源码来自：https://elixir.bootlin.com/glibc/glibc-2.23/source/malloc/malloc.c

MMAP support

```
/* ------------------ MMAP support ------------------  */


#include <fcntl.h>
#include <sys/mman.h>

#if !defined(MAP_ANONYMOUS) && defined(MAP_ANON)
# define MAP_ANONYMOUS MAP_ANON
#endif

#ifndef MAP_NORESERVE
# define MAP_NORESERVE 0
#endif

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

Chunk representations

malloc_chunk 结构体

```
/*
  -----------------------  Chunk representations -----------------------
*/


/*
  This struct declaration is misleading (but accurate and necessary).
  It declares a "view" into memory allowing access to necessary
  fields at known offsets from a given base. See explanation below.
*/

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;
};
```

之前可以看到 `#define INTERNAL_SIZE_T size_t` 也就是说在 64 位机器上这个类型就是 `unsigned long` 类型

prev_size：如果前一个块处于空闲状态，那么该值就是前一个块的大小

size：用来记录当前块的大小

fd：记录前驱节点

bk：记录后记节点

fd_nextsize：记录 large bin 的前驱节点

bk_nextsize：记录 large bin 的后继节点

malloc_chunk 的细节

```
/*
   malloc_chunk details:

    (The following includes lightly edited explanations by Colin Plumb.)

    Chunks of memory are maintained using a `boundary tag' method as
    described in e.g., Knuth or Standish.  (See the paper by Paul
    Wilson ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps for a
    survey of such techniques.)  Sizes of free chunks are stored both
    in the front of each chunk and at the end.  This makes
    consolidating fragmented chunks into bigger chunks very fast.  The
    size fields also hold bits representing whether chunks are free or
    in use.

    An allocated chunk looks like this:


    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                                     |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+


    Where "chunk" is the front of the chunk for the purpose of most of
    the malloc code, but "mem" is the pointer that is returned to the
```

```
    user.  "Nextchunk" is the beginning of the next contiguous chunk.

    Chunks always begin on even word boundaries, so the mem portion
    (which is returned to the user) is also on an even word boundary, and
    thus at least double-word aligned.

    Free chunks are stored in circular doubly-linked lists, and look like this:

    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                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    The P (PREV_INUSE) bit, stored in the unused low-order bit of the
    chunk size (which is always a multiple of two words), is an in-use
    bit for the *previous* chunk.  If that bit is *clear*, then the
    word before the current chunk size contains the previous chunk
    size, and can be used to find the front of the previous chunk.
    The very first chunk allocated always has this bit set,
    preventing access to non-existent (or non-owned) memory. If
    prev_inuse is set for any given chunk, then you CANNOT determine
    the size of the previous chunk, and might even get a memory
    addressing fault when trying to do so.

    Note that the `foot' of the current chunk is actually represented
    as the prev_size of the NEXT chunk. This makes it easier to
    deal with alignments etc but can be very confusing when trying
    to extend or adapt this code.

    The two exceptions to all this are

```

```
      1. The special chunk `top' doesn't bother using the
         trailing size field since there is no next contiguous chunk
         that would have to index off it. After initialization, `top'
         is forced to always exist.  If it would become less than
         MINSIZE bytes long, it is replenished.

      2. Chunks allocated via mmap, which have the second-lowest-order
         bit M (IS_MMAPPED) set in their size fields.  Because they are
         allocated one-by-one, each must contain its own trailing size
field.

*/
```

Size and alignment checks and conversions

chunk2mem(p) 宏

```
/* conversion from malloc headers to user pointers, and back */

#define chunk2mem(p)   ((void*)((char*)(p) + 2*SIZE_SZ))
```

该宏的作用是找到堆块 p 内用来存储 fd 指针的地址

说白了就是 p 其实就是用来存储当前堆块 prev_size 的地址，但是我们不需要用来存储当前堆块 prev_size 和 size 的地址

因为用户输入的内容都是存储到**那个能够存储 fd 指针的地址**，也就是存储 size 的地址的下一个地址

fd 和 bk 都是在堆块空闲的时候才会存储在这个地址上，当堆块正在被使用的时候这里就是正常的存储区域

mem2chunk(mem) 宏

