Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心

Glibc内存管理2

博士研究生　魏源

2019年05月20日

源代码分析

- 源码分析，希望能够终极无惑。
- 我们主要从arena.c和malloc.c进行分析
  - 这两个文件包含ptmalloc的核心实现
  - Arena.c支持多线程的实现
  - Malloc.c定义了malloc()和free()
  - Ptmalloc主要支持32位平台和64位平台
    - 只要基于linux 基于 linux86平台

源代码分析-边界标记法

- Ptmalloc对chunk的内存管理，采用独特边界标记法：
  - 每一个chunk最小的size，不同平台，地址方式对齐是不同的。
    - 32位以4字节对齐
    - 64位以可能是4字节也可能是8字节

```
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size */
#define SIZE_SZ                (sizeof(INTERNAL_SIZE_T))

/*
  MALLOC_ALIGNMENT is the minimum alignment for malloc'ed chunks.
  It must be a power of two at least 2 * SIZE_SZ, even on machines
  for which smaller alignments would suffice. It may be defined as
  larger than this though. Note however that code and data structures
  are optimized for the case of 8-byte alignment.
*/

#ifndef MALLOC_ALIGNMENT
#define MALLOC_ALIGNMENT       (2 * SIZE_SZ)
#endif

/* The corresponding bit mask value */
#define MALLOC_ALIGN_MASK      (MALLOC_ALIGNMENT - 1)
```

1. 分配chunk时，必须以2*SIZE_SZ对齐

2. MALLOC_ALIGNMENT 和MALLOC_ALIGN_MASK是用来处理chunk地址对齐的宏。

源代码分析-边界标记法

- fd：指向前一块;Bk：指向后一块
- 这两个指正只有在空闲才存在，加入链表统一管理
- Fd_nextsize:指向下一个比当前chunk大小大的第一个空闲chunk
- Bk_nextsize:指向前一个比当前chunk大小小的第一个空闲chunk

```
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;
```

源代码分析-边界标记法

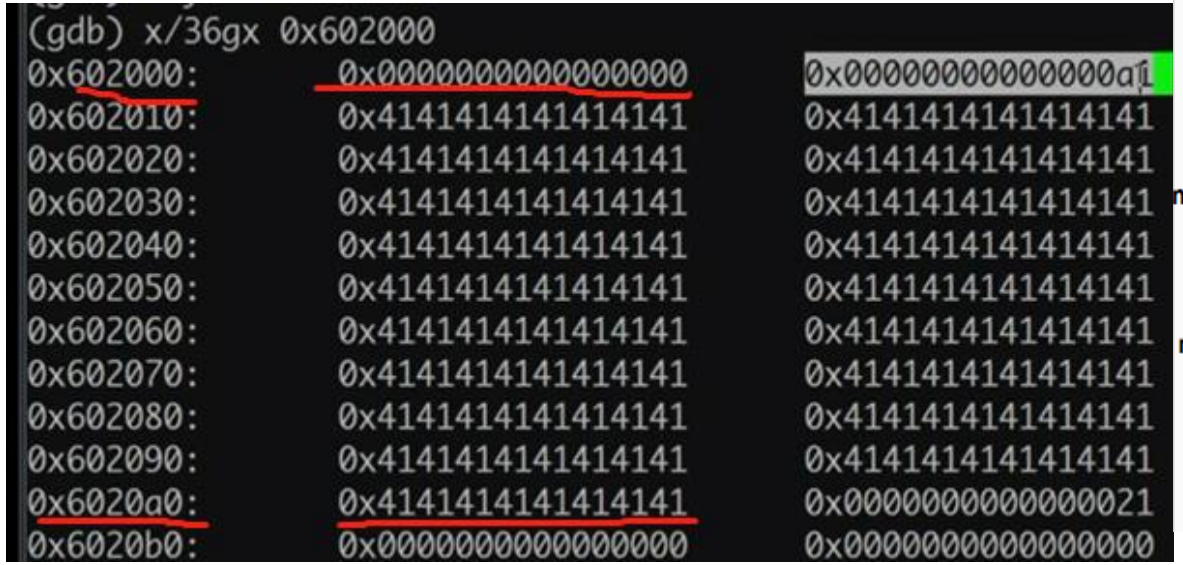
- Chunk2mem:根据chunk地址获得返回给用户的内存地址
- mem2chunk宏根据mem地址得到chunk地址
- 宏aligned_OK和misaligned_chunk(p)用于校验地址是否是按2*SIZE_SZ对齐的
- MINSIZE定义了最小的分配的内存大小

