[译] ptmalloc介绍

Anciety / 2018-10-16 00:00:47 / 浏览数 1960 技术文章 技术文章 顶(0) 踩(0)

原文链接

https://dangokyo.me/2017/12/05/introduction-on-ptmalloc-part1/

ptmalloc介绍

ptmalloc是在libc里用的内存分配及，我打算分两个部分细致的阐述一下ptmalloc的相关内容，这篇文章是第一部分。在这篇文章里，我会介绍ptmalloc里使用到的数据

Ptmalloc Chunk 块

在ptmalloc中最基本的分配单位是malloc_chunk，包括6个元数据域。如下所示，每一个元数据在x86平台下为4字节长，在x64平台下为8字节长。在这篇文章接下来的

```
#ifndef INTERNAL_SIZE_T
# define INTERNAL_SIZE_T size_t
#endif
#define SIZE_SZ (sizeof (INTERNAL_SIZE_T))

struct malloc_chunk;
typedef struct malloc_chunk* mchunkptr;

struct malloc_chunk {

 INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  ■■■■■■■■■■■■■■ */
 INTERNAL_SIZE_T      mchunk_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.  */
 /* ■■large■■■■■■■■■■■■■■■■■■■■ */
 struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
 struct malloc_chunk* bk_nextsize;
};
```

为了避免混淆，我们首先需要强调一下chunk（块）的概念。在ptmalloc里，一个块指的是通过内存管理分配器分配的一段内存区域，用来储存元数据和应用数据。

在ptmalloc里，一共有三种基本类型的块：

- 已分配块(allocated chunk)
- 释放块(freed chunk)
- top块(top chunk)

在介绍这三种块之前，我们首先来看一下块的操作，通过这些操作，我们可以观察到在mchunk_size中的最后三位分别用来表示了chunk的状态：

- A: 当chunk由非main arena获取时被设置为1
- M: 当chunk由mmap获取时设置为1
- P: 当前一个邻接的chunk被使用时设置为1

在CTF的堆利用题目中我们大多数时候只需要关注P位(0x1)，这里P在前一个邻接块被使用时为1，而前一个邻接块被释放时为0。前一个邻接块在这里主要指在当前块位置前
chunk)不同，前块的相关内容将会在后面块管理部分讨论。

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

/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use */
#define PREV_INUSE 0x1
/* extract inuse bit of previous chunk */
#define prev_inuse(p)       ((p)->mchunk_size & PREV_INUSE)
/* size field is or'ed with IS_MMAPPED if the chunk was obtained with mmap() */
#define IS_MMAPPED 0x2
```

```
/* check for mmap()'ed chunk */
#define chunk_is_mmapped(p) ((p)->mchunk_size & IS_MMAPPED)
/* 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
/* Check for chunk from main arena.  */
#define chunk_main_arena(p) (((p)->mchunk_size & NON_MAIN_ARENA) == 0)
/* Mark a chunk as not being on the main arena.  */
#define set_non_main_arena(p) ((p)->mchunk_size |= NON_MAIN_ARENA)

