Ptmalloc

> 如有错误以及图片不清晰等问题请提交 issue，谢谢～
>
> 源路径：[https://github.com/HATTER-LONG/glibc_Learnging]

_int_malloc 之 large bin

- 参考文章--浅析 largebin attack
- 参考文章--largebin 学习从源码到做题

前文已经介绍了 small bin 以及当其无法满足需求时，需要使用 large bin，并且会进行 fast bin 的内存合并相关操作，接下来开始分析 _int_malloc 后续的分配流程。

1. 首先判断 tcache 是否存在当前申请的内存块，初始化一些局部变量。

```
#if USE_TCACHE
INTERNAL_SIZE_T tcache_nb = 0;
size_t tc_idx = csize2tidx (nb);
if (tcache && tc_idx < mp_.tcache_bins)
    tcache_nb = nb;
int return_cached = 0;

tcache_unsorted_count = 0;
#endif
```

2. 两个循环，最外层用于无法满足需求用于合并重试，里层遍历所有 unsorted 节点：

```
//这里需要外循环，因为可能直到 malloc 结束时才意识到应该合并，所以必须这样做并重试。
//这最多发生一次，并且仅在需要扩展内存以服务"小"请求时才发生。
for (;; )
    {
    int iters = 0;
    while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av)) // 判断 unsorted 不为空
        {
        bck = victim->bk; //取出 unsorted bin 中倒数第二个空闲的 chunk
        size = chunksize (victim);
        mchunkptr next = chunk_at_offset (victim, size);  // 偏移到再下一个 chunk 头 用于检查链表是否正常

        if (__glibc_unlikely (size <= 2 * SIZE_SZ)
            || __glibc_unlikely (size > av->system_mem))
            malloc_printerr ("malloc(): invalid size (unsorted)");
        if (__glibc_unlikely (chunksize_nomask (next) < 2 * SIZE_SZ)
            || __glibc_unlikely (chunksize_nomask (next) > av->system_mem))
            malloc_printerr ("malloc(): invalid next size (unsorted)");
        if (__glibc_unlikely ((prev_size (next) & ~(SIZE_BITS)) != size))
            malloc_printerr ("malloc(): mismatching next->prev_size (unsorted)");
        if (__glibc_unlikely (bck->fd != victim)
            || __glibc_unlikely (victim->fd != unsorted_chunks (av)))
            malloc_printerr ("malloc(): unsorted double linked list corrupted");
        if (__glibc_unlikely (prev_inuse (next)))
            malloc_printerr ("malloc(): invalid next->prev_inuse (unsorted)");

            .......
        }
    }
```

3. 经过检查后进入到内存分配流程，当小内存请求，且 unsorted 中仅剩余一个 chunk 时，使用 last_remiainder 进行切割分配：

```
      /*
         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
         no exact fit for a small chunk.
         如果是小请求，则尝试使用最后一个余数只有未排序 bin 中的块。
         这有助于促进本地化运行连续的小请求。
         这是唯一最佳拟合的例外，仅适用于不完全合适一小块内存。
       */

      if (in_smallbin_range (nb) &&
          bck == unsorted_chunks (av) && //当剩余 1 个未使用的 chunk
          victim == av->last_remainder && // 指向上一个 chunk 分配出一个 small chunk 之后剩余的部分
          (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
        {
          /* split and reattach remainder */
          remainder_size = size - nb; //减去所需的内存
          remainder = chunk_at_offset (victim, nb);
          unsorted_chunks (av)->bk = unsorted_chunks (av)->fd = remainder;
          av->last_remainder = remainder; //更新
          remainder->bk = remainder->fd = unsorted_chunks (av); //将删减过剩余的内存重新添加回链表
          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_malloced_chunk (av, victim, nb);
          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;
        }
```

4. 将 victim 从 unsorted list 中移除，判断是否符合申请的内存大小，如果符合且 tcache 相应链表未满，则先将对应 chunk 放入 tcache 队列中，反之 返回给用户使用。

```
        /* 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) //如果当前 chunk 的大小符合申请
          {
              set_inuse_bit_at_offset (victim, size); //设置当前块为使用状态
              if (av != &main_arena) //非主分区设置标志位
                set_non_main_arena (victim);
      #if USE_TCACHE
              /* Fill cache first, return to user only if cache fills.
              We may return one of these chunks later.
              先填充缓存，缓存满才返回给用户。我们稍后可能会返回这些块之一。*/
              if (tcache_nb
              && tcache->counts[tc_idx] < mp_.tcache_count)
              {
                  tcache_put (victim, tc_idx);
                  return_cached = 1;
                  continue;
              }
              else
              {
      #endif
                  check_malloced_chunk (av, victim, nb);
                  void *p = chunk2mem (victim); // 返回用户内存地址
                  alloc_perturb (p, bytes);
                  return p;
      #if USE_TCACHE
              }
      #endif
          }
```

5. 当前将遍历到的 unrsored bin 插入对应的 small bin 或者 large bin 中，需要了解一点 large chunk 中有两个特殊的指针，如下：

   - 指针 fd_nextsize : 该指针指向下一个大小更小的 chunk (注意可能不是相邻 chunk) ;
   - 指针 bk_nextsize : 该指针指向前一个大小更大的 chunk (注意可能不是相邻 chunk) 。

```
    /* place chunk in bin */

    if (in_smallbin_range (size))
      {
        // 如果申请的大小在 small bin 的范围则取出对应的 对应大小链的头节点与前一个个节点
        victim_index = smallbin_index (size);
        bck = bin_at (av, victim_index);
        fwd = bck->fd;
      }
    else
      {
        // 不符合 small bin 的范围则使用 large bin，同样是取出对应大小链的头节点与前一个节点
        victim_index = largebin_index (size);
        bck = bin_at (av, victim_index);
        fwd = bck->fd;

        /* maintain large bins in sorted order
          large bins 是按照顺序进行维护的*/
        if (fwd != bck) // large bins 不为空
          {
            /* 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))
              {
                // 如果小于当前 large bin 链表最小的 chunk 则直接加到链表尾即可
                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; // large bin 需要将 nextsize 链表加上
              }
            else
              {
                // 否则遍历链表 nextsize 找到合适位置插入
                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;
      }

    //最终插入 victim 到合适的链表上，并将 large bin 所对应 binmap 的相应 bit 置位。 关于 binmap 后边会介绍
    mark_bin (av, victim_index);
    victim->bk = bck;
    victim->fd = fwd;
    fwd->bk = victim;
    bck->fd = victim;
```