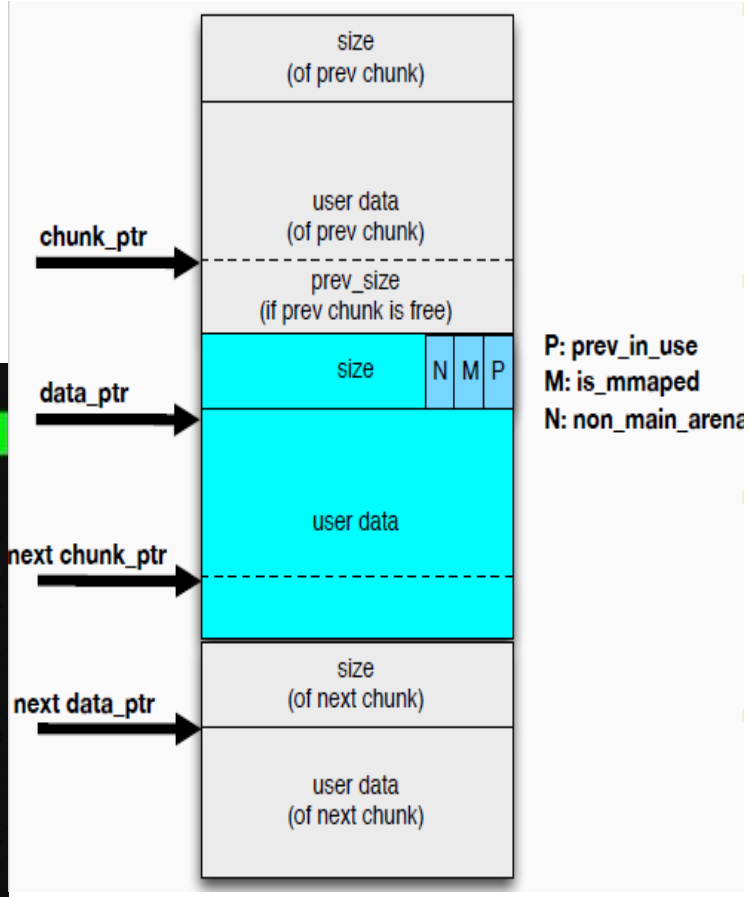
MIN_CHUNK_SIZE 定义了最小的chunk的大小

```
/* conversion from malloc headers to user pointers, and back */
#define chunk2mem(p)    ((Void_t*)((char*)(p) + 2*SIZE_SZ))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))
/* The smallest size we can malloc is an aligned minimal chunk */
#define MINSIZE  \
  (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK))

/* Check if m has acceptable alignment */
#define aligned_OK(m)  (((unsigned long)(m) & MALLOC_ALIGN_MASK) == 0)
#define misaligned_chunk(p) \
  ((uintptr_t)(MALLOC_ALIGNMENT == 2 * SIZE_SZ ? (p) : chunk2mem (p)) \
   & MALLOC_ALIGN_MASK)
```

源代码分析-边界标记法

1. 若前一个chunk被使用，下一个chunk的prev_size是无效的，这个空间是可以被复用的





1. 可以看出，下一个chunk的prev_size，被复用了

源代码分析-分箱式内存管理

根据空闲chunk的大小和处于的状态将其放在四个不同的bin中，这四个空闲chunk的容器包括fast bins，unsorted bin， small bins和large bins。

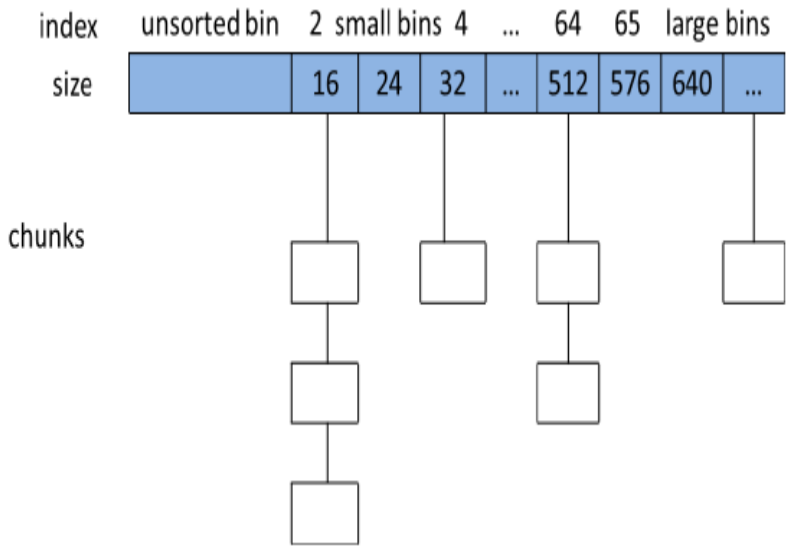
- **Fast bins**：是小内存块的高速缓存
- **Usorted bin**：只有一块，相当于一个过渡区：
  - 需要再次分配时，查看unsorted bin，如果有直接分配用户，否则，将它里面的所欲chunk进行分类到smallbin和lagerbin中。
- **Small bins**：共64个bins，最小的是16字节或者32字节，每个bin的大小相差8字节或是16字节。每一bin中chunk的大小一样。
- **Large bins**：用于存储大于等于512B或1024B的空闲chunk ，采用双向链表形式。

源代码分析-分箱式内存管理

Small bin：

- chunk，每个small bin中的chunk的大小与bin的index有如下关系：

  Chunk_size=2 * SIZE_SZ * index
- 每个链表的大小都是样，这样当有分配的需求直接从链表中摘取。
- 这样做能够很好的解决用户的申请需求，也不会出现太多的内存碎片。



源代码分析-分箱式内存管理

Large bins：一共包含63个bin，32位，>512B;64位，>1024B

- 每个bin中的chunk大小不是一个固定公差的等差数列
- 分为6组，
- 每组的bin数量依次为32、16、8、4、2、1，公差依次为64B、512B、4096B、32768B、262144B等
- 等差数列满足的关系：
- 第一个大小：512B，共32个bin，公差为64B
- Chunk_size=512 + 64 * index
- 第二个位第一个结束的位置
- Chunk_size=512 + 64 * 32 + 512 * index

组	数量	公差
1	32	64B
2	16	512B
3	8	4096B
4	4	32768B
5	2	262144B
6	1	不限制

源代码分析-分箱式内存管理

Large bins：一共包含63个bin，32位，>512B;64位，>1024B

- Chunk_size=512 + 64 * index
- 第二个位第一个结束的位置
- Chunk_size=512 + 64 * 32 + 512 * index

```
#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)

// 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.
```

开始(字节)	结束（字节）	Bin index
0	7	不存在
8	15	不存在
16	23	2
24	31	3
32	39	4
40	47	5
48	55	6
56	63	7
64	71	8
72	79	9
80	87	10
88	95	11
96	103	12
104	111	13
112	119	14
120	127	15
128	135	16
136	143	17
144	151	18

源代码分析-分箱式内存管理

Large bins：一共包含63个bin，32位，>512B;64位，>1024B

- 宏bin_at(m, i)通过bin index获得bin的链表头
- 宏next_bin(b)用于获得下一个bin的地址
- 宏first(b)用于获得bin的第一个可用chunk，
- 宏last(b)用于获得bin的最后一个可用的chunk，
- 宏unlink(P, BK, FD) ，
  - 删除节点操作

```
#define bin_at(m, i) \
  (mbinptr) (((char *) &((m)->bins[((i) - 1) * 2])) \
             - offsetof (struct malloc_chunk, fd))

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

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

