



首页
最新文章
经典回顾
开发
设计
IT技术
职场
业界
极客
创业
访谈
在国外

- 导航条 -

[伯乐在线](#) > [首页](#) > [所有文章](#) > [IT技术](#) > 理解 glibc 内存分配器的机制与实现

理解 glibc 内存分配器的机制与实现

2016/08/03 · [IT技术](#) · [glibc](#), [内存分配器](#)

分享到: 9

[Unity 3D地形编辑器
使用wordpress极速建站](#)[MySQL5.7复制功能实战
神奇的JpGraph类库](#)

本文由 [伯乐在线](#) - [巽离](#) 翻译，[黄利民](#) 校稿。未经许可，禁止转载！
英文出处：[sploitfun](#)。欢迎加入[翻译组](#)。

一直以来，我都沉迷于堆内存的一切。脑子里一直充斥着这些问题：

- 怎样从内核获取堆内存？
- 内存管理的效率如何？
- 是什么在管理它？内核？库函数？还是应用程序？
- 堆内存可以扩展吗？

直到最近我才有时间思考并理解这些问题。所以想在本文和大家分享我的思考过程。除了我们即将讨论的内存分配器，还有如下几种的存在：

- dlmalloc - General purpose allocator
- ptmalloc2 - glibc
- jemalloc - FreeBSD and Firefox
- tcmalloc - Google
- libumem - Solaris ...

每个内存分配器都说自己可以快速分配内存、可扩展而且高效。但并不是所有的分配器都适用于我们的应用程序。像那些对内存异常渴求的应用程序来说，它的性能很大程度上依赖于内存分配器的性能。在这篇文章中，我打算只介绍 ‘glibc malloc’ 内存分配器。希望未来有机会可以介绍其他几种。为了帮助大家深入理解 ‘glibc malloc’，本文会贴一些源代码。现在，准备好了吗？开始 glibc malloc 吧！

历史

ptmalloc2 是 dlmalloc 的分支。于 2006 年发布，在这条分支上，添加了对线程的支持。在正式版发布后，ptmalloc2 被集成到 glibc 的源代码中。在这之后，对这个内存分配器的修改直接在 glibc malloc 的源码中进行。今后，ptmalloc2 和 glibc 的 malloc 之间可能差异会越来越大。

系统调用：从本文的分析中，我们会发现 malloc 内部要么调用 brk，要么调用 mmap。

线程化：

在早期的 linux 中，将 dlmalloc 作为默认的内存分配器。不过由于 ptmalloc2 对线程的支持，它便取代了 dlmalloc 的地位。线程

化可以提高内存分配器的性能从而提高应用程序的性能。在 `dlmalloc` 中，如果两个线程同时调用 `malloc`，由于数据结构 `freelist` 在所有线程间共享，只有一个线程可以进入临界区。这样导致了在多线程的应用程序中，`malloc` 会很浪费时间，从而降低应用程序的性能。而在 `ptmalloc2` 中，当两个线程同时调用 `malloc` 就不会出现这种窘境，因为每个线程都有自己独立的 `heap` 段，管理 `heap` 的 `freelist` 结构也是相互独立的，从而不管多少线程同时请求内存，都会立即分配完成。这种每个线程都有独立 `heap`、独立 `freelist` 的行为称为“per thread arena”。

例子

```
1  /* Per thread arena example. */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  #include <unistd.h>
6  #include <sys/types.h>
7
8  void* threadFunc(void* arg) {
9      printf("Before malloc in thread 1n");
10     getchar();
11     char* addr = (char*) malloc(1000);
12     printf("After malloc and before free in thread 1n");
13     getchar();
14     free(addr);
15     printf("After free in thread 1n");
16     getchar();
17 }
18
19 int main() {
20     pthread_t t1;
21     void* s;
22     int ret;
23     char* addr;
24
25     printf("Welcome to per thread arena example::%dn",getpid());
26     printf("Before malloc in main threadn");
27     getchar();
28     addr = (char*) malloc(1000);
29     printf("After malloc and before free in main threadn");
30     getchar();
31     free(addr);
32     printf("After free in main threadn");
33     getchar();
34     ret = pthread_create(&t1, NULL, threadFunc, NULL);
35     if(ret)
36     {
37         printf("Thread creation errorn");
38         return -1;
39     }
40     ret = pthread_join(t1, &s);
41     if(ret)
42     {
43         printf("Thread join errorn");
44         return -1;
45     }
46     return 0;
47 }
```

