_SIZE**

```
#define MIN_LARGE_SIZE    ((NSMALLBINS - SMALLBIN_CORRECTION) *
SMALLBIN_WIDTH)
32: NSMALLBINS=64   SMALLBIN_CORRECTION=0  SMALLBIN_WIDTH=8
MIN_LARGE_SIZE=512
64：NSMALLBINS=64   SMALLBIN_CORRECTION=1/0  SMALLBIN_WIDTH=16
MIN_LARGE_SIZE=1008/1024，通常是1024
```

- **BINMAPSIZE**

```
#define BINMAPSHIFT      5
#define BITSPERMAP       (1U << BINMAPSHIFT)
#define BINMAPSIZE       (NBINS / BITSPERMAP)
BINMAPSIZE=4
```

- **global_max_fast** Maximum size of memory handled in fastbins

```
//#define DEFAULT_MXFAST (64 * SIZE_SZ / 4)
set_max_fast (DEFAULT_MXFAST);//main_arena在第一次初始化时会调用，设置
global_max_fast
#define set_max_fast(s) \
global_max_fast = (((size_t) (s) <= MALLOC_ALIGN_MASK - SIZE_SZ)    \
                    ? MIN_CHUNK_SIZE / 2 : ((s + SIZE_SZ) &
~MALLOC_ALIGN_MASK))
 32:DEFAULT_MXFAST=64  MALLOC_ALIGN_MASK=7 SIZE_SZ=4
 global_max_fast=64
 64:DEFAULT_MXFAST=64/128,通常128  MALLOC_ALIGN_MASK=15  SIZE_SZ=4/8,通常为8
 因此，在SIZE_SZ=4时，global_max_fast=64；在SIZE_SZ=8时，global_max_fast=128；通
常为128
```

- **TCACHE_MAX_BINS**

```
/* We want 64 entries.  This is an arbitrary limit, which tunables can
reduce.  */
# define TCACHE_MAX_BINS        64
```

- **MAX_TCACHE_SIZE** 指的是用户实际可用的内存大小，不会影响元数据

```
# define MAX_TCACHE_SIZE    tidx2usize (TCACHE_MAX_BINS-1)

/* Only used to pre-fill the tunables.  */
//这么计算原因是将下一个chunkheader中第一个元数据在使用时可作为用户数据
# define tidx2usize(idx)    (((size_t) idx) * MALLOC_ALIGNMENT + MINSIZE -
SIZE_SZ)
32: SIZE_SZ=4   MINSIZE=16  MALLOC_ALIGNMENT=8
tidx2usize(64-1) = 63*8+16-4 =516
64: SIZE_SZ=4/8 MINSIZE=32  MALLOC_ALIGNMENT=16
当SIZE=4  tidx2usize(64-1)=63*16+32-4=1036
当SIZE=8  tidx2usize(64-1)=63*16+32-8=1032 通常是这种情况
```

- **DEFAULT_MMAP_THRESHOLD**

```
#define DEFAULT_MMAP_THRESHOLD DEFAULT_MMAP_THRESHOLD_MIN
#define DEFAULT_MMAP_THRESHOLD_MIN (128 * 1024)
```

- **DEFAULT_TRIM_THRESHOLD**

```
#define DEFAULT_TRIM_THRESHOLD (128 * 1024)
```

- **TCACHE_FILL_COUNT**

```
/* This is another arbitrary limit, which tunables can change.  Each
   tcache bin will hold at most this number of chunks.  */
# define TCACHE_FILL_COUNT 7
```

- **DEFAULT_MMAP_THRESHOLD_MAX**

```
# if __WORDSIZE == 32
#  define DEFAULT_MMAP_THRESHOLD_MAX (512 * 1024)// 32位 512K
# else
#  define DEFAULT_MMAP_THRESHOLD_MAX (4 * 1024 * 1024 * sizeof(long))//64位
16\32M  通常32M
# endif
```

- **HEAP_MIN_SIZE**

```
#define HEAP_MIN_SIZE (32 * 1024)
```

- **HEAP_MAX_SIZE**

```
#define HEAP_MAX_SIZE (2 * DEFAULT_MMAP_THRESHOLD_MAX)
32:1M  64:32\64M 通常是64M
```

- **request2size(req)**

```
//应该是转换成能容下请求req 的合理的CHUNKSIZE大小,考虑了chunk之后的下一个chunk的
prev_size空间做为req使用
#define request2size(req)                                         \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) &
~MALLOC_ALIGN_MASK)//MALLOC_ALIGN_MASK=2*SIZE-1
32: SIZE_SZ=4  MALLOC_ALIGN_MASK=7  MINSIZE=16  request2size(req)  req转换后最
小为16字节chunk
64: SIZE_SZ=4/8  MALLOC_ALIGN_MASK=15  MINSIZE=32  request2size(req)  req转换
后最小为32字节chunk
```

结构特征

tcache

- TCAHCE使用的前提是必须编译时启用了TCACHE机制，而且变量tcache_shutting_down=false
- malloc时首先会尝试通过TCACHE获取chunk，只有失败了，才会进入_int_malloc
- 每一个线程有独立的TCACHE(tcache_perthread_struct)结构，每个结构有默认TCACHE_MAX_BINS(64)数量的tcache bin
  - 当前线程第一次调用malloc时，会初始化tcache_perthread_struct，但是tcache中不存在任何chunk，**只有free才会填充?**
  - malloc申请chunk时最先尝试通过tcache获取
- 每一个tchace bin将会形成最多默认TCACHE_FILL_COUNT(7)数量的chunk
  - 单链表(利用chunk的fd加密后构成单链表)，后入先出LIFO，分配与释放都从头部，地址相邻的下一个的 chunk 的 prev_inuse总为1（在链表或被分配后都是如此，始终被视为Allocated chunk）
  - 默认情况下（至少在2.34版本下），单链表除了头部外，后续的指针都被PROTECT_PTR加密保护了地址
- per thread tcache bin [0,63]
  - 32位
    - 最小chunk16字节，按8字节增长，最大chunk520字节，共计64个tcache bin
  - 64位
    - 最小chunk32字节，按16字节增长，最大chunk1040字节，共计64个tcache bin
  - 通常用户申请大小x，会先执行request2size(x)转换成chunk大小(同时限定了最小的chunk)，再进行申请。request2size(x)认为chunk之后的下一个chunk的prev_size空间属于申请内存范围来进行计算应申请的chunk大小，使用时需要注意

分析

1. 数据结构
   - ```
     //__thread是gcc线程局部存储，每个线程有独立的该变量互不影响
     static __thread tcache_perthread_struct *tcache = NULL;

     typedef struct tcache_perthread_struct
     {
     ```
     ```

```
  uint16_t counts[TCACHE_MAX_BINS];//TCACHE_MAX_BINS=64,可调
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;

typedef struct tcache_entry
{
  struct tcache_entry *next;//这里对应chunk的fd
  /* This field exists to detect double frees.  */
  uintptr_t key;//对应bk，检测free两次异常，在链表中时是tcache_key，不在链表时被设置为0
} tcache_entry;
```

2. 行为算法

- index计算 依据request size/chunk size计算应该所属tcache链表的索引

```
/* When "x" is from chunksize().  */
# define csize2tidx(x) (((x) - MINSIZE + MALLOC_ALIGNMENT - 1) / MALLOC_ALIGNMENT)
//32: x>=MINSIZE MINSIZE=16  MALLOC_ALIGNMENT=8    可以看出TCACHE之间按照8字节增长，最小16字节
//64: x>=MINSIZE MINSIZE=32  MALLOC_ALIGNMENT=16   可以看出TCACHE之间按照16字节增长，最小32字节
/* When "x" is a user-provided size.  */
# define usize2tidx(x) csize2tidx (request2size (x))
//应该是转换成能容下请求req 的合理的CHUNKSIZE大小,考虑了chunk之后的下一个chunk的prev_size空间做为申请使用
#define request2size(req)                                         \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) &
~MALLOC_ALIGN_MASK)//MALLOC_ALIGN_MASK=2*SIZE-1
```

- 初始化