#define SIZE_BITS (PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)
/* Get size, ignoring use bits */
#define chunksize(p) (chunksize_nomask (p) & ~(SIZE_BITS))
/* Like chunksize, but do not mask SIZE_BITS.  */
#define chunksize_nomask(p)         ((p)->mchunk_size)
/* Ptr to next physical malloc_chunk. */
#define next_chunk(p) ((mchunkptr) (((char *) (p)) + chunksize (p)))
/* Size of the chunk below P.  Only valid if prev_inuse (P).  */
#define prev_size(p) ((p)->mchunk_prev_size)
/* Set the size of the chunk below P.  Only valid if prev_inuse (P).  */
#define set_prev_size(p, sz) ((p)->mchunk_prev_size = (sz))
/* Ptr to previous physical malloc_chunk.  Only valid if prev_inuse (P).  */
#define prev_chunk(p) ((mchunkptr) (((char *) (p)) - prev_size (p)))
/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr) (((char *) (p)) + (s)))
/* extract p's inuse bit */
#define inuse(p)                                  \
 ((((mchunkptr) (((char *) (p)) + chunksize (p)))->mchunk_size) & PREV_INUSE)
/* set/clear chunk as being inuse without otherwise disturbing */
#define set_inuse(p)                                  \
 ((mchunkptr) (((char *) (p)) + chunksize (p)))->mchunk_size |= PREV_INUSE
#define clear_inuse(p)                                \
 ((mchunkptr) (((char *) (p)) + chunksize (p)))->mchunk_size &= ~(PREV_INUSE)
/* check/set/clear inuse bits in known places */
#define inuse_bit_at_offset(p, s)                         \
 (((mchunkptr) (((char *) (p)) + (s)))->mchunk_size & PREV_INUSE)
#define set_inuse_bit_at_offset(p, s)                         \
 (((mchunkptr) (((char *) (p)) + (s)))->mchunk_size |= PREV_INUSE)
#define clear_inuse_bit_at_offset(p, s)                       \
 (((mchunkptr) (((char *) (p)) + (s)))->mchunk_size &= ~(PREV_INUSE))
/* Set size at head, without disturbing its use bit */
#define set_head_size(p, s)  ((p)->mchunk_size = (((p)->mchunk_size & SIZE_BITS) | (s)))
/* Set size/use field */
#define set_head(p, s)       ((p)->mchunk_size = (s))
/* Set size at footer (only when chunk is not in use) */
#define set_foot(p, s)       (((mchunkptr) ((char *) (p) + (s)))->mchunk_prev_size = (s))
```

已分配块

对于一个已分配块，其size域中上一个块（previous chunk）将在其邻接前块是被释放的状态，而且P位为设置时会被设置。在下一个邻接块中，P位将会被设置。需要注意的一点是如果当前的块为已分配块时，mchunk_prev

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of previous chunk                             |<= chunk
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of chunk, in bytes                     |A|M|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             User data starts here...                           .<= mem
.                                                                .
.             (malloc_usable_size() bytes)                       .
.                                                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             (size of chunk, but used for application data)     |<= next chunk
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of next chunk, in bytes                |A|0|1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

释放块

对于一个释放块来说，上一块的大小（previous chunk size)在上一邻接块被释放了，且P位为设置时会被设置。对于下一个邻接块，P位将会被清零，mchunk_prev_size将会被设置为当前块的大小。前块指针(forward)和后块

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of previous chunk                            |<= chunk
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of chunk, in bytes                     |A|0|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Forward pointer to next chunk in list             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Back pointer to previous chunk in list            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Unused space (may be 0 bytes long)                .
.                                                               .
.                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of chunk, in bytes                           |<= next chunk
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of next chunk, in bytes                |A|0|0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

top块

在top块里，块大小表示main arena当前还有多少剩余大小。如果新的大小比当前大小要大，则brk()或mmap()将会被调用来扩大top块

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of previous chunk                            |<= top chunk
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Size of chunk, in bytes                     |A|0|P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+                                                               +
|                                                               |
+                                                               +
.                                                               .
.                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

块管理

在理解了ptmalloc中的malloc块之后，我们就来到理解块是如何由内存分配器管理的部分了。在ptmalloc中，一共有四种类型的bin，用来存储不同类型的释放块：Fas Unsorted bin, Small bin, Large bin。结构体malloc_state用来存储top chunk指针，last remainder，fastbins和bins，如下所示：

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

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

 /* 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;

 /* Number of threads attached to this arena.  0 if the arena is on
    the free list.  Access to this field is serialized by
    free_list_lock in arena.c.  */
 INTERNAL_SIZE_T attached_threads;

 /* Memory allocated from the system in this arena.  */
 INTERNAL_SIZE_T system_mem;
 INTERNAL_SIZE_T max_system_mem;
};
```

为了直观的展示一下`malloc_state`的情况，我们把`malloc_state`的内存列出如下。`0x804b0a8`是top chunk指针，`fastbinsY`位于`0xf7fac788`，长度为10，Unsorted bin，small bin和large bin都位于`bins`中，从`0xf7fac7b0`开始。