输出分析：

在 `main` 线程 `malloc` 之前：从下面的输出我们可以看出，由于此时还没有创建线程 `thread1`，从而没有 `heap` 段，也没有线程栈。

```
1  sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
2  Welcome to per thread arena example::6501
3  Before malloc in main thread
4  ...
5  sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
6  08048000-08049000 r-xp 00000000 08:01 539625      /home/sploitfun/ptmalloc.ppt/mthread/mthread
7  08049000-0804a000 r--p 00000000 08:01 539625      /home/sploitfun/ptmalloc.ppt/mthread/mthread
8  0804a000-0804b000 rw-p 00001000 08:01 539625      /home/sploitfun/ptmalloc.ppt/mthread/mthread
9  b7e05000-b7e07000 rw-p 00000000 00:00 0
10 ...
11 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

在 `main` 线程 `malloc` 后：从下面输出能看出，`heap segment` 已经创建了，并且就处于 `data segment(0804b000-0806c000)` 之上。这就意味着：`heap` 内存是通过增加程序中断点(program break location)来创建的(例如，系统调用 `brk`)。同时请注意，尽管程序中只申请 1000字节的内存，创建的 `heap` 内存却是 132KB。这一段连续的 `heap` 内存被称为“arena”。由于它是在 `main` 线程中创建，所以也被称为“main arena”。在这之后的内存请求都会使用这一块“arena”，直到消耗完这片空间。当 `arena` 耗完，可以通过增加程序中断点的方式来增加 `arena`(在增加之后，`top chunk` 的大小也会随之调整)。与之类似，如果 `top chunk` 中有过多的空闲空间，`arena` 的区域也会收缩。

注意：所谓“top chunk”指的是“arena”最顶部的内存块。想要了解更多知识，请看下面“Top Chunk”的内容。

```
1 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
2 Welcome to per thread arena example::6501
3 Before malloc in main thread
4 After malloc and before free in main thread
5 ...
6 sploitfun@sploitfun-VirtualBox:~/lsplotts/hof/ptmalloc.ppt/mthread$ cat /proc/6501/maps
7 08048000-08049000 r-xp 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
8 08049000-0804a000 r--p 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
9 0804a000-0804b000 rw-p 00001000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
10 0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
11 b7e05000-b7e07000 rw-p 00000000 00:00 0
12 ...
13 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

Main 线程 free 内存后：从下面输出可以看出，当我们在程序中释放已分配的内存后，实际上这块内存并没有立即还给操作系统。这块已分配的内存(大小为 1000 字节)仅仅是释放给了‘glibc malloc’的库，这个库将这块空闲出来的内存块添加到“main arena bin”中(在 glibc malloc 中，freelist 结构被认为是一个个容器 bins)。在这之后，一旦用户请求内存，‘glibc malloc’不再从 kernel 请求 heap 内存，而是从这些 bin 中找到空闲的内存块分配出去。只有当找不到空闲的内存块时，它才会从 kernel 请求新的内存。

```
1 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
2 Welcome to per thread arena example::6501
3 Before malloc in main thread
4 After malloc and before free in main thread
5 After free in main thread
6 ...
7 sploitfun@sploitfun-VirtualBox:~/lsplotts/hof/ptmalloc.ppt/mthread$ cat /proc/6501/maps
8 08048000-08049000 r-xp 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
9 08049000-0804a000 r--p 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
10 0804a000-0804b000 rw-p 00001000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
11 0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
12 b7e05000-b7e07000 rw-p 00000000 00:00 0
```

Thread1 线程 malloc 之前：从下面的输出可以看出，此时没有 thread1 的 heap segment，但是线程栈已经创建了。

```
1 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
2 Welcome to per thread arena example::6501
3 Before malloc in main thread
4 After malloc and before free in main thread
5 After free in main thread
6 Before malloc in thread 1
7 ...
8 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
9 08048000-08049000 r-xp 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
10 08049000-0804a000 r--p 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
11 0804a000-0804b000 rw-p 00001000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
12 0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
13 b7604000-b7605000 ---p 00000000 00:00 0
14 b7605000-b7e07000 rw-p 00000000 00:00 0 [stack:6594]
15 ...
16 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

Thread1 线程 malloc 之后：从下面输出可以看出，thread1 的 heap segment 也已经创建了。并且就处于内存映射段(b7500000-b7521000 大小为 132 Kb)中，由此我们可以得出结论：不同于 main 线程，这里的 heap 内存是通过系统调用 mmap 来创建的。同样，即便用户只请求了 1000 字节，实际映射到进程地址空间的大小为 1M。在这 1M 内存中，只有 132K 的内存被赋予了读写权限，作为该线程的 heap 内存。这一块连续的内存块被称为“thread arena”。

注意：如果用户请求内存大小超过 128K (比如 malloc(132*1024))，并且单个 arena 无法满足用户需求的情况下，不管这个内存请求来自 main arena 还是 thread arena，都会采用 mmap 的方式来分配内存。

