free 内存释放

__lib_free

- 参考文章--How does LIBC_PROBE macro actually work in Glibc?

1. 初始化过程与 __lib_malloc 类似可以指定 free_hook 替换掉 glibc 中的 free 实现。默认 __free_hook 为 NULL。

```
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;
```

2. 判断是否是通过 mmap 申请的内存，如果是则尝试使用 munmap_chunk 进行释放：

```
#ifndef DEFAULT_MMAP_THRESHOLD_MIN
#define DEFAULT_MMAP_THRESHOLD_MIN (128 * 1024)  //mmap_threshold 默认值
#endif

#define DEFAULT_MMAP_THRESHOLD_MAX (4 * 1024 * 1024 * sizeof(long))

/* These variables are used for undumping support.  Chunked are marked
as using mmap, but we leave them alone if they fall into this
range.  NB: The chunk size for these chunks only includes the
initial size field (of SIZE_SZ bytes), there is no trailing size
field (unlike with regular mmapped chunks).  */
static mchunkptr dumped_main_arena_start; /* Inclusive.  */ // = 0
static mchunkptr dumped_main_arena_end;   /* Exclusive.  */ // = 0


/* True if the pointer falls into the dumped arena.  Use this after
chunk_is_mmapped indicates a chunk is mmapped.
如果指针落入转储区域，则为真。之后使用这个 chunk_is_mmapped 表示块被 mmapped 管理*/
#define DUMPED_MAIN_ARENA_CHUNK(p) \
((p) >= dumped_main_arena_start && (p) < dumped_main_arena_end)

.................................

  void
__libc_free (void *mem)
{

    .............
#ifdef USE_MTAG
    /* Quickly check that the freed pointer matches the tag for the memory.
    This gives a useful double-free detection.
    快速检查释放的指针是否与内存标签匹配。
        这提供了有用的双重免费检测。*/
    *(volatile char *)mem;
#endif

    int err = errno;

    p = mem2chunk (mem);

    /* Mark the chunk as belonging to the library again.  */
    (void)TAG_REGION (chunk2rawmem (p), CHUNK_AVAILABLE_SIZE (p) - CHUNK_HDR_SZ);

    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 //起始值为 128KB，它是系统选择 mmap() 还是 brk() 进行分配的一个门槛值，它是动态变化的。
                                                         //大于 mp._mmap_threshold 时，系统会选择 mmap() 进行分配，否则选择 brk() 进行分配。
            && chunksize_nomask (p) <= DEFAULT_MMAP_THRESHOLD_MAX
        && !DUMPED_MAIN_ARENA_CHUNK (p))
        {
            mp_.mmap_threshold = chunksize (p);
            mp_.trim_threshold = 2 * mp_.mmap_threshold; //trim_threshold 是归还系统的阈值
            LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                        mp_.mmap_threshold, mp_.trim_threshold);
        }
        munmap_chunk (p);
    }
```

3. 非 mmap 则使用 _int_free 进行释放:

```
else
    {
    MAYBE_INIT_TCACHE ();

    ar_ptr = arena_for_chunk (p);
    _int_free (ar_ptr, p, 0);
    }

__set_errno (err);
}
```

_int_free

- 参考文章--heap-exploitation
- 参考文章--heap-9-_int_free 源码及其部分分析

1. 安全检查：

```
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.
    不会影响性能的小安全检查：分配器永远不会在地址空间的末尾回绕。
    因此我们可以排除一些可能偶然出现的尺寸值或者是某些入侵者的“设计”*/

//  如果当前chunk的大小异常大，或者地址不是按 2SIZE_SZ 对齐，则报错退出
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.
    我们知道每个块的大小至少为 MINSIZE 字节或 MALLOC_ALIGNMENT 的倍数 */
if (__glibc_unlikely (size < MINSIZE || !aligned_OK (size)))
    malloc_printerr ("free(): invalid size");
```

