福建堆培训-Glibc源码分析

> 福建堆培训-Glibc源码分析, 源码版本为 Glibc2.27 (未修复tcache double free的版本)

分配器和areana

ptmalloc2 来自于 dlmalloc 的分支

在早期的 Linux 里，dlmalloc 被用做默认的内存分配器，但它在处理多线程并行的程序时，只会维护一个共享的堆内存空间，因此当多个线程同时申请分配堆块时会进行加锁，影响程序性能

之后在ptmalloc2中增加了对多线程的支持，每个线程都会维护一个独享的堆内存空间，成为 `arena`，其中主线程的arnea称为“main_arena”。子线程的arnea称为“thread_arena”。

因此目前Glibc使用 ptmalloc2作为堆分配器

重要结构体

malloc_chunk

`chunk` 的基本结构

```
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};

// 典型的chunk结构
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of previous chunk, if allocated            | |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
        |             Size of chunk, in bytes                       |M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             User data starts here...                          .
        .                                                               .
        .             (malloc_usable_size() bytes)                      .
        .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of chunk                                     |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

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

- `prev_size`，当前的`chunk`为`free`的时候，表示物理相邻的前一个`chunk`的大小，并可以通过这个大小找到前一个`chunk`的起始位置。当该块为`use`的时候，该位可以被前一块的数据占用。
- `size`表示当前`chunk`的大小，由于`size`最小需要`8`字节对齐，因此其低`3`空闲，分别用作`falg`即`NMP`，其中`N`表示的是当前的`chunk`是否属于主进程，`M`表示的是当前的`chunk`是否是由`mmap`分配的`P`表示的是前一个`chunk`是否是空闲的。最开始的一个`chunk`该位总是置`1`，防止引用不存在的内存地址。
- `fd`当`chunk`空闲时，指向下一个空闲的`chunk`，`bk`指向上一个空闲的`chunk`
- `fd_nextsize`当为`largebin`的时候才是用，指向下一个与当前的`chunk`大小不同的`chunk`，`largebin`链表中相同大小的一组`chunk`仅第一块`chunk`的`fd_nextsize,bk_nextsize`指向下一个和上一个与当前大小不同的堆块，其余的`chunk`该位为空。

malloc_state

`malloc_state`是非常常用的一个结构体了，通常用`mstate`来表示。

```
struct malloc_state
{
```

```
  /* Serialize access.  */
  mutex_t mutex;//互斥锁

  /* Flags (formerly in max_fast).  */
  int flags;//标志位，用来表示当前的arena中是否存在fastbin或者内存是否连续等

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];//存放每一个fastbin链表的头指针，最多支持的bin的个数为10个

  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;//top chunk堆顶

  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;//指向上一个chunk分配出一个small chunk之后剩余的部分

  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];//存储unsorted_bin,small_bin.large_bin的链表头

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];//每一个bit表示对应的bin中是否存在空闲chunk

  /* Linked list */
  struct malloc_state *next;

  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;

  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */
  INTERNAL_SIZE_T attached_threads;

  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;//当前内存的分配量
  INTERNAL_SIZE_T max_system_mem;
};
```

每个分配区是一个 `malloc_state` 结构体的实例， `ptmalloc` 使用这个结构体来管理每一个分配区，而参数的管理使用的是 `malloc_par` 结构体，全局拥有一个该结构体的实例。

```
#define fastbin_index(sz) \
  ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)


/* The maximum fastbin request size we support */
#define MAX_FAST_SIZE     (80 * SIZE_SZ / 4)

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

针对 `fastbin` 中最大的 `bin` ，对 `32` 为系统来说，其数据空间最大为 `80` 字节，对 `64` 位系统来说其最大的数据空间为 `160` 字节。注意这里的数据空间都不不包含 `chunk` 头部的。

`bins` 变量中存储了 `unsorted_bin, small_bin (62) , large_bin (63)` 的链表头指针，其中 `bins[0],bins[127]` 都不存在， `bins[1]` 中存储了 `unsorted_bin` 的 `chunk` 的链表头部，`mchunkptr` 是 `malloc_chunk` 的结构体指针。

- `small bin` ，从 `small bin` 定义的宏来看，其数组下标 `index` 与存储的 `chunk` 的大小的关系是： `chunk_size = MALLOC_ALIGNMENT *index` ，其中 `MALLOC_ALIGNMENT=2 * SIZE_SZ` 。对 `32` 位系统来说，其最大的 `chunk` 的大小为 `504` 字节， `64` 位为 `1008` 字节

  ```
  #define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
  #define SMALLBIN_CORRECTION (MALLOC_ALIGNMENT > 2 * SIZE_SZ)
  #define MIN_LARGE_SIZE    ((NSMALLBINS - SMALLBIN_CORRECTION) * SMALLBIN_WIDTH)

  #define in_smallbin_range(sz)  \
    ((unsigned long) (sz) < (unsigned long) MIN_LARGE_SIZE)

  #define smallbin_index(sz) \
    ((SMALLBIN_WIDTH == 16 ? (((unsigned) (sz)) >> 4) : (((unsigned) (sz)) >> 3))\
     + SMALLBIN_CORRECTION)
  ```

- `large bin` ，共包含 `63` 个 `bin` ， 每个 `bin` 中的 `chunk` 的大小不一致，而是出于一定的范围之内，此外这 `63` 个 `bin` 被分成了 `6` 组，每组 `bin` 的 `chunk` 的大小之间的公差一致。

  `largebin` 排列方式为从大到小依次排列，链表头的 `bk` 指针指向最小的堆块。

  ```
  #define largebin_index_32(sz)
     \
    (((((unsigned long) (sz)) >> 6) <= 38) ?  56 + (((unsigned long) (sz)) >> 6) :\
     ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9) :\
     ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
  ```

```
    ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
    ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
    126)

#define largebin_index_32_big(sz)                                            \
  (((((unsigned long) (sz)) >> 6) <= 45) ?  49 + (((unsigned long) (sz)) >> 6) :\
    ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9) :\
    ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
    ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
    ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
    126)

// XXX It remains to be seen whether it is good to keep the widths of
// XXX the buckets the same or whether it should be scaled by a factor
// XXX of two as well.
#define largebin_index_64(sz)                                                \
  (((((unsigned long) (sz)) >> 6) <= 48) ?  48 + (((unsigned long) (sz)) >> 6) :\
    ((((unsigned long) (sz)) >> 9) <= 20) ?  91 + (((unsigned long) (sz)) >> 9) :\
    ((((unsigned long) (sz)) >> 12) <= 10) ? 110 + (((unsigned long) (sz)) >> 12) :\
    ((((unsigned long) (sz)) >> 15) <= 4) ? 119 + (((unsigned long) (sz)) >> 15) :\
    ((((unsigned long) (sz)) >> 18) <= 2) ? 124 + (((unsigned long) (sz)) >> 18) :\
    126)

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

`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` 个字节，也就是 `128` 个 `bit`。

- malloc_par

```
struct malloc_par
{
  /* Tunable parameters */
  unsigned long trim_threshold;//收缩阈值
  INTERNAL_SIZE_T top_pad;//分配内存时是否添加额外的pad，默认为0
  INTERNAL_SIZE_T mmap_threshold;//mmap的分配阈值
  INTERNAL_SIZE_T arena_test;//最小分配区
  INTERNAL_SIZE_T arena_max;//最大分配区

  /* Memory map support */
  int n_mmaps;//当前进程使用mmap()分配的内存块的个数
  int n_mmaps_max;//可以使用mmap()分配的内存块的最大数量
  int max_n_mmaps;
  /* the mmap_threshold is dynamic, until the user sets
     it manually, at which point we need to disable any
     dynamic behavior. */
  int no_dyn_threshold;//是否开启mmap分配阈值动态调整，0表示开启，默认为0

  /* Statistics */
  INTERNAL_SIZE_T mmapped_mem;//mmap分配的内存大小
  /*INTERNAL_SIZE_T  sbrked_mem;*/
  /*INTERNAL_SIZE_T  max_sbrked_mem;*/
  INTERNAL_SIZE_T max_mmapped_mem;//mmap分配的内存大小，一般与mmaped_mem相等
  INTERNAL_SIZE_T max_total_mem;  /* only kept for NO_THREADS *///单线程情况下统计进程分配的内存总数

  /* First address handed out by MORECORE/sbrk.  */
  char *sbrk_base;
};
```

- `thrim_threshold` 字段表示收缩阈值，默认为 `128KB`，当 `top chunk` 的大小大于这个阈值的时候，在调用 `free` 函数的时候可能会缩小 `top chunk`，收缩内存。收缩阈值可以通过 `mallocopt` 函数设置。由于 `mmap` 分配阈值动态调整，`free` 函数最大可以将收缩阈值设置为分配阈值的 `2` 倍。
- `mmap_threshold` 表示分配阈值，默认值为 `128KB`，`32` 位系统中最大值为 `512KB`，`64` 位系统中的最大值为 `32MB`。默认开启了 `mmap` 分配阈值的动态调整，但是不会超过最大值。
- `arena_test` 和 `arena_max`，当每个进程中的分配区的数量大于 `arena_max` 的时候不会创建新的分配区，当分配区数量小于 `arena_test` 的时候不会重用现有的分配区。

__libc_malloc

malloc执行的开始

位于 `malloc/malloc.c`