6. (4) 中描述了将符合申请块大小的 chunk 缓存进入了 Tcache 中后没有立即返回给调用者，而是继续遍历所有相同的块都放入 Tcache 中，直到这里才会返回合适的内存。

```
#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 && mp_.tcache_unsorted_limit > 0 &&
                tcache_unsorted_count > mp_.tcache_unsorted_limit) {
            return tcache_get(tc_idx);
            }
#endif

#define MAX_ITERS 10000
            if (++iters >= MAX_ITERS)
            break;
        } // while 循环退出，表示 unsorted bin 以处理完，如果 return_cached 同样没有置为 1 表示没有符合的 chunk.

#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
```

7. 遍历 large bin 查找合适的 chunk 块，并尝试进行切割节约内存：

```
      /*
     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.
     如果请求很大，则扫描当前 bin 中的块排序顺序以找到适合的最小的。
   */

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

      /* 如果空块或最大块太小则跳过扫描*/
      if ((victim = first (bin)) != bin
      && (unsigned long) chunksize_nomask (victim)
        >= (unsigned long) (nb))
        {
            // 遍历查找合适大小的 chunk
          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.
             跳过相同大小的下一个 chunk ，避免后续无效遍历
            */
          if (victim != last (bin)
      && chunksize_nomask (victim)
        == chunksize_nomask (victim->fd))
            victim = victim->fd;

          remainder_size = size - nb;
          unlink_chunk (av, victim); //从链表中删除 chunk

          /* Exhaust 如果申请大小与 chunksize 差距小于 MINSIZE 直接使用此 chunk, 反之进行切割*/
          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); // 偏移取出多分配的内存
              /* We cannot assume the unsorted list is empty and therefore
                 have to perform a complete insert here.  */
              bck = unsorted_chunks (av); // 将其存入 unsorted 中
              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_malloced_chunk (av, victim, nb);
          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;
        }
    }
```

8. binmap 中的每一位表示对应的 bin 中是否存在空闲的 chunk，4 个 block 来管理，每个 block 有 4 字节，也就是 128 个 bit。通过 binmap 机制快速查找所有 large bin 所包含的空闲内存是否有符合标准的。

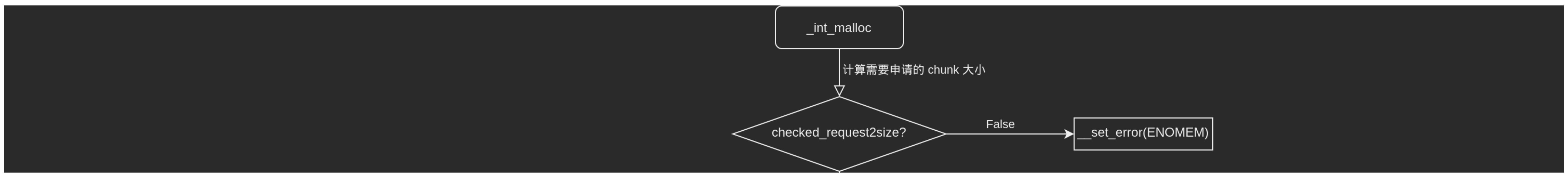
    - 参考文章--glibc-malloc 源码分析

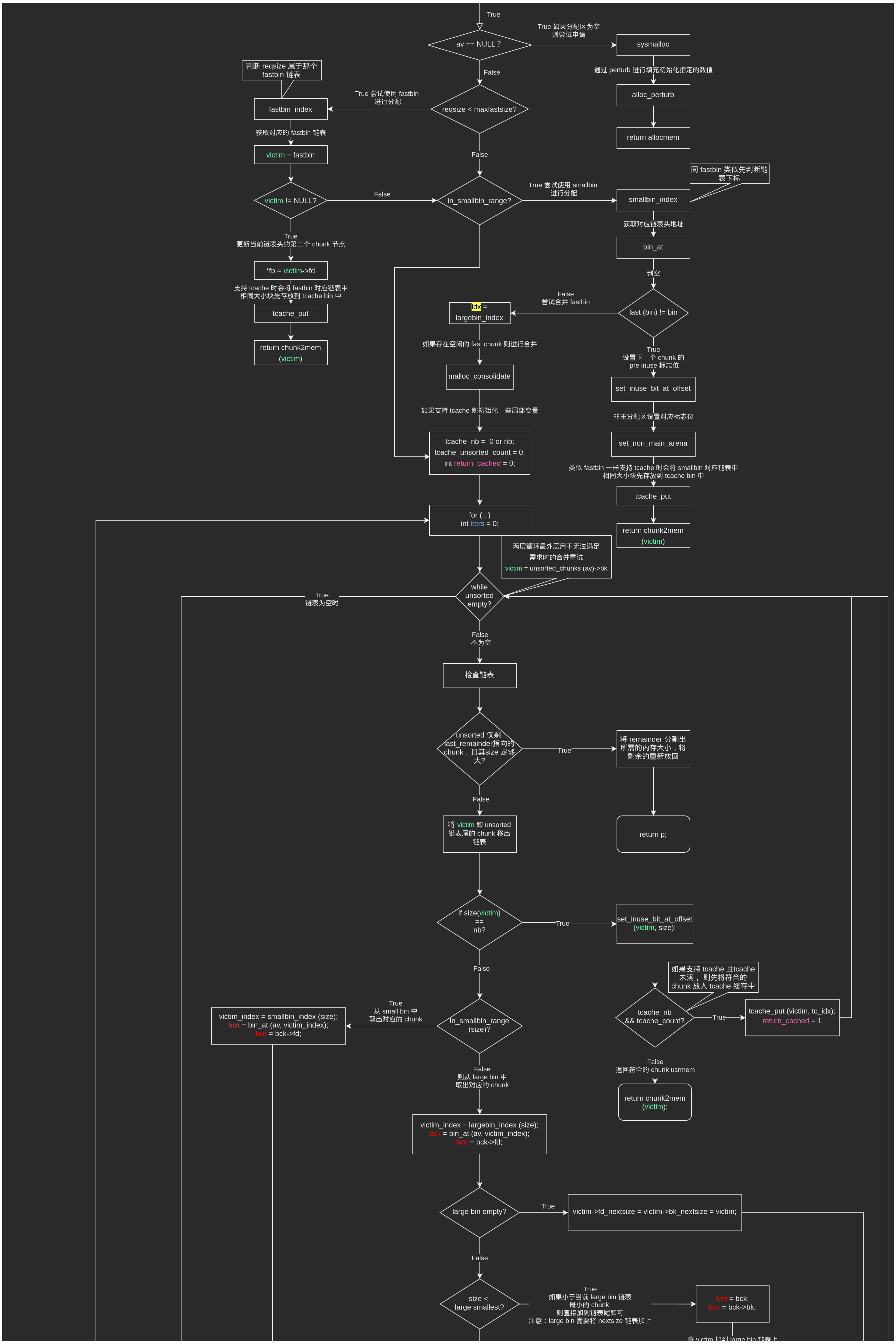
```
  ///////////////////////////////////////
  /*
  Binmap

      To help compensate for the large number of bins, a one-level index
      structure is used for bin-by-bin searching.  `binmap' is a
      bitvector recording whether bins are definitely empty so they can
      be skipped over during during traversals.  The bits are NOT always
      cleared as soon as bins are empty, but instead only
      when they are noticed to be empty during traversal in malloc.
  二进制映射

  为了帮助补偿大量的 bin，一个一级索引结构用于逐个 bin 搜索。 `binmap' 是一个 bitvector 记录 bin 是否绝对为空，
  以便它们可以在遍历过程中被跳过。这些位并不总是一旦 bin 为空，就会被清除，而只会当在 malloc 中遍历期间注意到它们为空时。
  */

  /* Conservatively use 32 bits per map word, even if on 64bit system */
  #define BINMAPSHIFT      5
  #define BITSPERMAP       (1U << BINMAPSHIFT)
  #define BINMAPSIZE       (NBINS / BITSPERMAP)

  #define idx2block(i)     ((i) >> BINMAPSHIFT)
  #define idx2bit(i)       ((1U << ((i) & ((1U << BINMAPSHIFT) - 1))))

  #define mark_bin(m, i)    ((m)->binmap[idx2block (i)] |= idx2bit (i))
  #define unmark_bin(m, i)  ((m)->binmap[idx2block (i)] &= ~(idx2bit (i)))
  #define get_binmap(m, i)  ((m)->binmap[idx2block (i)] & idx2bit (i))

  ///////////////////////////////////////
/*
   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.
 */
  /*
  当前 index 的 large bin 可能为空导致无法分配 [(7) 小结中有判断]， idx 递增开始查看比当前 bin 的 index 大的 small bin 或 large bin 是否有空闲 chunk 可利用来分配所需的 chunk。
  这里使用到了 Binmap 进行索引管理，快速判断哪些 bin 有合适的 chunk 块
  */
++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.
            们知道这个 bin 中的第一个块足够大可以使用*/
        assert ((unsigned long) (size) >= (unsigned long) (nb));

        remainder_size = size - nb;

        /* unlink */
        unlink_chunk (av, victim);

        /* 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.  */
```