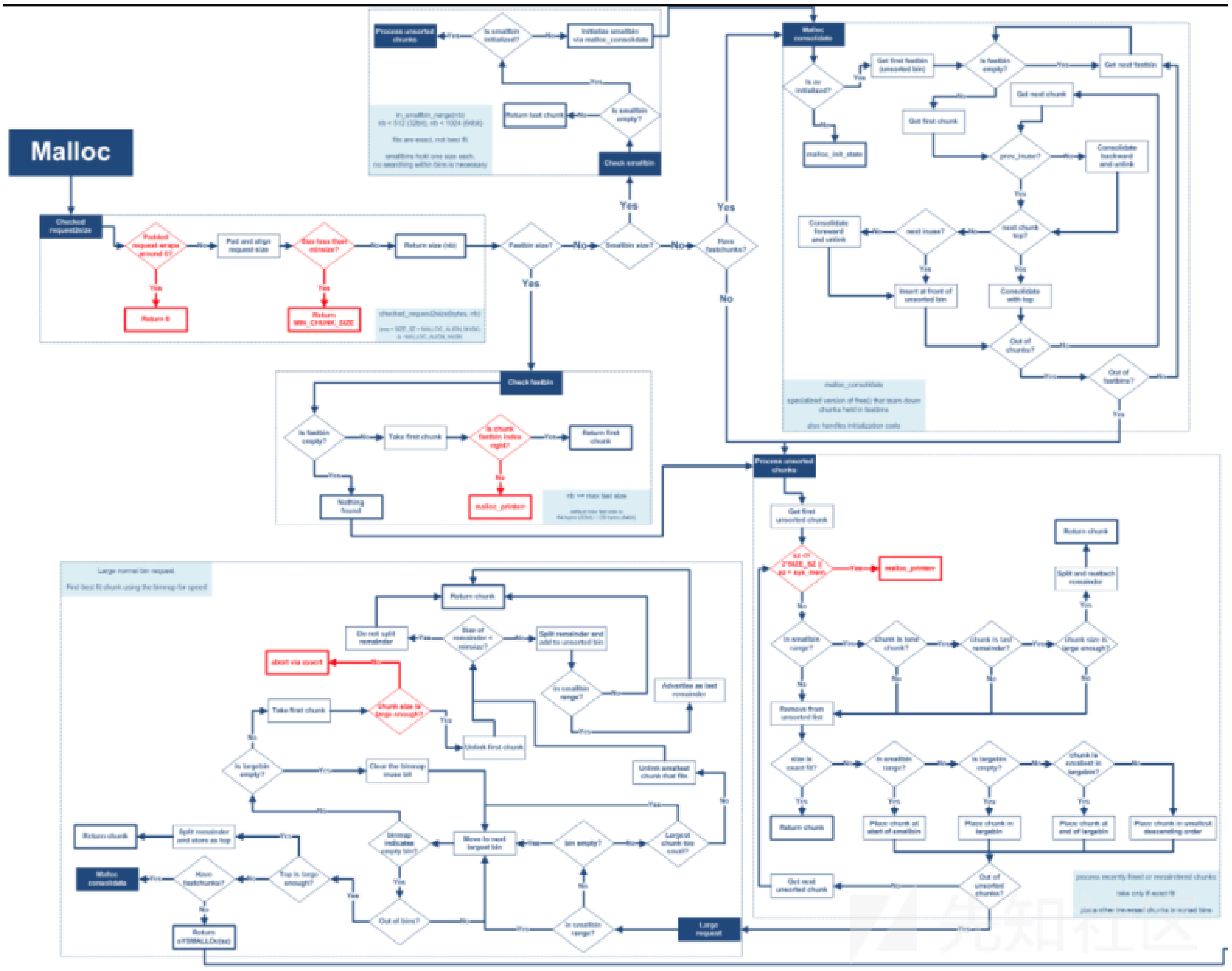
```
0xf7fac780: 0x00000000  0x00000001  0x00000000  0x00000000
0xf7fac790: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7a0: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7b0: 0x0804b0a8  0x00000000  0xf7fac7b0  0xf7fac7b0
0xf7fac7c0: 0xf7fac7b8  0xf7fac7b8  0xf7fac7c0  0xf7fac7c0
0xf7fac7d0: 0xf7fac7c8  0xf7fac7c8  0xf7fac7d0  0xf7fac7d0
0xf7fac7e0: 0xf7fac7d8  0xf7fac7d8  0xf7fac7e0  0xf7fac7e0
0xf7fac7f0: 0xf7fac7e8  0xf7fac7e8  0xf7fac7f0  0xf7fac7f0
0xf7fac800: 0xf7fac7f8  0xf7fac7f8  0xf7fac800  0xf7fac800
0xf7fac810: 0xf7fac808  0xf7fac808  0xf7fac810  0xf7fac810
```

ptmalloc 分配

这一章将会分配两个部分，第一个部分会基于`libc`的源码讨论`ptmalloc`当中的分配策略，比较无聊枯燥，第二部分会对第一部分的讨论的细节进行一个归纳。不感兴趣的

malloc内部

根据`malloc.c`的源代码，以及如下给出的malloc工作流，我会对每一个部分相应的代码在源代码等级进行展示。



malloc_init_state函数

```
#define NBINS             128
#define NSMALLBINS         64
#define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
#define SMALLBIN_CORRECTION (MALLOC_ALIGNMENT > 2 * SIZE_SZ)
#define MIN_LARGE_SIZE    ((NSMALLBINS - SMALLBIN_CORRECTION) * SMALLBIN_WIDTH)
#define in_smallbin_range(sz)  \
 ((unsigned long) (sz) < (unsigned long) MIN_LARGE_SIZE)

#ifndef DEFAULT_MXFAST
#define DEFAULT_MXFAST     (64 * SIZE_SZ / 4)
#endif

#define set_max_fast(s) \
 global_max_fast = (((s) == 0)                           \
                   ? SMALLBIN_WIDTH : ((s + SIZE_SZ) & ~MALLOC_ALIGN_MASK))
#define get_max_fast() global_max_fast


malloc_init_state (mstate av)
{
 int i;
 mbinptr bin;

 /* Establish circular links for normal bins */
 for (i = 1; i < NBINS; ++i)
   {
     bin = bin_at (av, i);
     bin->fd = bin->bk = bin;
   }
#if MORECORE_CONTIGUOUS
 if (av != &main_arena)
#endif
 set_noncontiguous (av);
 if (av == &main_arena)
   set_max_fast (DEFAULT_MXFAST);
 atomic_store_relaxed (&av->have_fastchunks, false);

 av->top = initial_top (av);
}
```