```
1 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
2 Welcome to per thread arena example::6501
3 Before malloc in main thread
4 After malloc and before free in main thread
5 After free in main thread
6 Before malloc in thread 1
7 After malloc and before free in thread 1
8 ...
9 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
10 08048000-08049000 r-xp 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
11 08049000-0804a000 r--p 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
12 0804a000-0804b000 rw-p 00001000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
13 0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
14 b7500000-b7521000 rw-p 00000000 00:00 0
15 b7521000-b7600000 ---p 00000000 00:00 0
```

```

16 b7604000-b7605000 ---p 00000000 00:00 0
17 b7605000-b7e07000 rw-p 00000000 00:00 0 [stack:6594]
18 ...
19 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$

```

Thread1 中释放内存之后：同样，这块内存实际上也并没有还给操作系统。而是被转交给了‘glibc malloc’，从而被添加到 thread arena bin 的空闲块中。

```

1 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
2 Welcome to per thread arena example::6501
3 Before malloc in main thread
4 After malloc and before free in main thread
5 After free in main thread
6 Before malloc in thread 1
7 After malloc and before free in thread 1
8 After free in thread 1
9 ...
10 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
11 08048000-08049000 r-xp 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
12 08049000-0804a000 r--p 00000000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
13 0804a000-0804b000 rw-p 00001000 08:01 539625 /home/sploitfun/ptmalloc.ppt/mthread/mthread
14 0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
15 b7500000-b7521000 rw-p 00000000 00:00 0
16 b7521000-b7600000 ---p 00000000 00:00 0
17 b7604000-b7605000 ---p 00000000 00:00 0
18 b7605000-b7e07000 rw-p 00000000 00:00 0 [stack:6594]
19 ...
20 sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$

```

Arena 的数量：从上面几个例子中，我们可以看到主线程有 main arena，线程1 有它自己的 thread arena。那么问题来了，我们是否可以不管线程个数多少，都将线程与 arena 的个数做一一映射呢？答案是 NO！某些应用程序可能有很多线程（大于 CPU 个数），在这种情况下，如果我们给每个线程配一个 arena 简直是自作孽，而且很没意义。所以呢，我们应用程序的 arena 数目受制于系统中 CPU 个数。

对 32 位系统:

arena 个数 = 2 核心个

对64 位系统: arena 个数 = 8 核心个数

例如：假设有个多线程程序（4 个线程 —— 主线程 + 3 个用户线程）跑在 32 位的单核系统上，线程数大于 2 乘以核心个数（2*1 = 2）。碰到这种情况，glibc malloc 会保证所有可用的线程间共享多个 arena。但是这种共享是怎么实现的呢？