```
void *
__libc_malloc (size_t bytes)
{
  mstate ar_ptr;  // 指向一个 arena
  void *victim;
  // 给名为hook的函数指针尝试读入malloc_hook,malloc_hook是一个调试变量,可以用来打印一些相关参数的值
  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);  // 读的原子操作
  // 查看malloc_hook中是否为空,若不为空则执行malloc_hook中的内容
  // 参数为调用malloc时指定的空间大小
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));
```

Tcache

FILO

Tcache 只用 fd 索引，bk始终指向 tcache struct

流程总结

- 第一次 malloc 时，会先 malloc 一块内存用来存放 `tcache_perthread_struct` 。
- free 内存，且 size 小于 small bin size 时
- tcache 之前会放到 fastbin 或者 unsorted bin 中
- tcache 后：
  - 先放到对应的 tcache 中，直到 tcache 被填满（默认是 7 个）
  - tcache 被填满之后，再次 free 的内存和之前一样被放到 fastbin 或者 unsorted bin 中
  - tcache 中的 chunk 不会合并（不取消 inuse bit）
- malloc 内存，且 size 在 tcache 范围内
- 先从 tcache 取 chunk，直到 tcache 为空
- tcache 为空后，从 bin 中找
- tcache 为空时，如果 `fastbin/smallbin/unsorted bin` 中有 size 符合的 chunk，会先把 `fastbin/smallbin/unsorted bin` 中的 chunk 放到 tcache 中，直到填满。之后再从 tcache 中取；因此 chunk 在 bin 中和 tcache 中的顺序会反过来

```
#if USE_TCACHE
  /* int_free also calls request2size, be careful to not pad twice.  */
  size_t tbytes;
```

```
    // bytes : 申请大小
    // tbytes:  转换后的大小
    checked_request2size (bytes, tbytes);
    // 根据申请的大小计算其在tcache中的位置
    size_t tc_idx = csize2tidx (tbytes);
    // 在 tcache 为空 (即第一次 malloc) 时调用tcache_init()
    MAYBE_INIT_TCACHE ();

    DIAG_PUSH_NEEDS_COMMENT;
  // 判断根据 size 得到的 idx 是否在合法的范围内
    if (tc_idx < mp_.tcache_bins
        /*&& tc_idx < TCACHE_MAX_BINS*/ /* to appease gcc */
        && tcache
        && tcache->entries[tc_idx] != NULL) // tcache->entries[tc_idx] 有 chunk
      {
        // 将tcache对应chunk指针返回给user
        return tcache_get (tc_idx);
      }
    DIAG_POP_NEEDS_COMMENT;
  #endif
```

checked_request2size

checked_request2size 宏将传入的n转换成符合内存对齐的值,然后计算idx

这其中涉及两个检查

```
#define checked_request2size(req, sz) \
({                  \
  (sz) = request2size (req);        \
  if (((sz) < (req))            \  //如果此时大小比req还小，说明肯定溢出了
      || REQUEST_OUT_OF_RANGE (sz)) \
    {                   \
      __set_errno (ENOMEM);     \
      return 0;             \
    }                   \
})
```

检查由 `request2size` 完成的填充是否导致溢出

```
// 将申请的size填充到可用的大小
#define request2size(req)                                         \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)
```

第一次读源码, 这里来详细的解释一下 `request2size` 和地址对齐

- 当一个chunk为空闲时，至少要有 `prev_size` 、 `size` 、 `fd` 和 `bk` 四个参数，因此 `MINSIZE` 就代表了这四个参数需要占用的内存大小；而当一个chunk被使用时， `prev_size` 可能会被前一个chunk用来存储数据
- 因此实际需要写入数据的内存大小就是 `req` (用户申请大小)+ `size` (大小同 size_sz),
- `MALLOC_ALIGN_MASK` 用来对齐，因此request2size就计算出了所需的chunk的大小。
- `MALLOC_ALIGN_MASK` 大小为 `01111` （15字节），主要是为了使低四位的 `size` 进位，之后再与上 `~MALLOC_ALIGN_MASK` 即 `10000` ，将第四位清空，完成对其

通过 `REQUEST_OUT_OF_RANGE` 检查申请的大小是否过大, 以至于对齐时会导致溢出

```
// 检查请求是否太大，以使其在填充和对齐时会回零
#define REQUEST_OUT_OF_RANGE(req)                                 \
  ((unsigned long) (req) >=                                \
   (unsigned long) (INTERNAL_SIZE_T) (-2 * MINSIZE))
```

tcache_init

```
static void
tcache_init(void)
{
  mstate ar_ptr;
  void *victim = 0;
    // 获得tcache_perthread_struct结构大小
  const size_t bytes = sizeof (tcache_perthread_struct);

  if (tcache_shutting_down)
    return;
    // 找到可用的 arena 存入ar_ptr中
  arena_get (ar_ptr, bytes);
    // 申请一个 sizeof(tcache_perthread_struct) 大小的 chunk
  victim = _int_malloc (ar_ptr, bytes);
    // 申请chunk或get_arena不成功的话重试
  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.  */
    // 初始化tcache
```

```
      // 分配成功则将块赋值给tcache，并清空其中的内容
      if (victim)
      {
        tcache = (tcache_perthread_struct *) victim;
        memset (tcache, 0, sizeof (tcache_perthread_struct));
      }

  }
```

tcache_entry

tcache的基本结构，是一个单项链表, 用于链接空闲的 chunk 结构体

`next` 指向下一个tcache的data段(非header)；每个tcache相当于一个缓存chunk

`一个固定大小的tcache链条对应一个 tcache_entry`

```
typedef struct tcache_entry
{
  struct tcache_entry *next;
} tcache_entry;
```

tcache_perthread_struct

这个数据结构为全局的 `tcache` 数据结构，存在于每个线程中，其中 `TCACHE_MAX_BINS` 默认值为64，即从24~1024每16字节一个 `tcache` 结构

```
typedef struct tcache_perthread_struct
{
  // 统计tc_idx中存在多少个缓存块
  char counts[TCACHE_MAX_BINS];
    // 对应有多少个 entry
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct;
```

它是整个 tcache 的管理结构，一共有 `TCACHE_MAX_BINS` 个计数器和 `TCACHE_MAX_BINS` 项 tcache_entry，其中

- `tcache_entry` 用单向链表的方式链接了相同大小的处于空闲状态（free 后）的 chunk，这一点上和 fastbin 很像。
- `counts` 记录了 `tcache_entry` 链上空闲 chunk 的数目，每条链上最多可以有 7 个 chunk。

tcache_put

向 tcache对应bin中存chunk

```
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);
  assert (tc_idx < TCACHE_MAX_BINS);
  e->next = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e;
  ++(tcache->counts[tc_idx]);
}
```

tcache_get

从tcache对应bin中取chunk

```
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  assert (tc_idx < TCACHE_MAX_BINS);
  assert (tcache->entries[tc_idx] > 0);
  tcache->entries[tc_idx] = e->next;
  --(tcache->counts[tc_idx]);
  return (void *) e;
}
```

正常堆分配

承接 `__libc_malloc` 主函数

```
   // 单线程,进入与无 tcache 时类似的流程
    if (SINGLE_THREAD_P)
      {
        // 申请对应大小的chunk
        victim = _int_malloc (&main_arena, bytes);
                //assert判断返回值,返回chunk是否属于mainarena,以及chunk是否是通过mmap
    分配的
        assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
                &main_arena == arena_for_chunk (mem2chunk (victim)));
        // 返回chunk地址
        return victim;
      }
```

多线程堆分配

多线程相关堆分配部分,除了一些互斥锁的加入,大体上还是一样的

```
//下面是线程相关 arena.c定义了arena_get宏,执行会使at_ptr = thread_ptr,然后一些互斥锁balabala
  arena_get (ar_ptr, bytes);
//同样在ar_ptr返回bytes,判断几个条件
  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;
}
```

流程总结

`__libc_malloc`

- 检查 `malloc_hook` ,如果存在值就执行
- 初始化 `tcache` (仅在程序运行时执行一次)
  - 如果 `tcache` 中有符合申请大小的chunk,直接分配
- 如果 `tcache` 中没有, 则调用 `_int_malloc` 申请堆块

_int_malloc

malloc关键的内存管理机制, 从 `__libc_malloc` 也能看出, 堆分配实际都是通过调用 `_int_malloc` 完成的

```
static void *
_int_malloc (mstate av, size_t bytes) // 两个参数分别是arena地址和用户申请的堆大小
{
    // 规范转换后的chunk申请大小
  INTERNAL_SIZE_T nb;               /* normalized request size */
```