（原文该部分代码错误，已修正）

首先，`malloc_init_state`被调用来初始化`malloc_state`。在这个过程当中，每一个在`bins`中的元素的`fd`和`bk`指针将会被设置为他自己的指针，然后`global_max_f`

`malloc_consolidate`函数

`malloc_consolidate`将会尽量去合并`fastbins`中的块，并且把他们放到`unsorted bin`中去，在从`fastbin`获取到一个释放块后，合并序列如下：

1. 检查是否上一个邻接块为使用状态：如果不为使用状态，则将当前块合并到上一个邻接块
2. 检查是否邻接的下一个块为top chunk：如果是，则把当前块设置为`top chunk`，并且修改`chunk_size`，如果不是，则进入下一步
3. 检查是否邻接的下一块为使用状态：如果是，则直接把当前块放入`unsorted bin`，并且清除下一个块的P位。如果不是，则把下一块合并到当前块中，将当前块放入`unsorted bin`，清除当前下一邻接块的P位.

```
/*
 ------------------------- malloc_consolidate -------------------------

 malloc_consolidate is a specialized version of free() that tears
 down chunks held in fastbins.  Free itself cannot be used for this
 purpose since, among other things, it might place chunks back onto
 fastbins.  So, instead, we need to use a minor variant of the same
 code.

 Also, because this routine needs to be called the first time through
 malloc anyway, it turns out to be the perfect place to trigger
 initialization code.
*/

static void malloc_consolidate(mstate av)
{
 mfastbinptr*    fb;                 /* current fastbin being consolidated */
 mfastbinptr*    maxfb;              /* last fastbin (for loop control) */
 mchunkptr       p;                  /* current chunk being consolidated */
 mchunkptr       nextp;              /* next chunk to consolidate */
 mchunkptr       unsorted_bin;       /* bin header */
 mchunkptr       first_unsorted;     /* chunk to link to */

 /* These have same use as in free() */
 mchunkptr       nextchunk;
 INTERNAL_SIZE_T size;
 INTERNAL_SIZE_T nextsize;
 INTERNAL_SIZE_T prevsize;
 int             nextinuse;
 mchunkptr       bck;
 mchunkptr       fwd;

 /*
   If max_fast is 0, we know that av hasn't
   yet been initialized, in which case do so below
 */

 if (get_max_fast () != 0) {
   clear_fastchunks(av);

   unsorted_bin = unsorted_chunks(av);

   /*
     Remove each chunk from fast bin and consolidate it, placing it
     then in unsorted bin. Among other reasons for doing this,
     placing in unsorted bin avoids needing to calculate actual bins
     until malloc is sure that chunks aren't immediately going to be
     reused anyway.
   */

   maxfb = &fastbin (av, NFASTBINS - 1);
   fb = &fastbin (av, 0);
   do {
     p = atomic_exchange_acq (fb, NULL);
     if (p != 0) {
   do {
     check_inuse_chunk(av, p);
     nextp = p->fd;

     /* Slightly streamlined version of consolidation code in free() */
     size = chunksize (p);
     nextchunk = chunk_at_offset(p, size);
     nextsize = chunksize(nextchunk);

     if (!prev_inuse(p)) {
       prevsize = prev_size (p);
       size += prevsize;
       p = chunk_at_offset(p, -((long) prevsize));
       unlink(av, p, bck, fwd);
     }

     if (nextchunk != av->top) {
       nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

       if (!nextinuse) {
         size += nextsize;
         unlink(av, nextchunk, bck, fwd);
       } else
         clear_inuse_bit_at_offset(nextchunk, 0);

       first_unsorted = unsorted_bin->fd;
       unsorted_bin->fd = p;
       first_unsorted->bk = p;

       if (!in_smallbin_range (size)) {
         p->fd_nextsize = NULL;
         p->bk_nextsize = NULL;
       }

       set_head(p, size | PREV_INUSE);
       p->bk = unsorted_bin;
       p->fd = first_unsorted;
       set_foot(p, size);
     }

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

   } while ( (p = nextp) != 0);

     }
   } while (fb++ != maxfb);
 }
 else {
   malloc_init_state(av);
   check_malloc_state(av);
 }
}
```

__int_malloc函数

__libc_malloc是从bins或者从main arena里返回应用请求的块的函数，现在我们来讨论一下__int_malloc，也就是libc中malloc的内部实现

```
checked_request2size (bytes, nb);
```