2. 如果块大小在 tcache 链表中：

```
check_inuse_chunk(av, p);

#if USE_TCACHE
{
    size_t tc_idx = csize2tidx (size); // 转化为 tcache index
    if (tcache != NULL && tc_idx < mp_.tcache_bins)
    {
    /* Check to see if it's already in the tcache.  */
    tcache_entry *e = (tcache_entry *) chunk2mem (p);

    /* This test succeeds on double free.  However, we don't 100%
    trust it (it also matches random payload data at a 1 in
    2^<size_t> chance), so verify it's not an unlikely
    coincidence before aborting.  */
    if (__glibc_unlikely (e->key == tcache)) //key 是在 put 到 tcache 中置位，检测 double free
    {
        tcache_entry *tmp;
        size_t cnt = 0;
        LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
        for (tmp = tcache->entries[tc_idx];
        tmp;
        tmp = REVEAL_PTR (tmp->next), ++cnt)
        {
        if (cnt >= mp_.tcache_count)
        malloc_printerr ("free(): too many chunks detected in tcache");
        if (__glibc_unlikely (!aligned_OK (tmp)))
        malloc_printerr ("free(): unaligned chunk detected in tcache 2");
        if (tmp == e)
        malloc_printerr ("free(): double free detected in tcache 2");
        /* If we get here, it was a coincidence.  We've wasted a
        few cycles, but don't abort.  */
        }
    }

    if (tcache->counts[tc_idx] < mp_.tcache_count) // 如果对应链表没有满则将其放入
    {
        tcache_put (p, tc_idx);
        return;
    }
    }
}
#endif
```

3. 如果块大小在 fastbin 链表中：

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

if ((unsigned long)(size) <= (unsigned long)(get_max_fast ()))

#if TRIM_FASTBINS
    /*
    If TRIM_FASTBINS set, don't place chunks
    bordering top into fastbins
    */
    // 如果 TRIM_FASTBINS 设置了，就不能把与 top chunk 相邻的 chunk 放入 fastbin 里
    // 默认 #define TRIM_FASTBINS 0，因此默认情况下下面的语句不会执行
    // 如果当前 chunk 是 fast chunk，并且下一个 chunk 是 top chunk，则不能插入,会尝试合并进 top
    && (chunk_at_offset(p, size) != av->top)
#endif
    ) {
        // 下一个chunk的大小不能小于两倍的SIZE_SZ,并且
        // 下一个chunk的大小不能大于system_mem， 一般为132k
```

```
    // 如果出现这样的情况，就报错。
if (__builtin_expect (chunksize_nomask (chunk_at_offset (p, size))
        <= CHUNK_HDR_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)) <= CHUNK_HDR_SZ
        || chunksize (chunk_at_offset (p, size)) >= av->system_mem);
    __libc_lock_unlock (av->mutex);
}

if (fail)
malloc_printerr ("free(): invalid next size (fast)");
}
// 将 chunk 的 mem 部分全部设置为 perturb_byte，默认为 0 ，注意减去的 CHUNK_HDR_SZ = 2 * siez_t
free_perturb (chunk2mem(p), size - CHUNK_HDR_SZ);

atomic_store_relaxed (&av->have_fastchunks, true); // 置位
unsigned int idx = fastbin_index(size);
// 获取对应 fastbin 的头指针
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).
    检查不要添加 bin 头，避免 double free
    */
    if (__builtin_expect (old == p, 0))
    malloc_printerr ("double free or corruption (fasttop)");
    // 将 p 插入 fastbin
    p->fd = PROTECT_PTR (&p->fd, old);
    *fb = p;
}
else
    do
    {
        /* Check that the top of the bin is not the record we are going to
            add (i.e., double free).  */
        // 判断fastbin的头chunk是否就是我们将要free的chunk，以防止正常用户的double free(hacker另说)
        if (__builtin_expect (old == p, 0))
            malloc_printerr ("double free or corruption (fasttop)");
        old2 = old;
        p->fd = PROTECT_PTR (&p->fd, old);
    }
        while ((old = catomic_compare_and_exchange_val_rel (fb, p, old2))
            != old2);