```
    // 关联bin索引
    unsigned int idx;                /* associated bin index */
    // 关联bin
    mbinptr bin;                     /* associated bin */
    // 选定的chunk
    mchunkptr victim;                /* inspected/selected chunk */
    // 被选中chunk的大小?
    INTERNAL_SIZE_T size;            /* its size */
      // 被选中chunk的索引?
    int victim_index;                /* its bin index */
      // 切割后剩余的chunk
    mchunkptr remainder;             /* remainder from a split */
      // last reminder的大小
    unsigned long remainder_size;    /* its size */

    unsigned int block;              /* bit map traverser */
    unsigned int bit;                /* bit map traverser */
    unsigned int map;                /* current word of binmap */

      // 临时链接?
    mchunkptr fwd;                   /* misc temp for linking */
    mchunkptr bck;                   /* misc temp for linking */

#if USE_TCACHE
      // 未分类的tcache?
    size_t tcache_unsorted_count;    /* count of unsorted chunks processed */
#endif
```

将传入值变成对齐内存合法的size值,然后判断main_arena是否存在,因为后面的分配是基于此的

```
// 将传入的值转换成符合条件的nb
checked_request2size (bytes, nb);/
/* There are no usable arenas.  Fall back to sysmalloc to get a chunk from
    mmap.  */
    // 当传入的arena指针为空的时候执行sysmalloc
 if (__glibc_unlikely (av == NULL))
   {
     void *p = sysmalloc (nb, av);
     if (p != NULL)
alloc_perturb (p, bytes);
     return p;
   }
```

- **fastbin**

fastbin只用fd索引，bk为空

```
#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);

    // 如果对齐后的大小在fastbin范围内
  if ((unsigned long) (nb) <= (unsigned long) (get_max_fast ()))
    {
      // 按照fastbin的size-index关系计算index
      idx = fastbin_index (nb);
      // 取对应fastbin位置的地址,即mainarena中一个固定偏移的地址,实际上就是对应位的链表
头指针
      mfastbinptr *fb = &fastbin (av, idx);
      mchunkptr pp;
      // victim尝试获取链表头指向的地址
      victim = *fb;
        // 如果存在对应的值,则进入从fastbin中获取堆块的流程,否则进入smallbin的判断
      if (victim != NULL)
    {
      // 如果是单线程,那么直接进行单链表删除节点的操作,链表头写入该节点指向的下一个地址
      if (SINGLE_THREAD_P)
        *fb = victim->fd;
      else
        REMOVE_FB (fb, pp, victim);
          // 判断victim是否为空.
      if (__glibc_likely (victim != NULL))
        {
          // 判断该处指向的size是否符合之前的idx
          size_t victim_idx = fastbin_index (chunksize (victim));
          if (__builtin_expect (victim_idx != idx, 0))
        malloc_printerr ("malloc(): memory corruption (fast)");
          check_remalloced_chunk (av, victim, nb);
#if USE_TCACHE
          // 如果存在tcache,且发现了其他同样大小的bin,将他们放入tcache中
          /* While we're here, if we see other chunks of the same size,
         stash them in the tcache.  */
          // 按照tcache的size-index关系计算index
          size_t tc_idx = csize2tidx (nb);
          // 判读合法性
          if (tcache && tc_idx < mp_.tcache_bins)
        {
          mchunkptr tc_victim;

          /* While bin not empty and tcache not full, copy chunks.  */
              // 如果目标块存在且tcache未满,将目标快放入tcache中
          while (tcache->counts[tc_idx] < mp_.tcache_count
             && (tc_victim = *fb) != NULL)
```

```
52                {
53                  // 在单线程中,执行链表删除,将后面的chunk赋值给当前fb
54                  if (SINGLE_THREAD_P)
55                *fb = tc_victim->fd;
56                  else
57                {
58                  REMOVE_FB (fb, pp, tc_victim);
59                  if (__glibc_unlikely (tc_victim == NULL))
60                    break;
61                }
62                  tcache_put (tc_victim, tc_idx);
63                }
64            }
65    #endif
66              // 将victim的chunk_ptr转换成user_ptr,然后return
67              void *p = chunk2mem (victim);
68              // 如果设置了perturb_type, 则将获取到的chunk初始化为 perturb_type ^ 0xff
69              alloc_perturb (p, bytes);
70              return p;
71            }
72        }
73        }
```

fastbin_index

根据size计算其在fastbin中的索引

```
1   // 这里如果不 '-2' 的话会索引不到前两个bin （因为fastbin是从0x20开始的）
2   #define fastbin_index(sz) \
3     ((((unsigned int) (sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)
```

流程总结

- 对其申请的size, 如果其大小在fastbin范围, 则进如fastbin分配流程
- 计算size在fastbin中对应的索引值
- 如果fastbin中该索引值内有chunk, 则进入chunk获取流程, 反之进如smallbin判断
- 在获取流程中, 直接进行单链表删除操作, 链表头写入选中chunk的下一个地址

  （控制这个fd,可日）.(两种不同size的fastbin 进行的覆写topchunk指针的操作,关键在与劫持fd,然后在mainarena中写入对应的值.）
- 判断分配出的chunk的索引是否符合当初计算的索引（防止分配出其他大小的chunk）
- 在有tcache时,如果满足以下条件便会将fastbin中该索引下的chunk转移到tcache中:
  - fastbin该size的索引下还有chunk
  - tcache该索引的bin未满
- 将取出chunk data段的指针返回给用户

smallbin

双链表 FIFO

fd指向下一个，bk指向前一个

最先分配的是bk链的第一个（即fd链的最后一个）

small bins 中每个 chunk 的大小与其所在的 bin 的 index 的关系为：chunk_size = 2 * SIZE_SZ *index，具体如下

下标	SIZE_SZ=4（32 位）	SIZE_SZ=8（64 位）
2	16	32
x	2*4*x	2*8*x
63	504	1008

```
/*
     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.)
   */
  // 判断是否在smallbin范围
   if (in_smallbin_range (nb))
     {
       // 获取smallbin索引
       idx = smallbin_index (nb);
       // 获取对应small bin中chunk的指针
       bin = bin_at (av, idx);
       // 先执行 victim = last(bin)，获取 small bin 的最后一个 chunk
         // 如果 victim = bin ，那说明该 bin 为空。
         // 如果不相等，则说明bin中存在chunk
       if ((victim = last (bin)) != bin)
         {
           // 取倒数第二块chunk
           bck = victim->bk;
           // 检查bck的fd是否为victim，防止伪造
       if (__glibc_unlikely (bck->fd != victim))
         malloc_printerr ("malloc(): smallbin double linked list corrupted");
           // 设置 victim 对应的 inuse 位（smallbin中的inuse位是被清零过的）
           set_inuse_bit_at_offset (victim, nb);
           // 修改 small bin 链表，将 small bin 的最后一个 chunk 取出来
           bin->bk = bck;
           bck->fd = bin;
             // 删除节点  设置inuse位
           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.  */
            //tcache部分将此处的所有smallbin归入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 over.  */
            while (tcache->counts[tc_idx] < mp_.tcache_count
               && (tc_victim = last (bin)) != bin)
          {
            if (tc_victim != 0)
              {
                bck = tc_victim->bk;
                set_inuse_bit_at_offset (tc_victim, nb);
                if (av != &main_arena)
              set_non_main_arena (tc_victim);
                bin->bk = bck;
                bck->fd = bin;

                tcache_put (tc_victim, tc_idx);
                  }
          }
          }
  #endif
            // 返回heap_ptr
            void *p = chunk2mem (victim);
            alloc_perturb (p, bytes);
            return p;
          }
      }
```

smallbin_index

计算smallbin_index对应的索引 这里的 `SMALLBIN_WIDTH` 在64位下为16

这里不用 `减2` 的原因是 smallbin 是从2开始索引的

```
1   #define smallbin_index(sz) \
2     ((SMALLBIN_WIDTH == 16 ? (((unsigned) (sz)) >> 4) : (((unsigned) (sz)) >>
    3))\
3      + SMALLBIN_CORRECTION)
```

bin_at

这个似乎和整个ptmalloc的三大结构体相关，是直接从结构体的bin中取chunk，稍后再补充

```
/* addressing -- note that bin_at(0) does not exist */
#define bin_at(m, i) \
  (mbinptr) (((char *) &((m)->bins[((i) - 1) * 2]))             \
             - offsetof (struct malloc_chunk, fd))
```

first & last

取 bin中 `fd` 和 `bk` 的宏, 和chunk中的fd和bk并不一样

```
#define first(b)     ((b)->fd)
#define last(b)      ((b)->bk)
```

流程总结

- 接下来是smallbins,同样判断是否在smallbin的范围,如果在的话,根据下标寻找双向链表头指针,如果存在则返回
- 将其余同样大小的smallbin归入tcache,然后返回chunk指针
- **这里有一个双链表的检查，之后在smallbin部分细说**

Large bin

双链表 fifo

fd指向下一个，bk指向前一个

最先分配的是bk链的第一个（即fd链的最后一个）

- 前面的都不符合的时候,到了这里,如果存在fastbin的话,合并所有的相邻fastbin,归入unsorted bin中
- 注意malloc_consolidate触发是在第一次初始化fastbin的时候和申请largebin的时候

