


# 404



## glibc malloc源码简析（二）

 wangshuo  
2021-02-27 浏览次

### # 1 前言

在上一篇文章中我们了解了glibc内存管理相关的数据结构，本文将结合glibc源码介绍malloc的具体执行流程。  
软件信息如下：

软件项	版本信息
OS	openEuler 20.03 (LTS)
kernel	4.19.90-2003.4.0.0036.oe1
glibc	2.28
gcc	7.3.0

### # 2 malloc执行流程分析

#### # 2.1 总体流程

glibc malloc操作的总体流程我们在《glibc malloc原理简析》中已有提及，建议结合3.2.1小结中的流程图来阅读接下来的内容。  
glibc malloc的代码量有几千行，由于glibc社区奇葩的编码规范，常常会出现几百行的for循环以及犬牙交错的缩进，因此阅读起来极为不便。为了便于理解，本文将先梳理malloc的主函数，然后再对较为关键的函数（如\_int\_malloc和sysmalloc）进行单独分析。  
glibc malloc主函数源码如下：

```
malloc/malloc.c

void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;
  void *victim;

  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));
  #if USE_TCACHE
```

```

/* int_free also calls request2size, be careful to not pad twice. */
size_t tbytes;
checked_request2size (bytes, tbytes);
size_t tc_idx = csize2tidx (tbytes);

MAYBE_INIT_TCACHE ();

DIAG_PUSH_NEEDS_COMMENT;
if (tc_idx < mp_.tcache_bins
    /*&& tc_idx < TCACHE_MAX_BINS*/ /* to appease gcc */
    && tcache
    && tcache->entries[tc_idx] != NULL)
{
    return tcache_get (tc_idx);
}
DIAG_POP_NEEDS_COMMENT;
#endif

if (SINGLE_THREAD_P)
{
    victim = _int_malloc (&main_arena, bytes);
    assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
        &main_arena == arena_for_chunk (mem2chunk (victim)));
    return victim;
}

arena_get (ar_ptr, bytes);

victim = _int_malloc (ar_ptr, bytes);
/* Retry with another arena only if we were able to find a usable arena
   before. */
if (!victim && ar_ptr != NULL)
{
    LIBC_PROBE (memory_malloc_retry, 1, bytes);
    ar_ptr = arena_get_retry (ar_ptr, bytes);
    victim = _int_malloc (ar_ptr, bytes);
}

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

assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
    ar_ptr == arena_for_chunk (mem2chunk (victim)));
return victim;
}
libc_hidden_def (__libc_malloc)

```

主函数首先会调用ptmalloc\_init进行初始化，而后，如果开启了tcache则会对相关的链表进行初始化，每次malloc时都优先查找tcache中的链表，如果满足条件直接返回。接下来，如果当前仅支持单线程，则直接调用\_int\_malloc来申请内存返回给malloc调用者，否则先获取分配区，然后基于分配区来调用\_int\_malloc申请内存。以上便是malloc主函数的流程。

## # 2.2 ptmalloc\_init

在进程首次调用malloc时，会进入\_\_malloc\_hook进行初始化，设置这个钩子的目的是实现mtrace调测功能，相关代码在malloc/mtrace.c路径下，本文不展开讲解。malloc\_hook\_ini的定义如下：

```

malloc/hooks.c

static void *
malloc_hook_ini (size_t sz, const void *caller)
{
    __malloc_hook = NULL;
    ptmalloc_init ();
    return __libc_malloc (sz);
}

```

默认情况下使用的是glibc的ptmalloc\_init函数，执行完初始化后会回到malloc主函数，ptmalloc\_init函数源码如下：

```
malloc/arena.c

static void
ptmalloc_init (void)
{
  if (__malloc_initialized >= 0)
    return;

  __malloc_initialized = 0;

#ifdef SHARED
  /* In case this libc copy is in a non-default namespace, never use brk.
     Likewise if dlopened from statically linked program. */
  DL_info di;
  struct link_map *l;

  if (_dl_open_hook != NULL
      || (_dl_addr (ptmalloc_init, &di, &l, NULL) != 0
          && l->l_ns != LM_ID_BASE))
    __morecore = __failing_morecore;
#endif

  thread_arena = &main_arena;

  malloc_init_state (&main_arena);

#ifdef HAVE_TUNABLES
  TUNABLE_GET (check, int32_t, TUNABLE_CALLBACK (set_mallopt_check));
  TUNABLE_GET (top_pad, size_t, TUNABLE_CALLBACK (set_top_pad));
  TUNABLE_GET (perturb, int32_t, TUNABLE_CALLBACK (set_perturb_byte));
  TUNABLE_GET (mmap_threshold, size_t, TUNABLE_CALLBACK (set_mmap_threshold));
  TUNABLE_GET (trim_threshold, size_t, TUNABLE_CALLBACK (set_trim_threshold));
  TUNABLE_GET (mmap_max, int32_t, TUNABLE_CALLBACK (set_mmaps_max));
  TUNABLE_GET (arena_max, size_t, TUNABLE_CALLBACK (set_arena_max));
  TUNABLE_GET (arena_test, size_t, TUNABLE_CALLBACK (set_arena_test));
# if USE_TCACHE
  TUNABLE_GET (tcache_max, size_t, TUNABLE_CALLBACK (set_tcache_max));
  TUNABLE_GET (tcache_count, size_t, TUNABLE_CALLBACK (set_tcache_count));
  TUNABLE_GET (tcache_unsorted_limit, size_t,
               TUNABLE_CALLBACK (set_tcache_unsorted_limit));
# endif
#else
  const char *s = NULL;
  if (__glibc_likely (_environ != NULL))
    {
      char **runp = _environ;
      char *envline;

      while (__builtin_expect ((envline = next_env_entry (&runp)) != NULL,
                              0))
        {
          size_t len = strcspn (envline, "=");

          if (envline[len] != '=')
            /* This is a "MALLOC_" variable at the end of the string
               without a '=' character. Ignore it since otherwise we
               will access invalid memory below. */
            continue;

          switch (len)
            {

```

```

case 6:
    if (memcmp (envline, "CHECK_", 6) == 0)
        s = &envline[7];
    break;
case 8:
    if (!__builtin_expect (__libc_enable_secure, 0))
    {
        if (memcmp (envline, "TOP_PAD_", 8) == 0)
            __libc_mallopt (M_TOP_PAD, atoi (&envline[9]));
        else if (memcmp (envline, "PERTURB_", 8) == 0)
            __libc_mallopt (M_PERTURB, atoi (&envline[9]));
    }
    break;
case 9:
    if (!__builtin_expect (__libc_enable_secure, 0))
    {
        if (memcmp (envline, "MMAP_MAX_", 9) == 0)
            __libc_mallopt (M_MMAP_MAX, atoi (&envline[10]));
        else if (memcmp (envline, "ARENA_MAX", 9) == 0)
            __libc_mallopt (M_ARENA_MAX, atoi (&envline[10]));
    }
    break;
case 10:
    if (!__builtin_expect (__libc_enable_secure, 0))
    {
        if (memcmp (envline, "ARENA_TEST", 10) == 0)
            __libc_mallopt (M_ARENA_TEST, atoi (&envline[11]));
    }
    break;
case 15:
    if (!__builtin_expect (__libc_enable_secure, 0))
    {
        if (memcmp (envline, "TRIM_THRESHOLD_", 15) == 0)
            __libc_mallopt (M_TRIM_THRESHOLD, atoi (&envline[16]));
        else if (memcmp (envline, "MMAP_THRESHOLD_", 15) == 0)
            __libc_mallopt (M_MMAP_THRESHOLD, atoi (&envline[16]));
    }
    break;
default:
    break;
}

}

if (s && s[0] != '\0' && s[0] != '0')
    __malloc_check_init ();
#endif

#if HAVE_MALLOC_INIT_HOOK
void (*hook) (void) = atomic_forced_read (__malloc_initialize_hook);
if (hook != NULL)
    (*hook)();
#endif
__malloc_initialized = 1;
}

```

ptmalloc\_init用来对整个ptmalloc框架进行初始化，首先检查全局变量\_\_malloc\_initialized是否大于等于0，如果该值大于0，表示ptmalloc已经初始化，如果改值为0，表示ptmalloc正在初始化，全局变量\_\_malloc\_initialized用来保证全局只初始化ptmalloc一次。接下来会调用malloc\_init\_state函数来初始化主分配区，其定义如下：

```

/*
    Initialize a malloc_state struct.

    This is called from ptmalloc_init () 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 */
  for (i = 1; i < NBINS; ++i)
    {
      bin = bin_at (av, i);
      bin->fd = bin->bk = bin;
    }

#ifdef MORECORE_CONTIGUOUS
  if (av != &main_arena)
#endif
  set_noncontiguous (av);
  if (av == &main_arena)
    set_max_fast (DEFAULT_MXFAST);
  atomic_store_relaxed (&av->have_fastchunks, false);

  av->top = initial_top (av);
  init_malloc_par();
}