在主线程中，当我们第一次调用 malloc 函数会创建 main arena，这是毋庸置疑的。

在线程1、线程2 中第一次调用 malloc 函数后，会分别给他俩创建各自的 arena。直到线程和 arena 达到了一对一映射。

在线程3中，第一次调用 malloc 函数，这时候会计算是否到了 arena 最大数。在这个例子中已经达到限额了，因此会尝试复用其他已存在的 arena（主线程或线程1 或线程2的arena）。

复用：遍历所有可用的 arena，并尝试给 arena 枷锁。

如果成功枷锁（假设 main arena 被锁住），将这个 arena 返回给用户。

如果没有找到空闲的 arena，那么就会堵塞在当前 arena 上。当线程 3 第二次调用 malloc，malloc 会尝试使用可访问 arena (main arena)。如果 main arena 空闲，直接使用。否则线程3 会被堵塞，直到 main arena 空闲。这时候 main arena 就在主线程和线程 3 之间共享啦！

下面三个主要数据结构可以在 ‘glibc malloc’ 源码中找到：

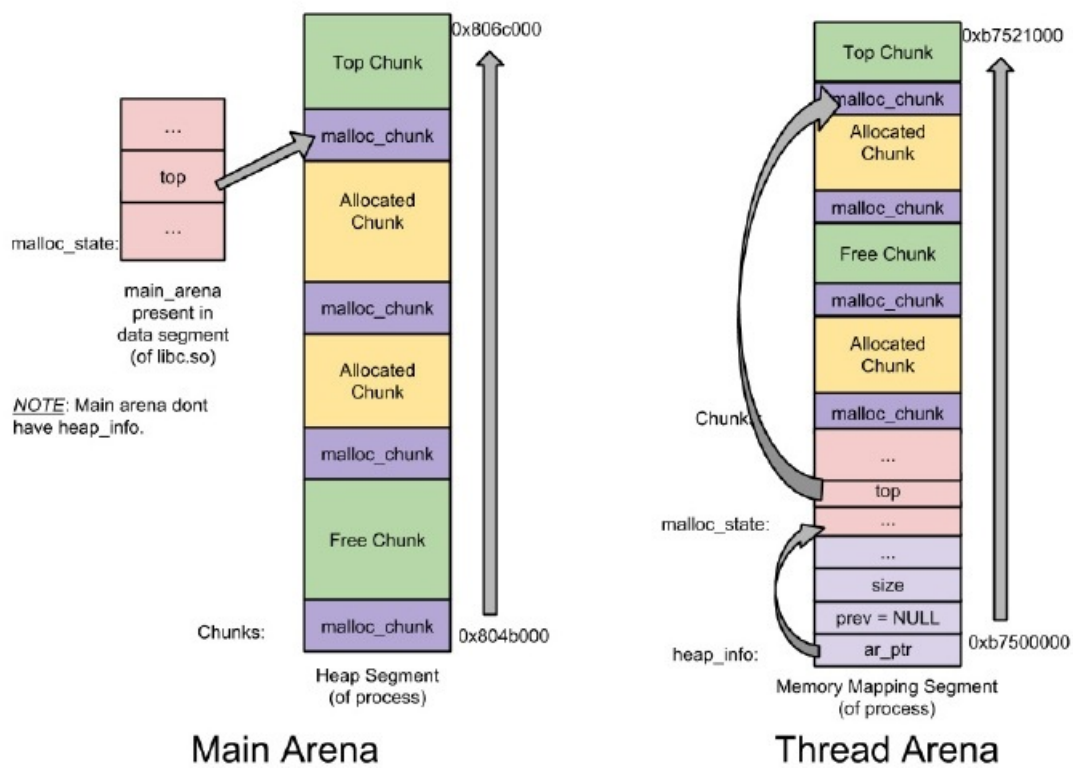
Heap_info - Heap Header - 每个线程 arena 都可以有多个 heap 段。每个 heap 段都有自己的 header。为什么需要多个 heap 段呢？一开始的时候，每个线程都只有一个 heap 段的，但是渐渐的，堆空间用光光了，就会调用 mmap 来获取新的 heap 段（非连续区域）。

Mallac_state - Arena Header - 每个线程 arena 可以有多个堆，但是，所有这些堆只能有一个 arena header。Arena header 结构中包含 bin、top chunk、last reminder chunk 等。

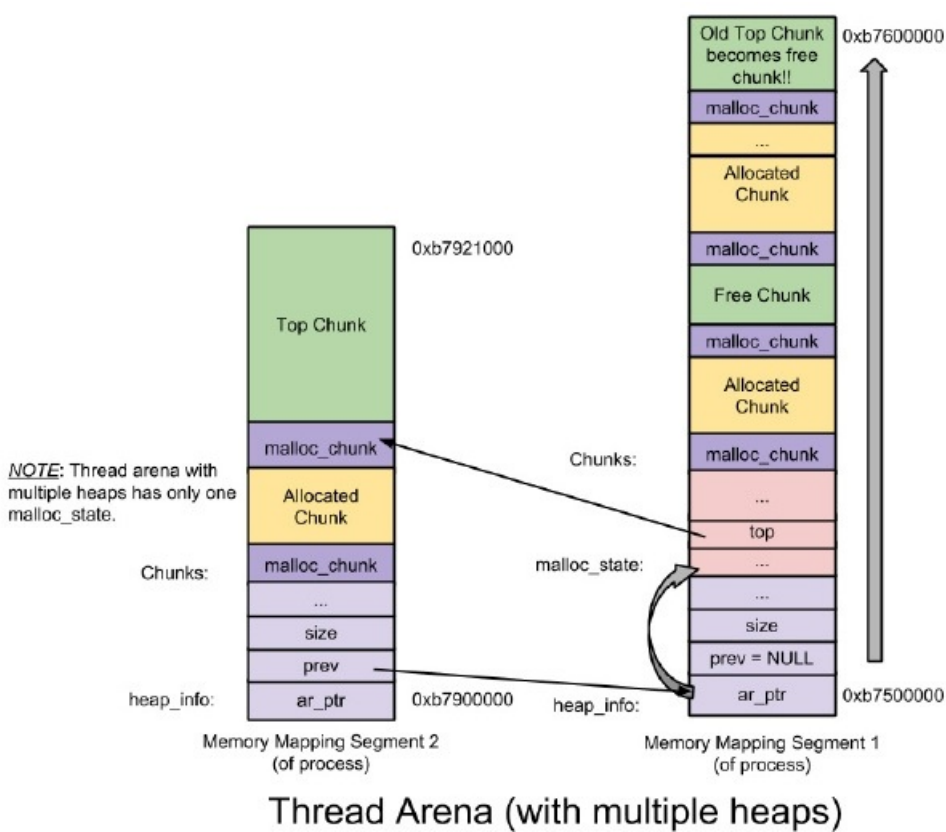
mallc_chunk - Chunk Header - 由于用户的内存请求，堆会被分割成许多块（chunk）。每个这样的块有着自己的 chunk header。

注意：Main arena 没有多个 heap 段，也因此没有 heap_info 结构。当 main arena 中堆空间耗尽了，就会调用 sbrk 来扩展堆空间（连续区域），直到 heap 段顶端达到了内存映射段。不同于线程 arena，main arena 的 arena header 结构不属于 sbrk 的 heap 段。它是一个全局变量，因此我们可以在 libc.so 的数据段中看到它的身影。

下图很形象的展示了 main arena 和线程 arena 的结构（单个 heap 段）：