```
//每个线程在第一次malloc时，会初始化tcahce，此后tcahce不再为null，但还没有存在任何真正的tcache chunk
//free非mmap chunk时，也会进行调用
if (__glibc_unlikely (tcache == NULL))
    tcache_init();//通过arean_get/_int_malloc申请tcache_perthread_struct结构体，初始化为0并赋值tcache
static void tcache_init(void)
{
  mstate ar_ptr;
  void *victim = 0;
  const size_t bytes = sizeof (tcache_perthread_struct);

  if (tcache_shutting_down)
    return;
  //acquires an arena and locks the corresponding mutex.
  //先尝试获取thread_arena，不行就的话且narenas_limit没达到上限则申请一个新的arena,否则reused_arena利用现有的arena
  arena_get (ar_ptr, bytes);
  victim = _int_malloc (ar_ptr, bytes);//尝试通过ar_ptr获取请求为bytes字节的内存
  if (!victim && ar_ptr != NULL)
```

```
    {
      /* If we don't have the main arena, then maybe the failure is due
to running   out of mmapped areas, so we can try allocating on the main
arena. Otherwise, it is likely that sbrk() has failed and there is still
a chance  to mmap(), so try one of the other arenas.*/
      //arena_get (ar_ptr, bytes); victim = _int_malloc (ar_ptr, bytes);
初次申请失败时进行如下补救
    //基本等同于如果入参ar_ptr不是main_arena，则返回main_arena;
    //否则，arena_get2的avoid_arena=main_arena来获取一个arena
      ar_ptr = arena_get_retry (ar_ptr, bytes);
      victim = _int_malloc (ar_ptr, bytes);
    }

  if (ar_ptr != NULL)
    __libc_lock_unlock (ar_ptr->mutex);

  /* In a low memory situation, we may not be able to allocate memory
     - in which case, we just keep trying later.  However, we
     typically do this very early, so either there is sufficient
     memory, or there isn't enough memory to do non-trivial
     allocations anyway.  */
  if (victim)
    {
      tcache = (tcache_perthread_struct *) victim;//初始化tcache
      memset (tcache, 0, sizeof (tcache_perthread_struct));
    }
}
```

- 获取tcache chunk

```
//malloc时，最开始先尝试从tcache中获取
if (tc_idx < mp_.tcache_bins//tc_idx计算参考index计算，这个判断确定是否落在
tcache范围
      && tcache//tcache初始化过
      && tcache->counts[tc_idx] > 0)//对应的链表存在可用tcache chunk
    {
      victim = tcache_get (tc_idx);//直接从tcache中获取并返回
      return tag_new_usable (victim);//此方法和mtag/arm有关，除去这部分后相当
于直接返回参数指针
    }

static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];//取出第一个tcache chunk
  if (__glibc_unlikely (!aligned_OK (e)))//取出的chunk地址必须
MALLOC_ALIGNMENT对齐，后12位为原始地址，因此可以这样检测
    malloc_printerr ("malloc(): unaligned tcache chunk detected");
  tcache->entries[tc_idx] = REVEAL_PTR (e->next);//揭示下一个出链表的真实指
针地址并放入链表头；链表头是没有加密的
  --(tcache->counts[tc_idx]);//减少计数
  e->key = 0;//将bk置零，用于检测free double异常
  return (void *) e;//这个e对应的就是chunk的mem
}

//a^b^a=b   异或自反定律,这里(&ptr)>>12获的是ptr变量所在的页地址
//用这个具有随机性的地址掩码真实地址ptr变量值，得到一个加密后的地址放入tcache链表
```

```
//这样，如果想单纯的劫持fastbin tcache的元数据指针就变得很困难
#define PROTECT_PTR(pos, ptr) \
  ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))
#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)

static __always_inline void *
tag_new_usable (void *ptr)
  if (__glibc_unlikely (mtag_enabled) && ptr)
    {
      mchunkptr cp = mem2chunk(ptr);
      ptr = __libc_mtag_tag_region (__libc_mtag_new_tag (ptr), memsize
(cp));
    }
  return ptr;
}
```

- 填充tcache链表
  - 方式一：fastbin取出时，如果发现申请的字节nb（转成chunk后的字节）也属于tcachebin，那么还会直接从fastbin中获取chunk放入tcachebin，直到*fb链表空或者tcachebin满为止
  - 方式二：smallbin取出时，如果发现申请的字节nb（转成chunk后的字节）也属于tcachebin，那么还会直接从smallbin中获取chunk放入tcachebin，直到smallbin链表为空或者tcachebin满为止
  - 方式三：_libc_free在free chunk时，如果不是mmap chunk，且freechunsize 属于tcache范围，对应tcachebin未满，加入到tcache

```
#if USE_TCACHE  方式一，可以结合后面获取fastbin阅读，属于获取fastbin过程
          /* While we're here, if we see other chunks of the same
size,
         stash them in the tcache.  */
     //如果nb也属于tcache范围，那么将直接从fastbin中获取chunk放入
tcachebin,直到*fb链表空或者tcachebin满为止
          size_t tc_idx = csize2tidx (nb);
          if (tcache && tc_idx < mp_.tcache_bins)//nb属于tcache范围内
        {
          mchunkptr tc_victim;
          /* While bin not empty and tcache not full, copy chunks.
*/
          while (tcache->counts[tc_idx] < mp_.tcache_count
             && (tc_victim = *fb) != NULL)
            {
              if (__glibc_unlikely (misaligned_chunk (tc_victim)))
            malloc_printerr ("malloc(): unaligned fastbin chunk
detected 3");
              if (SINGLE_THREAD_P)
        //将fastbin中的后续chunk取出，放到tcache中，直到cache满或者fastbin
中下一个为NULL为止
            *fb = REVEAL_PTR (tc_victim->fd);
              else
            {
              REMOVE_FB (fb, pp, tc_victim);
              if (__glibc_unlikely (tc_victim == NULL))
                break;
            }
```

```
              tcache_put (tc_victim, tc_idx);
            }
        }
#endif

/* Caller must ensure that we know tc_idx is valid and there's room
for more chunks.  */
static __always_inline void tcache_put (mchunkptr chunk, size_t
tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);//说明cache-
>entries中放的都是mem而不是chunk，next都是指向的mem(实际就是chunk的fd)
  /* Mark this chunk as "in the tcache" so the test in _int_free
will
     detect a double free.  */
  e->key = tcache_key;
  //利用位置信息对指针信息进行保护
  e->next = PROTECT_PTR (&e->next, tcache->entries[tc_idx]);
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}

--------------------------------
方式二，可以结合后面获取smallbin阅读，属于获取smallbin过程
if (in_smallbin_range (nb))//在smallbin范围
    {
      ........
#if USE_TCACHE
      /* While we're here, if we see other chunks of the same size,
         stash them in the tcache.  */
      size_t tc_idx = csize2tidx (nb);
      if (tcache && tc_idx < mp_.tcache_bins)//如果启用了tcache机制，并
且属于tc_index在tcache范围内
        {
          mchunkptr tc_victim;
          /* While bin not empty and tcache not full, copy chunks
over.  */
          while (tcache->counts[tc_idx] < mp_.tcache_count
             && (tc_victim = last (bin)) != bin)//tcache未满且
smallbin未空，则类似fastbin中的处理，将small list中chunk移除并放到tcache
中;相比fastbin，多了一步设置prev_inuse;
        {
          if (tc_victim != 0)
            {
              bck = tc_victim->bk;
              set_inuse_bit_at_offset (tc_victim, nb);//fastbin对应此
处没有这个操作，是因为fastbin/tcache的prev_inuse都为1
              if (av != &main_arena)
            set_non_main_arena (tc_victim);
              bin->bk = bck;
              bck->fd = bin;

              tcache_put (tc_victim, tc_idx);
                }
        }
        }
#endif
          void *p = chunk2mem (victim);
```

```
          alloc_perturb (p, bytes);
          return p;
        }
    }
```

方式三：_libc_free在free chunk时，如果不是mmap chunk，且freechunsize 属于tcache范围，对应tcachebin未满，加入到tcache

```
#if USE_TCACHE//启用了tcache机制
  {
    size_t tc_idx = csize2tidx (size);
    if (tcache != NULL && tc_idx < mp_.tcache_bins)//tcache初始化了并且size属于tcache范围
      {
    /* Check to see if it's already in the tcache.  */
    tcache_entry *e = (tcache_entry *) chunk2mem (p);