```

该函数做了四件事情，第一是初始化malloc\_state中的bins数组，初始化的结果是对bins数组中的每一个fd和对应的bk，都初始化为fd的地址，即fd=bk=&fd；第二是设置fastbin可管理的内存块的最大值，即global\_max\_fast，第三是设置一些标志位；第四是初始化分配去中的top chunk，就是一个malloc\_chunk指针，fd保存在bins[0]中（smallbin中不使用bins[0]和bins[1]）。

回到ptmalloc\_init，之后的代码虽然很长，其实只做了一件事，那就是读取当前的环境变量来设置malloc的一些参数，相关参数的定义和作用可以参考《glibc malloc原理简析》的4.1小节。需要注意的是，这部分代码包含了两种设置方式：TUNABLES和非TUNABLES，编译的时候通过HAVE\_TUNABLES宏来决定开启或关闭，《glibc malloc原理简析》的4.2小节给出了两种方式的区别和使用方法。

## # 2.3 tcache init

tcache的初始化操作最终会通过MAYBE\_INIT\_TCACHE宏调用tcache\_init函数实现，后者的源码如下：

```

malloc/malloc.c

static void
tcache_init(void)
{
  mstate ar_ptr;
  void *victim = 0;
  const size_t bytes = sizeof (tcache_perthread_struct);

  if (tcache_shutting_down)
    return;

  arena_get (ar_ptr, bytes);
  victim = _int_malloc (ar_ptr, bytes);
  if (!victim && ar_ptr != NULL)
    {
      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;
    memset (tcache, 0, sizeof (tcache_perthread_struct));
}

}

```

如果将这部分代码与malloc主函数对比的话会发现非常相似，没错，这部分实际上是使用malloc操作申请了sizeof (tcache\_perthread\_struct)大小的一块内存（我们在glibc malloc源码简析（一）的第6节提到tcache\_perthread\_struct是tcache管理内存块的基本数据结构），并通过memset初始化。内存申请这一块涉及的函数我们将在接下来几个小节中陆续进行分析。

## # 2.4 arena\_get

上文提到，对于默认的非单线程模式，需要先获取一个分配区，从分配区中获取内存，获取分配区通过宏定义arena\_get来实现，源码如下：

```

/* arena_get() acquires an arena and locks the corresponding mutex.
   First, try the one last locked successfully by this thread. (This
   is the common case and handled with a macro for speed.) Then, loop
   once over the circularly linked list of arenas. If no arena is
   readily available, create a new one. In this latter case, 'size'
   is just a hint as to how much memory will be required immediately
   in the new arena. */

#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)

```

在ptmalloc\_init中有一个对变量thread\_arena的赋值操作，该变量为线程变量，定义如下：

```

/* Thread specific data. */

static __thread mstate thread_arena attribute_tls_model_ie;

```

而根据ptmalloc\_init仅在第一次malloc时被调用，此时thread\_arena线程变量获得main\_arena的值，之后再次malloc则不会对thread\_arena初始化，结合arena\_get定义可知对于第一次malloc，则直接在main\_arena中申请内存，对于后续的malloc则通过arena\_get2获取一个thread分配区，arena\_get2函数的源码如下：

```

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)
        {
            if (mp_arena_max != 0)
                narenas_limit = mp_arena_max;
            /* ... */
        }
    }
}

```

```

else if (narenas > mp_arena_test)
{
    int n = __get_nprocs ();

    if (n >= 1)
        narenas_limit = NARENAS_FROM_NCORES (n);
    else
        /* We have no information about the system. Assume two
        cores. */
        narenas_limit = NARENAS_FROM_NCORES (2);
}
}
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))
        goto repeat;
    a = _int_new_arena (size);
    if (__glibc_unlikely (a == NULL))
        catomic_decrement (&narenas);
}
else
    a = reused_arena (avoid_arena);
}
return a;
}

```

glibc维护了一个全局的arena free list，当进入arena\_get2后，会优先从free list中查找是否有可用的arena，如果有则直接加锁并返回，否则，通过\_int\_new\_arena创建一个新的arena加锁并返回。获得分配区的指针后，就会调用\_int\_malloc开始分配内存。

## # 2.5 \_int\_malloc

\_int\_malloc是实现glibc malloc的核心函数，主体代码有600多行，其中包括大量百行的判断和循环等语句，因此我们分步分析。  
\_int\_malloc开头部分的源码如下：

```

static void *
_int_malloc (mstate av, size_t bytes)
{
    ...

    /*
    Convert request size to internal form by adding SIZE_SZ bytes
    overhead plus possibly more to obtain necessary alignment and/or
    to obtain a size of at least MINSIZE, the smallest allocatable
    size. Also, checked_request2size traps (returning 0) request sizes
    that are so large that they wrap around zero when padded and
    aligned.
    */

    checked_request2size (bytes, nb);

    /* There are no usable arenas. Fall back to sysmalloc to get a chunk from
    mmap. */
    if (__glibc_unlikely (av == NULL))
    {
        void *p = sysmalloc (nb, av);
        if (p != NULL)

```

```

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

...

}

```

`_int_malloc`首先调用`checked_request2size`将需要分配的内存大小`bytes`转换为`chunk`的大小。`checked_request2size`是个宏定义，主要调用`request2size`进行计算，定义如下：

```

/* pad request bytes into a usable size -- internal version */

#define request2size(req) \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE) ? \
   MINSIZE : \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)

/* Same, except also perform an argument and result check.  First, we check
   that the padding done by request2size didn't result in an integer
   overflow.  Then we check (using REQUEST_OUT_OF_RANGE) that the resulting
   size isn't so large that a later alignment would lead to another integer
   overflow.  */
#define checked_request2size(req, sz) \
({ \
  \
  (sz) = request2size (req); \
  if (((sz) < (req)) \
      || REQUEST_OUT_OF_RANGE (sz)) \
  { \
    \
    __set_errno (ENOMEM); \
    return 0; \
  } \
})

```

在glibc malloc源码简析（一）的第2小节中曾提到，当一个`chunk`为空闲时，至少要有`mchunk_prev_size`、`mchunk_size`、`fd`和`bk`四个参数，而当一个`chunk`被使用时，`mchunk_prev_size`可能会被前一个`chunk`用来存储，`fd`和`bk`也会被当作内存存储数据，因此当`chunk`被使用时，只剩下了`mchunk_size`参数需要设置，`request2size`中的`SIZE_SZ`就是`INTERNAL_SIZE_T`类型的大小，因此至少需要`req+SIZE_SZ`的内存大小。`MALLOC_ALIGN_MASK`用来对齐，因此`request2size`就计算出了所需的`chunk`的大小。传入的参数`av`即为调用`arena_get`获得的分配区指针，如果为`null`，就表示没有分配区可用，这时候就直接调用`sysmalloc`通过`mmap`获取`chunk`，关于`sysmalloc`我们将在2.7小节进行分析。

`_int_malloc`接下来会处理申请的内存大小落入`fastbin`区间时的情况，源码如下：

```

static void *
_int_malloc (mstate av, size_t bytes)
{
  ...

  if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
  {
    idx = fastbin_index (nb);
    mfastbinptr *fb = &fastbin (av, idx);
    mchunkptr pp;
    victim = *fb;

    if (victim != NULL)
    {
      if (SINGLE_THREAD_P)
        *fb = victim->fd;
      else
        REMOVE_FB (fb, pp, victim);
      if (__glibc_likely (victim != NULL))
        f

```



```

    \
    size_t victim_idx = fastbin_index (chunksize (victim));
    if (__builtin_expect (victim_idx != idx, 0))
    malloc_printerr ("malloc(): memory corruption (fast)");
    check_reallocated_chunk (av, victim, nb);
#endif 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)
    {
        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 (SINGLE_THREAD_P)
            *fb = tc_victim->fd;
            else
            {
                REMOVE_FB (fb, pp, tc_victim);
                if (__glibc_unlikely (tc_victim == NULL))
                    break;
            }
            tcache_put (tc_victim, tc_idx);
        }
    }
#endif

    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
}

}

...

}

```

我们先不考虑USE\_TCACHE判断中的内容进行分析，如果需要分配的内存大小nb落在fastbin的范围内，首先调用fastbin\_index获得chunk大小nb对应的fastbin索引。获得索引idx后，就通过fastbin取出空闲chunk链表指针，mfastbinptr其实就是malloc\_chunk指针，其定义如下

```
#define fastbin(ar_ptr, idx) ((ar_ptr)->fastbinsY[idx])
```