```
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

该宏的作用和 **chunk2mem 宏**是反过来的

由堆块内用于给用户输入的存储区地址找到堆块的起始地址，也就是用于存储当前堆块 prev_size 的地址

MIN_CHUNK_SIZE 宏

```
/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))
```

首先要了解 **offsetof 宏**的定义

```
# define offsetof(type,ident) ((size_t)&(((type*)0)->ident))
```

也就是通过一个结构体的元素获得该结构体的起始地址到该元素的距离

这个宏的作用是来规定一整个 chunk 的最小值是多少，包括 prev_size 域和 size 域

由此可以了解，在 32 位的系统下，MIN_CHUNK_SIZE 的值是 0x10

在 64 位的系统下，MIN_CHUNK_SIZE 的值是 0x20

MINSIZE 宏

```
/* The smallest size we can malloc is an aligned minimal chunk */

#define MINSIZE  \
  (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK))
```

用来规定最小的堆块可用空间，也就是说申请的堆块至少会有 MINSIZE 的大小

在 32 位下，MINSIZE 的值是 0x10 字节

在 64 位下，MINSIZE 的值是 0x20 字节

aligned_OK(m) 宏

```
/* Check if m has acceptable alignment */

#define aligned_OK(m)  (((unsigned long)(m) & MALLOC_ALIGN_MASK) == 0)
```

用来判断申请到的堆块中的地址是否为对齐的地址

misaligned_chunk(p) 宏

```
#define misaligned_chunk(p) \
  ((uintptr_t)(MALLOC_ALIGNMENT == 2 * SIZE_SZ ? (p) : chunk2mem (p)) \
   & MALLOC_ALIGN_MASK)
```

如果 `MALLOC_ALIGNMENT == 2 * SIZE_SZ`

即如果 `long double` 对齐所需的字节数大于 `2 * sizeof(size_t)`

那么就返回 p 的地址，也就是堆块的起始地址；否则就返回该堆块 fd 指针所处的地址

一般的架构都是返回 `chunk2mem (p)` 的

REQUEST_OUT_OF_RANGE(req) 宏（缺）

```
/*
   Check if a request is so large that it would wrap around zero when
   padded and aligned. To simplify some other code, the bound is made
   low enough so that adding MINSIZE will also not wrap around zero.
 */

#define REQUEST_OUT_OF_RANGE(req)                                \
  ((unsigned long) (req) >=
   \
   (unsigned long) (INTERNAL_SIZE_T) (-2 * MINSIZE))
```

未补充

request2size(req) 宏（缺）

```
/* pad request bytes into a usable size -- internal version */

#define request2size(req)                                         \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)
```

未补充

checked_request2size(req, sz) 宏（缺）

```
/*  Same, except also perform argument check */

#define checked_request2size(req, sz)                             \
  if (REQUEST_OUT_OF_RANGE (req)) {                               \
      __set_errno (ENOMEM);                                       \
      return 0;                                                   \
    }                                                             \
  (sz) = request2size (req);
```

未补充

Physical chunk operations

PREV_INUSE 宏

```
/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use */
#define PREV_INUSE 0x1
```

该宏的意思是当前堆块的前一个堆块处于非空闲状态，规定值为 0x1

prev_inuse(p) 宏

```
/* extract inuse bit of previous chunk */
#define prev_inuse(p)       ((p)->size & PREV_INUSE)
```

检查前一个堆块是否处于非空闲状态

如果前一个堆块处于非空闲状态，那么返回 0x1；否则返回 0

IS_MMAPPED 宏

```
/* size field is or'ed with IS_MMAPPED if the chunk was obtained with mmap() */
#define IS_MMAPPED 0x2
```

该宏的意思是当前的堆块是通过 mmap() 得到的

chunk_is_mmapped(p) 宏

```
/* check for mmap()'ed chunk */
#define chunk_is_mmapped(p) ((p)->size & IS_MMAPPED)
```