```
#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)                                     \
                FD->fd_nextsize = FD->bk_nextsize = FD;                    \
              else {                                                       \
                  FD->fd_nextsize = P->fd_nextsize;                        \
                  FD->bk_nextsize = P->bk_nextsize;                        \
                  P->fd_nextsize->bk_nextsize = FD;                        \
                  P->bk_nextsize->fd_nextsize = FD;                        \
                }                                                          \
            } else {                                                       \
              P->fd_nextsize->bk_nextsize = P->bk_nextsize;                \
              P->bk_nextsize->fd_nextsize = P->fd_nextsize;                \
            }                                                              \
        }                                                                  \
    }                                                                      \
}
```

源代码分析-分箱式内存管理

Unsorted bin ：它可以看做是small bins和large bins的cache。

- 双向链表管理空闲chunk，空闲chunk不排序
- 在回收时，都放在unsorted bin
- unsorted bin中没有合适的chunk，就会把他放在相应的bins中
- Bins数组中的元素bin[1]用于存储unsorted bin的chunk链表头

源代码分析-分箱式内存管理

Fast bins ： 主要提高小内存的分配效率；32;<64B;64 <128B

在回收时，都放在unsorted bin

- Fast bins可以看着是small bins的一小部分cache
- Bins数组中的元素bin[1]用于存储unsorted bin的chunk链表头
- Fast bins可以看着是LIFO的栈，使用单向链表实现。
- 宏fastbin_index(sz)用于获得fast bin在fast bins数组中的index

Fast bin #	Holds chunk sizes	Read chunk size
0	00 – 12	16
1	13 – 20	24
2	21 – 28	32
3	29 – 36	40
4	37 – 44	48
5	45 – 52	56
6	53 – 60	64
7	61 – 68	72
8	69 – 76	80
9	77 - 80	88

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

源代码分析-核心结构体分析

- struct malloc_state
  - 每个分配区是struct malloc_state的一个实例
  - ptmalloc使用malloc_state来管理分配区
- struct malloc_par
  - 而参数管理使用struct malloc_par，全局拥有一个唯一的malloc_par实例。

源代码分析-核心结构体分析

- malloc_state
  - Mutex:用于加锁操作
  - Flags记录了分配区的一些标志
  - Malloc_consolidate():合并所有fast_bin的chunk
  - Clear_fastbin:用于标志分配区已经没有fast chunk
  - Set_fast_chunks,表示分配区有fast chunk

```
#define FASTCHUNKS_BIT        (1U)
#define have_fastchunks(M)     (((M)->flags &  FASTCHUNKS_BIT) == 0)
#ifdef ATOMIC_FASTBINS
#define clear_fastchunks(M)    catomic_or (&(M)->flags, FASTCHUNKS_BIT)
#define set_fastchunks(M)      catomic_and (&(M)->flags, ~FASTCHUNKS_BIT)
#else
#define clear_fastchunks(M)    ((M)->flags |=  FASTCHUNKS_BIT)
#define set_fastchunks(M)      ((M)->flags &= ~FASTCHUNKS_BIT)
#endif
```

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);

  /* Flags (formerly in max_fast).  */
  int flags;

  /* Set if the fastbin chunks contain recently inserted free blocks.  */
  /* Note this is a bool but not all targets support atomics on booleans.
  int have_fastchunks;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

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

  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;

  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];

  /* 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;
```

源代码分析-核心结构体分析

- malloc_state

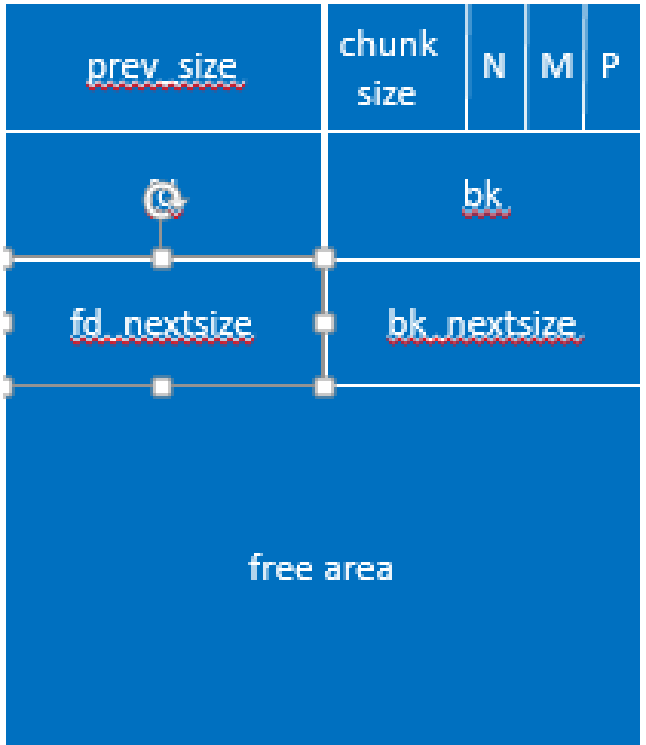
  NONCONTIGUOUS_BIT：表示MORCORE返回连续空间，

  - 主分配区： MORCORE使用sbr（）默认返回俩连续的虚拟地址
  - 非主分区：使用mmap()分配大块的虚拟内存。
    - 非主分配区默认不分配连续的空间
- fastbinsY拥有10（NFASTBINS）个元素的数组
- top是一个chunk指针，指向分配区的top chunk
- last_remainder是一个chunk指针
  - 分裂small_bin，一半给用户，一半给就是chunk，last_remainder指向这个chunk。

```
#define NONCONTIGUOUS_BIT      (2U)
#define contiguous(M)          (((M)->flags &  NONCONTIGUOUS_BIT) == 0)
#define noncontiguous(M)       (((M)->flags &  NONCONTIGUOUS_BIT) != 0)
#define set_noncontiguous(M)   ((M)->flags |=  NONCONTIGUOUS_BIT)
#define set_contiguous(M)      ((M)->flags &= ~NONCONTIGUOUS_BIT)
```