分配器首先将需要的大小转换为了实际分配块的大小，然后尝试按照以下顺序去获取需要的块：fast bin, unsorted bin, small bin, large bin和top chunk。我们在这里一个一个讨论。

fastbin

```
/*
  If the size qualifies as a fastbin, first check corresponding bin.
  This code is safe to execute even if av is not yet initialized, so we
  can try it without checking, which saves some time on this fast path.
*/

if ((unsigned long) (nb) fd, victim))!= victim);
   if (victim != 0)
   {
        if (__builtin_expect (fastbin_index (chunksize (victim)) != idx, 0))
        {
             errstr = "malloc(): memory corruption (fast)";
           errout:
             malloc_printerr (check_action, errstr, chunk2mem (victim), av);
             return NULL;
        }
        check_remalloced_chunk (av, victim, nb);
        void *p = chunk2mem (victim);
        alloc_perturb (p, bytes);
        return p;
    }
}
```

如果大小比global_max_fast小，或等于，分配器会尝试去搜索fastbin来找适合的块，fastbin的index由块大小决定。

small bin

```
/*
  If a small request, check regular bin.  Since these "smallbins"
  hold one size each, no searching within bins is necessary.
  (For a large request, we need to wait until unsorted chunks are
  processed to find best fit. But for small ones, fits are exact
  anyway, so we can check now, which is faster.)
*/

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

   if ((victim = last (bin)) != bin)
     {
       if (victim == 0) /* initialization check */
         malloc_consolidate (av);
       else
       {
           bck = victim->bk;
       if (__glibc_unlikely (bck->fd != victim))
           {
               errstr = "malloc(): smallbin double linked list corrupted";
               goto errout;
           }
           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);
           void *p = chunk2mem (victim);
           alloc_perturb (p, bytes);
           return p;
        }
     }
 }
```

small bin的index由块大小决定，在被认为应当使用small bin时，分配器会尝试在small bin中移除掉第一个释放块。

unsorted bin

```
while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
{
    bck = victim->bk;
    if (__builtin_expect (chunksize_nomask (victim)  av->system_mem, 0))
       malloc_printerr (check_action, "malloc(): memory corruption",
                            chunk2mem (victim), av);
    size = chunksize (victim);

    /*
        If a small request, try to use last remainder if it is the
        only chunk in unsorted bin.  This helps promote locality for
        runs of consecutive small requests. This is the only
        exception to best-fit, and applies only when there is
        no exact fit for a small chunk.
    */

    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;
     }

     /* remove from unsorted list */
     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);
         check_malloced_chunk (av, victim, nb);
         void *p = chunk2mem (victim);
         alloc_perturb (p, bytes);
         return p;
      }
      /* 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)
             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 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;
                      fwd->bk_nextsize = victim;
                      victim->bk_nextsize->fd_nextsize = victim;
                  }
                  bck = fwd->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;

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

分配器会循环迭代unsorted bin，如果第一个块满足一下几个条件，这个块将会被分成一个请求大小的块，和一个剩余块。剩余块将会重新被插入到unsorted bin当中。

1. 请求大小在small 范围内
2. 这是`unsorted bin`当中唯一的一块
3. 这个块同事也是`last remainder`块
4. 切开之后的剩余大小足够大

如果`unsorted`块的大小正好为请求大小，直接返回这个块，否则，`unsorted bin`上的迭代将会继续检查`unsorted`块的状态：

1. 如果`unsorted`块为small范围，该块会被插入到相应的`small bin`，之后在下一个`unsorted`块中重复以上过程
2. 否则，如果`unsorted`块是large范围，且相应的`large bin`为空(bck == fwd)，`unsorted`块将会被直接插入到相应的`large bin`中，之后在下一个块中重复以上过程
3. 否则，如果`unsorted`块是large范围，且相应的`large bin`为非空，该块将按照大小降序插入到`large bin`当中

在所有的`unsorted`块都无法正好被当做返回值范围的时候，例如没有`unsorted`块，或者`small`块来提供请求的块时，分配器将会继续到下一步。

Large bin

```
/*
    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.
*/

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)) 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 (av, victim, bck, fwd);

             /* Exhaust */
        if (remainder_size fd;
         if (__glibc_unlikely (fwd->bk != bck))
             {
                   errstr = "malloc(): corrupted unsorted chunks";
                   goto errout;
             }
             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;
  }
}
```

如果在large bin中没有large块或者第一个large bin中large块的大小比请求大小要小，分配器会跳到下一步，否则分配器会尝试在当前large bin中找到一个块。

large块的搜索过程主要根据"最好适应"的原则，也就是找到最小的大小大于请求大小的块。在找到large块之后，将他从large bin中移除，然后计算切分后的剩余大小，如果剩余大小比MIN_SIZE小，直接将整块作为返回值，否则将当前块气氛，并且把剩余块插入到unsorted bin中。

top chunk 切分