    /* This test succeeds on double free.  However, we don't 100%
       trust it (it also matches random payload data at a 1 in
       2^<size_t> chance), so verify it's not an unlikely
       coincidence before aborting.  */
    if (__glibc_unlikely (e->key == tcache_key))//tcache_put时会设置tcache_key;如果出现了相等的巧合，遍历链表来确认是否为double free
      {
        tcache_entry *tmp;
        size_t cnt = 0;
        LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
        for (tmp = tcache->entries[tc_idx];
         tmp;
         tmp = REVEAL_PTR (tmp->next), ++cnt)//遍历对应的tcache链表
          {
        if (cnt >= mp_.tcache_count)//检测是否有过多的tcache_count
          malloc_printerr ("free(): too many chunks detected in tcache");
        if (__glibc_unlikely (!aligned_OK (tmp)))//对齐检测
          malloc_printerr ("free(): unaligned chunk detected in tcache 2");
        if (tmp == e)//检测到double free
          malloc_printerr ("free(): double free detected in tcache 2");/* Abort with an error message.  */
        /* If we get here, it was a coincidence.  We've wasted a
           few cycles, but don't abort.  */
          }
      }

    if (tcache->counts[tc_idx] < mp_.tcache_count)//tcahce未满
      {
        tcache_put (p, tc_idx);//释放到tchach put,这时prev_inuse为1
        return;
      }
      }
  }
#endif
```

3. 利用方式
    - 待完善

fastbin

- Fastbin机制目前来看，除非源码不存在相关代码，不然一直都是开启的
- malloc时无法通过TCACHE获取chunk，失败后会进入_int_malloc，在该方法中：
  - 如果arean为null，那么会调用sysmalloc（尝试通过mmap获取并返回），否则首先尝试Fastbin获取
- malloc_state->fastbinsY[NFASTBINS]
  - 32位
    - NFASTBINS=10，fastbin_index[0,9]，理论共计10个fastbin
    - 最小chunk16字节，按8字节增长，理论最大chunk88字节，实际最大可用chunk为64字节，实际索引范围fastbin_index[0,6]，实际可用共计7个fastbin
      - nb(chunk大小) <= get_max_fast ()判断是否属于fastbin，32位默认情况下get_max_fast ()为64字节
  - 64位
    - SIZE_SZ=4
      - NFASTBINS=11，fastbin_index[2,10]，理论共计9个fastbin可用
      - 最小chunk32字节，按8字节增长，理论最大chunk96字节，实际最大chunk为64字节，实际索引范围fastbin_index[2,6]，实际可用共计5个fastbin
        - nb(chunk大小) <= get_max_fast ()判断是否属于fastbin，64位且SIZE_SZ=4情况下默认get_max_fast ()为64字节
    - SIZE_SZ=8
      - NFASTBINS=10，fastbin_index[0,9]，共计10个fastbin可用
      - 最小chunk32字节，按16字节增长，理论最大chunk176字节，实际最大chunk为128字节，实际索引范围fastbin_index[0,6]，实际可用共计7个fastbin
        - nb(chunk大小) <= get_max_fast ()判断是否属于fastbin，64位且SIZE_SZ=8情况下默认get_max_fast ()为128字节
  - 通常用户申请大小x，会先执行request2size(x)转换成chunk大小(同时限定了最小的chunk)，再进行申请。request2size(x)认为chunk之后的下一个chunk的prev_size空间属于申请内存范围来进行计算应申请的chunk大小，使用时需要注意
- 单向链表(利用chunk的fd加密后构成单链表)，分配与释放都从头部，LIFO后入先出，地址相邻的下一个的 chunk 的 prev_inuse总为1（在链表或被分配后都是如此，始终被视为Allocated chunk），单链表除头部外，后续的指针都被PROTECT_PTR加密保护了地址

分析

1. 数据结构

```
//每一个malloc_state(arena)中存在mfastbinptr fastbinsY[NFASTBINS],其中每一个fastbin是由malloc_chunk*构成的链表
//typedef struct malloc_chunk *mfastbinptr;
//NFASTBINS   32位长度10  ;  64位长度11/10，通常是10；
```

2. 行为算法
   - index计算

```
typedef struct malloc_chunk *mfastbinptr;
//ar_ptr 即arena ptr ，数据结构：malloc_state
//从mstate获取指定索引的fastbin
#define fastbin(ar_ptr, idx) ((ar_ptr)->fastbinsY[idx])

/* offset 2 to use otherwise unindexable first 2 bins */
//通过sz获取所属fastbin的索引
//-2的偏移，是为了让最小chunk size除完后能够定位到索引0,类似这样的作用
//也可以推出  SIZE_SZ=8时，sz最小为32字节，不然无法定位到0
//                      SIZE_SZ=4时，sz最小为16字节，不然无法定位到0
//sz 是Chunk的大小，通常会先用request2size转换成Chunk大小再计算索引
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)
```

- 初始化

```
main_arena 后调用 malloc_init_state
new_heap 后调用  malloc_init_state
arean在初始化后，fastbinsY中都为元素0，即NULL
```

- 获取fastbin

```
#define REMOVE_FB(fb, victim, pp)           \
  do                                \
    {                               \
      victim = pp;                      \
      if (victim == NULL)                   \
    break;                          \//获取解除指针保护的原始指针
      pp = REVEAL_PTR (victim->fd);
\
      if (__glibc_unlikely (pp != NULL && misaligned_chunk (pp)))
 \//检测对齐
    malloc_printerr ("malloc(): unaligned fastbin chunk detected"); \
    }                               \//CAS 操作直到成功从FASTBIN中获取到一个
victim(malloc_chunk)
  while ((pp = catomic_compare_and_exchange_val_acq (fb, pp, victim)) \
     != victim);                        \

  if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))//在
fastbin范围内
    {
      idx = fastbin_index (nb);//nb to fastbin index
      mfastbinptr *fb = &fastbin (av, idx);//获取第一个指向fastbin的指针
      mchunkptr pp;
      victim = *fb;//victim，指向fastbin的指针（实际就是一个malloc_chunk）

      if (victim != NULL)//指向的malloc_chunk存在
    {
      if (__glibc_unlikely (misaligned_chunk (victim)))//对齐检查
        malloc_printerr ("malloc(): unaligned fastbin chunk detected
2");

      if (SINGLE_THREAD_P)
        *fb = REVEAL_PTR (victim->fd);//单线程环境版本
      else
```

```
        REMOVE_FB (fb, pp, victim);//多线程版本，让fb指向下一个
malloc_chunk，victim指向取出的fastbin；分配与释放都从头部，LIFO，后入先出
      if (__glibc_likely (victim != NULL))//取出的fastbin存在
        {
          size_t victim_idx = fastbin_index (chunksize (victim));//重新获
取victim所对应的fastbin索引
          if (__builtin_expect (victim_idx != idx, 0))//结果很可能人为为
假，一般就是应该是一样的
        malloc_printerr ("malloc(): memory corruption (fast)");
          check_remalloced_chunk (av, victim, nb);//依据当前使用的arean 取
出的fastbin  规范化的bytes进行一系列检查，也是防止人为利用
          //后续...部分：如果nb也属于tcache范围，那么将直接从fastbin中获取chunk
放入tcache,直到*fb链表空或者tcachebin满为止；可以查看前面tcache部分的填充部分
          ....
        }
#endif
          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;//将找到的chunck返回
        }
    }
    }

static inline INTERNAL_SIZE_T get_max_fast (void)
{
  /* Tell the GCC optimizers that global_max_fast is never larger
     than MAX_FAST_SIZE.  This avoids out-of-bounds array accesses in
     _int_malloc after constant propagation of the size parameter.
     (The code never executes because malloc preserves the
     global_max_fast invariant, but the optimizers may not recognize
     this.)  */
  if (global_max_fast > MAX_FAST_SIZE)
    __builtin_unreachable ();
  return global_max_fast;
}

if (av == &main_arena)//仅在初始化main_arena时会走此路径（就执行一次），设置
global_max_fast
     //Maximum size of memory handled in fastbins.
     //在SIZE_SZ=4时，global_max_fast=64；在SIZE_SZ=8时，
global_max_fast=128；通常为128
    set_max_fast (DEFAULT_MXFAST);

#define DEFAULT_MXFAST     (64 * SIZE_SZ / 4)

#define set_max_fast(s) \
  global_max_fast = (((size_t) (s) <= MALLOC_ALIGN_MASK - SIZE_SZ)  \
                     ? MIN_CHUNK_SIZE / 2 : ((s + SIZE_SZ) &
~MALLOC_ALIGN_MASK))
```

- 填充fastbin
  - _libc_free在free chunk时，如果不是mmap chunk，且释放时未能加入tcachebin，小于get_max_fast()，加入fastbin，并设置av->have_fastchunks为true
- 合并fastbin
  - 方式一：_libc_malloc中发生

