malloc

在用户调用malloc后,一般会进入`__libc_malloc()`函数内.

首先,函数定义了一个用来返回分配的内存的指针和一个arena指针.

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

用来防止稀奇古怪的平台导致的问题的断言,以及初始化(如果还没有初始化)

```
  _Static_assert (PTRDIFF_MAX <= SIZE_MAX / 2,
                  "PTRDIFF_MAX is not more than half of SIZE_MAX");

  if (!__malloc_initialized)
    ptmalloc_init ();
```

如果启用了tcache,则会根据申请的大小在tcache中检查对应bin的下标,如果非空则从tcache中取出chunk

```
#if USE_TCACHE
  /* int_free also calls request2size, be careful to not pad twice.  */
  size_t tbytes;
  if (!checked_request2size (bytes, &tbytes))
    {
      __set_errno (ENOMEM);
      return NULL;
    }
  size_t tc_idx = csize2tidx (tbytes);

  MAYBE_INIT_TCACHE ();

  DIAG_PUSH_NEEDS_COMMENT;
  if (tc_idx < mp_.tcache_bins
      && tcache
      && tcache->counts[tc_idx] > 0)
    {
      victim = tcache_get (tc_idx);
      return tag_new_usable (victim);
    }
  DIAG_POP_NEEDS_COMMENT;
#endif
```

一些处理多线程的东西,另外据ai说断言(assert)对于 NODEBUG版本是不生效的.

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

  arena_get (ar_ptr, bytes);

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

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

  victim = tag_new_usable (victim);

  assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
          ar_ptr == arena_for_chunk (mem2chunk (victim)));
  return victim;
```

可以看到,除了tcache以外的东西,都会在_int_malloc处理.

在int_malloc的开头,首先是将请求的堆块尺寸规范化,然后是一些特殊情况处理:

```

static void *
_int_malloc (mstate av, size_t bytes)
{
  INTERNAL_SIZE_T nb;               /* normalized request size */
  unsigned int idx;                 /* associated bin index */
  mbinptr bin;                      /* associated bin */

  mchunkptr victim;                 /* inspected/selected chunk */
  INTERNAL_SIZE_T size;             /* its size */
  int victim_index;                 /* its bin index */

  mchunkptr remainder;              /* remainder from a split */
  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 */

  mchunkptr fwd;                    /* misc temp for linking */
  mchunkptr bck;                    /* misc temp for linking */

#if USE_TCACHE
  size_t tcache_unsorted_count;     /* count of unsorted chunks processed */
#endif

  /*
```

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

  if (!checked_request2size (bytes, &nb))
    {
      __set_errno (ENOMEM);
      return NULL;
    }

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

然后,程序首先会处理fast_bin的部分.程序会根据大小将对应的fast_bin的指针赋给fb,如果fb不为空,则会进入从fast_bin分配的流程.

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

      if (victim != NULL)
    {
      if (__glibc_unlikely (misaligned_chunk (victim)))
        malloc_printerr ("malloc(): unaligned fastbin chunk detected 2");

      if (SINGLE_THREAD_P)
        *fb = REVEAL_PTR (victim->fd);
      else
        REMOVE_FB (fb, pp, victim);
      if (__glibc_likely (victim != NULL))
        {
          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
          /* While we're here, if we see other chunks of the same size,
         stash them in the tcache.  */
          size_t tc_idx = csize2tidx (nb);
          if (tcache && tc_idx < mp_.tcache_bins)
        {
          mchunkptr tc_victim;

          /* While bin not empty and tcache not full, copy chunks.  */
          while (tcache->counts[tc_idx] < mp_.tcache_count
             && (tc_victim = *fb) != NULL)
            {
              if (__glibc_unlikely (misaligned_chunk (tc_victim)))
            malloc_printerr ("malloc(): unaligned fastbin chunk detected 3");
              if (SINGLE_THREAD_P)
            *fb = REVEAL_PTR (tc_victim->fd);
              else
            {
              REMOVE_FB (fb, pp, tc_victim);
              if (__glibc_unlikely (tc_victim == NULL))
                break;
            }
              tcache_put (tc_victim, tc_idx);
            }
        }
#endif
          void *p = chunk2mem (victim);
          alloc_perturb (p, bytes);
          return p;
        }
    }
    }
```

大概还是找到对应的fast_bin,然后尝试从里面取出chunk的过程.主要的检查是fast_bin本身的地址对齐,以及chunk的size的合法性检查(这也是为什么fast_bin_attack 需要伪造size)

值得注意的是,从libc2.32开始,还引入了对fd指针对齐的检查,用fast_bin_attack打free_hook就没有那么方便了.

在执行完这些操作后,程序会尝试将fast_bin中的chunk搬到tcache中.然后返回获取的chunk指针.

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

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

          if (av != &main_arena)
        set_non_main_arena (victim);
          check_malloced_chunk (av, victim, nb);
#if USE_TCACHE
      /* While we're here, if we see other chunks of the same size,
	   stash them in the tcache.  */
	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
            void *p = chunk2mem (victim);
            alloc_perturb (p, bytes);
            return p;
          }
      }
```

这段是从small_bin中取出chunk,并将同一个bin中的其他chunk移动到tcache.仅有的一个检查是检查该chunk的前驱的后继是不是chunk自身.