```
/*
   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);
    }
```

malloc_cinsolidate

堆块合并

```
static void malloc_consolidate(mstate av)
{
  mfastbinptr*    fb;                 /* current fastbin being consolidated */
  mfastbinptr*    maxfb;              /* last fastbin (for loop control) */
  mchunkptr       p;                  /* current chunk being consolidated */
  mchunkptr       nextp;              /* next chunk to consolidate */
  mchunkptr       unsorted_bin;       /* bin header */
  mchunkptr       first_unsorted;     /* chunk to link to */

  /* These have same use as in free() */
  mchunkptr       nextchunk;
  INTERNAL_SIZE_T size;
  INTERNAL_SIZE_T nextsize;
  INTERNAL_SIZE_T prevsize;
  int             nextinuse;
  mchunkptr       bck;
  mchunkptr       fwd;

    // 清除malloc_state中fastbin相关的标志位,表示该分配区中不包含fastbin
  atomic_store_relaxed (&av->have_fastchunks, false);
    // 获取usbin头指针
  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.
  */

  // 获取fastbin中表示最大和最小chunk链表的头指针
  maxfb = &fastbin (av, NFASTBINS - 1);
  fb = &fastbin (av, 0);
  do {
  // 首先是将当前的fastbin链表的头指针赋值给p，并将指针置为0表示该fastbin链表不包含任何的chunk。
    p = atomic_exchange_acq (fb, NULL);
  // 若fastbin当前链表不为空，则对每一个chunk执行下面的操作
    if (p != 0) {
```

```
40          do {
41        {
42          unsigned int idx = fastbin_index (chunksize (p));
43          if ((&fastbin (av, idx)) != fb)
44            malloc_printerr ("malloc_consolidate(): invalid chunk size");
45        }
46        // 检查chunk是否在min_address和max_address之间,并检查物理相邻的下一个chunk的
    prev_inuse位是否为1。
47        check_inuse_chunk(av, p);
48        nextp = p->fd;
49
50        /* Slightly streamlined version of consolidation code in free() */
51        // 接着获取了该chunk的size，和物理相邻的下一个chunk的起始地址以及chunk_size。
52        size = chunksize (p);
53        nextchunk = chunk_at_offset(p, size);
54        nextsize = chunksize(nextchunk);
55        // 判断物理相邻的上一个chunk是否是空闲的，若是空闲的则将这两个chunk合并，并调用
    unlink函数将上一个chunk从链表中删除。
56        if (!prev_inuse(p)) {
57          prevsize = prev_size (p);
58          size += prevsize;
59          p = chunk_at_offset(p, -((long) prevsize));
60          unlink(av, p, bck, fwd);
61        }
62        // 接着就是判断当前的chunk（或者是合并完之后的chunk）是否和top chunk相邻。
63        // 若与top chunk相邻，则将当前的chunk与top chunk进行合并。并设置top chunk的
    prev_inuse位为1。
64        // 如果与top chunk不相邻则执行下面的操作
65        if (nextchunk != av->top) {
66
67          nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
68            // 判断当前chunk物理相邻的下一个chunk是否是空闲的，若是空闲的则继续合并
69          if (!nextinuse) {
70            size += nextsize;
71            unlink(av, nextchunk, bck, fwd);
72          } else
73            // 若不是空闲的则清除下一个chunk的prev_inuse位，表示当前的chunk已经是空闲的
    了。
74            clear_inuse_bit_at_offset(nextchunk, 0);
75
76          first_unsorted = unsorted_bin->fd;
77          // 并将当前的chunk插入到unsorted bin的链表头部，
78          unsorted_bin->fd = p;
79          first_unsorted->bk = p;
80
81          if (!in_smallbin_range (size)) {
82            p->fd_nextsize = NULL;
83            p->bk_nextsize = NULL;
84          }
85        // 并设置当前chunk的prev_inuse为1,下一个chunk的prev_size为当前的size。
86          set_head(p, size | PREV_INUSE);
87          p->bk = unsorted_bin;
```

```
88          p->fd = first_unsorted;
89          set_foot(p, size);
90        }
91
92        else {
93          size += nextsize;
94          set_head(p, size | PREV_INUSE);
95          av->top = p;
96        }
97
98          } while ( (p = nextp) != 0);
99
100       }
101     } while (fb++ != maxfb);
102   }
```

- 当请求大小大于smallbin时，会先根据size获得largebin索引，接着如果程序存在fastbin的话会合并fastbin中的chunk并放入usbin中，具体流程为
  - 循环遍历每个bin中的每个chunk
  - 向上合并，查看物理相邻上一个chunk是否空闲，空闲则合并并用ulink删除该chunk
  - 判断当前chunk与topchunk是否相邻，相邻则合并
  - 向下合并，查看物理相邻下一个chunk是否空闲，若空闲则继续合并
    - 若下一个chunk不为空闲，则将其pre_in_use置0，表示当前chunk已空闲，并将当前chunk放入usbin中
    - 并设置当前chunk的prev_inuse为1,下一个chunk的prev_size为当前的size。

unlink

```
1   #define unlink(AV, P, BK, FD) {                                            \
2       // 由于 P 已经在双向链表中，所以有两个地方记录其大小，所以检查一下其大小是否一致。
3       if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))
    \
4         malloc_printerr ("corrupted size vs. prev_size");                    \
5       FD = P->fd;
          \
6       BK = P->bk;
          \
7       // 防止攻击者简单篡改空闲的 chunk 的 fd 与 bk 来实现任意写的效果。
8       if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
    \
9         malloc_printerr (check_action, "corrupted double-linked list", P, AV);
    \
10      else {
        \
11          FD->bk = BK;
          \
```

```
12            BK->fd = FD;                                                      \
13            // 下面主要考虑 P 对应的 nextsize 双向链表的修改
14            if (!in_smallbin_range (chunksize_nomask (P))                     \
15                // 如果P->fd_nextsize为 NULL，表明 P 未插入到 nextsize 链表中。
16                // 那么其实也就没有必要对 nextsize 字段进行修改了。
17                // 这里没有去判断 bk_nextsize 字段，可能会出问题。
18                && __builtin_expect (P->fd_nextsize != NULL, 0)) {            \
19                // 类似于小的 chunk 的检查思路
20                if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)    \
21                    || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
22                  malloc_printerr (check_action,                              \
23                                   "corrupted double-linked list (not small)",    \
24                                   P, AV);                                    \
25                // 这里说明 P 已经在 nextsize 链表中了。
26                // 如果 FD 没有在 nextsize 链表中
27                if (FD->fd_nextsize == NULL) {                                \
28                    // 如果 nextsize 串起来的双链表只有 P 本身，那就直接拿走 P
29                    // 令 FD 为 nextsize 串起来的
30                    if (P->fd_nextsize == P)                                  \
31                      FD->fd_nextsize = FD->bk_nextsize = FD;                 \
32                    else {                                                    \
33                        // 否则我们需要将 FD 插入到 nextsize 形成的双链表中
34                        FD->fd_nextsize = P->fd_nextsize;                     \
35                        FD->bk_nextsize = P->bk_nextsize;                     \
36                        P->fd_nextsize->bk_nextsize = FD;                     \
37                        P->bk_nextsize->fd_nextsize = FD;                     \
38                      }                                                       \
39                  } else {                                                    \
40                    // 如果在的话，直接拿走即可
41                    P->fd_nextsize->bk_nextsize = P->bk_nextsize;            \
42                    P->bk_nextsize->fd_nextsize = P->fd_nextsize;            \
```

```
                }
        \
            }
    \
        }
        \
}

```

unsorted bin

usbin也是双链表索引

fd的末尾即 bk的开头为第一个要分配的

```
#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

  for (;; )
    {
      int iters = 0;
      // 按照 FIFO 的方式逐个将 unsorted bin 中的 chunk 取出来
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
        {
          // victim 为 unsorted bin 的最后一个 chunk
            // bck 为 unsorted bin 的倒数第二个 chunk
          bck = victim->bk;
          // 如果对齐后的chunk size小于最小size，大于最大size，则报错
          if (__builtin_expect (chunksize_nomask (victim) <= 2 * SIZE_SZ, 0)
              || __builtin_expect (chunksize_nomask (victim)
                   > av->system_mem, 0))
            malloc_printerr ("malloc(): memory corruption");
          // 获取victim的size
          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
```