```
 //tcache失败
 //fastbin失败
 //视为smallbin失败
 //考虑视为largebin，并在继续后面的执行步骤前先在malloc_consolidate中合并fastbin
 if (in_smallbin_range (nb))//在smallbin范围
 {
 ...
 }
 else
 {
      idx = largebin_index (nb);//依据normalized bytes计算largebin_index
      //atomic_load含义应该是指加载读取原子操作
      //relaxed目前觉得和内存一致性模型相关，关键词：Relaxed Memory Model
      //暂时理解为如果该标识为真，进行consolideate
      if (atomic_load_relaxed (&av->have_fastchunks))//consolidate之后，这个变量就会被设置为false
        malloc_consolidate (av);//合并所有fastbin，放入unsortedbin中
 }
//上述代码执行后，到此处及以后的含义：
//说明如果是smallbin,说明其对应的链表为空了，但没进行consolidate;
//或者如果是largebin，说明可能进行了consolidate(fastchunks存在合并，不存在不合并)
```

- 方式二：_libc_free中发生

```
_libc_free在free chunk时，如果不是mmap chunk，且释放时未能加入tcachebin，未能加入fastbin，合并前后可用freechunk，如果size（相邻free合并后释放的大小）>= fastbin合并阈值（64K）,且fastbin中存在数据，则触发fastbin的合并操作
 if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {//如果size（相邻free合并后释放的大小）>= fastbin合并阈值（64K）,fastbin中存在数据，则触发fastbin的合并操作
      if (atomic_load_relaxed (&av->have_fastchunks))
    malloc_consolidate(av);
```

3. 利用方式
    - 待完善

bins

bins分布

> 通常用户申请大小x，会先执行request2size(x)转换成chunk大小，再进行申请。request2size(x)认为chunk之后的下一个chunk的prev_size空间属于申请内存范围来进行计算应申请的chunk大小，下面的总结性结论都是按照chunk进行考虑的。

- unsorted bin
    - bin_at [1]
- smallbin
    - 32位
        - bin_at[2, 63]

        - smallbin中最小chunk为16字节，以8字节增长，直到504字节chunk，共计62个smallbin
    - 64位
        - SIZE_SZ=4(简单理解就是chunk的单元数据大小为4字节)
            - bin_at[3,63]
            - smallbin中最小chunk32字节，以16字节增长，最大992字节chunk，共计61个smallbin
        - SIZE_SZ=8(简单理解就是chunk的单元数据大小为8字节) 通常是此情况
            - bin_at[2,63]
            - smallbin中最小chunk32字节，以16字节增长，最大1008字节chunk，共计62个smallbin
- largebin

> largebin中每个bin中的chunk大小不一致，而是处于一定区间范围内
>
> 下方数据说明：idx即bin_at索引范围 sz即chunk所属字节数 sz-k即sz/1024 bins是链表个数 sep是这段idx区间，字节的跨度

    - 32位
        - bin_at[64, 126]
        - sz>=512

```
idx[ 64 , 94 ]   sz[ 512 , 2488 ]         sz-k[ 0.5 k, 2.4296875 k]
  bins: 31 sep: 64 , 0.0625 k
idx[ 95 , 111 ]  sz[ 2496 , 10744 ]       sz-k[ 2.4375 k, 10.4921875 k]
  bins: 17 sep: 512 , 0.5 k
idx[ 112 , 120 ]  sz[ 10752 , 45048 ]     sz-k[ 10.5 k, 43.9921875 k]
bins: 9 sep: 4096 , 4.0 k
idx[ 120 , 123 ]  sz[ 45056 , 163832 ]    sz-k[ 44.0 k, 159.9921875 k]
 bins: 4 sep: 32768 , 32.0 k
idx[ 124 , 126 ]  sz[ 163840 , 786424 ]   sz-k[ 160.0 k, 767.9921875 k]
bins: 3 sep: 262144 , 256.0 k
idx[126,126]    sz-k[768k,for complete)
```

    - 64位
        - SIZE_SZ=4(简单理解就是chunk的元数据大小为4字节)
            - bin_at[64, 94]， bin_at[96, 126] 比较特殊，没有95
            - sz>=1008

```
idx[ 64 , 94 ]   sz[ 1008 , 2928 ]     sz-k[ 0.984375 k, 2.859375 k]
bins: 31 sep: 64 , 0.0625 k
idx[ 96 , 111 ]  sz[ 2944 , 10736 ]    sz-k[ 2.875 k, 10.484375 k]
bins: 16 sep: 512 , 0.5 k
idx[ 112 , 120 ] sz[ 10752 , 45040 ]   sz-k[ 10.5 k, 43.984375 k]
 bins: 9 sep: 4096 , 4.0 k
idx[ 120 , 123 ] sz[ 45056 , 163824 ]  sz-k[ 44.0 k, 159.984375 k]
bins: 4 sep: 32768 , 32.0 k
idx[ 124 , 126 ] sz[ 163840 , 786416 ] sz-k[ 160.0 k, 767.984375 k]
 bins: 3 sep: 262144 , 256.0 k
idx[126,126]    sz-k[768k,for complete)
```

        - SIZE_SZ=8(简单理解就是chunk的元数据大小为8字节)
            - bin_at[64, 126]

- sz>=1024

```
idx[ 64 , 96 ]   sz[ 1024 , 3120 ]     sz-k[ 1.0 k, 3.046875 k]
 bins: 33 sep: 64 , 0.0625 k
idx[ 97 , 111 ]  sz[ 3136 , 10736 ]    sz-k[ 3.0625 k, 10.484375 k]
 bins: 15 sep: 512 , 0.5 k
idx[ 112 , 120 ] sz[ 10752 , 45040 ]   sz-k[ 10.5 k, 43.984375 k]
 bins: 9 sep: 4096 , 4.0 k
idx[ 120 , 123 ] sz[ 45056 , 163824 ]  sz-k[ 44.0 k, 159.984375 k]
bins: 4 sep: 32768 , 32.0 k
idx[ 124 , 126 ] sz[ 163840 , 786416 ] sz-k[ 160.0 k, 767.984375 k]
 bins: 3 sep: 262144 , 256.0 k
idx[126,126]    sz-k[768k,for complete)
```

```
//bins[NBINS * 2 - 2],NBINS=128, 里面一共可视为127个malloc_chunk链表
//&bins[0]-2*SIZE  ，将这个地址视为假想的malloc_Chunk，这样可以方便的通过->fd,fk进行访问bin中包含的malloc_chunk链表
#define bin_at(m, i) \
  (mbinptr) (((char *) &((m)->bins[((i) - 1) * 2]))               \
             - offsetof (struct malloc_chunk, fd))
//从使用bin_at的角度考虑bins，那么i的输入范围1-127
//bin_at[1]是unsorted bin
#define unsorted_chunks(M)   (bin_at (M, 1))

//sz为chunk size,定位它在bin_at中的索引
#define bin_index(sz)  ((in_smallbin_range (sz)) ? smallbin_index (sz) : largebin_index (sz))

//判断是否在smallbin范围内
#define in_smallbin_range(sz)  ((unsigned long) (sz) < (unsigned long) MIN_LARGE_SIZE)
32: MIN_LARGE_SIZE=512        MINSIZE=16  True:16<= sz <= 504
False:sz>=512
63: MIN_LARGE_SIZE=1008/1024  MINSIZE=32  True:32<= sz <= 992/1008
 False:sz>=1008/1024，通常是1024

#define smallbin_index(sz) \
  ((SMALLBIN_WIDTH == 16 ? (((unsigned) (sz)) >> 4) : (((unsigned) (sz)) >> 3))\
   + SMALLBIN_CORRECTION)
源码有：
    idx = smallbin_index (nb);//nb(chunk size) to index
    bin = bin_at (av, idx);
//可以看到返回的值即为bin_at的索引
32: SMALLBIN_WIDTH=8    SMALLBIN_CORRECTION=0   16<= sz <= 504
smallbin_index(sz)=sz>>3
    //smallbin_index(sz) Range: bin_at[2,63]
    //可以看出，smallbin中最小16字节，以8字节增长，直到504字节，共计62个smallbin
64: SMALLBIN_WIDTH=16   SMALLBIN_CORRECTION=1/0 32<= sz <= 992/1008
 smallbin_index(sz)=sz>>4 + 1/0
    //smallbin_index(sz) Range(SMALLBIN_CORRECTION=1): bin_at[3,63]  32<=sz<=992
    //smallbin_index(sz) Range(SMALLBIN_CORRECTION=0): bin_at[2,63]  32<=sz<=1008
通常是此情况
    //可以看出，smallbin中最小32字节，以16字节增长，最大992/1008字节，共计61/62个smallbin