源代码分析-核心结构体分析

- malloc_state
  - Bins：unstored bin，small bins和large bins的chunk链表头
    - small bins一共62个，large bins一共63个，来一共125个bin
  - NBINS定义为128，
    - bin[0]和bin[127]不存在，bin[1]为unsorted bin的chunk链表头



- “头节点”，prev_size和size字段是没有任何实际作用。
- fd_nextsize和bk_nextsize字段只有large bins中的空闲chunk才会用到
- large bins的空闲chunk链表头不需要这两个字段
- 这四个字段不合理运用。浪费

源代码分析-核心结构体分析

- malloc_state:
  - Binmap:是一个int数组,用一个bit的位表示bit对应的空闲chunk
  - idx2bit宏取第i位为1，其它位都为0的掩码
    - 举例：：idx2bit(3) 为 “0000 1000”
  - mark_bin设置第i个bin在binmap中对应的bit位为1
  - unmark_bin设置第i个bin在binmap中对应的bit位为0
  - get_binmap获取第i个bin在binmap中对应的bit

```
#define BINMAPSHIFT      5
#define BITSPERMAP       (1U << BINMAPSHIFT)
#define BINMAPSIZE       (NBINS / BITSPERMAP)

#define idx2block(i)     ((i) >> BINMAPSHIFT)
#define idx2bit(i)       ((1U << ((i) & ((1U << BINMAPSHIFT)-1))))

#define mark_bin(m,i)    ((m)->binmap[idx2block(i)] |=  idx2bit(i))
#define unmark_bin(m,i)  ((m)->binmap[idx2block(i)] &= ~(idx2bit(i)))
#define get_binmap(m,i)  ((m)->binmap[idx2block(i)] &   idx2bit(i))
```

源代码分析-核心结构体分析

Malloc_par

- trim_threshold字段表示收缩阈值，默认为128KB
- TOP_pad：填充，默认是0
- mmap_threshold字段表示mmap分配阈值，默认值为128KB
- arena_test和arena_max用于PER_THREAD优化
- n_mmaps：表示当前进程使用mmap()函数分配的内存块的个数
- n_mmaps_max字段表示进程使用mmap()函数分配的内存块的最大数量。默认值为65536

```
1709 struct malloc_par
1710 {
1711   /* Tunable parameters */
1712   unsigned long trim_threshold;
1713   INTERNAL_SIZE_T top_pad;
1714   INTERNAL_SIZE_T mmap_threshold;
1715   INTERNAL_SIZE_T arena_test;
1716   INTERNAL_SIZE_T arena_max;
1717
1718   /* Memory map support */
1719   int n_mmaps;
1720   int n_mmaps_max;
1721   int max_n_mmaps;
1722   /* the mmap_threshold is dynamic, until the user sets
1723      it manually, at which point we need to disable any
1724      dynamic behavior. */
1725   int no_dyn_threshold;
1726
1727   /* Statistics */
1728   INTERNAL_SIZE_T mmapped_mem;
1729   INTERNAL_SIZE_T max_mmapped_mem;
1730
1731   /* First address handed out by MORECORE/sbrk.  */
1732   char *sbrk_base;
1733
1734 #if USE_TCACHE
1735   /* Maximum number of buckets to use.  */
1736   size_t tcache_bins;
1737   size_t tcache_max_bytes;
1738   /* Maximum number of chunks in each bucket.  */
1739   size_t tcache_count;
1740   /* Maximum number of chunks to remove from the unsorted list, which
1741      aren't used to prefill the cache.  */
1742   size_t tcache_unsorted_limit;
1743 #endif
```

源代码分析-核心结构体分析

Malloc_par

- max_n_mmaps：使用mmap()函数分配的内存块的数量的最大值
- no_dyn_threshold：字段表示是否开启mmap分配阈值动态调整机制，默认值为0
- pagesize表示系统的页大小，默认为4KB
- mmapped_mem和max_mmapped_mem都用于统计mmap分配的内存大小
- max_total_mem字段在单线程情况下用于统计进程分配的内存总数
- sbrk_base字段表示堆的起始地址。

```
1709 struct malloc_par
1710 {
1711   /* Tunable parameters */
1712   unsigned long trim_threshold;
1713   INTERNAL_SIZE_T top_pad;
1714   INTERNAL_SIZE_T mmap_threshold;
1715   INTERNAL_SIZE_T arena_test;
1716   INTERNAL_SIZE_T arena_max;
1717
1718   /* Memory map support */
1719   int n_mmaps;
1720   int n_mmaps_max;
1721   int max_n_mmaps;
1722   /* the mmap_threshold is dynamic, until the user sets
1723      it manually, at which point we need to disable any
1724      dynamic behavior. */
1725   int no_dyn_threshold;
1726
1727   /* Statistics */
1728   INTERNAL_SIZE_T mmapped_mem;
1729   INTERNAL_SIZE_T max_mmapped_mem;
1730
1731   /* First address handed out by MORECORE/sbrk.  */
1732   char *sbrk_base;
1733
1734 #if USE_TCACHE
1735   /* Maximum number of buckets to use.  */
1736   size_t tcache_bins;
1737   size_t tcache_max_bytes;
1738   /* Maximum number of chunks in each bucket.  */
1739   size_t tcache_count;
1740   /* Maximum number of chunks to remove from the unsorted list, which
1741      aren't used to prefill the cache.  */
1742   size_t tcache_unsorted_limit;
1743 #endif
```