其中REMOVE\_FB实际是一个do、while循环，其定义如下，本质上是一个CAS操作，其作用是从刚刚得到的空闲chunk链表指针中取出第一个空闲的chunk(victim)，并将链表头设置为该空闲chunk的下一个chunk(victim->fd)。

```

#define REMOVE_FB(fb, victim, pp) \
do \
{ \
    victim = pp; \
    if (victim == NULL) \
        break; \
} \
while ((pp = catomic_compare_and_exchange_val_acq (fb, victim->fd, victim)) \
    != victim);

```

这里注意，fastbin中使用的是单链表，而后面smallbin使用的是双链表。获得空闲chunk后，需要转换为可以存储的内存指针，假设fastbin中没有找到空闲chunk，或者fastbin根本没有初始化，或者其他原因，就进入下一步，从smallbin中获取内存。

让我们回到USE\_TCACHE判断，当fastbin中有不止一个相同大小的chunk时，会尝试把其它的chunk放入tcache，这样的好处是下一次malloc如果还申请这个大小的内存，则可以直接从tcache中获取，由于tcache是线程变量，因此消除了加锁解锁的开销，提升了性能。这一流程在接下来

本处中通过 1 个小时的讨论，刘云昌及从libc中获取，由于libc是线性变量，因此用陈江洪双链表的开闭，证明其性能，这个函数在技术上的步骤中还会多次遇到。

另外，为了保证小块内存申请释放的性能，fast bin和tcache都不会对free chunk进行合并操作，因此当free chunk被放入fastbin和tcache时，系统会保留其使用标志位以避免自动合并操作。

```
static void *
_int_malloc (mstate av, size_t bytes)
{
  ...

  /*
   * If a small request, check regular bin. Since these "smallbins"
   * hold one size each, no searching within bins is necessary.
   * (For a large request, we need to wait until unsorted chunks are
   * processed to find best fit. But for small ones, fits are exact
   * anyway, so we can check now, which is faster.)
   */

  if (in_smallbin_range (nb))
    {
      idx = smallbin_index (nb);
      bin = bin_at (av, idx);

      if ((victim = last (bin)) != bin)
        {
          bck = victim->bk;
          if (__glibc_unlikely (bck->fd != victim))
            malloc_printerr ("malloc(): smallbin double linked list corrupted");
          set_inuse_bit_at_offset (victim, nb);
          bin->bk = bck;
          bck->fd = bin;

          if (av != &main_arena)
            set_non_main_arena (victim);
          check_malloced_chunk (av, victim, nb);
        }
    }

  #if USE_TCACHE
    /* While we're here, if we see other chunks of the same size,
     * stash them in the tcache. */

    ...

  #endif
    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
  }
}

/*
 * If this is a large request, consolidate fastbins before continuing.
 * While it might look excessive to kill all fastbins before
 * even seeing if there is space available, this avoids
 * fragmentation problems normally associated with fastbins.
 * Also, in practice, programs tend to have runs of either small or
 * large requests, but less often mixtures, so consolidation is not
 * invoked all that often in most programs. And the programs that
 * it is called frequently in otherwise tend to fragment.
 */

else
  {
    idx = largebin_index (nb);
    if (atomic_load_relaxed (&av->have_fastchunks))
      malloc_consolidate (av);
  }
}
```

```
...
}
```

当申请内存的大小落在small bin区间时，将会执行以上逻辑。我们同样先不考虑USE\_TCACHE判断中的内容进行分析，其实这里的操作就很简单了，设置标志位，将chunk返回。同样，如果使能了tcache，且申请的chunk的大小在tcache管理的chunk的大小的范围内，则会遍历smallbin中同样大小的chunk放入tcache中。下次如果同一线程申请同样大小的内存，则会优先从tcache中获取。

当申请内存的大小落入large bin区间时，malloc会先试图通过malloc\_consolidate函数执行一次合并操作，具体代码如下所示，该函数的作用是遍历fastbin中每个chunk链表的每个malloc\_chunk指针，合并前一个不在使用中的chunk，如果后一个chunk是top chunk，则直接合并到top chunk中，如果后一个chunk不是top chunk，则合并后一个chunk并添加进unsorted\_bin中。

```
/*
----- malloc_consolidate -----

malloc_consolidate is a specialized version of free() that tears
down chunks held in fastbins. Free itself cannot be used for this
purpose since, among other things, it might place chunks back onto
fastbins. So, instead, we need to use a minor variant of the same
code.
*/

static void malloc_consolidate(mstate av)
{
    ...

    atomic_store_relaxed (&av->have_fastchunks, false);

    unsorted_bin = unsorted_chunks(av);

    /*
    Remove each chunk from fast bin and consolidate it, placing it
    then in unsorted bin. Among other reasons for doing this,
    placing in unsorted bin avoids needing to calculate actual bins
    until malloc is sure that chunks aren't immediately going to be
    reused anyway.
    */

    maxfb = &fastbin (av, NFASTBINS - 1);
    fb = &fastbin (av, 0);
    do {
        p = atomic_exchange_acq (fb, NULL);
        if (p != 0) {
            do {
                {
                    unsigned int idx = fastbin_index (chunksize (p));
                    if ((&fastbin (av, idx)) != fb)
                        malloc_printerr ("malloc_consolidate(): invalid chunk size");
                }
            }

            check_inuse_chunk(av, p);
            nextp = p->fd;

            /* Slightly streamlined version of consolidation code in free() */
            size = chunksize (p);
            nextchunk = chunk_at_offset(p, size);
            nextsize = chunksize(nextchunk);

            if (!prev_inuse(p)) {
                prevsize = prev_size (p);
                size += prevsize;
                p = chunk_at_offset(p, -((long) prevsize));
                if (__glibc_unlikely (chunksize(p) != prevsize))
                    malloc_printerr ("corrupted size vs. prev_size in fastbins");
                unlink(av, p, bck, fwd);
            }
        }
    } while (p);
}
```

```

        unlink(av, nextchunk, bck, fwd);
    } else
        clear_inuse_bit_at_offset(nextchunk, 0);

    first_unsorted = unsorted_bin->fd;
    unsorted_bin->fd = p;
    first_unsorted->bk = p;

    if (!in_smallbin_range (size)) {
        p->fd_nextsize = NULL;
        p->bk_nextsize = NULL;
    }

    set_head(p, size | PREV_INUSE);
    p->bk = unsorted_bin;
    p->fd = first_unsorted;
    set_foot(p, size);
}

else {
    size += nextsize;
    set_head(p, size | PREV_INUSE);
    av->top = p;
}

} while ( (p = nextp) != 0);

}
} while (fb++ != maxfb);
}

```

如果走到这儿还没找到所需的内存，将会执行最后一步，一个近500行的for循环，这里再拆出几步进行分析。

```

static void *
_int_malloc (mstate av, size_t bytes)
{
    ...

    for (;;)
    {
        int iters = 0;
        while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
        {
            bck = victim->bk;
            if (__builtin_expect (chunksize_nomask (victim) <= 2 * SIZE_SZ, 0)
                || __builtin_expect (chunksize_nomask (victim)
                    > av->system_mem, 0))
                malloc_printerr ("malloc(): memory corruption");
            size = chunksize (victim);

            /*
             * If a small request, try to use last remainder if it is the
             * only chunk in unsorted bin. This helps promote locality for
             * runs of consecutive small requests. This is the only
             * exception to best-fit, and applies only when there is
             * only one chunk in the bin.
             */

```

```

        no exact fit for a small chunk.
    */

    if (in_smallbin_range (nb) &&
        bck == unsorted_chunks (av) &&
        victim == av->last_remainder &&
        (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
    {
        /* split and reattach remainder */
        remainder_size = size - nb;
        remainder = chunk_at_offset (victim, nb);
        unsorted_chunks (av)->bk = unsorted_chunks (av)->fd = remainder;
        av->last_remainder = remainder;
        remainder->bk = remainder->fd = unsorted_chunks (av);
        if (!in_smallbin_range (remainder_size))
        {
            remainder->fd_nextsize = NULL;
            remainder->bk_nextsize = NULL;
        }

        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_malallocated_chunk (av, victim, nb);
        void *p = chunk2mem (victim);
        alloc_perturb (p, bytes);
        return p;
    }

    /* remove from unsorted list */
    if (__glibc_unlikely (bck->fd != victim))
        malloc_printerr ("malloc(): corrupted unsorted chunks 3");
    unsorted_chunks (av)->bk = bck;
    bck->fd = unsorted_chunks (av);

    /* Take now instead of binning if exact fit */

    if (size == nb)
    {
        set_inuse_bit_at_offset (victim, size);
        if (av != &main_arena)
            set_non_main_arena (victim);
#ifdef USE_TCACHE
        /* Fill cache first, return to user only if cache fills.
           We may return one of these chunks later. */
        if (tcache_nb
            && tcache->counts[tc_idx] < mp_.tcache_count)
        {
            tcache_put (victim, tc_idx);
            return_cached = 1;
            continue;
        }
        else
        {
#endif
            check_malallocated_chunk (av, victim, nb);
            void *p = chunk2mem (victim);
            alloc_perturb (p, bytes);
            return p;
        }
    }
#ifdef USE_TCACHE
    }
#endif
#endif