```
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 ((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;
}
```

如果top chunk足够大，并且以上所有过程都没办法返回一个合适的块，top chunk会被切分成请求大小，之后将剩余大小重置为top chunk。

总结

首先我们总结一下malloc里常用的宏：

	x86	x86-64
SIZE_SZ	4	8
MIN_CHUNK_SIZE	16	32
MALLOC_ALIGNMENT	8	16
MALLOC_ALIGN_MASK	7	15
NBINS	128	128
NFASTBINS	10	10
NSMALLBINS	64	64
SMALLBIN_WIDTH	8	16
DEFAULT_MXFAST	64	128
MAX_FAST_SIZE	80	160
MIN_LARGE_SIZE	512	1024

在以上对于ptmalloc内部原理的讨论之后，我们给出一个ptmalloc中不同类型的bins是和组织和管理的总结。我们也会给出一些示例来展示一下这些bins的内存布局。

fast bin

1. fast bin中的块由单链表管理
2. fast bin中的块大小小于0x40
3. 当前块下一邻接块的P位不会被清除
4. 在从fastbin中取出块时，分配器遵循先进后出原则

```
#include
#include

int main()
{
   char *p1, *p2, *p3, *p4;

   p1 = malloc(0x20);
   p2 = malloc(0x20);
   p3 = malloc(0x20);
   p4 = malloc(0x20);

   free(p1);
   free(p2);
   free(p3);

   return 0;
}

/*
(gdb) x/20wx 0xf7fac780
0xf7fac780: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac790: 0x00000000  0x0804b050  0x00000000  0x00000000
0xf7fac7a0: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7b0: 0x0804b0a0  0x00000000  0xf7fac7b0  0xf7fac7b0
0xf7fac7c0: 0xf7fac7b8  0xf7fac7b8  0xf7fac7c0  0xf7fac7c0
(gdb) x/12wx 0x0804b050
0x804b050:  0x00000000  0x00000029  0x0804b028  0x00000000
0x804b060:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b070:  0x00000000  0x00000000  0x00000000  0x00000029
(gdb) x/12wx 0x0804b028
0x804b028:  0x00000000  0x00000029  0x0804b000  0x00000000
0x804b038:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b048:  0x00000000  0x00000000  0x00000000  0x00000029
(gdb) x/12wx 0x0804b000
0x804b000:  0x00000000  0x00000029  0x00000000  0x00000000
0x804b010:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b020:  0x00000000  0x00000000  0x00000000  0x00000029
*/
```

当下一个`malloc(0x20)`被调用时，分配器将会返回`0x804b058`(chunk2mem后的结果)给应用使用。

unsorted bin

1. `unsorted bin`中的chunk由双链表维护
2. `unsorted bin`中的chunk大小必须大于`0x40`。
3. 在分配时，分配器会迭代`unsorted bin`中的`unsorted`块，在找到合适的块之后，将其从`unsorted`块中取出，并且处理这个块

```
#include
#include

int main()
{
   char *p1, *p2, *p3, *p4;

   p1 = malloc(0xa0);
   p2 = malloc(0x30);
   p3 = malloc(0x100);
   p4 = malloc(0x30);

   free(p1);
   free(p3);

   return 0;
}

/*
(gdb) x/20wx 0xf7fac780
0xf7fac780: 0x00000000  0x00000001  0x00000000  0x00000000
```

```
0xf7fac790: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7a0: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7b0: 0x0804b220  0x00000000  0x0804b0e0  0x0804b000
0xf7fac7c0: 0xf7fac7b8  0xf7fac7b8  0xf7fac7c0  0xf7fac7c0
(gdb) x/20wx 0x0804b0e0
0x804b0e0:  0x00000000  0x00000109  0x0804b000  0xf7fac7b0
0x804b0f0:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b100:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b110:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b120:  0x00000000  0x00000000  0x00000000  0x00000000
(gdb) x/20wx 0x0804b000
0x804b000:  0x00000000  0x000000a9  0xf7fac7b0  0x0804b0e0
0x804b010:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b020:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b030:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b040:  0x00000000  0x00000000  0x00000000  0x00000000
*/
```

small bin

1. `small bin`中的chunk也由双链表维护
2. `small bin`中的大小必须小于`0x200`
3. 与`unsorted bin`不同，释放后的块不会在释放后插入`small bin`，只有`unsorted bin`中的切分块会被插入到`small bin`（更多的细节将在第二部分讨论）
4. 在从`small bin`中取出时，分配器遵循先进先出原则