下图展示了线程 arena 的结构（多个 heap 段）：



Chunk

heap 段中的块可以是以下几种类型：

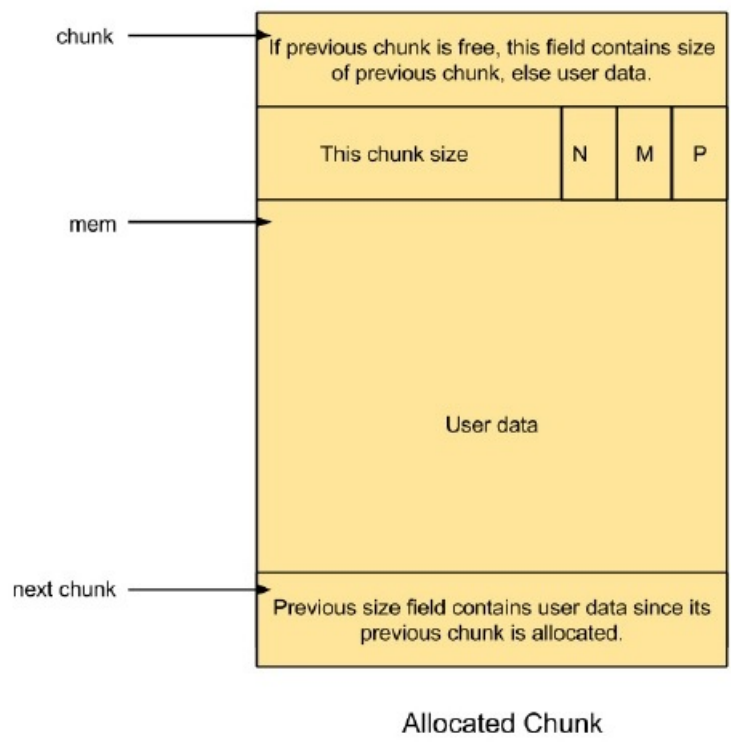
已分配的内存块

空闲内存块

顶层块

最终提示块

Allocated chunk:



Prev_size: 如果前一个内存块是空闲的，这个字段保存前一个内存块的大小。否则说明前一个内存块已经分配出去了，这个字段保存前一个内存块的用户数据。

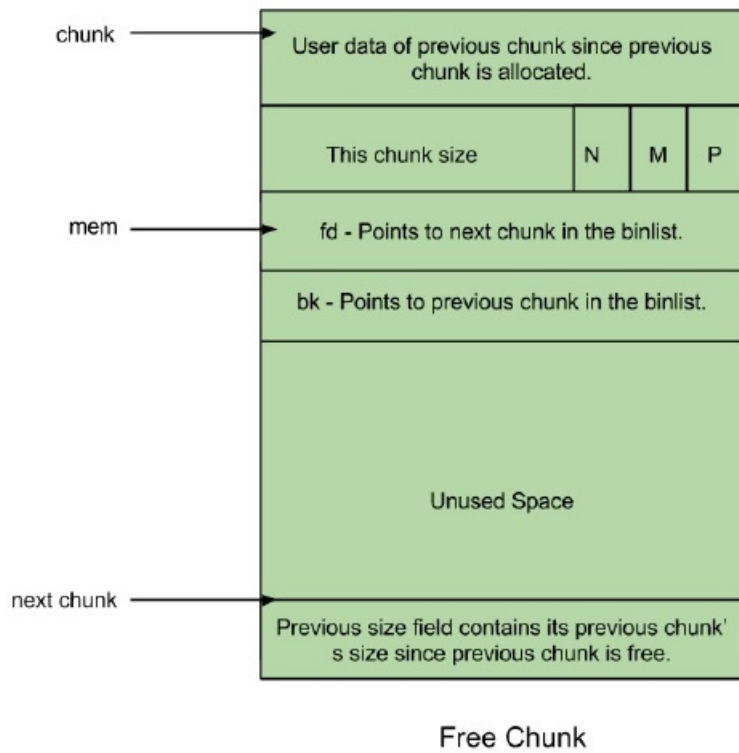
Size: 这个字段保存已分配内存块的大小。最后三个位包含标志信息。

- PREV_USE (P) - 前块内存已分配
 - IS_MAPPED (M) - 内存块通过 mmap 申请
 - NON_MAIN_ARENA (N) - 内存块属于线程 arena
- NOTE:

Malloc_chunk 的其他字段（例如 fd, bk）不存在于已分配内存块。因此这种内存块的用户数据就保存在这些字段中。

为了存储 malloc_chunk 结构，还有出于内存对齐的目的，我们需要额外的空间，因此将用户请求的内存大小转化为可用大小（内部表示形式）。转化方式为：可用大小的后三位不置位，用于存储标志信息。

Free Chunk:



Prev_size : 要知道两块空闲内存块不可能相邻。一旦两个内存块都空闲，它们就会被整合成一整个空闲内存块。也就是说，我们这块空闲内存块的前一个内存块肯定是已分配的，因此， prev_size 字段保存的是前一个内存块的用户数据。

Size: 这个字段保存当前空闲内存块的大小。

fd: 前向指针 - 指向同一容器中的下一块内存块（注意是容器，这里并不是物理内存上的下一个 chunk）。

Bk: 后向指针 - 指向同一容器中的前一个内存块（同上）。

Bins:

容器（bins）： 容器指的是空闲链表结构 freelist。用于管理多个空闲内存块。根据块大小的不同，容器可以分为：

用于保存这些容器的数据结构有：

fastbinsY： 这个数组用于保存 fast bins。

Bins： 这个数组用于后三种容器。一共有 126 个容器，划分为三组：

Bin 1 - Unsorted bin

Bin 2 to Bin 63 - Small bin

Bin 64 to Bin 126 - Large bin

Fast Bin：

大小为 16~80字节的内存块称为 fast chunk。保存这些 fast chunk 的容器就是 fast bins。在所有这些容器中， fast bins 操作内存效率高。

Number of bins - 10