//largebin中每个bin中的chunk大小不一致，而是处于一定区间范围内
#define largebin_index(sz) \
```

```
  (SIZE_SZ == 8 ? largebin_index_64 (sz)                                    \
   : MALLOC_ALIGNMENT == 16 ? largebin_index_32_big (sz)                    \
   : largebin_index_32 (sz))
```

源码有：

```
    victim_index = largebin_index (size);//size为chunk size
    bck = bin_at (av, victim_index);
//可以看到返回的值即为bin_at的索引
//32: SIZE_SZ=4   MALLOC_ALIGNMENT=8  sz>=512            执行largebin_index_32
(sz)
#define largebin_index_32(sz)                                               \
  (((((unsigned long) (sz)) >> 6) <= 38) ?  56 + (((unsigned long) (sz)) >> 6)
:\
   ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9)
:\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12)
:\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15)
:\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18)
:\
   126)
idx[ 64 , 94 ]   sz[ 512 , 2488 ]        sz-k[ 0.5 k, 2.4296875 k]       bins:
31 sep: 64 , 0.0625 k
idx[ 95 , 111 ]  sz[ 2496 , 10744 ]      sz-k[ 2.4375 k, 10.4921875 k]   bins:
17 sep: 512 , 0.5 k
idx[ 112 , 120 ]    sz[ 10752 , 45048 ]     sz-k[ 10.5 k, 43.9921875 k]
bins: 9 sep: 4096 , 4.0 k
idx[ 120 , 123 ]    sz[ 45056 , 163832 ]    sz-k[ 44.0 k, 159.9921875 k]
 bins: 4 sep: 32768 , 32.0 k
idx[ 124 , 126 ]    sz[ 163840 , 786424 ]   sz-k[ 160.0 k, 767.9921875 k]
bins: 3 sep: 262144 , 256.0 k
idx[126,126]    sz-k[768k,for complete)

//64: SIZE_SZ=4/8 MALLOC_ALIGNMENT=16 sz>=1008/1024
//当SIZE_SZ=4时，有sz>=1008,执行largebin_index_32_big (sz)
#define largebin_index_32_big(sz)                                           \
  (((((unsigned long) (sz)) >> 6) <= 45) ?  49 + (((unsigned long) (sz)) >> 6)
:\
   ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9)
:\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12)
:\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15)
:\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18)
:\
   126)
idx[ 64 , 94 ]   sz[ 1008 , 2928 ]       sz-k[ 0.984375 k, 2.859375 k]   bins:
31 sep: 64 , 0.0625 k
idx[ 96 , 111 ]  sz[ 2944 , 10736 ]      sz-k[ 2.875 k, 10.484375 k]     bins:
16 sep: 512 , 0.5 k
idx[ 112 , 120 ]    sz[ 10752 , 45040 ]     sz-k[ 10.5 k, 43.984375 k]
 bins: 9 sep: 4096 , 4.0 k
idx[ 120 , 123 ]    sz[ 45056 , 163824 ]    sz-k[ 44.0 k, 159.984375 k]
bins: 4 sep: 32768 , 32.0 k
idx[ 124 , 126 ]    sz[ 163840 , 786416 ]   sz-k[ 160.0 k, 767.984375 k]
 bins: 3 sep: 262144 , 256.0 k
idx[126,126]    sz-k[768k,for complete)
```

这种方式居然不存在95..

```
//当SIZE_SZ=8时，有sz>=1024,执行largebin_index_64 (sz) ，64位通常为此种方式
#define largebin_index_64(sz)                                                \
  (((((unsigned long) (sz)) >> 6) <= 48) ?  48 + (((unsigned long) (sz)) >> 6)
:\
   ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9)
:\
   ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12)
:\
   ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15)
:\
   ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18)
:\
   126)
idx[ 64 , 96 ]   sz[ 1024 , 3120 ]       sz-k[ 1.0 k, 3.046875 k]        bins:
33 sep: 64 , 0.0625 k
idx[ 97 , 111 ]  sz[ 3136 , 10736 ]      sz-k[ 3.0625 k, 10.484375 k]    bins:
15 sep: 512 , 0.5 k
idx[ 112 , 120 ]    sz[ 10752 , 45040 ]     sz-k[ 10.5 k, 43.984375 k]
 bins: 9 sep: 4096 , 4.0 k
idx[ 120 , 123 ]    sz[ 45056 , 163824 ]    sz-k[ 44.0 k, 159.984375 k]
bins: 4 sep: 32768 , 32.0 k
idx[ 124 , 126 ]    sz[ 163840 , 786416 ]   sz-k[ 160.0 k, 767.984375 k]
 bins: 3 sep: 262144 , 256.0 k
idx[126,126]    sz-k[768k,for complete)
```

python分析脚本

```
def calc(a,b,c,d,minsz,align):
    minidx="unknow"
    if minsz!=-1:
        minidx=c+(minsz>>d)
    maxsz=(b<<a)+((1<<a)-1)
    maxidx=c+(maxsz>>d)

 print("idx[",minidx,",",maxidx,"]\t","sz[",minsz,",",int(maxsz/align)*align,"]\
t","sz-k[",minsz/1024,"k,",int(maxsz/align)*align/1024,"k]\t","bins:",maxidx-
minidx+1,"sep:",1<<d,",",(1<<d)/1024,"k")
    return maxsz+1

#largebin_index_32
a32=calc(6,38,56,6,512,8)
b32=calc(9,20,91,9,a32,8)
c32=calc(12,10,110,12,b32,8)
d32=calc(15,4,119,15,c32,8)
e32=calc(18,2,124,18,d32,8)
print("idx[126,126]\tsz-k[768k,for complete)")

#largebin_index_32_big
abig32=calc(6,45,49,6,1008,16)
bbig32=calc(9,20,91,9,abig32,16)
cbig32=calc(12,10,110,12,bbig32,16)
dbig32=calc(15,4,119,15,cbig32,16)
ebig32=calc(18,2,124,18,dbig32,16)
print("idx[126,126]\tsz-k[768k,for complete)")

#largebin_index_64
a64=calc(6,48,48,6,1024,16)
b64=calc(9,20,91,9,a64,16)
```

```
c64=calc(12,10,110,12,b64,16)
d64=calc(15,4,119,15,c64,16)
e64=calc(18,2,124,18,d64,16)
print("idx[126,126]\tsz-k[768k,for complete)")
```

unsorted bin

- malloc时会依次尝试tcache，fastbin，smallbin只有失败了，才会进入unsortedbin处理逻辑(在进入之前如果申请的是largebin，且fastbin存在chunk数据则还会先malloc_consolidate进行合并fastbin到unsortedbin中)

1. 数据结构
    - ```
      //bin_at[1]是unsorted bin
      #define unsorted_chunks(M)   (bin_at (M, 1))
      ```
    - 双链表，FIFO，内部操作时chunk从头部放入，尾部取出，chunk大小不同，无序
        - unsortedbin中仅使用fd/bk。fd/bk_nextsize仅在largebin中使用，unsortedbin中都为NULL
        - 地址相邻的下一个chunk的prev_inuse为0（即unsortedbin中的chunk被视为free chunk)
2. 行为算法
    - index计算
        - 参考前文bins分布
    - 初始化

      ```
      //此处时arena初始化时代码，会将unsorted/small/largebin都指向自己
        for (i = 1; i < NBINS; ++i)
        {
        bin = bin_at (av, i);
        bin->fd = bin->bk = bin;
        }
      ```

    - 处理unsortedbin

      ```
      #if USE_TCACHE  //初始化一些值
      //如果填充过tcache，那么该变量就会设为1
      int return_cached = 0;
      //初始化为0，用于计数移出unsortedbin且不是填充tcache的chunk数
      //当 tcache_unsorted_count > mp_.tcache_unsorted_limit 且 return_cached为1，那么就直接从tcache中取出chunk返回,防止处理unsortedbin过程占用太多时间
      tcache_unsorted_count = 0;
      #endif

      for (;; )
      {
            int iters = 0;
            //unsorted chunks bk不指向自己，说明列表非空;victim为列表尾部chunk;采用FIFO，头部放入，尾部取出
            //unsortedbin非空，进入while循环
      ```
      ```