```

```

        ...

    }

    ...

}

}

```

这部分代码的整体意思就是遍历unsortedbin，从中查找是否有符合用户要求大小的chunk并返回，我们同样先忽略掉tcache。

第一个while循环从尾到头依次取出unsortedbin中的所有chunk，将该chunk对应的前一个chunk保存在bck中，并将大小保存在size中。如果用户需要分配的内存大小对应的chunk属于smallbin，unsortedbin中只有这一个chunk，并且该chunk属于last remainder chunk且其大小大于用户需要分配内存大小对应的chunk大小加上最小的chunk大小（保证可以拆分成两个chunk），就将该chunk拆分成两个chunk，分别为victim和remainder，进行相应的设置后，将用户需要的victim返回。如果不能拆开，就从unsortedbin中取出该chunk（victim）。再下来，如果刚刚从unsortedbin中取出的victim正好是用户需要的大小nb，就设置相应的标志位，直接返回该victim。

在加入tcache重新看这段代码可以发现，在这里依旧是之前的思想，多出来的chunk优先放入tcache中，下次申请内存可以快速返回。

```

static void *
_int_malloc (mstate av, size_t bytes)
{
    ...

    for (;;)
    {
        int iters = 0;
        while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
        {
            ...

            /* place chunk in bin */

            if (in_smallbin_range (size))
            {
                victim_index = smallbin_index (size);
                bck = bin_at (av, victim_index);
                fwd = bck->fd;
            }
            else
            {
                victim_index = largebin_index (size);
                bck = bin_at (av, victim_index);
                fwd = bck->fd;

                /* maintain large bins in sorted order */
                if (fwd != bck)
                {
                    /* Or with inuse bit to speed comparisons */
                    size |= PREV_INUSE;
                    /* if smaller than smallest, bypass loop below */
                    assert (chunk_main_arena (bck->bk));
                    if ((unsigned long) (size)
                        < (unsigned long) chunksize_nomask (bck->bk))
                    {
                        fwd = bck;
                        bck = bck->bk;

                        victim->fd_nextsize = fwd->fd;
                        victim->bk_nextsize = fwd->fd->bk_nextsize;
                        fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
                    }
                }
            }
            else
            {

```

```

        assert (chunk_main_arena (fwd));
        while ((unsigned long) size < chunksize_nomask (fwd))
        {
            fwd = fwd->fd_nextsize;
        }
        assert (chunk_main_arena (fwd));
    }

    if ((unsigned long) size
== (unsigned long) chunksize_nomask (fwd))
        /* Always insert in the second position. */
        fwd = fwd->fd;
    else
    {
        victim->fd_nextsize = fwd;
        victim->bk_nextsize = fwd->bk_nextsize;
        if (__glibc_unlikely (fwd->bk_nextsize->fd_nextsize != fwd))
            malloc_printerr ("malloc(): largebin double linked list corrupted (nextsize)");
        fwd->bk_nextsize = victim;
        victim->bk_nextsize->fd_nextsize = victim;
    }
    bck = fwd->bk;
    if (bck->fd != fwd)
        malloc_printerr ("malloc(): largebin double linked list corrupted (bk)");
}
}
else
    victim->fd_nextsize = victim->bk_nextsize = victim;
}

mark_bin (av, victim_index);
victim->bk = bck;
victim->fd = fwd;
fwd->bk = victim;
bck->fd = victim;

#if USE_TCACHE
    /* If we've processed as many chunks as we're allowed while
    filling the cache, return one of the cached ones. */
    ++tcache_unsorted_count;
    if (return_cached
88 mp_.tcache_unsorted_limit > 0
88 tcache_unsorted_count > mp_.tcache_unsorted_limit)
    {
        return tcache_get (tc_idx);
    }

#define MAX_ITERS    10000
    if (++iters >= MAX_ITERS)
        break;
}

#if USE_TCACHE
    /* If all the small chunks we found ended up cached, return one now. */
    if (return_cached)
    {
        return tcache_get (tc_idx);
    }
#endif
...
}
}

```

这部分代码的整体意思是如果从unsortedbin中取出的chunk不符合用户要求的大小，就将该chunk合并到smallbin或者largebin中，我们依旧先忽略掉tcache进行分析。

首先如果取出的chunk ( victim ) 属于smallbin，就通过smallbin\_index计算需要插入的位置victim\_index，然后获取smallbin中对应位置的链表头指针保存在bck中，最后直接插入到smallbin中，由于smallbin中的chunk不使用fd\_nextsize和bk\_nextsize指针，插入操作只需要更新bk和fd指针，具体的插入操作在后面。这里需解释一下，fd\_nextsize指针指向的是chunk双向链表中下一个大小不同的chunk，bk\_nextsize指向的是chunk双向链表中前一个大小不同的chunk。

如果取出的chunk ( victim ) 属于largebin，通过largebin\_index计算需要插入的位置victim\_index，然后获取largebin链表头指针保存在bck中。

如果fwd等于bck，即bck->fd=bck，则表示largebin中对应位置上的chunk双向链表为空，直接进入后面的else部分中，代码victim->fd\_nextsize = victim->bk\_nextsize = victim表示插入到largebin中的victim是唯一的chunk，因此其fd\_nextsize和bk\_nextsize指针都指向自己。

如果fwd不等于bck，对应的chunk双向链表存在空闲chunk，这时就要在该链表中找到合适的位置插入了。因为largebin中的chunk链表是按照chunk大小从大到小排序的，如果victim的size小于bck->bk->size即最后一个chunk的大小，则表示即将插入victim的大小在largebin的chunk双向链表中是最小的，因此要把victim插入到该链表的最后。

如果要插入的victim的size不是最小的，就要通过一个while循环遍历找到合适的位置，这里是从双向链表头bck->fd开始遍历，利用fd\_nextsize加快遍历的速度，找到第一个size>=fwd->size的chunk。如果size=fwd->size，就只是改变victim以及前后相应chunk的bk、fd指针就行。

再往下，如果size不相等，则直接插入在fwd的左边，这样也能保证所有的fd\_nextsize和bk\_nextsize指针设置在相同chunk大小的最地地址处（最左边）。

再往下，mark\_bin用来标识malloc\_state中的binmap，标识相应位置的chunk空闲。然后就更改fd、bk指针，插入到双向链表中，这个插入操作同时适用于smallbin和largebin，因此放在这里。最后如果在unsortedbin中处理了超过10000个chunk，就直接退出循环，这里保证不会因为unsortedbin中chunk太多，处理的时间太长了。

重新加入tcache分析，这里有两块用到tcache，其思路都是上文在合并时通过return\_cached变量标记是否已找到所需大小的内存（如果找到会优先放入tcache中），然后等合并完毕执行到这一步时，直接从tcache拿出满足条件的chunk返回。

```
static void * _int_malloc(mstate av, size_t bytes) {
    ...

    for (;;) {

        ...

        /*
         * If a large request, scan through the chunks of current bin in
         * sorted order to find smallest that fits. Use the skip list for this.
         */

        if (!in_smallbin_range (nb))
        {
            bin = bin_at (av, idx);

            /* skip scan if empty or largest chunk is too small */
            if ((victim = first (bin)) != bin
&& (unsigned long) chunksize_nomask (victim)
>= (unsigned long) (nb))
            {
                victim = victim->bk_nextsize;
                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;

                remainder_size = size - nb;
                unlink (av, victim, bck, fwd);

                /* Exhaust */
                if (remainder_size < MINSIZE)
                {
                    set_inuse_bit_at_offset (victim, size);
                    if (av != &main_arena)
                        set_non_main_arena (victim);
                }
                /* Split */
                else
            }
        }
    }
}
```



```

    {
        remainder = chunk_at_offset (victim, nb);
        /* We cannot assume the unsorted list is empty and therefore
           have to perform a complete insert here. */
        bck = unsorted_chunks (av);
        fwd = bck->fd;
        if (__glibc_unlikely (fwd->bk != bck))
            malloc_printerr ("malloc(): corrupted unsorted chunks");
        remainder->bk = bck;
        remainder->fd = fwd;
        bck->fd = remainder;
        fwd->bk = remainder;
        if (!in_smallbin_range (remainder_size))
            {
                remainder->fd_nextsize = NULL;
                remainder->bk_nextsize = NULL;
            }
        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_malallocated_chunk (av, victim, nb);
    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
}
}

...
}
}