源代码分析-配置选项

分配区的初始化

- av->flags |= FASTCHUNKS_BIT;
  - 是否存在fastbin
- ，mp_是全局唯一的一个malloc_par实例，用于管理参数和统计信息

```
1799 malloc_init_state (mstate av)
1800 {
1801   int i;
1802   mbinptr bin;
1803
1804   /* Establish circular links for normal bins */
1805   for (i = 1; i < NBINS; ++i)
1806     {
1807       bin = bin_at (av, i);
1808       bin->fd = bin->bk = bin;
1809     }
1810
1811 #if MORECORE_CONTIGUOUS
1812   if (av != &main_arena)
1813 #endif
1814   set_noncontiguous (av);
1815   if (av == &main_arena)
1816     set_max_fast (DEFAULT_MXFAST);
1817   atomic_store_relaxed (&av->have_fastchunks, false);
1818
1819   av->top = initial_top (av);
1820 }
```

```
ptmalloc_init_minimal (void)
{
#if DEFAULT_TOP_PAD != 0
  mp_.top_pad        = DEFAULT_TOP_PAD;
#endif
  mp_.n_mmaps_max    = DEFAULT_MMAP_MAX;
  mp_.mmap_threshold = DEFAULT_MMAP_THRESHOLD;
  mp_.trim_threshold = DEFAULT_TRIM_THRESHOLD;
  mp_.pagesize       = malloc_getpagesize;
#ifdef PER_THREAD
# define NARENAS_FROM_NCORES(n) ((n) * (sizeof(long) == 4 ? 2 : 8))
  mp_.arena_test     = NARENAS_FROM_NCORES (1);
  narenas = 1;
```

源代码分析-配置选项

配置选项

- Ptmalloc的配置选项不多
  - mallopt()函数配置前，需要检查主分配区是否初始化了，如果没有初始化，调用ptmalloc_init()函数初始化ptmalloc，然后获得主分配区的锁，调用malloc_consolidate()函数，malloc_consolidate()函数会判断主分配区是否已经初始化，如果没有，则初始化主分配区。同时我们也看到，mp_都没有锁，对mp_中参数字段的修改，是通过主分配区的锁来同步的。

```
switch(param_number) {
case M_MXFAST:
  if (value >= 0 && value <= MAX_FAST_SIZE) {
    set_max_fast(value);
  }
  else
    res = 0;
  break;

case M_TRIM_THRESHOLD:
  mp_.trim_threshold = value;
  mp_.no_dyn_threshold = 1;
  break;
```

源代码分析- Ptmalloc的初始化

在 ptmalloc 中 malloc()函数的实际接口函数为 public_mALLOc()，这个函数最开始会执行如下的一段代码：

```
__malloc_ptr_t (*hook) (size_t, __const __malloc_ptr_t)
  = force_reg (__malloc_hook);
if (__builtin_expect (hook != NULL, 0))
  return (*hook)(bytes, RETURN_ADDRESS (0));
```

在定义了__malloc_hook()全局函数的情况下，只是执行__malloc_hook()函数，在进程初始化时__malloc_hook 指向的函数为 malloc_hook_ini()。

```
__malloc_ptr_t weak_variable (*__malloc_hook)
     (size_t __size, const __malloc_ptr_t) = malloc_hook_ini;
```

malloc_hook_ini()函数定义在 hooks.c 中，实现代码如下：

```
static Void_t*
#if __STD_C
malloc_hook_ini(size_t sz, const __malloc_ptr_t caller)
#else
malloc_hook_ini(sz, caller)
     size_t sz; const __malloc_ptr_t caller;
#endif
{
  __malloc_hook = NULL;
  ptmalloc_init();
  return public_mALLOc(sz);
```

源代码分析- Ptmalloc的初始化

Ptmalloc未初始化时分配/释放内存

- ptmalloc_init()之前，Glibc中可能需要分配内存。
- 为了解决上述问题，ptmalloc分装了分配释放函数提供下面三个函数：
  - malloc_starter()
  - memalign_starter()
  - free_starter()

源代码分析- Ptmalloc的初始化

ptmalloc_init()函数

ptmalloc_init()函数比较长，将分段对这个函数做介绍

```
if(__malloc_initialized >= 0) return;
__malloc_initialized = 0;
```

- 是否malloc初始化

```
save_malloc_hook = __malloc_hook;
save_memalign_hook = __memalign_hook;
save_free_hook = __free_hook;
__malloc_hook = malloc_starter;
__memalign_hook = memalign_starter;
__free_hook = free_starter;
```

- 对于多线程的版本，对于一些私有变量，需要保存当前的hook函数

```
mutex_init(&main_arena.mutex);
main_arena.next = &main_arena;
```

- 初始化主分配区的mutex，并将主分配区的next指针指向自身组成环形链表

源代码分析- Ptmalloc的初始化

ptmalloc_init()函数

- 主分配区才能使用sbrk()分配连续虚拟内存空间
- 加载动态连接库glibc
  - Glibc在内存中只有一份拷贝
  - 数据段在内存中也只有一份拷贝
- morecore函数指针指向 failing morecore就可以禁止使用

```
#if defined _LIBC && defined SHARED
  /* In case this libc copy is in a non-default namespace, never use brk.
     Likewise if dlopened from statically linked program.  */
  Dl_info di;
  struct link_map *l;

  if (_dl_open_hook != NULL
      || (_dl_addr (ptmalloc_init, &di, &l, NULL) != 0
          && l->l_ns != LM_ID_BASE))
    __morecore = __failing_morecore;
```

源代码分析- Ptmalloc的初始化

ptmalloc_init()函数

- **初始化全局锁**
  - list_lock，list_lock主要用于同步分配区的单向循环链表
- arena_key是私有实例
  - 保存的是分配区（arena）的malloc_state实例指针
  - 调用ptmalloc_init()，首选从主分配区分配内存
- thread_atfork()设置当前进程在fork子线程
- ptmalloc_lock_all()获得所有分配区的锁
- **子线程调用ptmalloc_unlock_all2()重新初始化每个分配区的锁**

mutex

```
mutex_init(&list_lock);
tsd_key_create(&arena_key, NULL);
tsd_setspecific(arena_key, (Void_t *)&main_arena);
thread_atfork(ptmalloc_lock_all, ptmalloc_unlock_all, ptmalloc_unlock_all2)
```