```
#include
#include

int main()
{
   char *p1, *p2, *p3, *p4, *p5, *p6;

   p1 = malloc(0xa0);
   p2 = malloc(0x30);
   p3 = malloc(0xa0);
   p4 = malloc(0x30);
   p5 = malloc(0xa0);
   p6 = malloc(0x30);

   free(p1);
   free(p3);
   free(p5);

   malloc(0x50);
   malloc(0x50);
   malloc(0x50);

   return 0;
}

/*
(gdb) x/40wx 0xf7fac780
0xf7fac780: 0x00000000  0x00000001  0x00000000  0x00000000
0xf7fac790: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7a0: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7b0: 0x0804b2a0  0x0804b218  0x0804b218  0x0804b218
0xf7fac7c0: 0xf7fac7b8  0xf7fac7b8  0xf7fac7c0  0xf7fac7c0
0xf7fac7d0: 0xf7fac7c8  0xf7fac7c8  0xf7fac7d0  0xf7fac7d0
0xf7fac7e0: 0xf7fac7d8  0xf7fac7d8  0xf7fac7e0  0xf7fac7e0
0xf7fac7f0: 0xf7fac7e8  0xf7fac7e8  0xf7fac7f0  0xf7fac7f0
0xf7fac800: 0x0804b138  0x0804b058  0xf7fac800  0xf7fac800
0xf7fac810: 0xf7fac808  0xf7fac808  0xf7fac810  0xf7fac810
(gdb) x/20wx 0x0804b138
0x804b138:  0x00000000  0x00000051  0x0804b058  0xf7fac7f8
0x804b148:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b158:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b168:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b178:  0x00000000  0x00000000  0x00000000  0x00000000
(gdb) x/20wx 0x0804b058
```

```
0x804b058:  0x00000000  0x00000051  0xf7fac7f8  0x0804b138
0x804b068:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b078:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b088:  0x00000000  0x00000000  0x00000000  0x00000000
0x804b098:  0x00000000  0x00000000  0x00000000  0x00000000
*/
```

large bin

1. `large bin`中的块也由双链表维护
2. `large bin`中的块大小必须大于`0x200`
3. 除了`fwd`和`bck`指针以外，`large`块中还有`fd_nextsize`和`bck_nextsize`域用来表明`large`块中的不同大小（降序排列）
4. 与`small`块类似，释放后的`large`块不会被插入到`large bin`当中，只有从`unsorted bin`中气氛的块会被插入到`large bin`中。
5. 在从`large bin`中取出chunk时，分配器遵循■■■■原则，也就是找到比需求大小大的最小块

```
#include
#include

int main()
{
   char *p1, *p2, *p3, *p4, *p5, *p6, *p7, *p8;

   p1 = malloc(0x1000);
   p2 = malloc(0x30);
   p3 = malloc(0x1000);
   p4 = malloc(0x30);
   p5 = malloc(0x1000);
   p6 = malloc(0x30);
   p7 = malloc(0x1000);
   p8 = malloc(0x30);

   free(p1);
   free(p3);
   free(p5);
   free(p7);

   malloc(0x810);
   malloc(0x810);
   malloc(0x840);
   malloc(0x840);

   return 0;
}
/*
(gdb) x/200wx 0xf7fac780
0xf7fac780: 0x00000000  0x00000001  0x00000000  0x00000000
0xf7fac790: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7a0: 0x00000000  0x00000000  0x00000000  0x00000000
0xf7fac7b0: 0x0804f100  0x00000000  0x0804b848  0x0804b848
0xf7fac7c0: 0xf7fac7b8  0xf7fac7b8  0xf7fac7c0  0xf7fac7c0
0xf7fac7d0: 0xf7fac7c8  0xf7fac7c8  0xf7fac7d0  0xf7fac7d0
0xf7fac7e0: 0xf7fac7d8  0xf7fac7d8  0xf7fac7e0  0xf7fac7e0
0xf7fac7f0: 0xf7fac7e8  0xf7fac7e8  0xf7fac7f0  0xf7fac7f0
0xf7fac800: 0xf7fac7f8  0xf7fac7f8  0xf7fac800  0xf7fac800
0xf7fac810: 0xf7fac808  0xf7fac808  0xf7fac810  0xf7fac810
0xf7fac820: 0xf7fac818  0xf7fac818  0xf7fac820  0xf7fac820
0xf7fac830: 0xf7fac828  0xf7fac828  0xf7fac830  0xf7fac830
0xf7fac840: 0xf7fac838  0xf7fac838  0xf7fac840  0xf7fac840
0xf7fac850: 0xf7fac848  0xf7fac848  0xf7fac850  0xf7fac850
0xf7fac860: 0xf7fac858  0xf7fac858  0xf7fac860  0xf7fac860
0xf7fac870: 0xf7fac868  0xf7fac868  0xf7fac870  0xf7fac870
0xf7fac880: 0xf7fac878  0xf7fac878  0xf7fac880  0xf7fac880
0xf7fac890: 0xf7fac888  0xf7fac888  0xf7fac890  0xf7fac890
0xf7fac8a0: 0xf7fac898  0xf7fac898  0xf7fac8a0  0xf7fac8a0
0xf7fac8b0: 0xf7fac8a8  0xf7fac8a8  0xf7fac8b0  0xf7fac8b0
0xf7fac8c0: 0xf7fac8b8  0xf7fac8b8  0xf7fac8c0  0xf7fac8c0
0xf7fac8d0: 0xf7fac8c8  0xf7fac8c8  0xf7fac8d0  0xf7fac8d0
0xf7fac8e0: 0xf7fac8d8  0xf7fac8d8  0xf7fac8e0  0xf7fac8e0
0xf7fac8f0: 0xf7fac8e8  0xf7fac8e8  0xf7fac8f0  0xf7fac8f0
```