```
  /*
     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);
    }
```

如果tcache,small_bin都无法满足分配的需求,程序就会尝试从unsorted_bin中获取堆块.首先,程序会将fast_bin中的堆块转移到unsorted_bin中并尝试合并(和free情况下堆块进入unsorted_bin时发生的合并和unlink类似)

```
#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
```

这一段在计算当前请求的大小对应的tcache索引,当之后从unsorted_bin中取出堆块时,会将其他大小符合的堆块放入tcache中.

最后则是一个超大的for循环,用于处理unsorted_bin和large_bin中的堆块.如果还是不行,则从top_chunk中分隔.:

```
  for (;; )
    {
      int iters = 0;
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
        {
          bck = victim->bk;
          size = chunksize (victim);
          mchunkptr next = chunk_at_offset (victim, size);

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

这是一些安全检查

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

              set_head (victim, nb | PREV_INUSE |
                        (av != &main_arena ? NON_MAIN_ARENA : 0));
              set_head (remainder, remainder_size | PREV_INUSE);
              set_foot (remainder, remainder_size);

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

一些优化

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

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

          if (size == nb)
            {
              set_inuse_bit_at_offset (victim, size);
              if (av != &main_arena)
        set_non_main_arena (victim);
#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
              check_malloced_chunk (av, victim, nb);
              void *p = chunk2mem (victim);
              alloc_perturb (p, bytes);
              return p;
#if USE_TCACHE
        }
```

```
#endif
            }

          /* place chunk in bin */

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

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

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

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

```
                          fwd->bk_nextsize = victim;
                          victim->bk_nextsize->fd_nextsize = victim;
                        }
                      bck = fwd->bk;
                      if (bck->fd != fwd)
                        malloc_printerr ("malloc(): largebin double linked list
corrupted (bk)");
                    }
                }
              else
                victim->fd_nextsize = victim->bk_nextsize = victim;
            }

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


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

遍历unsorted_bin,如果有大小符合要求的chunk且能放入tcache,则将其放入tcache并在完成遍历后返回指针;如果有大小符合要求的chunk且不能放入tcache,则直接返回该chunk的指针.

否则,将这些chunk按照大小放入small_bin或large_bin.

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

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

      /*
         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.
       */
```

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

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

        /* Avoid removing the first entry for a size so that the skip
           list does not have to be rerouted.  */
        if (victim != last (bin)
    && chunksize_nomask (victim)
      == chunksize_nomask (victim->fd))
          victim = victim->fd;

        remainder_size = size - nb;
        unlink_chunk (av, victim);

        /* Exhaust */
        if (remainder_size < MINSIZE)
          {
            set_inuse_bit_at_offset (victim, size);
            if (av != &main_arena)
      set_non_main_arena (victim);
          }
        /* Split */
        else
          {
            remainder = chunk_at_offset (victim, nb);
            /* We cannot assume the unsorted list is empty and therefore
               have to perform a complete insert here.  */
            bck = unsorted_chunks (av);
            fwd = bck->fd;
    if (__glibc_unlikely (fwd->bk != bck))
      malloc_printerr ("malloc(): corrupted unsorted chunks");
            remainder->bk = bck;
            remainder->fd = fwd;
            bck->fd = remainder;
            fwd->bk = remainder;
            if (!in_smallbin_range (remainder_size))
              {
                remainder->fd_nextsize = NULL;
                remainder->bk_nextsize = NULL;
              }
            set_head (victim, nb | PREV_INUSE |
                      (av != &main_arena ? NON_MAIN_ARENA : 0));
            set_head (remainder, remainder_size | PREV_INUSE);
            set_foot (remainder, remainder_size);
          }
```

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

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

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

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

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

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

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

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

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

```
                bit <<= 1;
              }

            else
              {
                size = chunksize (victim);

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

                remainder_size = size - nb;

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

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

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

                    /* We cannot assume the unsorted list is empty and therefore
                       have to perform a complete insert here.   */
                    bck = unsorted_chunks (av);
                    fwd = bck->fd;
            if (__glibc_unlikely (fwd->bk != bck))
              malloc_printerr ("malloc(): corrupted unsorted chunks 2");
                    remainder->bk = bck;
                    remainder->fd = fwd;
                    bck->fd = remainder;
                    fwd->bk = remainder;

                    /* advertise as last remainder */
                    if (in_smallbin_range (nb))
                      av->last_remainder = remainder;
                    if (!in_smallbin_range (remainder_size))
                      {
                        remainder->fd_nextsize = NULL;
                        remainder->bk_nextsize = NULL;
                      }
                    set_head (victim, nb | PREV_INUSE |
                              (av != &main_arena ? NON_MAIN_ARENA : 0));
                    set_head (remainder, remainder_size | PREV_INUSE);
                    set_foot (remainder, remainder_size);
                  }
                check_malloced_chunk (av, victim, nb);
                void *p = chunk2mem (victim);
                alloc_perturb (p, bytes);
                return p;
```

```
          }
      }
```

如果分配的chunk不属于small_bin则继续在large_bin中寻找chunk

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

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

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

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

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

          check_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.  */
      else if (atomic_load_relaxed (&av->have_fastchunks))
        {
          malloc_consolidate (av);
          /* restore original bin index */
          if (in_smallbin_range (nb))
            idx = smallbin_index (nb);
          else
            idx = largebin_index (nb);
        }
      /*
         Otherwise, relay to handle system-dependent cases
       */
      else
        {
          void *p = sysmalloc (nb, av);
          if (p != NULL)
            alloc_perturb (p, bytes);
          return p;
        }
    }
}
```

如果large_bin中还是没有找到合适的chunk,从top_chunk分隔出内存.

主要的检查是double_free(通过下一个chunk的pre_inuse检测)和地址是否落在arena边界内