源代码分析- Ptmalloc的初始化

ptmalloc_init()函数

- 在ptmalloc_init()函数结束处
  - 查看是否存在__malloc_initialize_hook函数，
    - 如果存在，执行该hook函数**arena_key**是私有实例
- 最后将全局变量__malloc_initialized设置为1，
  - 表示ptmalloc_init()已经初始化完成。

```
void (*hook) (void) = force_reg (__malloc_initialize_hook);
if (hook != NULL)
  (*hook)();
__malloc_initialized = 1;
```

源代码分析- Ptmalloc的初始化

ptmalloc_init()函数

- 父进程中的某个子线程，分配内存
  - 将使用malloc_atfork()函数分配内存
  - 判断：指针为ATFORK_ARENA_PTR，
    - 锁住全局锁和分配区
    - 当前只有本线程可以分配内存
  - 否则：是常规分配
    - 等待另外的线程完成分配
    - 获得全局锁list_lock
      - 等待完成fork子线程
    - 释放全局所list_lock
    - public_mALLOc()分配内存

```
static Void_t*
malloc_atfork(size_t sz, const Void_t *caller)
{
  Void_t *vptr = NULL;
  Void_t *victim;

  tsd_getspecific(arena_key, vptr);
  if(vptr == ATFORK_ARENA_PTR) {
    /* We are the only thread that may allocate at all.  */
    if(save_malloc_hook != malloc_check) {
      return _int_malloc(&main_arena, sz);
    } else {
      if(top_check()<0)
        return 0;
      victim = _int_malloc(&main_arena, sz+1);
      return mem2mem_check(victim, sz);
    }
  } else {
    /* Suspend the thread until the `atfork' handlers have completed.
       By that time, the hooks will have been reset as well, so that
       mALLOc() can be used again. */
    (void)mutex_lock(&list_lock);
    (void)mutex_unlock(&list_lock);
    return public_mALLOc(sz);
```

源代码分析- Ptmalloc的初始化

ptmalloc_init()函数:

- 父进程在fork子进程时:
  - 需要保证分配区的一致性
  - 获得全局锁，
  - 然后对所有的分配区加锁
  - 保存分配释放函数
  - 早fork期间，使用 malloc_atfork，free_atfork 来作为分配释放函数使用
  - 保存当前线程的私有实例中的原有分配区指针
- atfork_recursive_cntr：用于对加锁的层数进行记录

```
static void
ptmalloc_lock_all (void)
{
  mstate ar_ptr;

  if(__malloc_initialized < 1)
    return;
  if (mutex_trylock(&list_lock))
    {
      Void_t *my_arena;
      tsd_getspecific(arena_key, my_arena);
      if (my_arena == ATFORK_ARENA_PTR)
        /* This is the same thread which already locks the global list.
           Just bump the counter.  */
        goto out;

      /* This thread has to wait its turn.  */
      (void)mutex_lock(&list_lock);
    }
  for(ar_ptr = &main_arena;;) {
    (void)mutex_lock(&ar_ptr->mutex);
    ar_ptr = ar_ptr->next;
    if(ar_ptr == &main_arena) break;
  }
  save_malloc_hook = __malloc_hook;
  save_free_hook = __free_hook;
  __malloc_hook = malloc_atfork;
  __free_hook = free_atfork;
  /* Only the current thread may perfo
  tsd_getspecific(arena_key, save_arena);
  tsd_setspecific(arena_key, ATFORK_ARENA_PTR);
out:
  ++atfork_recursive_cntr;
```

wy
2019/5/3 12:19:48

//将线程的私有变量先设置成主分区

源代码分析- Ptmalloc的初始化

ptmalloc_init()函数:

- atfork_recursive_cntr减1
  - 判断他是否为0;
  - 这保证了递归fork子线程只会解锁一次
  - 接着将当前线程的私有实例还原为原来的分配区

```
if(__malloc_initialized < 1)
  return;
if (--atfork_recursive_cntr != 0)
  return;
tsd_setspecific(arena_key, save_arena)
__malloc_hook = save_malloc_hook;
__free_hook = save_free_hook;
for(ar_ptr = &main_arena;;) {
  (void)mutex_unlock(&ar_ptr->mutex);
  ar_ptr = ar_ptr->next;
  if(ar_ptr == &main_arena) break;
}
(void)mutex_unlock(&list_lock);
```

源代码分析- Ptmalloc的初始化

ptmalloc_init()函数:

- ptmalloc_unlock_all2（）
  - unlock从父线程（进程）中继承的mutex不安全，会导致资源泄漏
- 重新初始化mutex是安全的

```
#endif
    ar_ptr = ar_ptr->next;
    if(ar_ptr == &main_arena) break;
  }
  mutex_init(&list_lock);
  atfork_recursive_cntr = 0;
}
#else
#define ptmalloc_unlock_all2 ptmalloc_unlock_all
#endif
```

源代码分析-多分配区支持

- 1.多个线程是不能同时调用sbrk()函数的
  - 多个线程都从主分配区中分配小内存块，效率很低效
  - 同时解决多线程使用sbrk（）
- **解决办法：**
  - ptmalloc使用非主分配区来模拟主分配区的功能
  - 非主分区使用mmap分配一大块内存模拟堆（sub_heap）
  - 从中按需求切割小块
  - Sub_heap不够用，或者用光了，则在使用mmap再申请一块，以单项链表链接在非主分配区中sub-heap的单向链表中
- 收缩堆的条件：
  - 当前free的chunk大小加上前后能合并chunk的大小大于64KB，
  - 并且top chunk的大小达到mmap收缩阈值，才有可能收缩堆。