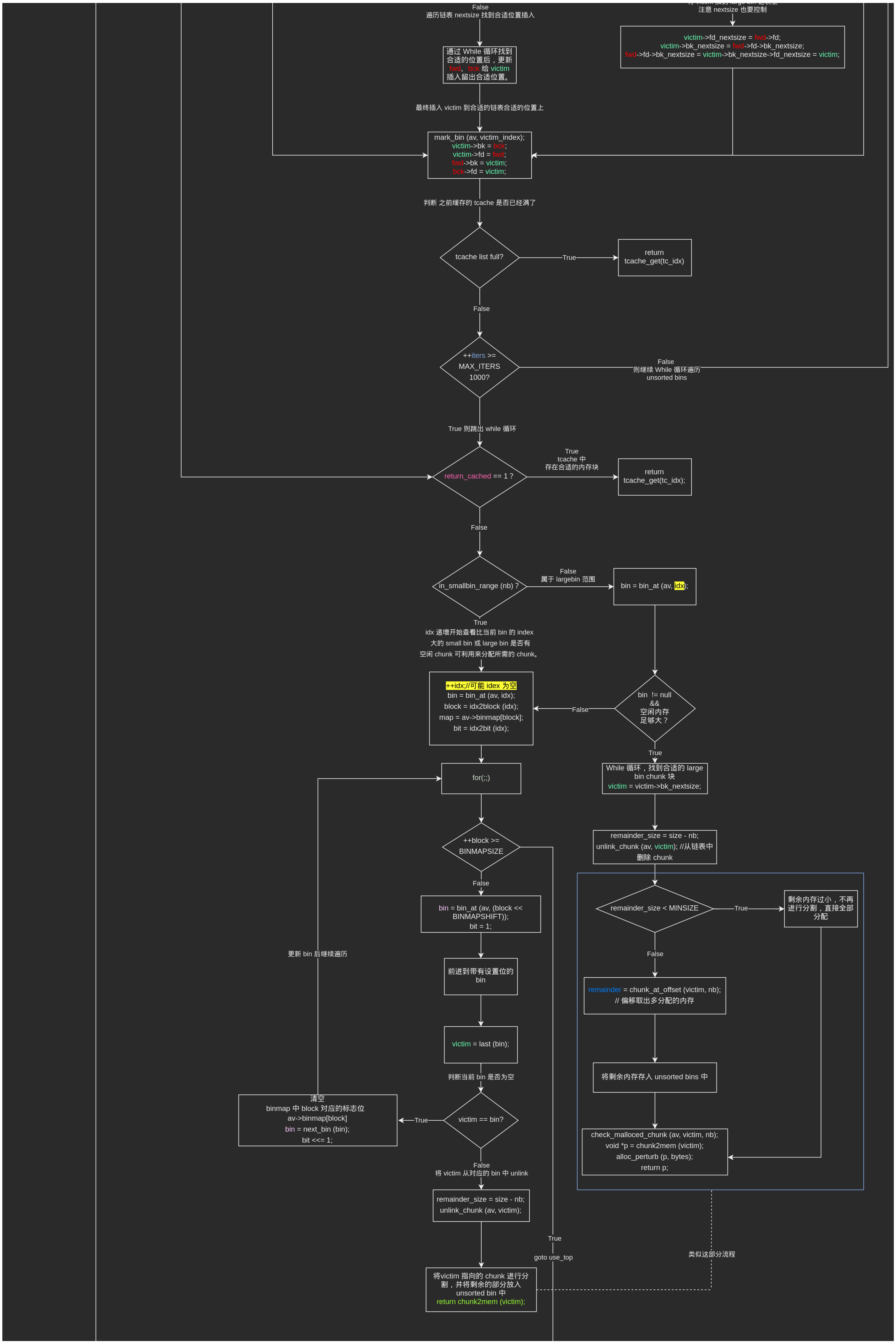
```
                bck = unsorted_chunks (av);
                fwd = bck->fd;
        if (__glibc_unlikely (fwd->bk != bck))
          malloc_printerr ("malloc(): corrupted unsorted chunks 2");
                remainder->bk = bck;
                remainder->fd = fwd;
                bck->fd = remainder;
                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_malloced_chunk (av, victim, nb);
            void *p = chunk2mem (victim);
            alloc_perturb (p, bytes);
            return p;
          }
      }
```