容器 bin 的个数 - 10 每个 fast bin 都包含一个空闲内存块的单链表。之所以采用单链表，是因为 fast bins 中内存块的添加、删除操作都在列表的头尾端进行，不涉及中间段的移除操作 —— 后进先出。

Chunk size - 8 bytes apart

Chunk 大小 - 以 8 字节划分 Fast bins 包含一条链表 binlist，以 8 字节对齐。例如，第一个 fast bin（索引为0）包含一条链表，它的内存块大小为 16字节，第二个 fast bin 中链表上的内存块大小为 24 字节，以此类推。

每个 fast bin 中的内存块大小是一致的。

在 malloc 初始化期间， fast bin 最大值设置为 64 字节（不是80字节）。因此，默认情况下16~64 字节的内存块都被视为 fast chunks。

不存在合并 ——在 fast bins 中允许相邻两块都是空闲块，不会被自动合并。虽然不合并会导致内存碎片，但是它大大加速了 free 操作！！

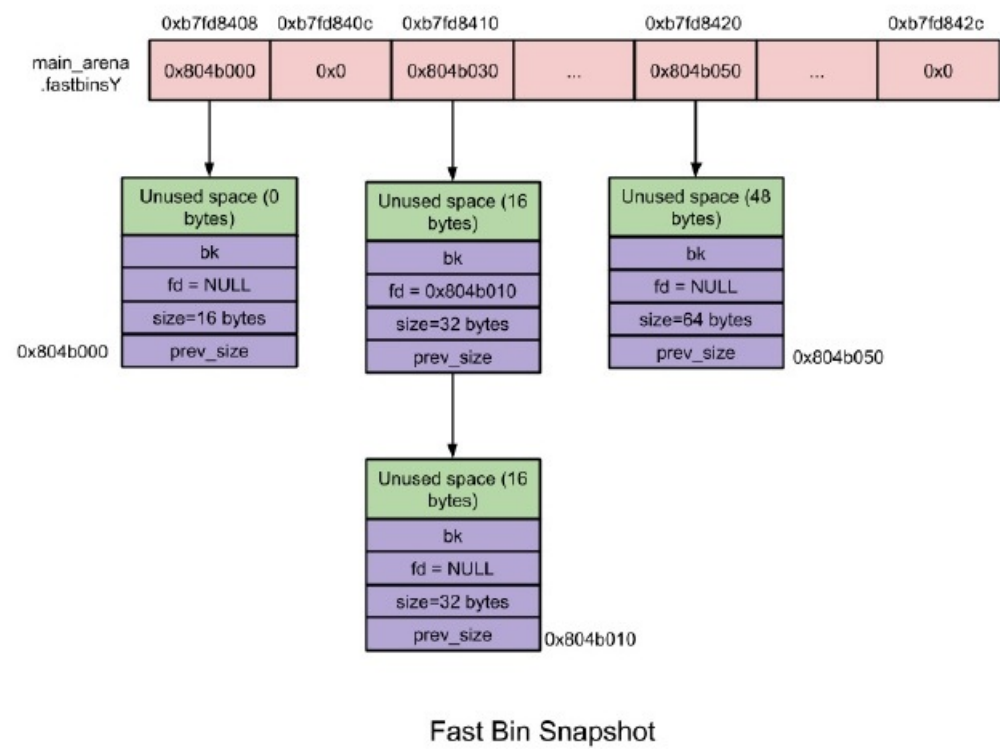
Malloc(fast chunk) - 一开始 fast bin 最大值和fast bin 都是空的，所以不管用户请求的块大小怎样，都不会走到 fast bin 的代码，而是 small bin code。

之后，由于内存释放等原因，fast bin 不为空，通过fast bin 索引来找到对应的 binlist。

最后，binlist 中的第一块内存块从链表中移除并返回给用户。

根据 fast bin 索引值找到对应的 binlist

这块待释放的内存块添加到上面 binlist 的链表头。



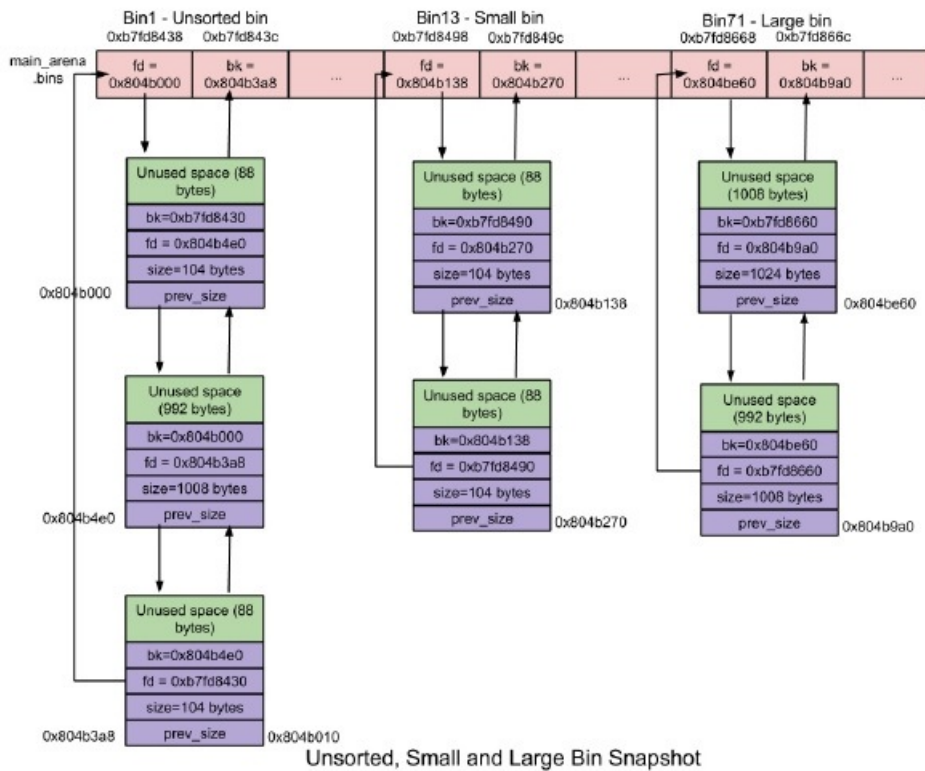
Unsorted Bin:

当释放small chunk 或者 large chunk时，不会将它们添加到 small bin 或者 large bin 中，而是添加到 unsorted bin。这种解决方案让‘glibc malloc’多了一种重复利用空闲内存块的方式。因为不需要花时间去各自的容器中去，从而也提高了分配和释放内存的效率。

Number of bins - 1

Unsorted bin 中包含空闲内存块的环形双向链表。

内存块大小 —— 无大小限制，任意大小的内存块都可以。



Unsorted, Small and Large Bin Snapshot

Small Bin:

小于 512 字节的内存块称为 small chunk。容纳这类 small chunk 的称为 small bins。在内存分配和释放时，Small bins 的处理速度优于 large bins（但是低于 fast bins）。