源代码分析-多分配区支持

Heap_info： 即Heap Header

- Pad字段用于保证sizeof (heap_info) + 2 * SIZE_SZ是按MALLOC_ALIGNMENT对齐的。MALLOC_ALIGNMENT_MASK为2 * SIZE_SZ - 1，无论SIZE_SZ为4或8，-6 * SIZE_SZ & MALLOC_ALIGN_MASK的值为0，如果sizeof (heap_info) + 2 * SIZE_SZ不是按MALLOC_ALIGNMENT对齐，编译的时候就会报错，编译时会执行下面的宏。

为什么一定要保证对齐呢？

```
typedef struct _heap_info
{
  mstate ar_ptr; /* Arena for this heap. */
  struct _heap_info *prev; /* Previous heap. */
  size_t size;   /* Current size in bytes. */
  size_t mprotect_size; /* Size in bytes that has been mprotected
                           PROT_READ|PROT_WRITE.  */
  /* Make sure the following data is properly aligned, particularly
     that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
     MALLOC_ALIGNMENT. */
  char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

源代码分析-多分配区支持

- arena_key存放的是线程的私用实例，该私有实例保存的是分配区（arena）的malloc_state实例的指针。
- list_lock用于同步分配区的单向环形链表。
- narenas全局变量表示当前分配区的数量
- free_list全局变量是空闲分配区的单向链表

```
static tsd_key_t arena_key;
static mutex_t list_lock;
#ifdef PER_THREAD
static size_t narenas;
static mstate free_list;
#endif

/* Mapped memory in non-main arenas (reliable only for NO_THREADS). */
static unsigned long arena_mem;
```

源代码分析-多分配区支持

- Arena_get2() ：
- arena_get宏尝试查看线程的私用实例中是否包含一个分配区
- arena_get2()函数获得一个分配区
- 下图：私有变量没有分配区，将主分配区做为候选变量
  - 如果私有变量存在分配区，但不能获得锁
    - 将分配区的下一个作为候选分配区
    - 如果这个候选分配区为空，意味这当前分配区在初始化

```
#else
  if(!a_tsd)
    a = a_tsd = &main_arena;
  else {
    a = a_tsd->next;
    if(!a) {
      /* This can only happen while initializing the new arena. */
      (void)mutex_lock(&main_arena.mutex);
      THREAD_STAT(++(main_arena.stat_lock_wait));
      return &main_arena;
    }
  }
```

源代码分析-多分配区支持

- Arena_get2() ：
- 当线程在创建子线程的时候，当前线程不能获得分配区
  - 首先尝试加全局锁
  - 如果不成功，就阻塞全局锁，直到过的全局锁
  - 所有的查找完成，之后创建新的分配区
    - 将分配区加入了全局分配区链表

```
/* If not even the list_lock can be obtained, try again.  This can
   happen during `atfork', or for example on systems where thread
   creation makes it temporarily impossible to obtain _any_
   locks. */
if(!retried && mutex_trylock(&list_lock)) {
  /* We will block to not run in a busy loop.  */
  (void)mutex_lock(&list_lock);

  /* Since we blocked there might be an arena available now.  */
  retried = true;
  a = a_tsd;
  goto repeat;
}

/* Nothing immediately available, so generate a new arena.  */
a = _int_new_arena(size);
(void)mutex_unlock(&list_lock);
```

源代码分析-多分配区支持

_int_new_arena() ：

- _int_new_arena()函数用于创建一个非主分配区，在arena_get2()函数中被调用
- ，malloc state实例紧接着heap_info实例

```
_int_new_arena(size_t size)
{
  mstate a;
  heap_info *h;
  char *ptr;
  unsigned long misalign;

  h = new_heap(size + (sizeof(*h) + sizeof(*a) + MALLOC_ALIGNMENT),
               mp_.top_pad);
  if(!h) {
    /* Maybe size is too large to fit in a single heap.  So, just try
       to create a minimally-sized arena and let _int_malloc() attempt
       to deal with the large request via mmap_chunk().  */
    h = new_heap(sizeof(*h) + sizeof(*a) + MALLOC_ALIGNMENT, mp_.top_pad);
    if(!h)
      return 0;

  /* Set up the top chunk, with proper alignment. */
  ptr = (char *)(a + 1);
  misalign = (unsigned long)chunk2mem(ptr) & MALLOC_ALIGN_MASK;
  if (misalign > 0)
    ptr += MALLOC_ALIGNMENT - misalign;
  top(a) = (mchunkptr)ptr;
  set_head(top(a), (((char*)h + h->size) - ptr) | PREV_INUSE);
```

1.ptr指向存储malloc_state实例后的空闲内存

2. ptr赋值给分配区的top chunk

源代码分析-多分配区支持

New_heap()：

- _New_heap()函数负责从mmap区域映射一块内存来作为sub_heap
  - 32位，每次映射1M，以1M对齐；
  - 64位，函数映射64M内存，映射的内存块地址按64M对齐

1. size的大小调整到最小值与最大值之间
2. 并以页对齐
3. 如果size大于最大值，直接报错

```
if(size+top_pad < HEAP_MIN_SIZE)
  size = HEAP_MIN_SIZE;
else if(size+top_pad <= HEAP_MAX_SIZE)
  size += top_pad;
else if(size > HEAP_MAX_SIZE)
  return 0;
else
  size = HEAP_MAX_SIZE;
size = (size + page_mask) & ~page_mask;
```

源代码分析-多分配区支持

New_heap()：

- 1. aligned_heap_area不为空，尝试从上次映射结束地址开始映射大小为HEAP_MAX_SIZE的内存块
- 2.全局变量aligned_heap_area没有锁保护
- 3.无论映射成功。全局变量aligned_heap_area设置为NULL
- 4.映射失败，或者映射返回地址没有对齐，则返回失败