```
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks
(av))
      {   ...
          size = chunksize (victim);//当前处理的victim的大小
          ...
          //第一部分：unsortedbin中唯一chunk就是lastremainder,申请的nb字节属于
smallbin范围，而且满足(size) > (nb + MINSIZE)，则拆分它，并返回chunk
          if (in_smallbin_range (nb) &&//说明申请nb属于small bin，nb是用户
req转换成chunk后的字节数
              bck == unsorted_chunks (av) &&//说明是unsortedbin中最后一个
chunk
              victim == av->last_remainder &&//说明这个chunk是
last_remainder :The remainder from the most recent split of a small
request 最近拆分的 small request 的剩余部分
              (unsigned long) (size) > (unsigned long) (nb + MINSIZE))//
说明可以拆分last remainder
          {.....  return p;}
          ...
          //第二部分：如果申请的nb字节刚好等于victim字节，且nb属于tcache范围，且
对应所属tcachebin未满，那么先填充tcache，return_cached置1;如果申请的nb字节刚好等
于victim字节，且（nb不属于tcache范围，或对应所属tcachebin已满),那么进行返回
           if (size == nb){
               #if USE_TCACHE
          if (tcache_nb && tcache->counts[tc_idx] < mp_.tcache_count)
        {//这一步说明申请的nb属于tcache范围，对应所属tcachebin未满，且当前
unsortedbin中处理的victim大小等同申请需要的大小
          tcache_put (victim, tc_idx);
          return_cached = 1;//标识填充过tcachebin,可以从tcachebin中返回
          continue;//while循环处理unsorted bin
        }
          else
        {
#endif      //到这说明申请的nb不属于tcache范围，且当前unsortedbin中处理的victim
大小等同申请需要的大小，则进行返回
              //或者说明申请的nb属于tcache范围，对应所属tcachebin已满，且当前
unsortedbin中处理的victim大小等同申请需要的大小，则进行返回
              check_malloced_chunk (av, victim, nb);//检测
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
#if USE_TCACHE
        }
#endif
        }

       //第三部分
       //如果size和nb不一样,会到此处,这里会将当前的victim分别放入对应的smallbin
和largebin中
          /* place chunk in bin */
          if (in_smallbin_range (size))//如果在small bin 范围内
            {//此处记录size对应的smallbin链表
              victim_index = smallbin_index (size);
              //后面最终会插入在bck fwd之间，位于smallbin的头部，这里先记录链表
相关信息
              bck = bin_at (av, victim_index);//待插入位置前一个地址;
              fwd = bck->fd;//带插入位置的后一个地址;
            }
          else
```

```
            {//large bin ; 每个索引位置的bin都包含了一个区间范围，其中chunk按大小递减排序
              victim_index = largebin_index (size);
              bck = bin_at (av, victim_index);//待插入位置前一个地址 但只是初步的值
              fwd = bck->fd;//带插入位置的后一个地址，但只是初步的值，默认这么设是是假定比头第一个chunk大来思考的，
              //后面代码会调整，直到位置满足largebin中chunk排序从大大小(对应从头到尾)
              ...
            }
            mark_bin (av, victim_index);//将bitmap置位,表示对应的bin存在chunk；bin为空时，不会改变置位，只有后面serach时发现bin为空才会清0
             ...
       //第四部分：当处理了足够多的unsorted_chunk后，从tcache中获取一个返回；主要用于提高分配速度，避免在unsorted chunk处理太长时间
      #if USE_TCACHE
      ++tcache_unsorted_count;
      if (return_cached
      && mp_.tcache_unsorted_limit > 0  //默认为0，无限制，不会提前通过tcahce返回；Maximum number of chunks to remove from the unsorted list, which aren't used to prefill the cache
      && tcache_unsorted_count > mp_.tcache_unsorted_limit)
    {
      return tcache_get (tc_idx);
    }
    #endif

#define MAX_ITERS       10000
       //超出10000次循环后，break while循环，停止unsortedbin处理，防止处理消耗太多时间
          if (++iters >= MAX_ITERS)
            break;

      }//此处while结尾
    #if USE_TCACHE
      /* If all the small chunks we found ended up cached, return one now.  */
      //如果上面所有unsortedbin chunk处理完，且tcahce放置过chunk
      //或者处理了10000次unsortedchunk，且tcache放置过chunk
      //则立马取出缓存中的一个进行返回
      if (return_cached)
    {
      return tcache_get (tc_idx);
    }
    #endif

}
```

- 填充unsortedbin
  - malloc_consolidate

```
malloc_consolidate发生条件：
 //tcache失败
 //fastbin失败
 //视为smallbin失败
```

```
 //考虑视为largebin，并在继续后面的执行步骤前先在malloc_consolidate中合并fastbin
 if (in_smallbin_range (nb))//在smallbin范围
 {
 ...
 }
 else
 {
      idx = largebin_index (nb);//依据normalized bytes计算largebin_index
      //atomic_load含义应该是指加载读取原子操作
      //relaxed目前觉得和内存一致性模型相关，关键词：Relaxed Memory Model
      //暂时理解为如果该标识为真，进行consolideate
      if (atomic_load_relaxed (&av->have_fastchunks))//consolidate之后，这个变量就会被设置为false
        malloc_consolidate (av);//合并所有fastbin，放入unsortedbin中
 }
//上述代码执行后，到此处及以后的含义：
//说明如果是smallbin,说明其对应的链表为空了，但没进行consolidate;
//或者如果是largebin，说明可能进行了consolidate(fastchunks存在合并，不存在不合并)

malloc_consolidate逻辑：
//从后往前，从大到小遍历所有fastbin，并对fastchunk的前后chunk尝试进行合并
//合并后的chunk如果不挨着topchunk则放入unsortedbin头部
//合并后的chunk如果挨着topchunk则并入topchunk
//合并过程中，前后chunk如果是freechunk，那么freechunk会从原先的bins链表（unsorted/smallbin/largebin）unlink，
//为什么这里只需要尝试合并前后chunk就可以呢？可以参考前面推荐的B站视频，讲述了Allocated/Free chunk的递归分析
细节参考源码
```

- _libc_free

```
free chunk时，如果不是mmap chunk，且freechunsize > fastbin max size 且不挨着topchunk,则合并空闲chunk并放入unsortedbin
```

3. 利用方式
   - 待完善

smallbin

1. 数据结构
   - 双链表，FIFO，从尾部分配，从头部插入，在链表中时地址相邻的下一个chunk的prev_inuse为0，分配后地址相邻的下一个chunk的prev_inuse为1，相同索引位置的双链表中的chunk大小相同
2. 行为算法
   - index计算
     - 参考前文bins分布
   - 初始化

```
//此处时arena初始化时代码，会将unsorted/small/largebin都指向自己
  for (i = 1; i < NBINS; ++i)
  {
  bin = bin_at (av, i);
  bin->fd = bin->bk = bin;
  }
```

- 获取small bin

```
if (in_smallbin_range (nb))//在smallbin范围
    {
      idx = smallbin_index (nb);//nb to index
      bin = bin_at (av, idx);//使用起来如同直接使用mbinptr

      if ((victim = last (bin)) != bin)//如果bin不是指向自己(指向自己说明不存在对应的bin)；采用FIFO，从尾部分配，而释放的加入到头部
        {
          bck = victim->bk;
      if (__glibc_unlikely (bck->fd != victim))//检测
        malloc_printerr ("malloc(): smallbin double linked list corrupted");
          set_inuse_bit_at_offset (victim, nb);//让虚拟地址上紧挨着的下一个chunk的prev_inuse置1
          bin->bk = bck;
          bck->fd = bin;//将victim从链表中移除

          if (av != &main_arena)
        set_non_main_arena (victim);//标记chunk为非主分配区内容
          check_malloced_chunk (av, victim, nb);//same as check_remalloced_chunk = do_check_remalloced_chunk 基本上能检查的都给检查了
          //下面部分是如果启用了tcache机制，并且属于tc_index在tcache范围内,tcache未满且smallbin未空，将small list中chunk移除并放到tcache中
          .........
          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;
        }
    }
....