检查当前堆块是否是由 mmap() 得到的

如果是由 mmap() 得到的，那么返回 0x2；否则返回 0

NON_MAIN_ARENA 宏

```
/* size field is or'ed with NON_MAIN_ARENA if the chunk was obtained
   from a non-main arena.  This is only set immediately before handing
   the chunk to the user, if necessary.  */
#define NON_MAIN_ARENA 0x4
```

表示当前 chunk 不属于主线程

chunk_non_main_arena(p) 宏

```
/* check for chunk from non-main arena */
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

检查当前堆块是否属于主线程

如果不属于主线程，那么返回 0x4；否则返回 0

SIZE_BITS 宏

```
/*
   Bits to mask off when extracting size

   Note: IS_MMAPPED is intentionally not masked off from size field in
   macros for which mmapped chunks should never be seen. This should
   cause helpful core dumps to occur if it is tried by accident by
   people extending or adapting this malloc.
 */
#define SIZE_BITS (PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)
```

表面看这个宏的返回值就是 7，也就是 `b0111`，作用在下面的宏中有体现

chunksize(p) 宏

```
/* Get size, ignoring use bits */
#define chunksize(p)         ((p)->size & ~(SIZE_BITS))
```

得到堆块 p 中的 size 位的值，因为堆块是对齐的，所以后三位没有用而且也不算是大小

算了后三位就破坏对齐机制了，所以这里要把后三位给清除掉

next_chunk(p) 宏

```
/* Ptr to next physical malloc_chunk. */
#define next_chunk(p) ((mchunkptr) (((char *) (p)) + ((p)->size &
~SIZE_BITS)))
```

mchunkptr 结构体指针变量的定义：`typedef struct malloc_chunk* mchunkptr;`

这个宏的作用就是得到当前堆块的下一个堆块的地址

看代码意思就是用当前宏的地址加上该宏的大小，那么得到的值就是下一个堆块的地址了

prev_chunk(p) 宏

```
/* Ptr to previous physical malloc_chunk */
#define prev_chunk(p) ((mchunkptr) (((char *) (p)) - ((p)->prev_size)))
```

得到当前堆块的前一个堆块地址

代码意思就是用当前堆块的地址减去前一个堆块的大小，就可以得到前一个堆块的地址

不过 prev_size 位只有在前一个堆块处于空闲状态时才会有值

chunk_at_offset(p, s) 宏

```
/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr) (((char *) (p)) + (s)))
```

也是获得一个堆块的地址，不过这种获得方式是指定偏移大小的

inuse(p) 宏

```
/* extract p's inuse bit */
#define inuse(p)
\
  ((((mchunkptr) (((char *) (p)) + ((p)->size & ~SIZE_BITS)))->size) & PREV_INUSE)
```

获取下一个堆块的 PREV_INUSE 位，也就是说这个宏是用来判断当前堆块是否处于空闲状态的

若是处于空闲状态就返回 1；否则返回 0

set_inuse(p) 宏

```
/* set/clear chunk as being inuse without otherwise disturbing */
#define set_inuse(p)
     \
  ((mchunkptr) (((char *) (p)) + ((p)->size & ~SIZE_BITS)))->size |= PREV_INUSE
```

这个宏的作用就是通过当前堆块的大小及地址得到下一个堆块的地址

然后将下一个堆块的 PREV_INUSE 位设置为 1

clear_inuse(p) 宏

```
#define clear_inuse(p)
       \
  ((mchunkptr) (((char *) (p)) + ((p)->size & ~SIZE_BITS)))->size &= ~(PREV_INUSE)
```

该函数的作用是清除掉 PREV_INUSE 位，`~(PREV_INUSE)` 的值是 -2

inuse_bit_at_offset(p, s) 宏

```
/* check/set/clear inuse bits in known places */
#define inuse_bit_at_offset(p, s) \
  (((mchunkptr) (((char *) (p)) + (s)))->size & PREV_INUSE)
```

类似于 `inuse(p)` 宏，区别是它可以自己指定偏移

set_inuse_bit_at_offset(p, s) 宏