```
p2 = MAP_FAILED;
if(aligned_heap_area) {
  p2 = (char *)MMAP(aligned_heap_area, HEAP_MAX_SIZE, PROT_NONE,
                    MAP_PRIVATE|MAP_NORESERVE);
  aligned_heap_area = NULL;
if (p2 != MAP_FAILED && ((unsigned long)p2 & (HEAP_MAX_SIZE-1))) {
  munmap(p2, HEAP_MAX_SIZE);
  p2 = MAP_FAILED;
```

源代码分析-多分配区支持

New_heap()：

- 1. 尝试映射2倍HEAP_MAX_SIZE大小的虚拟内存，便于地址对齐
- 2. 将大于等于p1并按HEAP_MAX_SIZE大小对齐的第一个虚拟地址赋值给p2
- 3. p2作为sub_heap的起始虚拟地址
- 4. p2+ HEAP_MAX_SIZE作为sub_heap的结束地址

```
if(p2 == MAP_FAILED) {
  p1 = (char *)MMAP(0, HEAP_MAX_SIZE<<1, PROT_NONE,
                    MAP_PRIVATE|MAP_NORESERVE);
if(p1 != MAP_FAILED) {
  p2 = (char *)(((unsigned long)p1 + (HEAP_MAX_SIZE-1))
                & ~(HEAP_MAX_SIZE-1));
  ul = p2 - p1;
  if (ul)
    munmap(p1, ul);
  else
    aligned_heap_area = p2 + HEAP_MAX_SIZE;
  munmap(p2 + HEAP_MAX_SIZE, HEAP_MAX_SIZE - ul);
```

5. 并将sub_heap的结束地址赋值给全局变量aligned_heap_area

源代码分析-多分配区支持

New_heap()：

- 1.映射2倍HEAP_MAX_SIZE大小的虚拟内存失败了
- 再尝试映射HEAP_MAX_SIZE大小的虚拟内存
- 调用mprotect()函数将size大小的内存设置为可读可写，
- 如果失败，解除整个sub_heap的映射。
- 然后更新heap_info实例中的相关字段。

```
/* Try to take the chance that an allocation of only HEAP_MAX_SIZE
   is already aligned. */
p2 = (char *)MMAP(0, HEAP_MAX_SIZE, PROT_NONE, MAP_PRIVATE|MAP_NORESERVE);
if(p2 == MAP_FAILED)
  return 0;
if((unsigned long)p2 & (HEAP_MAX_SIZE-1)) {
  munmap(p2, HEAP_MAX_SIZE);
  return 0;
}
if(mprotect(p2, size, PROT_READ|PROT_WRITE) != 0) {
  munmap(p2, HEAP_MAX_SIZE);
  return 0;
```

源代码分析-多分配区支持

get_free_list()和reused_arena()

- 1.开启了PER_THRAD优化时用于获取分配区（arena）
  - rena_get2首先调用get_free_list()尝试获得arena
  - 如果失败在尝试调用reused_arena()获得arena
  - 如果仍然没有获得分配区，调用_int_new_arena()创建一个新的分配区
  - 首先判断全局分配区的总数是否小于分配区的个数的限定值（arena_test），在32位系统上arena_test默认值为2，64位系统上的默认值为8，如果当前进程的分配区数量没有达到限定值，直接返回NULL。

reused_arena()函数的源代码实现如下：

```
static mstate
reused_arena (void)
{
  if (narenas <= mp_.arena_test)
    return NULL;
```

源代码分析-多分配区支持

grow_heap(),shrink_heap(),delete_heap(),heap_trim()

这几个函数实现sub_heap和增长和收缩

- grow_heap()函数主要将sub_heap中可读可写区域扩大
- shrink_heap()函数缩小sub_heap的虚拟内存区域
- delete_heap()为一个宏，如果sub_heap中所有的内存都空闲，使用该宏函数将sub_heap的虚拟内存还回给操作系统
- heap_trim()函数根据sub_heap的top chunk大小调用shrink_heap()函数收缩sub_heap。

```
  if (__builtin_expect (__libc_enable_secure, 0))
#else
  if (1)
#endif
    {
      if((char *)MMAP((char *)h + new_size, diff, PROT_NONE,
                      MAP_PRIVATE|MAP_FIXED) == (char *) MAP_FAILED)
        return -2;
      h->mprotect_size = new_size;
```

源代码分析-多分配区支持

grow_heap(),shrink_heap(),delete_heap(),heap_trim()

- 1.**释放**sub_heap
- 2.**把sub_heap的前一块指向heap**
- 3. p指向的是被释放sub_heap的前一个sub_heap的倒数第二个chunk
- 4.如果p的前一个chunk为空闲状态,
  - **就把p的前一块赋给p,并将其从空闲链表中删除**

```
ar_ptr->system_mem -= heap->size;
arena_mem -= heap->size;
delete_heap(heap);
heap = prev_heap;
if(!prev_inuse(p)) { /* consolidate backward */
  p = prev_chunk(p);
  unlink(p, bck, fwd);
}
assert(((unsigned long)((char*)p + new_size) & (pagesz-1)) == 0);
assert( ((char*)p + new_size) == ((char*)heap + heap->size) );
top(ar_ptr) = top_chunk = p;
set_head(top_chunk, new_size | PREV_INUSE);
```

5.将该空闲chunk赋值给sub_heap的top chunk

6.设置top chunk的size，标识top chunk的前一个chunk处于inuse状态

源代码分析-多分配区支持

grow_heap(),shrink_heap(),delete_heap(),heap_trim()

- 1.保证缩小在一页之内
- 2.否则调用shrink_heap()函数对当前sub_heap进行收缩
- 3.最后，更新内存使用统计，更新top chunk的大小

```
top_size = chunksize(top_chunk);
extra = ((top_size - pad - MINSIZE + (pagesz-1))/pagesz - 1) * pagesz;
if(extra < (long)pagesz)
  return 0;
/* Try to shrink. */
if(shrink_heap(heap, extra) != 0)
  return 0;
ar_ptr->system_mem -= extra;
arena_mem -= extra;

/* Success. Adjust top accordingly. */
set_head(top_chunk, (top_size - extra) | PREV_INUSE);
/*check_chunk(ar_ptr, top_chunk);*/
```

谢谢！

大成若缺，其用不弊。大盈若冲，其用不穷。大直若屈。大巧若拙。大辩若讷。静胜躁，寒胜热。清静为天下正。