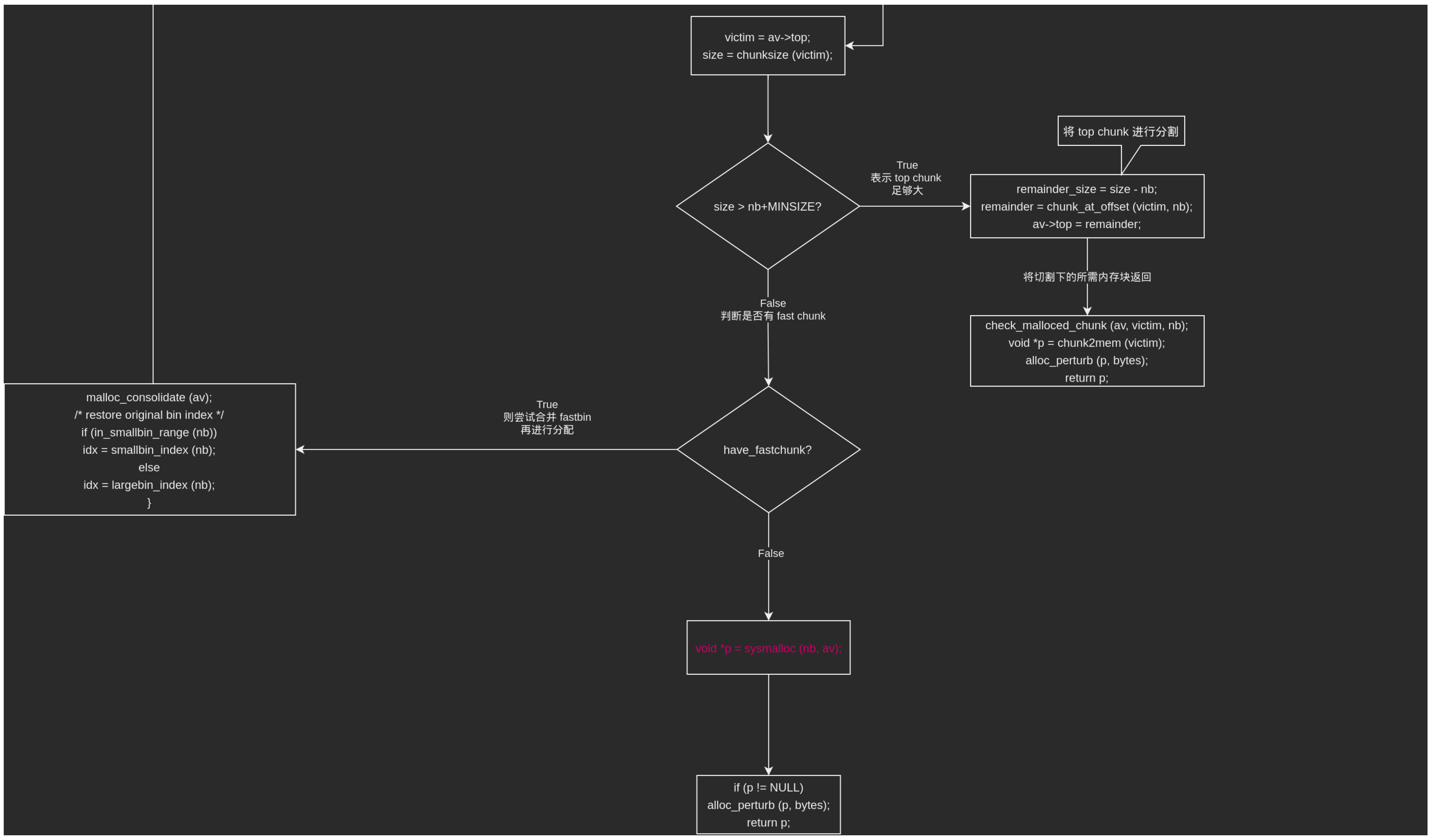
9. use_top 处理，尝试使用分配区 top 节点进行，如果上边所有方法都不能再分配出内存只能通过 sysmalloc 向系统申请：

```
    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.)

        如果足够大，则拆分与内存末尾接壤的块（保存在 av->top 中）。请注意，这符合最佳拟合搜索规则。
        实际上， av->top 被视为更大（因此不太适合）比任何其他可用的块，因为它可以根据需要扩展到尽可能大（直到系统限制）。

        我们要求 av->top 始终存在（即，大小 >= MINSIZE) 初始化后，所以如果否则当前请求耗尽，它被补充。
      （主要的确保它存在的原因是我们可能需要 MINSIZE 空间在 sysmalloc 中放入 bin。）
    */

    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))
        {
            //将 top chunk 划分
            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_malloced_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.  */
    //如果 top chunk 不够，则合并 fastbin，再次尝试使用 small bin 或者 large bin 进行分配
    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
    */
    //如果仍然不够，则调用 sysmalloc 继续分配堆块
    else
        {
        void *p = sysmalloc (nb, av);
        if (p != NULL)
            alloc_perturb (p, bytes);
        return p;
        }
    }
}
```