```
33                no exact fit for a small chunk.
34             */
35              // size是否在smallbin范围内
36             // usbin中是否只有一个chunk (bk指向自身)
37             // 选中的chunk是否位于usbin开头(是否为lastreminder)
38             // 判断usbin的所选chunk的size是否大于申请size+chunk头大小
39             if (in_smallbin_range (nb) &&
40                 bck == unsorted_chunks (av) &&
41                 victim == av->last_remainder &&
42                 (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
43               {
44                 /* split and reattach remainder */
45                 // 减去分配大小
46                 remainder_size = size - nb;
47                 //指向下一个偏移的地方
48                 remainder = chunk_at_offset (victim, nb);
49                 //改写unsortedbin,也就是切割以后的值
50                 av->last_remainder = remainder;
51                 //更新last remainder的指针
52                 unsorted_chunks (av)->bk = unsorted_chunks (av)->fd =
    remainder;
53                 av->last_remainder = remainder;
54                 remainder->bk = remainder->fd = unsorted_chunks (av);
55                 //切割后如果不在smallbin的范围内,则设置largebin的指针
56                 if (!in_smallbin_range (remainder_size))
57                   {
58                     remainder->fd_nextsize = NULL;
59                     remainder->bk_nextsize = NULL;
60                   }
61
62                 set_head (victim, nb | PREV_INUSE |
63                           (av != &main_arena ? NON_MAIN_ARENA : 0));
64                 set_head (remainder, remainder_size | PREV_INUSE);
65                 set_foot (remainder, remainder_size);
66
67                 check_malloced_chunk (av, victim, nb);
68                 void *p = chunk2mem (victim);
69                 alloc_perturb (p, bytes);
70                 // 返回分配的chunk
71                 return p;
72               }
73
74             /* remove from unsorted list */
75             unsorted_chunks (av)->bk = bck;
76             bck->fd = unsorted_chunks (av);
77
78             /* Take now instead of binning if exact fit */
79           // 如果size恰好等于申请的大小, 直接分配
80             if (size == nb)
81               {
82                 set_inuse_bit_at_offset (victim, size);
83                 if (av != &main_arena)
```

```
          set_non_main_arena (victim);
                // 如果存在tcache且对应bin未满，则将chunk放入tcache中
#if 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       // 反之将chunk返回给user
               check_malloced_chunk (av, victim, nb);
               void *p = chunk2mem (victim);
               alloc_perturb (p, bytes);
               return p;
#if USE_TCACHE
         }
#endif
             }

           /* place chunk in bin */
         // 把取出来的 chunk 放到对应的 small bin 中。
           if (in_smallbin_range (size))
             {
               victim_index = smallbin_index (size);
               bck = bin_at (av, victim_index);
               fwd = bck->fd;
             }
           // 反之则放到largebin中
           else
             {
               victim_index = largebin_index (size);
               bck = bin_at (av, victim_index);
               fwd = bck->fd;

               /* maintain large bins in sorted order */
               /* 从这里我们可以总结出，largebin 以 fd_nextsize 递减排序。
                    同样大小的 chunk，后来的只会插入到之前同样大小的 chunk 后，
                    而不会修改之前相同大小的fd/bk_nextsize，这也很容易理解，
                    可以减低开销。此外，bin 头不参与 nextsize 链接。*/
               // largebin链表不为空
               if (fwd != bck)
                 {
                   /* Or with inuse bit to speed comparisons */
                   size |= PREV_INUSE;
                   /* if smaller than smallest, bypass loop below */
                    // bck->bk 存储着相应 large bin 中最小的chunk。
```

```
135                       // 如果遍历的 chunk 比当前最小的还要小，那就只需要插入到链表尾
    部。
136                       // 判断 bck->bk 是不是在 main arena。
137                     assert (chunk_main_arena (bck->bk));
138                     if ((unsigned long) (size)
139                   < (unsigned long) chunksize_nomask (bck->bk))
140                       {
141                         // 令 fwd 指向 large bin 头
142                         fwd = bck;
143                         // 令 bck 指向 largin bin 尾部 chunk
144                         bck = bck->bk;
145                         // victim 的 fd_nextsize 指向 largin bin 的第一个 chunk
146                         victim->fd_nextsize = fwd->fd;
147                         // victim 的 bk_nextsize 指向原来链表的第一个 chunk 指向的
    bk_nextsize
148                         victim->bk_nextsize = fwd->fd->bk_nextsize;
149                         // 原来链表的第一个 chunk 的 bk_nextsize 指向 victim
150                           // 原来指向链表第一个 chunk 的 fd_nextsize 指向 victim
151                         fwd->fd->bk_nextsize = victim->bk_nextsize-
    >fd_nextsize = victim;
152                       }
153                     else
154                       {
155                         // 当前要插入的 victim 的大小大于最小的 chunk
156                           // 判断 fwd 是否在 main arena
157                         assert (chunk_main_arena (fwd));
158                         // 从链表头部开始找到不比 victim 大的 chunk
159                         while ((unsigned long) size < chunksize_nomask (fwd))
160                           {
161                             fwd = fwd->fd_nextsize;
162                   assert (chunk_main_arena (fwd));
163                           }
164                         // 如果找到了一个和 victim 一样大的 chunk，
165                         // 那就直接将 chunk 插入到该chunk的后面，并不修改 nextsize
    指针。
166                         if ((unsigned long) size
167                   == (unsigned long) chunksize_nomask (fwd))
168                           /* Always insert in the second position.  */
169                           fwd = fwd->fd;
170                         else
171                           {
172                             // 如果找到的chunk和当前victim大小不一样
173                               // 那么就需要构造 nextsize 双向链表了
174                             victim->fd_nextsize = fwd;
175                             victim->bk_nextsize = fwd->bk_nextsize;
176                             fwd->bk_nextsize = victim;
177                             victim->bk_nextsize->fd_nextsize = victim;
178                           }
179                         bck = fwd->bk;
180                       }
181                   }
182                 else
```

```
183                 // 如果该largebin空的话，直接简单使得 fd_nextsize 与 bk_nextsize 构
    成一个双向链表即可。
184                   victim->fd_nextsize = victim->bk_nextsize = victim;
185               }
186               // 放到对应的 bin 中，构成 bck<-->victim<-->fwd。
187             mark_bin (av, victim_index);
188             victim->bk = bck;
189             victim->fd = fwd;
190             fwd->bk = victim;
191             bck->fd = victim;
192
193   #if USE_TCACHE
194         /* If we've processed as many chunks as we're allowed while
195        filling the cache, return one of the cached ones.  */
196
197         ++tcache_unsorted_count;
198         if (return_cached
199         && mp_.tcache_unsorted_limit > 0
200         && tcache_unsorted_count > mp_.tcache_unsorted_limit)
201       {
202         return tcache_get (tc_idx);
203       }
204   #endif
205
206   #define MAX_ITERS       10000
207             // while 最多迭代 10000 次后退出。
208             if (++iters >= MAX_ITERS)
209               break;
210           }
```

流程总结

如果程序执行到这里，说明和size大小一致的fastbin及smallbin中都没有满足要求的chunk

因此需要一个大循环来检索usbin、largebin并重新分配chunk

- 按照FIFO的规则循环取usbin中的chunk，并获取大小
- 如果size位于smallbin范围内，usbin中有且只有一个chunk，且该chunk大小大于size，则
  - 切割该chunk，返回给user
  - 切剩下的chunk放回usbin中
- 如果usbin中chunk大小恰好等于size
  - 如果存在tcache且对应bin未满，则将chunk放入tcache中
  - 反之直接将chunk返回给用户
- 此时已经可以确定选中chunk大小小于申请的size,则
  - 如果选中chunk属于smallbin范围,放入smallbin
  - 反之放入largebin

Large chunk

如果请求的 chunk 在 large chunk 范围内，就在对应的 bin 中从小到大进行扫描，找到第一个合适的。

```
#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

      /*
         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 */
          // 如果对应的 bin 为空或者其中的chunk最大的也很小，那就跳过
        // first(bin)=bin->fd 表示当前链表中最大的chunk
          if ((victim = first (bin)) != bin
          && (unsigned long) chunksize_nomask (victim)
            >= (unsigned long) (nb))
            {
              // 反向遍历链表,直到找到第一个不小于所需chunk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.  */
              // 如果最终取到的chunk不是该bin中的最后一个chunk，并且该chunk与其前面的
chunk
                // 的大小相同，那么我们就取其前面的chunk，这样可以避免调整
bk_nextsize,fd_nextsize
              //  链表。因为大小相同的chunk只有一个会被串在nextsize链上。
              if (victim != last (bin)
          && chunksize_nomask (victim)
            == chunksize_nomask (victim->fd))
                victim = victim->fd;
                // remainder_size = size - nb;
              // 计算分配后剩余大小
              remainder_size = size - nb;
              // 进行unlink
              unlink (av, victim, bck, fwd);
```