```

这部分代码的整体意思就是要尝试从largebin中取出对应的chunk了。这里的idx是在前面根据宏largebin\_index计算的。接下来就要根据idx获得largebin中的双向链表头指针bin，然后从bin->bk开始从尾到头（根据chunk大小从小到大）遍历整个双向链表，找到第一个大于用户需要分配的chunk大小nb的chunk指针victim。找到victim后，需要将其拆分成两部分，第一部分是要返回给用户的chunk，第二部分分为两种情况，如果其大小小于MINSIZE，则不能构成一个最小chunk，这种情况下就将拆开前的整个victim返回给用户；如果大于MINSIZE，就将拆开后的第二部分remainder插入到unsortedbin中，然后把第一部分victim返回给用户。

```

static void * _int_malloc(mstate av, size_t bytes) {
    ...

    for (;;) {
        ...

        /*
           Search for a chunk by scanning bins, starting with next largest
           bin. This search is strictly by best-fit; i.e., the smallest
           (with ties going to approximately the least recently used) chunk
           that fits is selected.

           The bitmap avoids needing to check that most blocks are nonempty.
           The particular case of skipping all bins during warm-up phases
           when no chunks have been returned yet is faster than it might look.
        */

        ++idx;
        bin = bin_at (av, idx);
        block = idx2block (idx);
        map = av->binmap[block];
        bit = idx2bit (idx);
    }
}

```

```

for (;;)
{
    /* Skip rest of block if there are no more set bits in this block. */
    if (bit > map || bit == 0)
    {
        do
        {
            if (++block >= BINMAPSIZE) /* out of bins */
                goto use_top;
        }
        while ((map = av->binmap[block]) == 0);

        bin = bin_at (av, (block << BINMAPSHIFT));
        bit = 1;
    }

    /* Advance to bin with set bit. There must be one. */
    while ((bit & map) == 0)
    {
        bin = next_bin (bin);
        bit <<= 1;
        assert (bit != 0);
    }

    /* Inspect the bin. It is likely to be non-empty */
    victim = last (bin);

    /* If a false alarm (empty bin), clear the bit. */
    if (victim == bin)
    {
        av->binmap[block] = map & ~bit; /* Write through */
        bin = next_bin (bin);
        bit <<= 1;
    }

    else
    {
        size = chunksize (victim);

        /* We know the first chunk in this bin is big enough to use. */
        assert ((unsigned long) (size) >= (unsigned long) (nb));

        remainder_size = size - nb;

        /* unlink */
        unlink (av, victim, bck, fwd);

        /* Exhaust */
        if (remainder_size < MINSIZE)
        {
            set_inuse_bit_at_offset (victim, size);
            if (av != &main_arena)
                set_non_main_arena (victim);
        }

        /* Split */
        else
        {
            remainder = chunk_at_offset (victim, nb);

            /* We cannot assume the unsorted list is empty and therefore
               have to perform a complete insert here. */
            bck = unsorted_chunks (av);
            fwd = bck->fd;
            if (__glibc_unlikely (fwd->bk != bck))

```

```

        malloc_printerr ("malloc(): corrupted unsorted chunks 2");

        remainder->bk = bck;
        remainder->fd = fwd;
        bck->fd = remainder;
        fwd->bk = remainder;

        /* advertise as last remainder */
        if (in_smallbin_range (nb))
            av->last_remainder = remainder;
        if (!in_smallbin_range (remainder_size))
        {
            remainder->fd_nextsize = NULL;
            remainder->bk_nextsize = NULL;
        }
        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_mallocked_chunk (av, victim, nb);
    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
}
}

...

}
}

```

这一部分的整体意思是，前面在largebin中寻找特定大小的空闲chunk，如果没找到，这里就要遍历largebin中的其他更大的chunk双向链表，继续寻找。

开头的++idx就表示，这里要从largebin中下一个更大的chunk双向链表开始遍历。ptmalloc中用一个bit表示malloc\_state的bins数组中对应的位置上是否有空闲chunk，bit为1表示有，为0则没有。ptmalloc通过4个block（一个block 4字节）一共128个bit管理bins数组。因此，代码中计算的block表示对应的idx属于哪一个block，map就表是block对应的bit组成的二进制数字。接下来进入for循环，如果bit > map，表示该map对应的整个block里都没有大于bit位置的空闲的chunk，因此就要找下一个block。因为后面的block只要不等于0，就肯定有空闲chunk，并且其大小大于bit位置对应的chunk，下面就根据block，取出block对应的第一个双向链表的头指针。这里也可以看出，设置map和block也是为了加快查找的速度。如果遍历完所有block都没有空闲chunk，这时只能从top chunk里分配chunk了，因此跳转到use\_top。

如果有空闲chunk，接下来就通过一个while循环依次比较找出到底在哪个双向链表里存在空闲chunk，最后获得空闲chunk所在的双向链表的头指针bin和位置bit。

接下来，如果找到的双向链表又为空，则继续前面的遍历，找到空闲chunk所在的双向链表的头指针bin和位置bit。如果找到的双向链表不为空，就和上面一部分再largebin中找到空闲chunk的操作一样了，这里就不继续分析了。

```

use_top:
/*
    If large enough, split off the chunk bordering the end of memory
    (held in av->top). Note that this is in accord with the best-fit
    search rule. In effect, av->top is treated as larger (and thus
    less well fitting) than any other available chunk since it can
    be extended to be as large as necessary (up to system
    limitations).

    We require that av->top always exists (i.e., has size >=
    MINSIZE) after initialization, so if it would otherwise be
    exhausted by current request, it is replenished. (The main
    reason for ensuring it exists is that we may need MINSIZE space
    to put in fenceposts in sysmalloc.)
*/

victim = av->top;
size = chunksize (victim);

if (__glibc_unlikely (size > av->system_mem))
    malloc_printerr ("malloc(): corrupted top size");

```

```

if ((unsigned long) (size) >= (unsigned long) (nb + MINSIZE))
{
    remainder_size = size - nb;
    remainder = chunk_at_offset (victim, nb);
    av->top = remainder;
    set_head (victim, nb | PREV_INUSE |
              (av != &main_arena ? NON_MAIN_ARENA : 0));
    set_head (remainder, remainder_size | PREV_INUSE);

    check_malallocated_chunk (av, victim, nb);
    void *p = chunk2mem (victim);
    alloc_perturb (p, bytes);
    return p;
}

/* When we are using atomic ops to free fast chunks we can get
   here for all block sizes. */
else if (atomic_load_relaxed (&av->have_fastchunks))
{
    malloc_consolidate (av);
    /* restore original bin index */
    if (in_smallbin_range (nb))
        idx = smallbin_index (nb);
    else
        idx = largebin_index (nb);
}

/*
   Otherwise, relay to handle system-dependent cases
*/
else
{
    void *p = sysmalloc (nb, av);
    if (p != NULL)
        alloc_perturb (p, bytes);
    return p;
}
}
}

```

这里就是 `_int_malloc` 的最后一部分了，这部分代码的整体意思分为三部分，首先从 `top chunk` 中尝试分配内存；如果失败，就检查 `fastbin` 中是否有空闲内存了（其他线程此时可能将释放的 `chunk` 放入 `fastbin` 中了），如果不空闲，就合并 `fastbin` 中的空闲 `chunk` 并放入 `smallbin` 或者 `largebin` 中，然后会回到 `_int_malloc` 函数中最前面的 `for` 循环，重新开始查找空闲 `chunk`；如果连 `fastbin` 中都没有空闲内存了，这时只能通过 `sysmalloc` 从系统分配内存了。

## # 2.6 sysmalloc

`sysmalloc` 顾名思义，就是在当前 `glibc malloc` 在用户态管理的内存不足的情况下从 `OS` 申请内存，因此应用场景有二：一个是初始化的时候，另一个便是内存不足的时候。考虑到 `sysmalloc` 代码量依旧庞大，这里继续分段来讲解。

第一部分代码如下，如果需要分配的内存大小大于实用 `mmap` 进行分配的阈值 `mp_.mmap_threshold`，且 `mp_.n_mmmaps` 判断系统还可以有可以使用 `mmap` 分配的空间时会执行 `try_mmap` 分支。

首先会重新计算需要分配多少内存，因为通过使用 `mmap` 直接分配的 `chunk` 不需要添加到链表中，因此不存在前后关系，当一个 `chunk` 被使用时，不能借用后一个 `chunk` 的 `prev_size` 字段，这里需要把该字段的长度 `SIZE_SZ` 加上。并且这里假设 `MALLOC_ALIGNMENT == 2 * SIZE_SZ`。接下来判断需要分配的内存大小是否会溢出，然后就调用 `MMAP` 分配内存，`MMAP` 是一个宏定义，最后就是通过系统调用来分配内存，后面来看这个函数。

再往下就是通过 `set_head` 在 `chunk` 中的 `size` 参数里设置标志位，因为 `chunk` 是按 8 字节对齐的，而 `size` 标识 `chunk` 占用的字节数，所以最后三位是没有用的，`ptmalloc` 将这三位用来作为标志位，这里便是设置其中一个标志位，用来标识该 `chunk` 是直接通过 `mmap` 分配的。