True
av == NULL ?
True 如果分配区为空
则尝试申请
sysmalloc
通过 perturb 进行填充初始化指定的数值
alloc_perturb
return allocmem
False
判断 reqsize 属于那个
fastbin 链表
fastbin_index
True 尝试使用 fastbin
进行分配
reqsize < maxfastsize?
获取对应的 fastbin 链表
victim = fastbin
False
同 fastbin 类似先判断链
表下标
victim != NULL?
False
in_smallbin_range?
True 尝试使用 smallbin
进行分配
smallbin_index
获取对应链表头地址
bin_at
True
更新当前链表头的第二个 chunk 节点
*fb = victim->fd
判空
支持 tcache 时会将 fastbin 对应链表中
相同大小块先存放到 tcache bin 中
tcache_put
return chunk2mem
(victim)
idx =
largebin_index
False
尝试合并 fastbin
last (bin) != bin
如果存在空闲的 fast chunk 则进行合并
True
设置下一个 chunk 的
pre inuse 标志位
malloc_consolidate
set_inuse_bit_at_offset
如果支持 tcache 则初始化一些局部变量
非主分配区设置对应标志位
tcache_nb = 0 or nb;
tcache_unsorted_count = 0;
int return_cached = 0;
set_non_main_arena
类似 fastbin 一样支持 tcache 时会将 smallbin 对应链表中
相同大小块先存放到 tcache bin 中
tcache_put
for (;;)
int iters = 0;
return chunk2mem
(victim)
两层循环最外层用于无法满足
需求时的合并重试
victim = unsorted_chunks (av)->bk
True
链表为空时
while
unsorted
empty?
False
不为空
检查链表
unsorted 仅剩
last_remainder指向的
chunk，且其size 足够
大?
True
将 remainder 分割出
所需的内存大小，将
剩余的重新放回
False
return p;
将 victim 即 unsorted
链表尾的 chunk 移出
链表
if size(victim)
==
nb?
True
set_inuse_bit_at_offset
(victim, size);
False
如果支持 tcache 且tcache
未满， 则先将符合的
chunk 放入 tcache 缓存中
victim_index = smallbin_index (size);
bck = bin_at (av, victim_index);
fwd = bck->fd;
True
从 small bin 中
取出对应的 chunk
in_smallbin_range
(size)?
tcache_nb
&& tcache_count?
True
tcache_put (victim, tc_idx);
return_cached = 1
False
则从 large bin 中
取出对应的 chunk
False
返回符合的 chunk usrmem
return chunk2mem
(victim);
victim_index = largebin_index (size);
bck = bin_at (av, victim_index);
fwd = bck->fd;
large bin empty?
True
victim->fd_nextsize = victim->bk_nextsize = victim;
False
size <
large smallest?
True
如果小于当前 large bin 链表
最小的 chunk
则直接加到链表尾即可
注意：large bin 需要将 nextsize 链表加上
fwd = bck;
bck = bck->bk;
将 victim 加到 large bin 链表上


False
遍历链表 nextsize 找到合适位置插入
注意 nextsize 也要控制
victim->fd_nextsize = fwd->fd;
victim->bk_nextsize = fwd->fd->bk_nextsize;
fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
通过 While 循环找到合适的位置后，更新 fwd、bck 给 victim 插入留出合适位置。
最终插入 victim 到合适的链表合适的位置上
mark_bin (av, victim_index);
victim->bk = bck;
victim->fd = fwd;
fwd->bk = victim;
bck->fd = victim;
判断 之前缓存的 tcache 是否已经满了
tcache list full?
True
return tcache_get(tc_idx)
False
++iters >= MAX_ITERS 1000?
False
则继续 While 循环遍历 unsorted bins
True 则跳出 while 循环
return_cached == 1 ?
True
tcache 中
存在合适的内存块
return tcache_get(tc_idx);
False
in_smallbin_range (nb) ?
False
属于 largebin 范围
bin = bin_at (av, idx);
True
idx 递增开始查看比当前 bin 的 index
大的 small bin 或 large bin 是否有
空闲 chunk 可利用来分配所需的 chunk。
++idx;//可能 idex 为空
bin = bin_at (av, idx);
block = idx2block (idx);
map = av->binmap[block];
bit = idx2bit (idx);
bin != null && 空闲内存足够大？
False
True
While 循环，找到合适的 large bin chunk 块
victim = victim->bk_nextsize;
for(;;)
remainder_size = size - nb;
unlink_chunk (av, victim); //从链表中删除 chunk
++block >= BINMAPSIZE
False
bin = bin_at (av, (block << BINMAPSHIFT));
bit = 1;
remainder_size < MINSIZE
True
剩余内存过小，不再进行分割，直接全部分配
False
remainder = chunk_at_offset (victim, nb);
// 偏移取出多分配的内存
将剩余内存存入 unsorted bins 中
check_malloced_chunk (av, victim, nb);
void *p = chunk2mem (victim);
alloc_perturb (p, bytes);
return p;
更新 bin 后继续遍历
前进到带有设置位的 bin
victim = last (bin);
判断当前 bin 是否为空
清空
binmap 中 block 对应的标志位
av->binmap[block]
bin = next_bin (bin);
bit <<= 1;
True
victim == bin?
False
将 victim 从对应的 bin 中 unlink
remainder_size = size - nb;
unlink_chunk (av, victim);
True
goto use_top
类似这部分流程
将victim 指向的 chunk 进行分割，并将剩余的部分放入 unsorted bin 中
return chunk2mem (victim);


victim = av->top;
size = chunksize (victim);
将 top chunk 进行分割
size > nb+MINSIZE?
True
表示 top chunk
足够大
remainder_size = size - nb;
remainder = chunk_at_offset (victim, nb);
av->top = remainder;
将切割下的所需内存块返回
check_malloced_chunk (av, victim, nb);
void *p = chunk2mem (victim);
alloc_perturb (p, bytes);
return p;
False
判断是否有 fast chunk
malloc_consolidate (av);
/* restore original bin index */
if (in_smallbin_range (nb))
idx = smallbin_index (nb);
else
idx = largebin_index (nb);
}
True
则尝试合并 fastbin
再进行分配
have_fastchunk?
False
void *p = sysmalloc (nb, av);
if (p != NULL)
alloc_perturb (p, bytes);
return p;

`_int_malloc` 之 `sysmalloc`

- 参考文章--申请内存块

1. sysmalloc 处理需要更多系统内存的情况：

```
/*
sysmalloc handles malloc cases requiring more memory from the system.
On entry, it is assumed that av->top does not have enough
space to service request for nb bytes, thus requiring that av->top
be extended or replaced.

sysmalloc 处理需要更多系统内存的 malloc 情况。
进入时，假设 av->top 没有足够的为 nb 字节服务请求的空间，因此需要 av->top 延长或更换。
*/

static void *
sysmalloc (INTERNAL_SIZE_T nb, mstate av)
{
mchunkptr old_top;               /* incoming value of av->top */
INTERNAL_SIZE_T old_size;        /* its size */
char *old_end;                   /* its end address */

long size;                       /* arg to first MORECORE or mmap call */
char *brk;                       /* return value from MORECORE */

long correction;                 /* arg to 2nd MORECORE call */
char *snd_brk;                   /* 2nd return val */

INTERNAL_SIZE_T front_misalign; /* unusable bytes at front of new space */
INTERNAL_SIZE_T end_misalign;   /* partial page left at end of new space */
char *aligned_brk;               /* aligned offset into brk */

mchunkptr p;                     /* the allocated/returned chunk */
mchunkptr remainder;             /* remainder from allocation */
unsigned long remainder_size;    /* its size */

size_t pagesize = GLRO (dl_pagesize);
bool tried_mmap = false;
```