```
#define set_inuse_bit_at_offset(p, s) \
  (((mchunkptr) (((char *) (p)) + (s)))->size |= PREV_INUSE)
```

类似于 `set_inuse(p)` 宏，区别是它可以自己指定偏移

clear_inuse_bit_at_offset(p, s) 宏

```
#define clear_inuse_bit_at_offset(p, s) \
  (((mchunkptr) (((char *) (p)) + (s)))->size &= ~(PREV_INUSE))
```

类似于 `clear_inuse(p)` 宏，区别是它可以自己指定偏移

set_head_size(p, s) 宏

```
/* Set size at head, without disturbing its use bit */
#define set_head_size(p, s)  ((p)->size = (((p)->size & SIZE_BITS) | (s)))
```

在堆块 p 的 size 位设置该堆块的大小，并且不会影响到该堆块的使用位

set_head(p, s) 宏

```
/* Set size/use field */
#define set_head(p, s)       ((p)->size = (s))
```

在堆块 p 的 size 位设置该堆块的大小，该方法能影响到该堆块的使用位

set_foot(p, s) 宏

```
/* Set size at footer (only when chunk is not in use) */
#define set_foot(p, s)       (((mchunkptr) ((char *) (p) + (s)))->prev_size = (s))
```

设置下一个堆块的 prev_size 位，该宏只有在当前堆块为空闲堆块时才会使用

看样子这个宏是专门在下一个堆块的 prev_size 位设置当前堆块的大小的

而且就算该堆块的地址被申请回来了，那么下一个堆块的 prev_size 位也不会改变

Internal data structures

mbinptr 结构体指针变量

```
/*
   -------------------- Internal data structures --------------------

   All internal state is held in an instance of malloc_state defined
   below. There are no other static variables, except in two optional
   cases:
 * If USE_MALLOC_LOCK is defined, the mALLOC_MUTEx declared above.
 * If mmap doesn't support MAP_ANONYMOUS, a dummy file descriptor
     for mmap.

   Beware of lots of tricks that minimize the total bookkeeping space
   requirements. The result is a little over 1K bytes (for 4byte
   pointers and size_t.)
 */

/*
   Bins

    An array of bin headers for free chunks. Each bin is doubly
    linked.  The bins are approximately proportionally (log) spaced.
    There are a lot of these bins (128). This may look excessive, but
    works very well in practice.  Most bins hold sizes that are
    unusual as malloc request sizes, but are more usual for fragments
    and consolidated sets of chunks, which is what these bins hold, so
    they can be found quickly.  All procedures maintain the invariant
    that no consolidated chunk physically borders another one, so each
    chunk in a list is known to be preceeded and followed by either
    inuse chunks or the ends of memory.

    Chunks in bins are kept in size order, with ties going to the
    approximately least recently used chunk. Ordering isn't needed
    for the small bins, which all contain the same-sized chunks, but
    facilitates best-fit allocation for larger chunks. These lists
    are just sequential. Keeping them in order almost never requires
    enough traversal to warrant using fancier ordered data
    structures.

    Chunks of the same size are linked with the most
    recently freed at the front, and allocations are taken from the
    back.  This results in LRU (FIFO) allocation order, which tends
    to give each chunk an equal opportunity to be consolidated with
    adjacent freed chunks, resulting in larger free chunks and less
    fragmentation.

    To simplify use in double-linked lists, each bin header acts
    as a malloc_chunk. This avoids special-casing for headers.
    But to conserve space and improve locality, we allocate
    only the fd/bk pointers of bins, and then use repositioning tricks
    to treat these as the fields of a malloc_chunk*.
 */

typedef struct malloc_chunk *mbinptr;
```

没啥说的，跟 `mchunkptr` 差不多，不过是用在 bin（空闲堆块）里的

bin_at(m, i) 宏

```c
/* 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))
```

获得某种类型的 bins 里某一个 bin 的地址，且该 bins 的基地址的下标是 1，而不能是 0

next_bin(b) 宏（缺具体）

```c
/* analog of ++bin */
#define next_bin(b)  ((mbinptr) ((char *) (b) + (sizeof (mchunkptr) << 1)))
```

获取下一个 bin 的地址

first(b) 宏