设置完标志位后，接下来就是设置全局变量 `mp_`，将 `mp_.n_mmmaps` 加 1，后者表示当前进程通过 `mmap` 分配的 `chunk` 个数，对应的 `mp_.max_n_mmmaps` 表示最大 `chunk` 个数。`mp_.mmapmed_mem` 标识已经通过 `mmap` 分配的内存大小，`mp_.max_mmapmed_mem` 对应可分配内存的最大值。最后，通过 `chunk2mem` 返回 `chunk` 中内存的起始指针。

回到 `_int_malloc` 函数中，假设通过 `sysmalloc` 分配成功，接下来就需要调用 `alloc_perturb` 对刚刚分配的内存进行初始化。

```

static void *
sysmalloc (INTERNAL_SIZE_T nb, mstate av)
{
    ...
}

```

```

/*
    If have mmap, and the request size meets the mmap threshold, and
    the system supports mmap, and there are few enough currently
    allocated mmapped regions, try to directly map this request
    rather than expanding top.
*/

if (av == NULL
    || ((unsigned long) (nb) >= (unsigned long) (mp->mmap_threshold)
        && (mp->n_mmmaps < mp->n_mmmaps_max)))
{
    char *mm;          /* return value from mmap call*/

try_mmap:
    /*
        Round up size to nearest page. For mmapped chunks, the overhead
        is one SIZE_SZ unit larger than for normal chunks, because there
        is no following chunk whose prev_size field could be used.

        See the front_misalign handling below, for glibc there is no
        need for further alignments unless we have high alignment.
    */
    if (MALLOC_ALIGNMENT == 2 * SIZE_SZ)
        size = ALIGN_UP (nb + SIZE_SZ, pagesize);
    else
        size = ALIGN_UP (nb + SIZE_SZ + MALLOC_ALIGN_MASK, pagesize);
    tried_mmap = true;

    /* Don't try if size wraps around 0 */
    if ((unsigned long) (size) > (unsigned long) (nb))
    {
        mm = (char *) (MMAP (0, size, PROT_READ | PROT_WRITE, 0));

        if (mm != MAP_FAILED)
        {
            /*
                The offset to the start of the mmapped region is stored
                in the prev_size field of the chunk. This allows us to adjust
                returned start address to meet alignment requirements here
                and in memalign(), and still be able to compute proper
                address argument for later munmap in free() and realloc().
            */

            if (MALLOC_ALIGNMENT == 2 * SIZE_SZ)
            {
                /*
                    For glibc, chunk2mem increases the address by 2*SIZE_SZ and
                    MALLOC_ALIGN_MASK is 2*SIZE_SZ-1. Each mmap'ed area is page
                    aligned and therefore definitely MALLOC_ALIGN_MASK-aligned. */
                assert (((INTERNAL_SIZE_T) chunk2mem (mm) & MALLOC_ALIGN_MASK) == 0);
                front_misalign = 0;
            }
            else
                front_misalign = (INTERNAL_SIZE_T) chunk2mem (mm) & MALLOC_ALIGN_MASK;
            if (front_misalign > 0)
            {
                correction = MALLOC_ALIGNMENT - front_misalign;
                p = (mchunkptr) (mm + correction);
                set_prev_size (p, correction);
                set_head (p, (size - correction) | IS_MMAPPED);
            }
            else
            {
                p = (mchunkptr) mm;
                set_prev_size (p, 0);
                set_head (p, size | IS_MMAPPED);
            }
        }
    }
}

```

```

    }

    /* update statistics */

    int new = atomic_exchange_and_add (&mp_.n_mmaps, 1) + 1;
    atomic_max (&mp_.max_n_mmaps, new);

    unsigned long sum;
    sum = atomic_exchange_and_add (&mp_.mmapped_mem, size) + size;
    atomic_max (&mp_.max_mmapped_mem, sum);

    check_chunk (av, p);

    return chunk2mem (p);
}
}
}

/* There are no usable arenas and mmap also failed. */
if (av == NULL)
    return 0;

...
}

```

接下来是第二部分，代码如下：

```

static void *
sysmalloc (INTERNAL_SIZE_T nb, mstate av)
{
    ...

    /* Record incoming configuration of top */

    old_top = av->top;
    old_size = chunksize (old_top);
    old_end = (char *) (chunk_at_offset (old_top, old_size));

    brk = snd_brk = (char *) (MORECORE_FAILURE);

    /*
       If not the first time through, we require old_size to be
       at least MINSIZE and to have prev_inuse set.
    */

    assert ((old_top == initial_top (av) && old_size == 0) ||
            ((unsigned long) (old_size) >= MINSIZE &&
             prev_inuse (old_top) &&
             ((unsigned long) old_end & (pagesize - 1)) == 0));

    /* Precondition: not enough current space to satisfy nb request */
    assert ((unsigned long) (old_size) < (unsigned long) (nb + MINSIZE));

    if (av != &main_arena)
    {
        heap_info *old_heap, *heap;
        size_t old_heap_size;

        /* First try to extend the current heap. */
        old_heap = heap_for_ptr (old_top);
        old_heap_size = old_heap->size;
    }
}

```

```

if ((long) (MINSIZE + nb - old_size) > 0)
    && grow_heap (old_heap, MINSIZE + nb - old_size) == 0)
{
    av->system_mem += old_heap->size - old_heap_size;
    set_head (old_top, (((char *) old_heap + old_heap->size) - (char *) old_top)
              | PREV_INUSE);
}
else if ((heap = new_heap (nb + (MINSIZE + sizeof (*heap)), mp_.top_pad)))
{
    /* Use a newly allocated heap. */
    heap->ar_ptr = av;
    heap->prev = old_heap;
    av->system_mem += heap->size;
    /* Set up the new top. */
    top (av) = chunk_at_offset (heap, sizeof (*heap));
    set_head (top (av), (heap->size - sizeof (*heap)) | PREV_INUSE);

    /* Setup fencepost and free the old top chunk with a multiple of
       MALLOC_ALIGNMENT in size. */
    /* The fencepost takes at least MINSIZE bytes, because it might
       become the top chunk again later. Note that a footer is set
       up, too, although the chunk is marked in use. */
    old_size = (old_size - MINSIZE) & ~MALLOC_ALIGN_MASK;
    set_head (chunk_at_offset (old_top, old_size + 2 * SIZE_SZ), 0 | PREV_INUSE);
    if (old_size >= MINSIZE)
    {
        set_head (chunk_at_offset (old_top, old_size), (2 * SIZE_SZ) | PREV_INUSE);
        set_foot (chunk_at_offset (old_top, old_size), (2 * SIZE_SZ));
        set_head (old_top, old_size | PREV_INUSE | NON_MAIN_ARENA);
        _int_free (av, old_top, 1);
    }
    else
    {
        set_head (old_top, (old_size + 2 * SIZE_SZ) | PREV_INUSE);
        set_foot (old_top, (old_size + 2 * SIZE_SZ));
    }
}
else if (!tried_mmap)
    /* We can at least try to use to mmap memory. */
    goto try_mmap;
}
else /* av == main_arena */
{
    ...
}

...
}

```

首先，old\_top、old\_size和old\_end分别保存了top chunk的指针，大小以及尾部的地址。如果是thread分配区，首先通过heap\_for\_ptr获得原top chunk对应的heap\_info指针，heap\_for\_ptr的定义如下。对于thread分配区，因为每个heap是按照HEAP\_MAX\_SIZE的大小分配且对齐的，而每个top chunk存在于每个heap的剩余空间（高地址处），因此通过heap\_for\_ptr就能取出heap\_info指针，heap\_info保存了每个heap的相关信息。获得heap\_info指针后，就能获得该heap当前被使用的大小并将其保存在old\_heap\_size中。

```

#define heap_for_ptr(ptr) \
    ((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))

```

由上文可知，进入到sysmalloc前会尝试在top chunk分配内存，因此代码执行到这里肯定失败了。所以这里只有MINSIZE + nb - old\_size>0这一种情况，即这时的top chunk空间不足了，因此首先通过grow\_heap尝试向heap的高地址处增加heap当前使用的大小，即top chunk的大小，函数的定义如下：

```

/* Grow a heap: size is automatically rounded up to a

```

```

/* Grow a heap. Size is automatically rounded up to a
   multiple of the page size. */

static int
grow_heap (heap_info *h, long diff)
{
    size_t pagesize = GLRO (dl_pagesize);
    long new_size;

    diff = ALIGN_UP (diff, pagesize);
    new_size = (long) h->size + diff;
    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,
                        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;
}