```
44
45                /* Exhaust */
46                  // 剩下的大小不足以当做一个块,则返回时直接返回整个chunk
47                if (remainder_size < MINSIZE)
48                  {
49                    set_inuse_bit_at_offset (victim, size);
50                    if (av != &main_arena)
51              set_non_main_arena (victim);
52                  }
53                /* Split */
54                //  剩下的大小还可以作为一个chunk，进行分割。
55                else
56                  {
57                    // 获取剩下那部分chunk的指针，称为remainder
58                    remainder = chunk_at_offset (victim, nb);
59                    /* We cannot assume the unsorted list is empty and
    therefore
60                       have to perform a complete insert here.  */
61                    // 插入unsorted bin中
62                    bck = unsorted_chunks (av);
63                    fwd = bck->fd;
64                    // 判断 unsorted bin 是否被破坏。
65            if (__glibc_unlikely (fwd->bk != bck))
66              malloc_printerr ("malloc(): corrupted unsorted chunks");
67                    // fd bk以及相应的标志位
68                    remainder->bk = bck;
69                    remainder->fd = fwd;
70                    bck->fd = remainder;
71                    fwd->bk = remainder;
72                    if (!in_smallbin_range (remainder_size))
73                      {
74                        remainder->fd_nextsize = NULL;
75                        remainder->bk_nextsize = NULL;
76                      }
77                    set_head (victim, nb | PREV_INUSE |
78                              (av != &main_arena ? NON_MAIN_ARENA : 0));
79                    set_head (remainder, remainder_size | PREV_INUSE);
80                    set_foot (remainder, remainder_size);
81                  }
82                check_malloced_chunk (av, victim, nb);
83                void *p = chunk2mem (victim);
84                alloc_perturb (p, bytes);
85                return p;
86              }
87            }
```

流程总结

该流程为在largebin中检索符合申请要求的chunk

- 如果请求大小不在smallbin范围内，根据之前计算的largebin索引获取对应的largebin
- 如果对应bin不为空且最大chunk大小大于申请size则:
  - 依据 `bk_nextsize` 反向搜索chunk(由小到大)，直到找到不小于申请大小的chunk
    - 如果该chunk不位于bin末尾, 且其后面还有同样大小的chunk, 则取后面的chunk (`为了避免修改nextsize`)
    - 反之直接取该chunk
  - 对该chunk进行切割:
    - 如果剩下的chunk小于最小chunk大小,则将原chunk全部返回
    - 如果剩下chunk大小大于最小chunk大小,则把切剩下的放入usbin中
    - 返回chunk指针给用户

寻找较大 chunk

如果走到了这里，那说明对于用户所需的 chunk，不能直接从其对应的合适的 bin 中获取 chunk，所以我们需要来查找比当前 bin 更大的 fast bin ， small bin 或者 large bin。

```
1         ++idx;
2       // 获取对应的bin
3         bin = bin_at (av, idx);
4           // 获取当前索引在binmap中的block索引
5           // #define idx2block(i) ((i) >> BINMAPSHIFT)  ,BINMAPSHIFT=5
6           // Binmap按block管理，每个block为一个int，共32个bit，可以表示32个bin中是否有空闲chunk存在
7           // 所以这里是右移5
8         block = idx2block (idx);
9           // 获取当前块大小对应的映射，这里可以得知相应的bin中是否有空闲块
10        map = av->binmap[block];
11          // #define idx2bit(i) ((1U << ((i) & ((1U << BINMAPSHIFT) - 1))))
12          // 将idx对应的比特位设置为1，其它位为0
13        bit = idx2bit (idx);
14
15        for (;; )
16          {
17            /* Skip rest of block if there are no more set bits in this block.
    */
18            // 如果bit>map，则表示该 map 中没有比当前所需要chunk大的空闲块
19              // 如果bit为0，那么说明，上面idx2bit带入的参数为0。
20            if (bit > map || bit == 0)
21              {
22                do
23                  {
```

```
24                      // 寻找下一个block，直到其对应的map不为0。
25                        // 如果已经不存在的话，那就只能使用top chunk了
26                      if (++block >= BINMAPSIZE) /* out of bins */
27                        goto use_top;
28                    }
29                  while ((map = av->binmap[block]) == 0);
30                  // 获取其对应的bin，因为该map中的chunk大小都比所需的chunk大，而且
31                    // map本身不为0，所以必然存在满足需求的chunk。
32                  bin = bin_at (av, (block << BINMAPSHIFT));
33                  bit = 1;
34                }
35
36              /* Advance to bin with set bit. There must be one. */
37              // 从当前map的最小的bin一直找，直到找到合适的bin。
38              // 这里是一定存在的
39                while ((bit & map) == 0) {
40                    bin = next_bin(bin);
41                    bit <<= 1;
42                    assert(bit != 0);
43                }
44
45
46              /* Inspect the bin. It is likely to be non-empty */
47              // 获取对应的bin
48              victim = last (bin);
49
50              /*  If a false alarm (empty bin), clear the bit. */
51              // 如果victim=bin，那么我们就将map对应的位清0，然后获取下一个bin
52              // 这种情况发生的概率应该很小。
53              if (victim == bin)
54                {
55                  av->binmap[block] = map &= ~bit; /* Write through */
56                  bin = next_bin (bin);
57                  bit <<= 1;
58                }
59
60              else
61                {
62                  // 获取对应victim的大小
63                  size = chunksize (victim);
64
65                  /*  We know the first chunk in this bin is big enough to use.
    */
66                  assert ((unsigned long) (size) >= (unsigned long) (nb));
67                    // 计算分割后剩余的大小
68                  remainder_size = size - nb;
69
70                  /* unlink */
71                  unlink (av, victim, bck, fwd);
72
73                  /* Exhaust */
74                  // 如果分割后不够一个chunk 整个返回
```

```
75                if (remainder_size < MINSIZE)
76                  {
77                    set_inuse_bit_at_offset (victim, size);
78                    if (av != &main_arena)
79              set_non_main_arena (victim);
80                  }
81
82                /* Split */
83                // 否则就切割
84                else
85                  {
86                    remainder = chunk_at_offset (victim, nb);
87
88                    /* We cannot assume the unsorted list is empty and
    therefore
89                       have to perform a complete insert here.  */
90                    bck = unsorted_chunks (av);
91                    fwd = bck->fd;
92            if (__glibc_unlikely (fwd->bk != bck))
93              malloc_printerr ("malloc(): corrupted unsorted chunks 2");
94                    remainder->bk = bck;
95                    remainder->fd = fwd;
96                    bck->fd = remainder;
97                    fwd->bk = remainder;
98
99                    /* advertise as last remainder */
100                   if (in_smallbin_range (nb))
101                     av->last_remainder = remainder;
102                   if (!in_smallbin_range (remainder_size))
103                     {
104                       remainder->fd_nextsize = NULL;
105                       remainder->bk_nextsize = NULL;
106                     }
107                   set_head (victim, nb | PREV_INUSE |
108                             (av != &main_arena ? NON_MAIN_ARENA : 0));
109                   set_head (remainder, remainder_size | PREV_INUSE);
110                   set_foot (remainder, remainder_size);
111                 }
112               check_malloced_chunk (av, victim, nb);
113               void *p = chunk2mem (victim);
114               alloc_perturb (p, bytes);
115               return p;
116             }
117         }
```

Top chunk

如果所有的 bin 中的 chunk 都没有办法直接满足要求（即不合并），或者说都没有空闲的 chunk。那么我们就只能使用 top chunk 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.)
       */
      // 获取当前的top chunk，并计算其对应的大小
      victim = av->top;
      size = chunksize (victim);
        // 如果分割之后，top chunk 大小仍然满足 chunk 的最小大小，那么就可以直接进行分割。
      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_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.  */
        // 否则，判断是否有 fast chunk
      else if (atomic_load_relaxed (&av->have_fastchunks))
        {
          // 先执行一次fast bin的合并
          malloc_consolidate (av);
          /* restore original bin index */
          // 判断需要的chunk是在small bin范围内还是large bin范围内
            // 并计算对应的索引
            // 等待下次再看看是否可以
```

```
46              if (in_smallbin_range (nb))
47                idx = smallbin_index (nb);
48              else
49                idx = largebin_index (nb);
50            }
51
52          /*
53             Otherwise, relay to handle system-dependent cases
54           */
55        // 如果堆内存不够，我们就需要使用 sysmalloc 来申请内存了。
56          else
57            {
58              void *p = sysmalloc (nb, av);
59              if (p != NULL)
60                alloc_perturb (p, bytes);
61              return p;
62            }
63        }
64    }
```

流程总结

- 获取topchunk指针，并计算大小
- topchunk大小是否大于size+MINISIZE
  - 大于的话，则进行分割，将chunk返回给用户
  - 反之判断是否存在fastchun
    - 如果存在则进行一次堆块合并，之后看看能不能那个切割
    - 如果不存在fastchunk，则表示堆内存不足，需要通过top chunk分割

流程总结

这里所说的用户申请的 `size` 包含 `chunk` 头部。

1. 检查是否设置了 `malloc_hook` ，若设置了则跳转进入 `malloc_hook` （第一次调用 `malloc` 时设置了 `malloc_hook` 来实现初始化操作），若未设置则获取当前的分配区，进入 `int_malloc` 函数。
   1. 如果当前的分配区为空，则调用sysmalloc分配空间，返回指向新 `chunk` 的指针，否则进入下一步。
   2. 若用户申请的大小符合 `fastbin` 大小范围，若相应大小的链表不为空则返回链表头部的 `chunk` ，否则进入下一步。
   3. 如果用户申请的大小符合 `small bin` 的范围，则在相应大小的链表中寻找 `chunk` ，若 `small bin` 未初始化，则进入第 `4` 步，否则验证链表是否为空，若不为空将链表尾部的 `chunk` 分配给用户，否则进入第 `5` 步。
   4. 调用malloc_consolidate函数将 `fastbin` 进行合并插入到 `unsorted bin` 链表中（通过 `get_max_fast` 若堆未初始化则初始化堆）。