   - 注意其中的 `pagesize` 定义：

```
#ifndef SHARED
#define GLRO(name) _##name

//将 dl_pagesize 转化为 _dl_pagesize

#ifndef EXEC_PAGESIZE
#define EXEC_PAGESIZE   4096   //0x1000
#endif

/* Cached value of `getpagesize ()'.  */
EXTERN size_t _dl_pagesize;
......
size_t _dl_pagesize = EXEC_PAGESIZE;

/* Return the system page size.  */
int
__getpagesize (void)
{
    #ifdef EXEC_PAGESIZE
    return EXEC_PAGESIZE;
    ......
}
```

```
libc_hidden_def (__getpagesize)
weak_alias (__getpagesize, getpagesize)
```

2. 如果存在且支持 mmap 并且请求大小满足 mmap 阈值则考虑使用 mmap、如果目前还不够分配的 mmapped 区域，尝试直接映射此请求而不是扩大顶部。

```
#ifndef DEFAULT_MMAP_MAX
#define DEFAULT_MMAP_MAX       (65536) // 0x10000
#endif

/* Align a value by rounding down to closest size.
e.g. Using size of 4096, we get this behavior:
通过向下舍入到最接近的大小来对齐值。
例如使用 4096 的大小，我们得到以下行为
{4095, 4096, 4097} = {0, 4096, 4096}.  */
#define ALIGN_DOWN(base, size) ((base) & -((__typeof__ (base)) (size)))

/* Align a value by rounding up to closest size.
e.g. Using size of 4096, we get this behavior:
    {4095, 4096, 4097} = {4096, 4096, 8192}.

Note: The size argument has side effects (expanded multiple times).  */
#define ALIGN_UP(base, size) ALIGN_DOWN ((base) + (size) - 1, (size))

#define MMAP(addr, size, prot, flags) \
__mmap((addr), (size), (prot), (flags)|MAP_ANONYMOUS|MAP_PRIVATE, -1, 0)

.......

/*
    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.
*/
//mp_.mmap_threshold 阈值 (128 * 1024)
if (av == NULL
    || ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_threshold)
    && (mp_.n_mmaps < mp_.n_mmaps_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 have high alignment.
        将大小四舍五入到最近的页面。对于 mmapped 块，开销比普通块大一个 SIZE_SZ 单位，
        因为有没有可以使用其 prev_size 字段的后续块。

        请参阅下面的 front_misalign 处理，对于 glibc 没有除非我们有高度对齐，否则需要进一步对齐。
    */
    if (MALLOC_ALIGNMENT == CHUNK_HDR_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))
        {
            //调用 mmap 映射内存
        mm = (char *) (MMAP (0, size,
                MTAG_MMAP_FLAGS | 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 == CHUNK_HDR_SZ)
                {
                /* For glibc, chunk2rawmem increases the address by
                    CHUNK_HDR_SZ and MALLOC_ALIGN_MASK is
                    CHUNK_HDR_SZ-1.  Each mmap'ed area is page
                    aligned and therefore definitely
                    MALLOC_ALIGN_MASK-aligned.  */
                assert (((INTERNAL_SIZE_T) chunk2rawmem (mm) & MALLOC_ALIGN_MASK) == 0);
                front_misalign = 0;
                }
            else
                front_misalign = (INTERNAL_SIZE_T) chunk2rawmem (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;
```

3. 记录并检查旧堆信息：

```
/* 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.
*/
// 检查 旧堆：1, 如果是刚初始化的 top 其 size 应该为 0; 2, 已经分配过的对结束地址应该是页对齐的
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));
```

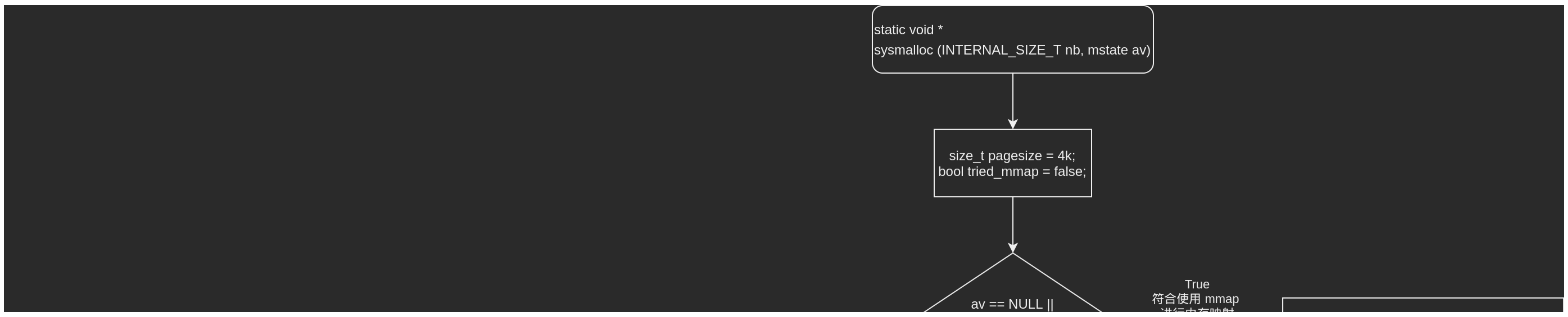
4. 非主分配区情况，通过 mmap 申请新的 heap header，放入线程的 heap 链表中进行内存扩展。首先获取当前非主分配区 heap header info：
    - 首先判断是否而可以进行 heap 扩展 `grow_heap`，成功则进行标志位更新；
    - 否则申请新的 heap 使用，`new_heap` 通过 mmap 申请新的 heap 内存；
    - 参考文章--Glibc 内存管理--ptmalloc2 源代码分析（二十四）
    - **参考文章--glibc+malloc 源码简析（二）**
    - //TODO: 可以详细解析下 sysmalloc 内存对齐相关细节参考上边这篇文章。