```

grow\_heap最终的结果是将h->size增大到new\_size，二者的差值等于所需内存的大小按照page size对齐。假设grow\_heap成功，即将top chunk的大小设置为MINSIZE + nb，则重新设置分配区使用的内存大小，并且设置top chunk的size至新值。假设grow\_heap失败，大部分情况下说明heap的使用大小已经接近其最大值HEAP\_MAX\_SIZE了，此时只能通过new\_heap重新分配一个heap，注意传入的参数mp\_.top\_pad表示在分配内存时，额外多分配的内存。

```

static heap_info *
new_heap (size_t size, size_t top_pad)
{
    ...

    if (size + top_pad < HEAP_MIN_SIZE)
        size = HEAP_MIN_SIZE;
    else if (size + top_pad <= HEAP_MAX_SIZE)
        size += top_pad;
    else if (size > HEAP_MAX_SIZE)
        return 0;
    else
        size = HEAP_MAX_SIZE;
    size = ALIGN_UP (size, pagesize);

    /* 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)
    {
        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)))
        {
            __munmap (p2, HEAP_MAX_SIZE);
            p2 = MAP_FAILED;
        }
    }
}

```



```

    }
}
if (p2 == MAP_FAILED)
{
    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));

        ul = p2 - p1;
        if (ul)
            __munmap (p1, ul);
        else
            aligned_heap_area = p2 + HEAP_MAX_SIZE;
        __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;
        }
    }
}
if (__mprotect (p2, size, PROT_READ | PROT_WRITE) != 0)
{
    __munmap (p2, HEAP_MAX_SIZE);
    return 0;
}
h = (heap_info *) p2;
h->size = size;
h->mprotect_size = size;
LIBC_PROBE (memory_heap_new, 2, h, h->size);
return h;
}

```

首先对需要分配的内存大小size做相应的调整。aligned\_heap\_area表示上一次MMAP分配后的结束地址，如果存在，就首先尝试从该地址分配大小为HEAP\_MAX\_SIZE的内存。MMAP最后是系统调用，这里只是一些标志位的区别。分配完后，会检查地址是否对齐，如果不对齐也是失败。如果第一次分配失败了，就会再尝试一次，这次分配HEAP\_MAX\_SIZE \* 2大小的内存，并且新内存的起始地址由内核决定。因为尝试分配了HEAP\_MAX\_SIZE\*2大小的内存，其中必定包含了大小为HEAP\_MAX\_SIZE且和HEAP\_MAX\_SIZE对齐的内存，因此一旦分配成功，就从中截取出这部分内存。

如果连第二次也分配失败了，就会通过MMAP进行第三次分配，这次只分配HEAP\_MAX\_SIZE大小的内存，并且起始地址由内核决定，如果又失败了就返回0。

如果三面三次分配内存任何一次成功，就设置相应的可读写位置，并且返回分配区的heap\_info指针。

重新回到sysmalloc中，假设分配成功，就会对刚刚分配得到的heap做相应的设置，其中ar\_ptr表示所属的分配区的指针，prev表示上一个heap，所有的heap通过prev形成单向链表，然后通过set\_head设置av分配区top chunk的size，这里也可以看出，对于刚分配的heap，包含了heap\_info指针、top chunk、以及大于size的未被使用的部分。

再接下来就要对原来的top chunk进行最后的处理，这里假设对齐，如果原top chunk的大小不够大，就将其分割成old\_size + 2 \* SIZE\_SZ和2 \* SIZE\_SZ大小；如果原top chunk的大小足够大，就将其分割成old\_size，2 \* SIZE\_SZ和2 \* SIZE\_SZ大小，并通过\_int\_free进行释放。

接下来是第三部分，代码如下：

```

static void *
sysmalloc (INTERNAL_SIZE_T nb, mstate av)
{
    ...

    if (av != &main_arena)
    {

```

```

...
}
else /* av == main_arena */
{ /* Request enough space for nb + pad + overhead */
    size = nb + mp_.top_pad + MINSIZE;

    /*
     * If contiguous, we can subtract out existing space that we hope to
     * combine with new space. We add it back later only if
     * we don't actually get contiguous space.
     */

    if (contiguous (av))
        size -= old_size;

    /*
     * Round to a multiple of page size.
     * If MORECORE is not contiguous, this ensures that we only call it
     * with whole-page arguments. And if MORECORE is contiguous and
     * this is not first time through, this preserves page-alignment of
     * previous calls. Otherwise, we correct to page-align below.
     */

    size = ALIGN_UP (size, pagesize);

    /*
     * Don't try to call MORECORE if argument is so big as to appear
     * negative. Note that since mmap takes size_t arg, it may succeed
     * below even if we cannot call MORECORE.
     */

    if (size > 0)
    {
        brk = (char *) (MORECORE (size));
        LIBC_PROBE (memory_sbrk_more, 2, brk, size);
    }

    if (brk != (char *) (MORECORE_FAILURE))
    {
        /* Call the 'morecore' hook if necessary. */
        void (*hook) (void) = atomic_forced_read (__after_morecore_hook);
        if (__builtin_expect (hook != NULL, 0))
            (*hook)();
    }
    else
    {
        /*
         * If have mmap, try using it as a backup when MORECORE fails or
         * cannot be used. This is worth doing on systems that have "holes" in
         * address space, so sbrk cannot extend to give contiguous space, but
         * space is available elsewhere. Note that we ignore mmap max count
         * and threshold limits, since the space will not be used as a
         * segregated mmap region.
         */

        /* Cannot merge with old top, so add its size back in */
        if (contiguous (av))
            size = ALIGN_UP (size + old_size, pagesize);

        /* If we are relying on mmap as backup, then use larger units */
        if ((unsigned long) (size) < (unsigned long) (MMAP_AS_MORECORE_SIZE))
            size = MMAP_AS_MORECORE_SIZE;

        /* Don't try if size wraps around 0 */
        if ((unsigned long) (size) > (unsigned long) (nb))
            r

```

```

1
char *mbrk = (char *) (MMAP (0, size, PROT_READ | PROT_WRITE, 0));

if (mbrk != MAP_FAILED)
{
    /* We do not need, and cannot use, another sbrk call to find end */
    brk = mbrk;
    snd_brk = brk + size;

    /*
     Record that we no longer have a contiguous sbrk region.
     After the first time mmap is used as backup, we do not
     ever rely on contiguous space since this could incorrectly
     bridge regions.
    */
    set_noncontiguous (av);
}
}
}

...

}

...

}

```

以上代码的关键是通过MORECORE向内核申请内存，MORECORE是一个宏定义，其最终是通过系统调用分配内存。假设分配成功，会查找是否有\_\_after\_morecore\_hook函数并执行，这里假设该函数指针为null。假设分配失败，则进入else部分，首先对需要分配的大小按地址对齐，并且设置分配size的最小值为MMAP\_AS\_MORECORE\_SIZE（1MB），然后通过MMAP宏分配内存。这里注意，如果是通过mmap分配的内存，则设置分配区为不连续标志位。

接下来是第四部分，代码如下：

```

static void * sysmalloc (INTERNAL_SIZE_T nb, mstate av) {
    ...
    if (av != &main_arena) {
        ...
    }
    else{

        ...

        if (brk != (char *) (MORECORE_FAILURE))
        {
            if (mp_.sbrk_base == 0)
                mp_.sbrk_base = brk;
            av->system_mem += size;

            /*
             If MORECORE extends previous space, we can likewise extend top size.
            */

            if (brk == old_end && snd_brk == (char *) (MORECORE_FAILURE))
                set_head (old_top, (size + old_size) | PREV_INUSE);

            else if (contiguous (av) && old_size && brk < old_end)
                /* Oops! Someone else killed our space.. Can't touch anything. */
                malloc_printerr ("break adjusted to free malloc space");

            /*
             Otherwise, make adjustments:

            * If the first time through or noncontiguous, we need to call sbrk

```

```

just to find out where the end of memory lies.

* We need to ensure that all returned chunks from malloc will meet
  MALLOC_ALIGNMENT

* If there was an intervening foreign sbrk, we need to adjust sbrk
  request size to account for fact that we will not be able to
  combine new space with existing space in old_top.

* Almost all systems internally allocate whole pages at a time, in
  which case we might as well use the whole last page of request.
  So we allocate enough more memory to hit a page boundary now,
  which in turn causes future contiguous calls to page-align.
*/

else
{
    front_misalign = 0;
    end_misalign = 0;
    correction = 0;
    aligned_brk = brk;

    /* handle contiguous cases */
    if (contiguous (av))
    {
        /* Count foreign sbrk as system_mem. */
        if (old_size)
            av->system_mem += brk - old_end;

        /* Guarantee alignment of first new chunk made from this space */

        front_misalign = (INTERNAL_SIZE_T) chunk2mem (brk) & MALLOC_ALIGN_MASK;
        if (front_misalign > 0)
        {
            /*
             * Skip over some bytes to arrive at an aligned position.
             * We don't need to specially mark these wasted front bytes.
             * They will never be accessed anyway because
             * prev_inuse of av->top (and any chunk created from its start)
             * is always true after initialization.
             */

            correction = MALLOC_ALIGNMENT - front_misalign;
            aligned_brk += correction;
        }

        /*
         * If this isn't adjacent to existing space, then we will not
         * be able to merge with old_top space, so must add to 2nd request.
         */

        correction += old_size;

        /* Extend the end address to hit a page boundary */
        end_misalign = (INTERNAL_SIZE_T) (brk + size + correction);
        correction += (ALIGN_UP (end_misalign, pagesize)) - end_misalign;

        assert (correction >= 0);
        snd_brk = (char *) (MORECORE (correction));

        /*
         * If can't allocate correction, try to at least find out current
         * brk. It might be enough to proceed without failing.

         Note that if second sbrk did NOT fail, we assume that space