5. 用户申请的大小符合 `large bin` 或 `small bin` 链表为空，开始处理 `unsorted bin` 链表中的 `chunk` 。在 `unsorted bin` 链表中查找符合大小的 `chunk` ，若用户申请的大小为 `small bin` ， `unsorted bin` 中只有一块 `chunk` 并指向 `last_remainder` ，且 `chunk size` 的大小大于 `size+MINSIZE` (保证拆分之后的 `remainder` 能组成一个 `chunk`)，则对当前的 `chunk` 进行拆分，更新分配区中的 `last_remainder` 。否则进入下一步。
6. 将当前的 `unsorted bin` 中的 `chunk` 取下，若其 `size` 恰好为用户申请的 `size` ，则将 `chunk` 返回给用户。否则进入下一步
7. 获取当前 `chunk size` 所对应的 `bins` 数组中的头指针。（ `large bin` 需要保证从大到小的顺序，因此需要遍历）将其插入到对应的链表中。如果处理的 `chunk` 的数量大于 `MAX_ITERS` 则不在处理。进入下一步。
8. 如果用户申请的空间的大小符合 `large bin` 的范围或者对应的 `small bin` 链表为空且 `unsorted bin` 链表中没有符合大小的 `chunk` ，则在对应的 `large bin` 链表中查找符合条件的 `chunk` （即其大小要大于用户申请的 `size` ）。若找到相应的 `chunk` 则对 `chunk` 进行拆分，返回符合要求的 `chunk` （无法拆分时整块返回）。否则进入下一步。
9. 根据 `binmap` 找到表示更大 `size` 的 `large bin` 链表，若其中存在空闲的 `chunk` ，则将 `chunk` 拆分之后返回符合要求的部分，并更新 `last_remainder` 。否则进入下一步。
10. 若 `top chunk` 的大小大于用户申请的空间的大小，则将 `top chunk` 拆分，返回符合用户要求的 `chunk` ，并更新 `last_remainder` ，否则进入下一步。
11. 若 `fast bin` 不为空（其他线程可能释放 `chunk` ），则调用 `malloc_consolidate` 合并 `fastbin` ，返回第 `5` 步继续执行。否则进入下一步。
12. 调用 `sysmalloc` 分配空间，返回指向新 `chunk` 的指针。

2. 若 `_int_malloc` 函数返回的 `chunk` 指针为空，且当前分配区指针不为空，则再次尝试 `_int_malloc`
3. 对 `chunk` 指针进行检查，主要检查 `chunk` 是否为 `mmap` ，且位于当前的分配区内。

__libc_free

`free(void* p)` 释放 `p` 指向的 `chunk` 指针，如果 `p` 是空值，则没有任何的效果。如果 `p` 已经被释放，那么将会触发未定义的行为。默认释放大容量内存的时候将直接交还给 `system` ，从而减少系统占用的空间。

```
void
__libc_free (void *mem)
{
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  void (*hook) (void *, const void *)
    = atomic_forced_read (__free_hook);
  if (__builtin_expect (hook != NULL, 0))
    {
      (*hook)(mem, RETURN_ADDRESS (0));
```

```
        return;
      }

    if (mem == 0)                              /* free(0) has no effect */
      return;

    p = mem2chunk (mem);

    if (chunk_is_mmapped (p))                       /* release mmapped memory.
   */
      {
        /* See if the dynamic brk/mmap threshold needs adjusting.
       Dumped fake mmapped chunks do not affect the threshold.  */
        if (!mp_.no_dyn_threshold
            && chunksize_nomask (p) > mp_.mmap_threshold
            && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX
        && !DUMPED_MAIN_ARENA_CHUNK (p))
          {
            mp_.mmap_threshold = chunksize (p);
            mp_.trim_threshold = 2 * mp_.mmap_threshold;
            LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                        mp_.mmap_threshold, mp_.trim_threshold);
          }
        munmap_chunk (p);
        return;
      }

    MAYBE_INIT_TCACHE ();

    ar_ptr = arena_for_chunk (p);
    _int_free (ar_ptr, p, 0);
  }
  libc_hidden_def (__libc_free)
```

程序首先判断 `free_hook` 是否被设置，如果被设置，则执行 `free_hook` 。如果未被设置，且需要释放的指针不为空值，则判断指针指向的 `chunk` 是否是 `mmap` 的。

如果 `chunk` 是由 `mmap` 分配的，则首先更新 `mmap` 分配和收缩阈值，然后调用 `munmap_chunk` 函数释放 `chunk` 。否则调用 `_int_free` 函数释放 `chunk` 。

_int_free

```
static void
_int_free (mstate av, mchunkptr p, int have_lock)
{
  INTERNAL_SIZE_T size;        /* its size */
  mfastbinptr *fb;             /* associated fastbin */
  mchunkptr nextchunk;         /* next contiguous chunk */
```

```
    INTERNAL_SIZE_T nextsize;    /* its size */
    int nextinuse;               /* true if nextchunk is used */
    INTERNAL_SIZE_T prevsize;    /* size of previous contiguous chunk */
    mchunkptr bck;               /* misc temp for linking */
    mchunkptr fwd;               /* misc temp for linking */

    size = chunksize (p);

    /* Little security check which won't hurt performance: the
       allocator never wrapps around at the end of the address space.
       Therefore we can exclude some size values which might appear
       here by accident or by "design" from some intruder.  */
    if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
        || __builtin_expect (misaligned_chunk (p), 0))
      malloc_printerr ("free(): invalid pointer");
    /* We know that each chunk is at least MINSIZE bytes in size or a
       multiple of MALLOC_ALIGNMENT.  */
    if (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
      malloc_printerr ("free(): invalid size");

    check_inuse_chunk(av, p);

#if USE_TCACHE
    {
      size_t tc_idx = csize2tidx (size);

      if (tcache
      && tc_idx < mp_.tcache_bins
      && tcache->counts[tc_idx] < mp_.tcache_count)
        {
      tcache_put (p, tc_idx);
      return;
        }
    }
#endif
```

判断待释放chunk size和指针的合法性

如果开启tcache且对应bin未满的话,将chunk放入bin中

• fastbin

如果大小属于fastbin，且通过检查后

连入fastbin中

```
    /*
      If eligible, place chunk on a fastbin so it can be found
      and used quickly in malloc.
    */
```

```
 // size在fastbin范围内
  if ((unsigned long)(size) <= (unsigned long)(get_max_fast ()))
// 和topchunk不相邻
#if TRIM_FASTBINS
      /*
    If TRIM_FASTBINS set, don't place chunks
    bordering top into fastbins
      */
      && (chunk_at_offset(p, size) != av->top)
#endif
      ) {
    // size检查
    if (__builtin_expect (chunksize_nomask (chunk_at_offset (p, size))
              <= 2 * SIZE_SZ, 0)
    || __builtin_expect (chunksize (chunk_at_offset (p, size))
                >= av->system_mem, 0))
      {
    bool fail = true;
    /* We might not have a lock at this point and concurrent modifications
       of system_mem might result in a false positive.  Redo the test after
       getting the lock.  */
    if (!have_lock)
      {
        __libc_lock_lock (av->mutex);
        fail = (chunksize_nomask (chunk_at_offset (p, size)) <= 2 * SIZE_SZ
            || chunksize (chunk_at_offset (p, size)) >= av->system_mem);
        __libc_lock_unlock (av->mutex);
      }

    if (fail)
      malloc_printerr ("free(): invalid next size (fast)");
      }

    free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);

    atomic_store_relaxed (&av->have_fastchunks, true);
      // 获取fastbin索引
    unsigned int idx = fastbin_index(size);
      // 获取fastbin头指针
    fb = &fastbin (av, idx);

    /* Atomically link P to its fastbin: P->FD = *FB; *FB = P;  */
      // 使用原子操作将P插入到链表中
    mchunkptr old = *fb, old2;

    if (SINGLE_THREAD_P)
      {
    /* Check that the top of the bin is not the record we are going to
       add (i.e., double free).  */
    if (__builtin_expect (old == p, 0))
      malloc_printerr ("double free or corruption (fasttop)");
    p->fd = old;
```

```
57        *fb = p;
58          }
59        else
60          do
61        {
62          /* Check that the top of the bin is not the record we are going to
63             add (i.e., double free).  */
64          if (__builtin_expect (old == p, 0))
65            malloc_printerr ("double free or corruption (fasttop)");
66          p->fd = old2 = old;
67        }
68          while ((old = catomic_compare_and_exchange_val_rel (fb, p, old2))
69             != old2);
70
71        /* Check that size of fastbin chunk at the top is the same as
72           size of the chunk that we are adding.  We can dereference OLD
73           only if we have the lock, otherwise it might have already been
74           allocated again.  */
75        if (have_lock && old != NULL
76        && __builtin_expect (fastbin_index (chunksize (old)) != idx, 0))
77          malloc_printerr ("invalid fastbin entry (free)");
78      }
```