    /* Check that size of fastbin chunk at the top is the same as
    size of the chunk that we are adding.  We can dereference OLD
    only if we have the lock, otherwise it might have already been
    allocated again.  */
     // 如果实际放入的fastbin index与预期的不同，则error
    if (have_lock && old != NULL
    && __builtin_expect (fastbin_index (chunksize (old)) != idx, 0))
    malloc_printerr ("invalid fastbin entry (free)");
}
```

4. 当前 chunk 不在 fastbin 中，也不是通过 mmap 分配得来的：

    - 参考文章--linux 堆溢出学习之malloc堆管理机制原理详解

```
/*
    Consolidate other non-mmapped chunks as they arrive.
*/

else if (!chunk_is_mmapped(p)) {

    /* If we're single-threaded, don't lock the arena.  */
if (SINGLE_THREAD_P)
  have_lock = true;

if (!have_lock)
  __libc_lock_lock (av->mutex);

nextchunk = chunk_at_offset(p, size);

/* Lightweight tests: check whether the block is already the
   top block.  */
// 判断当前chunk是否top chunk
if (__glibc_unlikely (p == av->top))
  malloc_printerr ("double free or corruption (top)");
/* Or whether the next chunk is beyond the boundaries of the arena.  */
// 或者下一个块是否超出了 arena 的边界。
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.  */
//或者块是否实际上没有被标记为使用。
if (__glibc_unlikely (!prev_inuse(nextchunk)))
  malloc_printerr ("double free or corruption (!prev)");

// 判断下一块 chunksize 是否正常
nextsize = chunksize(nextchunk);
if (__builtin_expect (chunksize_nomask (nextchunk) <= CHUNK_HDR_SZ, 0)
|| __builtin_expect (nextsize >= av->system_mem, 0))
  malloc_printerr ("free(): invalid next size (normal)");

free_perturb (chunk2mem(p), size - CHUNK_HDR_SZ);
```

```
/* consolidate backward  向后合并*/
if (!prev_inuse(p)) {
  prevsize = prev_size (p);
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  if (__glibc_unlikely (chunksize(p) != prevsize))
    malloc_printerr ("corrupted size vs. prev_size while consolidating");
  unlink_chunk (av, p);
}

if (nextchunk != av->top) {
  /* get and clear inuse bit */
  nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

  /* consolidate forward 向前合并*/
  if (!nextinuse) {
unlink_chunk (av, nextchunk);
size += nextsize;
  } else
clear_inuse_bit_at_offset(nextchunk, 0); //清空使用标志位

  /*
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.
  */
//对fastbin中的chunk进行合并并添加到unsortedbin中
  bck = unsorted_chunks(av);
  fwd = bck->fd;
  if (__glibc_unlikely (fwd->bk != bck))
malloc_printerr ("free(): corrupted unsorted chunks");
  p->fd = fwd;
  p->bk = bck;
  if (!in_smallbin_range(size))
{
  p->fd_nextsize = NULL;
  p->bk_nextsize = NULL;
}
  bck->fd = p;
  fwd->bk = p;

  set_head(p, size | PREV_INUSE);
  set_foot(p, size);

  check_free_chunk(av, p);
}

/*
  If the chunk borders the current high end of memory,
  consolidate into top
*/

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

/*
  If freeing a large space, consolidate possibly-surrounding
  chunks. Then, if the total unused topmost memory exceeds trim
  threshold, ask malloc_trim to reduce top.

  Unless max_fast is 0, we don't know if there are fastbins
  bordering top, so we cannot tell for sure whether threshold
  has been reached unless fastbins are consolidated.  But we
  don't want to consolidate on each free.  As a compromise,
  consolidation is performed if FASTBIN_CONSOLIDATION_THRESHOLD
  is reached.