//在上面方法没有获取到smallbin，并且程序执行到后续的binmap部分时，可能仍然没有找到对应的chunk，则通过最小适配原则匹配查找第一个满足要求的chunk；这一部分执行时，申请的字节nb为smallbin/largebin都有可能；这个过程会利用binmap，且发现信息失效的binmap部分并更新，如果能找到容纳的chunk，且chunksize>=nb+MINSIZE，则进行分裂，分裂的一部分返回，一部分插入unsortedbin(如果nb是samllbin范围，这部分还会设置为lastreaminder)
...
//通常上面过程基本完成chunk的分配，如果还不行，就尝试进行topchunk分配，一部分返回，一部分继续成为topchunk;如果topchunk也不行，那么先查看是否有fastbin，如果有那么尝试malloc_consolidate,之后切回unsorted部分重新处理；如果也不行，则通过sysmalloc尝试直接向系统分配内存
```

3. 利用方式

- 待完善

largebin

1. 数据结构
    - 双链表，在链表中时地址相邻的下一个chunk的prev_inuse为0，分配后地址相邻的下一个chunk的prev_inuse为1
        - 每一个chunk都存在于fd/bk构成的双链表中，头部chunksize最大，尾部chunksize最小
        - 相邻chunks大小相同的为一组，每一组第一个index最小chunk之间通过fd/bk_nextsize构成了一个快速查找大小的双链表
        - 插入时chunk时会自动依据chunsize维护largebin中chunk的顺序，严格保持头大尾小。插入时是从头往后寻找合适的位置，而且找到后，如果存在相同大小组的话总是插在相同大小一组中的第二个位置，否则就是新组的第一个
        - 取出时是从后往前找，即从小往大找，找到第一个满足要求的chunk
2. 行为算法
    - index计算
        - 参考前文bins分布
    - 初始化

```
//此处时arena初始化时代码，会将unsorted/small/largebin都指向自己
  for (i = 1; i < NBINS; ++i)
  {
  bin = bin_at (av, i);
  bin->fd = bin->bk = bin;
  }
```

    - 获取largebin

```
//unsortedbin while循环处理完成后，且tcahce尝试返回后，进入下面代码
//一般是largebin；   如果之前unsortedbin中不存在nb大小的chunk，那么可能走到这仍然时smallrange范围内的bin，这种情况下此处不执行，往下走
//此处逻辑尝试从nb对应的largebin中，从后往前从小往大获取第一个满足要求的chunk，如果不满足，则放入后面binmap处理
if (!in_smallbin_range (nb))
        {
          bin = bin_at (av, idx);//利用largebin idx找到对应链表头

          /* skip scan if empty or largest chunk is too small */
          if ((victim = first (bin)) != bin
          && (unsigned long) chunksize_nomask (victim)
            >= (unsigned long) (nb))//largebin最大值大于nb
            {
              victim = victim->bk_nextsize;//看这个样子是从后往前找，即从小往大找，找到第一个满足要求的victim  : victim >=nb
              while (((unsigned long) (size = chunksize (victim)) <
                      (unsigned long) (nb)))
                victim = victim->bk_nextsize;
              /* Avoid removing the first entry for a size so that the skip
                 list does not have to be rerouted.  */
              if (victim != last (bin)
          && chunksize_nomask (victim)
            == chunksize_nomask (victim->fd))
```

```
                victim = victim->fd;//先避免移出找到的victim，而是移出相同大小的下一个victim   这样就不需要维护nextsize等指针
              remainder_size = size - nb;//记录remainder size
              unlink_chunk (av, victim);//将victim从链表中移出
              /* Exhaust */
              if (remainder_size < MINSIZE)//如果分裂的话不满足MINSIZE要求，即没办法分裂成两个chunk
                {
                  set_inuse_bit_at_offset (victim, size);//设置相关prev_inuse标识
                  if (av != &main_arena)
            set_non_main_arena (victim);//取出的时候，会设置相关arena标识
                }
              /* Split */
              else//对从bin中取出的chunk进行分裂;分裂后新的victim返回，而remainder加入到unsorted_chunks头部
                {
                  ...
                }
              check_malloced_chunk (av, victim, nb);//检查
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
            }
        }

//上面逻辑处理完后，可能仍然没有找到对应的chunk，则通过最小适配原则匹配查找第一个满足要求的chunk；这一部分逻辑中申请的字节nb为smallbin/largebin都有可能；这个过程会利用binmap，且发现信息失效的binmap部分并更新，如果能找到容纳的chunk，且chunksize>=nb+MINSIZE，则进行分裂，分裂的一部分返回，一部分插入unsortedbin(如果nb是samllbin范围，这部分还会设置为lastreaminder)
...
//通常上面过程基本完成chunk的分配，如果还不行，就尝试进行topchunk分配，一部分返回，一部分继续成为topchunk;如果topchunk也不行，那么先查看是否有fastbin，如果有那么尝试malloc_consolidate,之后切回unsorted部分重新处理；如果也不行，则通过sysmalloc尝试直接向系统分配内存
```

    - 填充largebin
        - 参见前文处理unsortedbin
3. 利用方式
    - 待完善

topchunk

- topchunk的prev_inuse必须设置为1
- topchunk块的尾部地址必须页对齐
- topchunk要是能分裂的话，剩下的topchunk必须>=MINSIZE（至少可以还分配一个MINSIZE大小的chunk)
- malloc_consolidate时，合并fastbin之后的chunk若紧邻着topchunk，则合并到topchunk

1. 数据结构
    - arean顶部区域
2. 行为算法

- 初始化

```
#define initial_top(M)      (unsorted_chunks (M))
在初始化topchunk时，有av->top = initial_top (av)，等同于av->top=&(av->top)
```

- 获取

```
//binmap处理完后仍然无法满足分配需要，进入top区域处理；
//_int_malloc的最后一部分，尝试进行topchunk分配，一部分返回，一部分继续成为topchunk;如果topchunk也不行，那么先查看是否有fastbin，如果有那么尝试malloc_consolidate,之后切回unsorted部分重新处理；如果也不行，则通过sysmalloc尝试直接向系统分配内存
```

- 填充

```
malloc_consolidate时，合并fastbin之后的chunk若紧邻着topchunk，则合并到topchunk
free chunk时，如果不是mmap chunk，且freechunsize > fastbin max size 且挨着topchunk,则合并到topchunk
```

3. 利用方式
   - 待完善

lastremainder

1. 数据结构
   - 注意：申请的字节数nb必须要**属于smallbind大小范围**，分裂后chunk另一部分才可能为lastremainder
2. 行为算法
   - 初始化

```
arena->last_remainder
在main_arena/非main_arena新建和初始化时，没有显示设置该值，默认值为NULL
```

   - 获取

```
//这部分unsortedbin处理不仅仅是获取，也是填充；获取后的剩余部分仍然作为last_remainder
....
//到这说明如果是smallbin,说明其对应的链表为空了，但没进行consolidate;
//或者如果是largebin，说明可能进行了consolidate(fastchunks存在合并，不存在则不合并)
...
//这一块简而言之就是unsortedbin中唯一chunk就是lastremainder,而且满足拆分出nb字节的smallbinchunk后还能大于MINSIZE，则拆分它，并返回chunk
          if (in_smallbin_range (nb) &&//说明属于small bin
          if (in_smallbin_range (nb) &&//说明属于small bin
              bck == unsorted_chunks (av) &&//说明是最后一个chunk
              victim == av->last_remainder &&//说明这个chunk是last_remainder :The remainder from the most recent split of a small request 最近拆分的 small request 的剩余部分
```