```

```

        is contiguous with first sbrk. This is a safe assumption unless
        program is multithreaded but doesn't use locks and a foreign sbrk
        occurred between our first and second calls.
    */

    if (snd_brk == (char *) (MORECORE_FAILURE))
    {
        correction = 0;
        snd_brk = (char *) (MORECORE (0));
    }
    else
    {
        /* Call the 'morecore' hook if necessary. */
        void (*hook) (void) = atomic_forced_read (__after_morecore_hook);
        if (__builtin_expect (hook != NULL, 0))
            (*hook)();
    }
}

...

}
}
}

...

}

```

假设增加了主分配区的top chunk成功，则更新sbrk\_base和分配区已分配的内存大小。然后，第一个判断表示，新分配的内存地址和原来的top chunk连续，并且不是通过MMAP分配的，这时只需要更新原来top chunk的大小size。第二个判断表示如果分配区的连续标志位置位，top chunk的大小大于0，但是分配的brk小于原来的top chunk结束地址，这里就判定出错了。进入第三个判断表示新分配的内存地址大于原来的top chunk的结束地址，但是不连续。这种情况下，如果分配区的连续标志位置位，则表示不是通过MMAP分配的，肯定有其他线程调用了brk在堆上分配了内存，av->system\_mem += brk - old\_end表示将其他线程分配的内存一并计入到该分配区分配的内存大小。然后将刚刚分配的地址brk按MALLOC\_ALIGNMENT对齐。再往下就要处理地址不连续的问题了，因为地址不连续，就要放弃原来top chunk后面一部分的内存大小，并且将这一部分内存大小“补上”到刚刚分配的新内存后面。首先计算堆上补上内存后的结束地址并保存在correction中，然后调用MORECORE继续分配一次，将新分配内存的开始地址保存在snd\_brk中。如果分配失败，则将correction设为0，并将snd\_brk重置为原来分配的内存的结束地址，表示放弃该次补偿操作；如果分配成功，就调用\_\_after\_morecore\_hook函数，这里假设该函数指针为null。

接下来是第五部分，代码如下：

```

static void * sysmalloc (INTERNAL_SIZE_T nb, mstate av) {
    ...
    if (av != &main_arena) {
        ...
    }
    else {
        ...
        if (brk != (char *) (MORECORE_FAILURE)) {
            ...
            if (brk == old_end && snd_brk == (char *) (MORECORE_FAILURE))
                ...
            else if (contiguous (av) && old_size && brk < old_end) {
                ...
            }
            else {
                ...
                if (contiguous (av)) {
                    ...
                }

                /* handle non-contiguous cases */
            }
            else
            {
                if (MALLOC_ALIGNMENT == 2 * SIZE_SZ)
                    ...
                ...
            }
        }
    }
}

```

```

/* MURKIN/mmap must correctly align */
assert (((unsigned long) chunk2mem (brk) & MALLOC_ALIGN_MASK) == 0);
else
{
    front_misalign = (INTERNAL_SIZE_T) chunk2mem (brk) & MALLOC_ALIGN_MASK;
    if (front_misalign > 0)
    {
        /*
         * Skip over some bytes to arrive at an aligned position.
         * We don't need to specially mark these wasted front bytes.
         * They will never be accessed anyway because
         * prev_inuse of av->top (and any chunk created from its start)
         * is always true after initialization.
         */

        aligned_brk += MALLOC_ALIGNMENT - front_misalign;
    }
}

/* Find out current end of memory */
if (snd_brk == (char *) (MORECORE_FAILURE))
{
    snd_brk = (char *) (MORECORE (0));
}

/* Adjust top based on results of second sbrk */
if (snd_brk != (char *) (MORECORE_FAILURE))
{
    av->top = (mchunkptr) aligned_brk;
    set_head (av->top, (snd_brk - aligned_brk + correction) | PREV_INUSE);
    av->system_mem += correction;

    /*
     * If not the first time through, we either have a
     * gap due to foreign sbrk or a non-contiguous region. Insert a
     * double fencepost at old_top to prevent consolidation with space
     * we don't own. These fenceposts are artificial chunks that are
     * marked as inuse and are in any case too small to use. We need
     * two to make sizes and alignments work out.
     */

    if (old_size != 0)
    {
        /*
         * Shrink old_top to insert fenceposts, keeping size a
         * multiple of MALLOC_ALIGNMENT. We know there is at least
         * enough space in old_top to do this.
         */
        old_size = (old_size - 4 * SIZE_SZ) & ~MALLOC_ALIGN_MASK;
        set_head (old_top, old_size | PREV_INUSE);

        /*
         * Note that the following assignments completely overwrite
         * old_top when old_size was previously MINSIZE. This is
         * intentional. We need the fencepost, even if old_top otherwise gets
         * lost.
         */
        set_head (chunk_at_offset (old_top, old_size),
            (2 * SIZE_SZ) | PREV_INUSE);
        set_head (chunk_at_offset (old_top, old_size + 2 * SIZE_SZ),
            (2 * SIZE_SZ) | PREV_INUSE);

        /* If possible, release the rest. */
        if (old_size >= MINSIZE)
        {

```

```

        _int_free (av, old_top, 1);
    }
}

}

}

} /* if (av != &main_arena) */

...

}

```

开头的else表示分配区的连续标志没有置位，这时只要按照MALLOc\_ALIGNMENT做简单的对齐就行了，如果是通过brk分配的内存，则通过MORECORE (0)得到新分配的内存的结束地址并保存在snd\_brk中。再往下进入if，设置分配区的top指针为经过对齐之后的起始地址aligned\_brk，设置top chunk的大小size，aligned\_brk表示对齐造成的误差，correction是因为要补偿原来top chunk剩余内存造成的误差，然后设置分配区已分配的内存大小。因为不连续，最后if内是设置原top chunk的fencepost，将原来top chunk的剩余空间拆成两个SIZE\_SZ\*2大小的chunk，如果剩下的的大小大于可分配的chunk的最小值MINSIZE，就通过\_int\_free释放掉整个剩余内存。

最后一部分代码如下：

```
static void * sysmalloc (INTERNAL_SIZE_T nb, mstate av) {
    ...

    if ((unsigned long) av->system_mem > (unsigned long) (av->max_system_mem))
        av->max_system_mem = av->system_mem;

    check_malloc_state (av);

    /* finally, do the allocation */
    p = av->top;
    size = chunksize (p);

    /* check that one of the above allocation paths succeeded */
    if ((unsigned long) (size) >= (unsigned long) (nb + MINSIZE))
    {
        remainder_size = size - nb;
        remainder = chunk_at_offset (p, nb);
        av->top = remainder;
        set_head (p, nb | PREV_INUSE | (av != &main_arena ? NON_MAIN_ARENA : 0));
        set_head (remainder, remainder_size | PREV_INUSE);
        check_mallocated_chunk (av, p, nb);
        return chunk2mem (p);
    }

    /* catch all failure paths */
    __set_errno (ENOMEM);
    return 0;
}
```

这里就是获得前面所有代码更新后的top chunk，然后从该top chunk中分配用户需要的大小chunk并返回，如果失败则返回0。

## # 参考资料

Ptmalloc2 源代码分析 华庭 (庄明强)

Linux 堆内存管理深入分析：<https://murphypei.github.io/blog/2019/01/linux-heap>

glibc tcache 机制：[https://firmianay.gitbooks.io/ctf-all-in-one/content/doc/4.14\\_glibc\\_tcache.html](https://firmianay.gitbooks.io/ctf-all-in-one/content/doc/4.14_glibc_tcache.html)

### malloc源码分析：

[illegible]

【版权声明】 Copyright © 2021 openEuler Community。本文由openEuler社区首发，欢迎遵照 CC-BY-SA 4.0 协议规定转载。转载时敬请在正文注明并保留原文链接和作者信息。

【免责声明】本文仅代表作者本人观点，与本网站无关。本网站对文中陈述、观点判断保持中立，不对所包含内容的准确性、可靠性或完整性提供任何明示或暗示的保证。本文仅供读者参考，由此产生的所有法律责任均由读者本人承担。

contact@openeuler.io

[品牌](#) [隐私政策](#) [法律声明](#)

版权所有 © 2021 openEuler 保留一切权利



扫码关