  如果腾出大空间，巩固可能的周围大块。
  然后，如果总未使用的最高内存超过trim阈值，要求 malloc_trim 减少顶部。

  除非max_fast为0，否则我们不知道是否有fastbins与顶部接壤，因此我们无法确定阈值是否除非合并 fastbins，否则已经达到。
  但我们不想在每个 free bin 上整合。作为妥协，如果 FASTBIN_CONSOLIDATION_THRESHOLD，则执行合并到达了。
*/
//如果是主分配区，并且主分配区的top chunk大于一定的值，就通过systrim缩小top chunk。如果是非主分配区，就获得top chunk对应的非主分配区的heap_info指针，调用heap_trim尝试缩小该heap。后面来看systrim和heap_trim这两个函数。
if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
  if (atomic_load_relaxed (&av->have_fastchunks))
malloc_consolidate(av);

  if (av == &main_arena) {
#ifndef MORECORE_CANNOT_TRIM
    if ((unsigned long)(chunksize(av->top)) >=
        (unsigned long)(mp_.trim_threshold))
    systrim(mp_.top_pad, av);
#endif
    } else {
    /* Always try heap_trim(), even if the top chunk is not
    large, because the corresponding heap might go away.  */
    heap_info *heap = heap_for_ptr(top(av));

    assert(heap->ar_ptr == av);
    heap_trim(heap, mp_.top_pad);
    }
    }

    if (!have_lock)
    __libc_lock_unlock (av->mutex);
}
/*
    If the chunk was allocated via mmap, release via munmap().
*/
else {
    munmap_chunk (p);
}
}
```

__libc_free

chunk_is_mmapped (p)

True
p 为 mmap 申请的内存

判断是否需要更新
mmap 阈值？

True
更新

mp_.mmap_threshold = chunksize (p);
mp_.trim_threshold = 2 * mp_.mmap_threshold;

False
使用 _int_free 进行释放

ar_ptr = arena_for_chunk (p);
_int_free (ar_ptr, p, 0);

False

munmap_chunk (p);

void
_int_free(mstate av, mchunkptr p, int have_lock)

size = chunksize (p);

tcache != NULL &&
tc size < tc max?

True

tcache->counts[tc_idx] <
mp_.tcache_count

True

tcache_put (p, tc_idx);
return;

False

