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

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

  /* Quickly check that the freed pointer matches the tag for the memory.
     This gives a useful double-free detection.  */
  if (__glibc_unlikely (mtag_enabled))
    *(volatile char *)mem;

  int err = errno;

  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)
        {
          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);
    }
  else
    {
      MAYBE_INIT_TCACHE ();

      /* Mark the chunk as belonging to the library again.  */
      (void)tag_region (chunk2mem (p), memsize (p));

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

  __set_errno (err);
}
```

这里没什么具体的逻辑,也就处理一下空指针,获取一下arena,初始化一下tcache.其他的逻辑都在_int_free里.

```
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);
```

检查大小和16字节对齐,以及next_chunk的inuse位(这也是为什么不会有unsorted_bin double_free)的原因

```
#if USE_TCACHE
  {
    size_t tc_idx = csize2tidx (size);
    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))
      {
        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)
      {
        tcache_put (p, tc_idx);
        return;
      }
      }
  }
#endif
```

尝试将chunk放入tcache,检查key.如果发现疑似double_free则会对对应的bin进行遍历.

TODO:查看tcache_put

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

#if TRIM_FASTBINS
      /*
    If TRIM_FASTBINS set, don't place chunks
    bordering top into fastbins
      */
      && (chunk_at_offset(p, size) != av->top)
#endif
      ) {

    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)");
      }

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

    atomic_store_relaxed (&av->have_fastchunks, true);
    unsigned int idx = fastbin_index(size);
```

```
    fb = &fastbin (av, idx);

    /* Atomically link P to its fastbin: P->FD = *FB; *FB = P;  */
    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 = 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).  */
      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.  */
    if (have_lock && old != NULL
    && __builtin_expect (fastbin_index (chunksize (old)) != idx, 0))
      malloc_printerr ("invalid fastbin entry (free)");
  }
```

将chunk放入fast_bin,其中我们耳熟能详的double_free检查在这里:

```
    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 = PROTECT_PTR (&p->fd, old);
    *fb = p;
      }
```

```
/*
    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.  */
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.  */
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.  */
if (__glibc_unlikely (!prev_inuse(nextchunk)))
  malloc_printerr ("double free or corruption (!prev)");

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 */
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 */
  if (!nextinuse) {
unlink_chunk (av, nextchunk);
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;
      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.
    */

    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);
  }
}
```

这段则是放入unsorted_bin,以及耳熟能详的触发unlink部分.