合并非mmap的空闲chunk

只有不是 fast bin 的情况下才会触发 unlink

首先我们先说一下为什么会合并 chunk，这是为了避免 heap 中有太多零零碎碎的内存块，合并之后可以用来应对更大的内存块请求。合并的主要顺序为

- 先考虑物理低地址空闲块
- 后考虑物理高地址空闲块

合并后的 chunk 指向合并的 chunk 的低地址。

```
1     else if (!chunk_is_mmapped(p)) {
2
3       /* If we're single-threaded, don't lock the arena.  */
4       if (SINGLE_THREAD_P)
5         have_lock = true;
6
7       if (!have_lock)
8         __libc_lock_lock (av->mutex);
9
10      nextchunk = chunk_at_offset(p, size);
11
12      /* Lightweight tests: check whether the block is already the
13         top block.  */
14        // 当前chunk不能是top chunk
15      if (__glibc_unlikely (p == av->top))
16        malloc_printerr ("double free or corruption (top)");
```

```
      /* Or whether the next chunk is beyond the boundaries of the arena.  */
        // 当前freechunk的下一个chunk不能超过arena边界
      if (__builtin_expect (contiguous (av)
                && (char *) nextchunk
                >= ((char *) av->top + chunksize(av->top)), 0))
      malloc_printerr ("double free or corruption (out)");
      /* Or whether the block is actually not marked used.  */
        // 当前要free的chunk的pre_in_use为0
      if (__glibc_unlikely (!prev_inuse(nextchunk)))
        malloc_printerr ("double free or corruption (!prev)");

      nextsize = chunksize(nextchunk);
        // 判断下一个chunk的大小是否不大于2*SIZE_SZ, 或者
          // nextsize是否大于系统可提供的内存
      if (__builtin_expect (chunksize_nomask (nextchunk) <= 2 * SIZE_SZ, 0)
      || __builtin_expect (nextsize >= av->system_mem, 0))
        malloc_printerr ("free(): invalid next size (normal)");

      free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);

      /* consolidate backward */
        // 后向合并 - 合并低地址 chunk
      if (!prev_inuse(p)) {
        prevsize = prev_size (p);
        size += prevsize;
        p = chunk_at_offset(p, -((long) prevsize));
        unlink(av, p, bck, fwd);
      }
      // 下一块不是 top chunk - 前向合并 - 合并高地址 chunk
        // 合并后放入usbin中
      if (nextchunk != av->top) {
        /* get and clear inuse bit */
        nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

        /* consolidate forward */
        if (!nextinuse) {
      unlink(av, nextchunk, bck, fwd);
      size += nextsize;
        } else
      clear_inuse_bit_at_offset(nextchunk, 0);

        /*
      Place the chunk in unsorted chunk list. Chunks are
      not placed into regular bins until after they have
      been given one chance to be used in malloc.
        */

        bck = unsorted_chunks(av);
        fwd = bck->fd;
        if (__glibc_unlikely (fwd->bk != bck))
      malloc_printerr ("free(): corrupted unsorted chunks");
        p->fd = fwd;
```

```
69          p->bk = bck;
70          if (!in_smallbin_range(size))
71        {
72          p->fd_nextsize = NULL;
73          p->bk_nextsize = NULL;
74        }
75          bck->fd = p;
76          fwd->bk = p;
77
78          set_head(p, size | PREV_INUSE);
79          set_foot(p, size);
80
81          check_free_chunk(av, p);
82        }
83
84        /*
85          If the chunk borders the current high end of memory,
86          consolidate into top
87        */
88        // 下一块是 top chunk - 合并到 top chunk¶
89        else {
90          size += nextsize;
91          set_head(p, size | PREV_INUSE);
92          av->top = p;
93          check_chunk(av, p);
94        }
95
96        /*
97          If freeing a large space, consolidate possibly-surrounding
98          chunks. Then, if the total unused topmost memory exceeds trim
99          threshold, ask malloc_trim to reduce top.
100
101         Unless max_fast is 0, we don't know if there are fastbins
102         bordering top, so we cannot tell for sure whether threshold
103         has been reached unless fastbins are consolidated.  But we
104         don't want to consolidate on each free.  As a compromise,
105         consolidation is performed if FASTBIN_CONSOLIDATION_THRESHOLD
106         is reached.
107       */
108       // 如果合并后的 chunk 的大小大于FASTBIN_CONSOLIDATION_THRESHOLD
109            // 一般合并到 top chunk 都会执行这部分代码。
110            // 那就向系统返还内存
111       if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
112           // 如果有 fast chunk 就进行合并
113         if (atomic_load_relaxed (&av->have_fastchunks))
114       malloc_consolidate(av);
115
116         if (av == &main_arena) {
117   #ifndef MORECORE_CANNOT_TRIM
118       if ((unsigned long)(chunksize(av->top)) >=
119           (unsigned long)(mp_.trim_threshold))
120         systrim(mp_.top_pad, av);
```

```
121   #endif
122         } else {
123       /* Always try heap_trim(), even if the top chunk is not
124          large, because the corresponding heap might go away.  */
125       heap_info *heap = heap_for_ptr(top(av));
126 
127       assert(heap->ar_ptr == av);
128       heap_trim(heap, mp_.top_pad);
129         }
130       }
131 
132       if (!have_lock)
133         __libc_lock_unlock (av->mutex);
134     }
```

- **总结**

1. 首先检查 `free_hook` ，如果已经设置，则跳转到 `free_hook` 处，否则进入第二步
2. 如果 `chunk` 是通过 `mmap` 分配的，则调用 `munmap` 函数释放 `chunk` ，否则调用 `_int_free` 函数（这部分只分析 `_int_free` 函数，munmap在这里
    1. 对传入的指针和其指向的 `chunk size` 进行合法性验证，判断传入的 `chunk` 为 `inuse`
    2. 若 `chunk size` 的大小满足 `fast bin` 的大小范围，则在经过指针和 `size` 的合法性验证之后将 `chunk` 插入到 `fast bin` 链表中。若 `chunk` 不是由 `mmap` 分配的，则进入下一步，否则进入第 `5` 步
    3. 此时 `chunk size` 的大小超过 `fast bin` 的规定范围，将 `chunk` 与物理相邻的前一个 `chunk` 进行前向合并，与物理相邻的后一个 `chunk` (包含 `top chunk`)进行后向合并。合并后的 `chunk` 插入到 `unsorted bin` 链表中。进入下一步
    4. 判断释放的 `chunk size` 的大小，超过阈值之后收缩内存。
    5. `chunk` 是通过 `mmap` 分配的，调用 `munmap` 释放 `chunk` 。

calloc

calloc 与 malloc 的区别是 **calloc 在分配后会自动进行清空，这对于某些信息泄露漏洞的利用来说是致命的**

```
1   calloc(0x20);
2   //等同于
3   ptr=malloc(0x20);
4   memset(ptr,0,0x20);
```

realloc

realloc 函数可以身兼 malloc 和 free 两个函数的功能。 `此外realloc函数是存在realloc_hook的`

realloc 的操作并不是像字面意义上那么简单，其内部会根据不同的情况进行不同操作

- 首先判断 `realloc_hook` 是否设置，如果设置则跳转到 `realloc_hook` 位置，否则进入下一步
- 当 realloc(ptr,size) 的 size 不等于 ptr 的 size 时
  - 如果申请 size > 原来 size
    - 如果 chunk 与 top chunk 相邻，直接扩展这个 chunk 到新 size 大小
    - 如果 chunk 与 top chunk 不相邻，相当于 free(ptr),malloc(new_size)
  - 如果申请 size < 原来 size
    - 如果相差不足以容得下一个最小 chunk(64 位下 32 个字节，32 位下 16 个字节)，则保持不变
    - 如果相差可以容得下一个最小 chunk，则切割原 chunk 为两部分，free 掉后一部分
- 当 realloc(ptr,size) 的 size 等于 0 时，相当于 free(ptr)
- 当 realloc(ptr,size) 的 size 等于 ptr 的 size，不进行任何操作

Linux x64 内存布局

- Kernel space

  内核空间

- Undefined Region

  未定义

- Stack

  栈 向低地址空间生长

- Memory Mapping Region

  内存映射段，可以将文件内容直接映射到内存，提高io效率

- Heap

  堆 向高地址方向生长

  与栈一样，堆用于运行时内存分配；但不同点是，堆用于存储那些生存期与函数调用无关的数据。

  - 为什么需要堆：
    - 光有栈，对于面向过程的程序设计还远远不够，因为栈上的数据在函数返回的时候就会被释放掉，所以无法将数据传递至函数外部。而全局变量没有办法动态地产生，只能在编译的时候定义，有很多情况下缺乏表现力，在这种情况下，堆（Heap）是一种唯一的选择。

- BSS

BSS段 可读可写

主要存储未初始化的全局变量或者初始化为0的全局变量和静态变量的一块内存区域

- Data

  数据段

  包含静态初始化的数据，所以有初值的全局变量和static变量在data区

- Text

  Text段 只读

  程序代码段