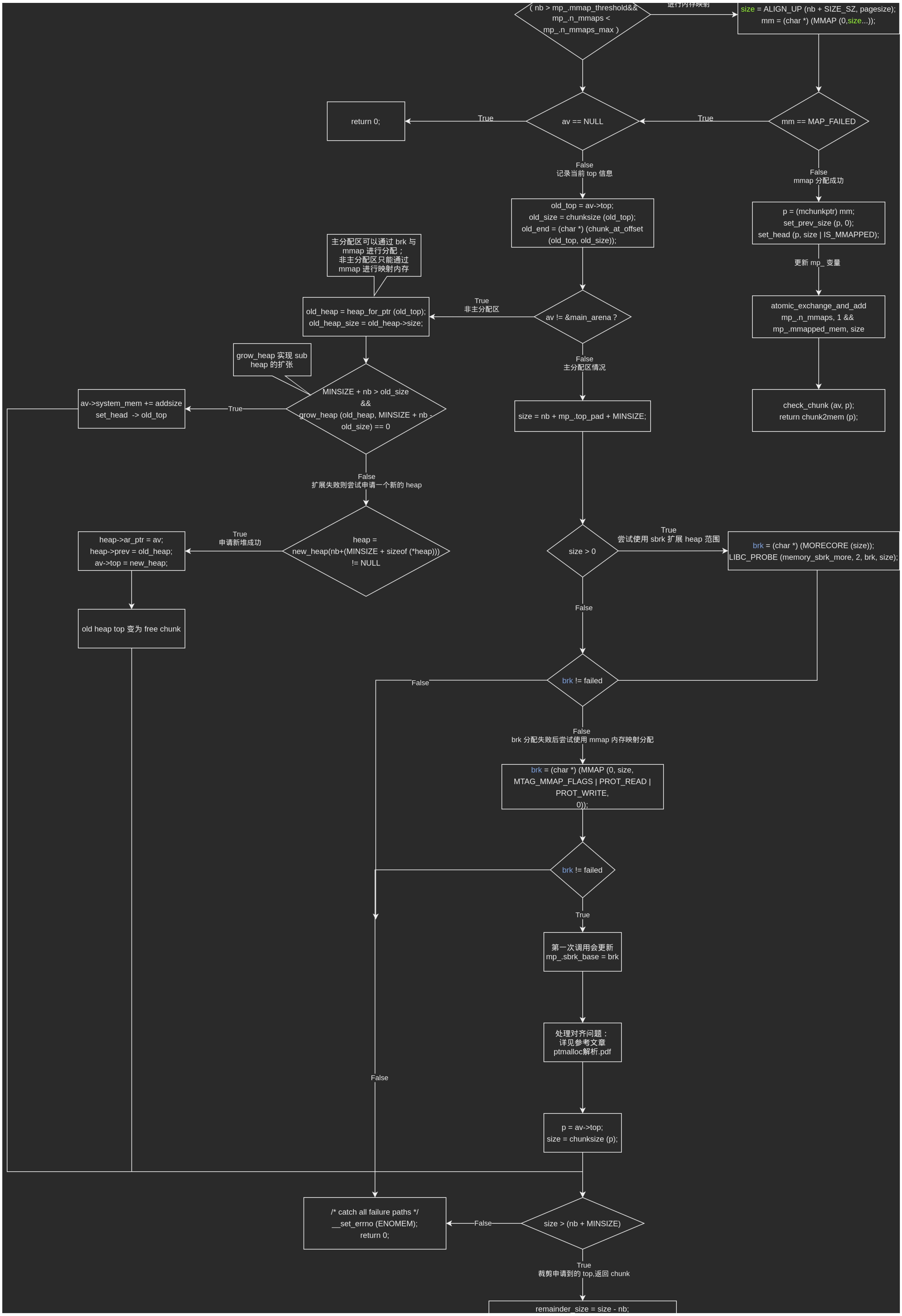
```
#define heap_for_ptr(ptr) \
((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE - 1)))
#define arena_for_chunk(ptr) \
(chunk_main_arena (ptr) ? &main_arena : heap_for_ptr (ptr)->ar_ptr)

.......
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) //grow_heap() 函数主要将 sub_heap 中可读可写区域扩大
        {
        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; // 更新 heap header info
        av->system_mem += heap->size; // 更新分配区信息
        /* Set up the new top.  */
        top (av) = chunk_at_offset (heap, sizeof (*heap)); // 更新 top 地址
        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. */
        /*设置 fencepost 并使用多个的释放旧的顶部块
            MALLOC_ALIGNMENT 大小。  */
        /*栅栏柱至少需要 MINSIZE 字节，因为它可能
            稍后再次成为顶级块。注意设置了页脚
            也一样，虽然块被标记为使用中。  */
        old_size = (old_size - MINSIZE) & ~MALLOC_ALIGN_MASK;
        set_head (chunk_at_offset (old_top, old_size + CHUNK_HDR_SZ),
            0 | PREV_INUSE);
        if (old_size >= MINSIZE)
            {
            set_head (chunk_at_offset (old_top, old_size),
            CHUNK_HDR_SZ | PREV_INUSE);
            set_foot (chunk_at_offset (old_top, old_size), CHUNK_HDR_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 + CHUNK_HDR_SZ) | PREV_INUSE);
            set_foot (old_top, (old_size + CHUNK_HDR_SZ));
            }
        }
    else if (!tried_mmap)
        /* We can at least try to use to mmap memory.  */
        goto try_mmap;
    }
```