```c
/* Reminders about list directionality within bins */
#define first(b)     ((b)->fd)
```

获得 bin 里的 fd 指针

last(b) 宏

```c
#define last(b)      ((b)->bk)
```

获取 bin 里的 bk 指针

unlink(AV, P, BK, FD) 宏（重点）（缺）

```c
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) { \
    FD = P->fd;                                   \
    BK = P->bk;                                   \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))          \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
    else {                                        \
        FD->bk = BK;                                  \
        BK->fd = FD;                                  \
        if (!in_smallbin_range (P->size)                      \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {            \
        if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)        \
        || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
          malloc_printerr (check_action,                      \
                   "corrupted double-linked list (not small)",    \
                   P, AV);                        \
            if (FD->fd_nextsize == NULL) {                    \
                if (P->fd_nextsize == P)                      \
```

```
19                      FD->fd_nextsize = FD->bk_nextsize = FD;             \
20                    else {                                \
21                        FD->fd_nextsize = P->fd_nextsize;                 \
22                        FD->bk_nextsize = P->bk_nextsize;                 \
23                        P->fd_nextsize->bk_nextsize = FD;                 \
24                        P->bk_nextsize->fd_nextsize = FD;                 \
25                      }                                   \
26                  } else {                                \
27                    P->fd_nextsize->bk_nextsize = P->bk_nextsize;             \
28                    P->bk_nextsize->fd_nextsize = P->fd_nextsize;             \
29                  }                                   \
30              }                                   \
31          }                                       \
32  }
```

3 - 4 行：`FD = P->fd;` 和 `BK = P->bk;` 是分别获取传入参数 P 的前驱节点和后继节点

5 行： if 语句用于判断 P 的前驱节点的后继节点是否为 P，P 的后继节点的前驱节点是否为 P，且要通过条件最后返回值应该是 0

6 行：如果返回值是 1，那么就调用 `malloc_printerr (check_action, "corrupted double-linked list", P, AV);`

7 - 9 行：如果返回值是 0，进入 else 语句，并让 **P 的前驱节点的后继节点变成 P 的后继节点**

再让 **P 的后继节点的前驱节点变成 P 的前驱节点**，完成删除双向链表上的 P 节点的操作

10 - 11 行：

Indexing

NBINS 宏

```
/*
   Indexing

    Bins for sizes < 512 bytes contain chunks of all the same size, spaced
    8 bytes apart. Larger bins are approximately logarithmically spaced:

    64 bins of size       8
    32 bins of size      64
    16 bins of size     512
     8 bins of size    4096
     4 bins of size   32768
     2 bins of size  262144
     1 bin  of size what's left

    There is actually a little bit of slop in the numbers in bin_index
    for the sake of speed. This makes no difference elsewhere.

    The bins top out around 1MB because we expect to service large
    requests via mmap.

    Bin 0 does not exist.  Bin 1 is the unordered list; if that would be
    a valid chunk size the small bins are bumped up one.
 */

```

```
#define NBINS             128
```

规定计算正常 bin 大小时的基准值

NSMALLBINS 宏

```
#define NSMALLBINS         64
```

规定计算正常 smallbin 大小时的基准值

SMALLBIN_WIDTH 宏

```
#define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
```

正常情况在 32 位下，这个值是 0x08；在 64 下这个值是 0x10

SMALLBIN_CORRECTION 宏

```
#define SMALLBIN_CORRECTION (MALLOC_ALIGNMENT > 2 * SIZE_SZ)
```

这种就是在非正常情况下会有返回值 1，即在满足 `2 *SIZE_SZ < __alignof__ (long double)` 时

MIN_LARGE_SIZE 宏

```
#define MIN_LARGE_SIZE    ((NSMALLBINS - SMALLBIN_CORRECTION) *
SMALLBIN_WIDTH)
```

用于规定 large bin 大小的最小值

正常情况下 64 位的最小值为 `(64 - 0) * 0x10 == 0x400`；32 位的最小值为 `(64 - 0) * 0x08 == 0x200`

in_smallbin_range(sz) 宏

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

555

```

```

555

```

```

555

```

```