False

size <= max fast&&
next chunk != top?

True

free_perturb 清空内存空间数据 注意避开 chunk 头指针(size - CHUNK_HDR_SZ);
atomic_store_relaxed (&av->have_fastchunks, true);

unsigned int idx = fastbin_index(size);
fb = &fastbin (av, idx);
mchunkptr old = *fb, old2;

!chunk_is_mmapped(p)

True

nextchunk = chunk_at_offset(p, size);
free_perturb (chunk2mem(p), size - CHUNK_HDR_SZ);

SINGLE_THREAD_P

True

// 将 p 插入 fastbin
p->fd = PROTECT_PTR (&p->fd, old);
*fb = p;

False

False

munmap_chunk (p);

通过 while 循环判断fastbin的头chunk是否就是我们将要free的chunk，以防止正常用户的double free

old2 = old;
p->fd = PROTECT_PTR (&p->fd, old);

!prev_inuse(p)

True
前一块 chunk 也是空闲的
向后合并

prevsize = prev_size (p);
size += prevsize;
p = chunk_at_offset(p, -((long) prevsize));

False

unlink_chunk (av, p);

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

False
插入 top chunk 中

nextchunk != av->top

True
next chunk 非 top chunk

nextinuse =
inuse_bit_at_offset(nextchunk,
nextsize);
获取 next chunk 是否在使用

!nextinuse

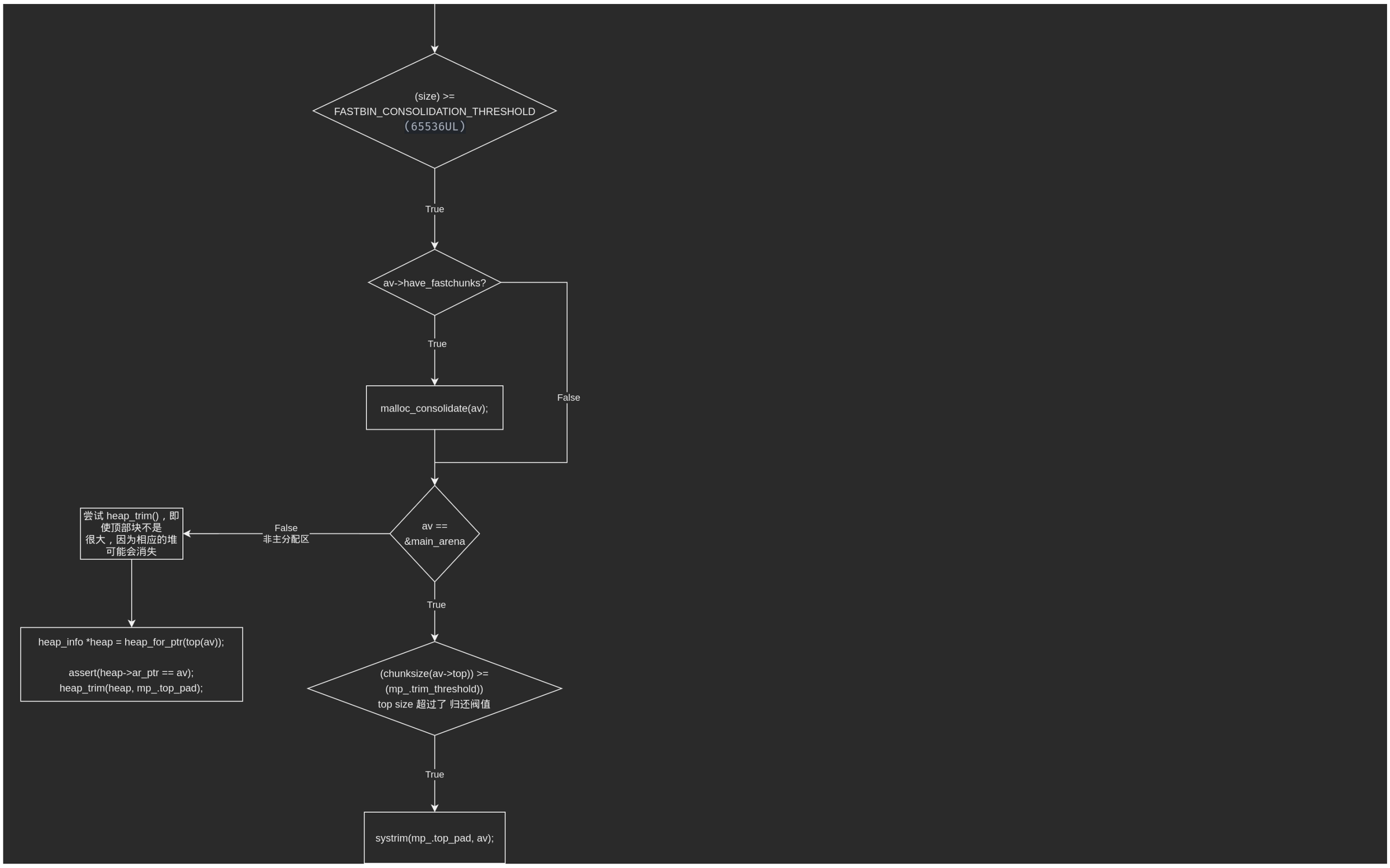
True

unlink_chunk (av, nextchunk);
size += nextsize;
合并

False

clear_inuse_bit_at_offset(nextchunk, 0);
清楚当前 chunk 的 inuse 标志

bck = unsorted_chunks(av);
将 p 插入到 unsorted bin 中


(size) >=
FASTBIN_CONSOLIDATION_THRESHOLD
(65536UL)
True
av->have_fastchunks?
True
False
malloc_consolidate(av);
av ==
&main_arena
False
非主分配区
尝试 heap_trim()，即
使顶部块不是
很大，因为相应的堆
可能会消失
True
heap_info *heap = heap_for_ptr(top(av));
assert(heap->ar_ptr == av);
heap_trim(heap, mp_.top_pad);
(chunksize(av->top)) >=
(mp_.trim_threshold))
top size 超过了 归还阈值
True
systrim(mp_.top_pad, av);