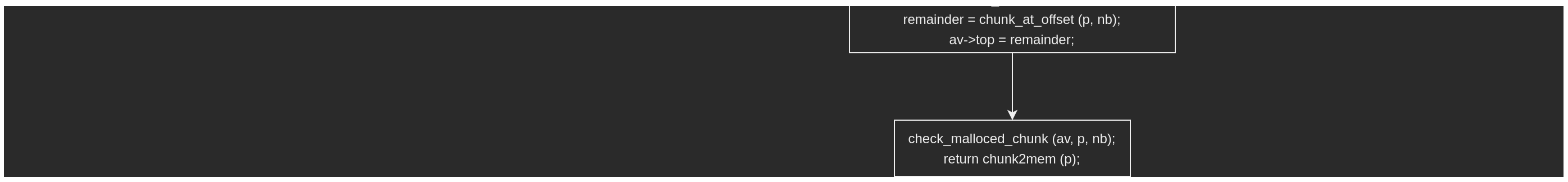
5. 主分配区情况：

```
else     /* av == main_arena */
    { /* Request enough space for nb + pad + overhead */
    // 计算可以满足请求的内存大小
    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.

        如果是连续的可以复用 old top 的内存
    */

    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.

        四舍五入为页面大小的倍数。如果 MORECORE 不连续，这确保我们只调用它使用整页参数。
        如果 MORECORE 是连续的并且这不是第一次通过，这保留了页面对齐以前的电话。
        否则，我们更正下面的页面对齐。
    */

    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.

        尝试申请内存使用 sbrk
    */

    if (size > 0)
        {
        brk = (char *) (MORECORE (size));
        LIBC_PROBE (memory_sbrk_more, 2, brk, size);
        }
    // 申请成功，调用了一个 __after_morecore_hook
    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 // 调用失败后 尝试使用 mmap 进行分配
        {
        /*
            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.
            如果有 mmap，请尝试在 MORECORE 失败时将其用作备份或
            不能使用。这对于有“漏洞”的系统来说是值得的
            地址空间，因此 sbrk 不能扩展以提供连续空间，但是
            其他地方有空位。请注意，我们忽略了 mmap 最大计数
            和阈值限制，因为空间不会被用作
            隔离的 mmap 区域。
        */

        /* 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))
            {
            char *mbrk = (char *) (MMAP (0, size,
                    MTAG_MMAP_FLAGS | 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);
                }
            }
        }

    // 当内存分配成功后
    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.
            空间是连续的，则可以尝试进行扩张 old top
        */

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

        // 意外的已申请内存被耗尽
        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.
            如果第一次连续或者不连续，需要调用 sbrk 只是为了找出内存的尽头在哪里。

        * We need to ensure that all returned chunks from malloc will meet
            MALLOC_ALIGNMENT
            我们需要确保所有从 malloc 返回的块都会满足 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.
            如果有外部 sbrk 介入，我们需要调整 sbrk 请求大小以说明我们将无法将新空间与 old_top 中的现有空间结合起来。

        * 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.
            几乎所有系统都在内部一次分配整个页面，在在这种情况下，我们不妨使用请求的整个最后一页。
            所以我们现在分配足够多的内存来达到页面边界，这反过来又会导致未来对页面对齐的连续调用。
        */

        else  // 新申请的内存并不连续
        /*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。*/
            {
            front_misalign = 0;
            end_misalign = 0;
            correction = 0;
            aligned_brk = brk;

            /* handle contiguous cases 处理连续内存 */
            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) chunk2rawmem (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)();
                    }
                }

            /* handle non-contiguous cases 处理不连续内存 */
            else
                {
                if (MALLOC_ALIGNMENT == CHUNK_HDR_SZ)
                    /* MORECORE/mmap must correctly align */
                    assert (((unsigned long) chunk2rawmem (brk) & MALLOC_ALIGN_MASK) == 0);
                else
                    {
```

```
                    front_misalign = (INTERNAL_SIZE_T) chunk2rawmem (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 根据第二个 sbrk 的结果调整顶部*/
            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 - 2 * CHUNK_HDR_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),
                CHUNK_HDR_SZ | PREV_INUSE);
            set_head (chunk_at_offset (old_top,
                        old_size + CHUNK_HDR_SZ),
                CHUNK_HDR_SZ | PREV_INUSE);

                    /* If possible, release the rest.
                    需要注意的是，在这里程序将旧的 top chunk 进行了释放，那么其会根据大小进入不同的 bin 或 tcache 中。*/
                    if (old_size >= MINSIZE)
                        {
                        _int_free (av, old_top, 1);
                        }
                    }
                }
            }
        }
    } /* if (av !=  &main_arena) */
//更新最大内存
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 分配内存块*/
p = av->top;
size = chunksize (p);

/* check that one of the above allocation paths succeeded 切分 TOP*/
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_malloced_chunk (av, p, nb);
    return chunk2mem (p);
    }

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




(nb > mp_.mmap_threshold&& mp_.n_mmaps < mp_.n_mmaps_max)
进行内存映射
size = ALIGN_UP (nb + SIZE_SZ, pagesize);
mm = (char *) (MMAP (0,size...));
return 0;
True
av == NULL
True
mm == MAP_FAILED
False
记录当前 top 信息
False
mmap 分配成功
old_top = av->top;
old_size = chunksize (old_top);
old_end = (char *) (chunk_at_offset (old_top, old_size));
p = (mchunkptr) mm;
set_prev_size (p, 0);
set_head (p, size | IS_MMAPPED);
主分配区可以通过 brk 与 mmap 进行分配；
非主分配区只能通过 mmap 进行映射内存
更新 mp_ 变量
old_heap = heap_for_ptr (old_top);
old_heap_size = old_heap->size;
True
非主分配区
av != &main_arena ?
atomic_exchange_and_add mp_.n_mmaps, 1 && mp_.mmapped_mem, size
grow_heap 实现 sub heap 的扩张
False
主分配区情况
av->system_mem += addsize
set_head -> old_top
True
MINSIZE + nb > old_size
&&
grow_heap (old_heap, MINSIZE + nb - old_size) == 0
size = nb + mp_.top_pad + MINSIZE;
check_chunk (av, p);
return chunk2mem (p);
False
扩展失败则尝试申请一个新的 heap
True
申请新堆成功
heap->ar_ptr = av;
heap->prev = old_heap;
av->top = new_heap;
heap = new_heap(nb+(MINSIZE + sizeof (*heap))) != NULL
size > 0
True
尝试使用 sbrk 扩展 heap 范围
brk = (char *) (MORECORE (size));
LIBC_PROBE (memory_sbrk_more, 2, brk, size);
old heap top 变为 free chunk
False
False
brk != failed
False
brk 分配失败后尝试使用 mmap 内存映射分配
brk = (char *) (MMAP (0, size, MTAG_MMAP_FLAGS | PROT_READ | PROT_WRITE, 0));
brk != failed
True
第一次调用会更新 mp_.sbrk_base = brk
处理对齐问题：
详见参考文章
ptmalloc解析.pdf
False
p = av->top;
size = chunksize (p);
/* catch all failure paths */
__set_errno (ENOMEM);
return 0;
False
size > (nb + MINSIZE)
True
裁剪申请到的 top,返回 chunk
remainder_size = size - nb;


remainder = chunk_at_offset (p, nb);
av->top = remainder;
check_malloced_chunk (av, p, nb);
return chunk2mem (p);