```
              (unsigned long) (size) > (unsigned long) (nb + MINSIZE))//
说明可以拆分last remainder
            {
              /* split and reattach remainder */
              remainder_size = size - nb;
              remainder = chunk_at_offset (victim, nb);
              unsorted_chunks (av)->bk = unsorted_chunks (av)->fd =
remainder;//拆分last remainder
              av->last_remainder = remainder;//设置arena 的 last
remainder属性
              remainder->bk = remainder->fd = unsorted_chunks (av);//设置
unsorted 链表
              if (!in_smallbin_range (remainder_size))//remainder不在
small bin范围内，设置相关指针;unsortedbin中的chunk(属于largebin范
围),fd/bk_nextsize必须都为NULL
                {
                  remainder->fd_nextsize = NULL;
                  remainder->bk_nextsize = NULL;
                }
              //个人理解此处设置prev_inuse，是为了满足A/F chunk的递归关系，Free
chunk之前的chunk prev_inuse必定为1
              set_head (victim, nb | PREV_INUSE |
                        (av != &main_arena ? NON_MAIN_ARENA : 0));
              set_head (remainder, remainder_size | PREV_INUSE);
              set_foot (remainder, remainder_size);

              check_malloced_chunk (av, victim, nb);//检测分配的chunk
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
            }
```

- 填充

  binmap处理时，找到的chunk size >= nb+MINSIZE,并且nb属于smallbin大小范围，那么分裂后的一部分作为chunk返回，另一部分设置为lastremainder

3. 利用方式
   - 待完善

sysmalloc

主要逻辑：（假设了topchunk不满足分配需要）
首先满足下面条件的进入try_mmap，尝试通过mmap进行分配：
1.在arena数量已满但是当前线程thread_arena还没创建过，那么arena_get会返回null，并传递到这
2.在arena数量未满但是地址空间不足以申请一个最小的heap(HEAP_MIN_SIZE)来创建arena，那么arena_get会返回null，并传递到这
3.av存在但是 nb字节书大于mmap_threshold且mmap region的数量没有达到最大值，尝试直接mmap映射，mmap_threshold 默认值：128K 最小值128k 最大值：32位512K 64位16\32M，通常32M
if(av == NULL || ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_threshold)&& (mp_.n_mmaps < mp_.n_mmaps_max))) //满足进行mmap分配，并返回申请区域；但是如果av==NULL且mmap失败，则直接返回NULL；其它则往下走

接着判断是否是非主分配区域
非主分配区：先尝试grow_heap，再尝试new_heap（会在oldheap topchunk处产生fencepost,来让一部分oldheap topchunk进行free），要是还是失败的话判断是否执行过try_mmap逻辑，若是没有则跳到try_mmap尝试直接映射
主分配区：正常情况下直接sbrk增长即可。少数情况下，会存在外部非glibc调用sbrk，造成区域不连续，oldtop需要加入fencepost来让oldtop部分尽量进行free；要么就无法sbrk，只能mmap补救，造成set_noncontiguous (main_arena)（默认是true的）；这部分非正常情况处理复杂，但很少见，暂不花费很多精力

最后分裂top,一部分作为chunk返回，一部分作为新的top

mmap chunk

```
//mmapchunk 本身是不太关注是主arena还是非主arena，默认下值A (NON_MAIN_ARENA)为0，表示为主分区，但是free时也不关注它，它只有在分配处时，会稍微校验一下，但也是A标志无关
//mmapchunk的分配过程参考sysmalloc部分
//mp_.mmap_threshold会在free mmapchunk时可能更新：
if (chunk_is_mmapped (p)) {
      if (!mp_.no_dyn_threshold//默认mmap_threshold是动态的，除非用户手动设置非动态;mmap_threshold默认值是DEFAULT_MMAP_THRESHOLD，为128K
          && chunksize_nomask (p) > mp_.mmap_threshold  //p的chunksize大于动态的mmap_threshold 且小于默认的DEFAULT_MMAP_THRESHOLD_MAX（32:512K/64:16\32M）
          && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX)
        {
          mp_.mmap_threshold = chunksize (p);//利用p的chunksize更新mmap_threshold
          mp_.trim_threshold = 2 * mp_.mmap_threshold;//更新trim_threshold收缩阈值，为2倍mmap_threshold即默认256K
          LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                      mp_.mmap_threshold, mp_.trim_threshold);
        }
      munmap_chunk (p);
    }
}
```

systrim

简单来说就是依据topchunksize向下页对齐的值为缩小量对topchunk进行缩减；只适用main_arena

heap_trim

从包含topchunk的heap开始，由后往前，依次判断整个heap能否delete，能删除就删，遇到第一个不能删除就开始尝试类似main_arena收缩的方法对heap进行收缩

check chunk

- 代码中检查体现了安全是动态的，而不是静态的，需要持续校验，因此代码中四散着各种check

引用

MallocInternals - glibc wiki (sourceware.org):

> ## Malloc Algorithm
>
> In a nutshell, malloc works like this:
>
> - If there is a suitable (exact match only) chunk in the tcache, it is returned to the caller. No attempt is made to use an available chunk from a larger-sized bin.
> - If the request is large enough, `mmap()` is used to request memory directly from the operating system. Note that the threshold for mmap'ing is dynamic, unless overridden by M_MMAP_THRESHOLD (see mallopt() documentation), and there may be a limit to how many such mappings there can be at one time.
> - If the appropriate fastbin has a chunk in it, use that. If additional chunks are available, also pre-fill the tcache.
> - If the appropriate smallbin has a chunk in it, use that, possibly pre-filling the tcache here also.
> - If the request is "large", take a moment to take everything in the fastbins and move them to the unsorted bin, coalescing them as you go.
> - Start taking chunks off the unsorted list, and moving them to small/large bins, coalescing as you go (note that this is the only place in the code that puts chunks into the small/large bins). If a chunk of the right size is seen, use that.
> - If the request is "large", search the appropriate large bin, and successively larger bins, until a large-enough chunk is found.
> - If we still have chunks in the fastbins (this may happen for "small" requests), consolidate those and repeat the previous two steps.
> - Split off part of the "top" chunk, possibly enlarging "top" beforehand.
>
> For an over-aligned malloc, such as `valloc`, `pvalloc`, or `memalign`, an overly-large chunk is located (using the malloc algorithm above) and divided into two or more chunks in such a way that most of the chunk is now suitably aligned (and returned to the caller), and the excess before and after that portion is re-turned to the unsorted list to be re-used later.
>
> ## Free Algorithm

Note that, in general, "freeing" memory does not actually return it to the operating system for other applications to use. The `free()` call marks a chunk of memory as "free to be reused" by the application, but from the operating system's point of view, the memory still "belongs" to the application. However, if the top chunk in a heap - the portion adjacent to unmapped memory - becomes large enough, some of that memory may be unmapped and returned to the operating system.

In a nutshell, free works like this:

- If there is room in the tcache, store the chunk there and return.
- If the chunk is small enough, place it in the appropriate fastbin.
- If the chunk was mmap'd, munmap it.
- See if this chunk is adjacent to another free chunk and coalesce if it is.
- Place the chunk in the unsorted list, unless it's now the "top" chunk.
- If the chunk is large enough, coalesce any fastbins and see if the top chunk is large enough to give some memory back to the system. Note that this step might be deferred, for performance reasons, and happen during a malloc or other call.

Realloc Algorithm

Note that realloc of NULL and realloc to zero size are handled separately and as per the relevant specs.

In a nutshell, realloc works like this:

For MMAP'd chunks...

Allocations that are serviced via individual `mmap` calls (i.e. large ones) are realloc'd by `mremap()` if available, which may or may not result in the new memory being at a different address than the old memory, depending on what the kernel does.

If the system does not support `munmap()` and the new size is smaller than the old size, nothing happens and the old address is returned, else a malloc-copy-free happens.

For arena chunks...

- If the size of the allocation is being reduced by enough to be "worth it", the chunk is split into two chunks. The first half (which has the old address) is returned, and the second half is returned to the arena as a free chunk. Slight reductions are treated as "the same size".
- If the allocation is growing, the next (adjacent) chunk is checked. If it is free, or the "top" block (representing the expandable part of the heap), and large enough, then that chunk and the current are merged, producing a large-enough block which can be possibly split (as above). In this case, the old pointer is returned.
- If the allocation is growing and there's no way to use the existing/following chunk, then a malloc-copy-free sequence is used.

Switching arenas

> The arena to which a thread is attached is generally viewed as an invariant that does not change over the lifetime of the process. This invariant, while useful for explaining general concepts, is not true. The scenario for changing arenas looks like this:
>
> - Thread fails to allocate memory from the attached arena.
>   - Assumes we tried coalescing, searched free list, processed unsorted list etc.
>   - Assumes we tried to expand the heap but either the `sbrk` failed or creating the new mapping failed.
> - If previously using a non-main arena with `mmap`'d heaps the thread is switched via `arena_get_retry` to the main arena with an `sbrk`-based heap, or switched to non-main arena (from free list or a new one) if previously using the main arena. As an optimization this is not done if the process is single threaded, and we fail at this point returning `ENOMEM` (it is assumed that if `sbrk` did not work, and we tried `mmap` to extend the main arena, that a non-main arena will not work either).
>
> Note that a thread may change frequently between arenas in low-memory conditions, switching from main-arena which `sbrk`-based to a non-main arena which is `mmap`-based, all in an attempt to find a heap with enough space to satisfy the allocation.