```
0xf7fac900: 0xf7fac8f8  0xf7fac8f8  0xf7fac900  0xf7fac900
0xf7fac910: 0xf7fac908  0xf7fac908  0xf7fac910  0xf7fac910
0xf7fac920: 0xf7fac918  0xf7fac918  0xf7fac920  0xf7fac920
0xf7fac930: 0xf7fac928  0xf7fac928  0xf7fac930  0xf7fac930
0xf7fac940: 0xf7fac938  0xf7fac938  0xf7fac940  0xf7fac940
0xf7fac950: 0xf7fac948  0xf7fac948  0xf7fac950  0xf7fac950
0xf7fac960: 0xf7fac958  0xf7fac958  0xf7fac960  0xf7fac960
0xf7fac970: 0xf7fac968  0xf7fac968  0xf7fac970  0xf7fac970
0xf7fac980: 0xf7fac978  0xf7fac978  0xf7fac980  0xf7fac980
0xf7fac990: 0xf7fac988  0xf7fac988  0xf7fac990  0xf7fac990
0xf7fac9a0: 0xf7fac998  0xf7fac998  0xf7fac9a0  0xf7fac9a0
0xf7fac9b0: 0xf7fac9a8  0xf7fac9a8  0xf7fac9b0  0xf7fac9b0
0xf7fac9c0: 0xf7fac9b8  0xf7fac9b8  0xf7fac9c0  0xf7fac9c0
0xf7fac9d0: 0xf7fac9c8  0xf7fac9c8  0xf7fac9d0  0xf7fac9d0
0xf7fac9e0: 0xf7fac9d8  0xf7fac9d8  0xf7fac9e0  0xf7fac9e0
0xf7fac9f0: 0xf7fac9e8  0xf7fac9e8  0xf7fac9f0  0xf7fac9f0
0xf7faca00: 0xf7fac9f8  0xf7fac9f8  0xf7faca00  0xf7faca00
0xf7faca10: 0xf7faca08  0xf7faca08  0xf7faca10  0xf7faca10
0xf7faca20: 0xf7faca18  0xf7faca18  0xf7faca20  0xf7faca20
0xf7faca30: 0xf7faca28  0xf7faca28  0xf7faca30  0xf7faca30
0xf7faca40: 0xf7faca38  0xf7faca38  0xf7faca40  0xf7faca40
0xf7faca50: 0xf7faca48  0xf7faca48  0xf7faca50  0xf7faca50
0xf7faca60: 0xf7faca58  0xf7faca58     {0x0804c858******0x0804e908}
0xf7faca70: 0xf7faca68  0xf7faca68  0xf7faca70  0xf7faca70
0xf7faca80: 0xf7faca78  0xf7faca78  0xf7faca80  0xf7faca80
0xf7faca90: 0xf7faca88  0xf7faca88  0xf7faca90  0xf7faca90

(gdb) x/20wx 0x804c858
0x804c858:  0x00000000  0x000007f1  0x0804d898  0xf7faca60
0x804c868:  0x0804e908  0x0804e908  0x00000000  0x00000000
0x804c878:  0x00000000  0x00000000  0x00000000  0x00000000
0x804c888:  0x00000000  0x00000000  0x00000000  0x00000000
0x804c898:  0x00000000  0x00000000  0x00000000  0x00000000
(gdb) x/20wx 0x804d898
0x804d898:  0x00000000  0x000007f1  0x0804e908  0x0804c858
0x804d8a8:  0x00000000  0x00000000  0x00000000  0x00000000
0x804d8b8:  0x00000000  0x00000000  0x00000000  0x00000000
0x804d8c8:  0x00000000  0x00000000  0x00000000  0x00000000
0x804d8d8:  0x00000000  0x00000000  0x00000000  0x00000000
(gdb) x/20wx 0x804e908
0x804e908:  0x00000000  0x000007c1  0xf7faca60  0x0804d898
0x804e918:  0x0804c858  0x0804c858  0x00000000  0x00000000
0x804e928:  0x00000000  0x00000000  0x00000000  0x00000000
0x804e938:  0x00000000  0x00000000  0x00000000  0x00000000
0x804e948:  0x00000000  0x00000000  0x00000000  0x00000000
*/
```

注明

原文中多处小错误已经修改。