Number of bins – 62

Bins 数目 – 62

每个 small bin 都包含一条环形双向链表（binlist）。之所以使用双向链表是因为内存块需要在链表中间解引用。在链表头添加内存块，在链表尾端删除（FIFO）。

Chunk Size – 8 bytes apart

内存块大小 —— 以 8 字节划分

Small bin 也包含一条内存块链表 binlist，链表中每个内存块按照 8 字节对齐。例如：第一个 small bin（bin 2），它的链表中内存块的大小为 16 字节，第二个 small bin（bin 3），它的链表中内存块大小为 24 字节，以此类推。每个 small bin 中的内存块大小一致，因此无需排序。

内存块合并 —— 相邻的两块内存块如果都是空闲的话会合并为一整块。合并会消除内存碎片，但是很显然会降低内存释放的效率。

Malloc(small chunk) —— 起初，所有的 small bins 都是 NULL，因此即使用户请求的是 small chunk，也不会走到 small bin 的代码，而是 unsorted bin 提供服务。同样，第一次调用 malloc 时，会初始化 malloc state 结构中的 small bin 和 large bin (bins)。比如，bins 可以指向自身，表明它是空的。之后当 small bin 不为空，malloc 会从 binlist 中移除最后一块内存块并返回给用户。

Free(small chunk) —— 释放 small chunk 时，首先检查他的前后内存块是否为空闲，如果空闲的话，需要合并内存块。例如，将内存块从对应的链表中解引用，将合并后的大内存块添加到 unsorted bin 链表头。

Large Bin:

大于或等于 512b 的内存块就是 large chunk。保存这类内存块的容器称为 large bins。在内存分配和释放时，large bins 效率低于 small bins。

Number of bins – 63

每个 large bin 都包含一条环形双向链表（也称为 binlist）。因为 large bin 中内存块可能在任何位置添加删除。

在这 63 个 bin 中：

其中有 32 个 bins，它们所包含的环形双向链表 binlist，其中每个内存块按照 64 字节划分。例如，第一个 large bin（bin 65）包含一条 binlist，它的内存块大小为 512b~568b，第二个 large bin（bin 66），它的链表中每个内存块的大小为 576b~632b，以此类推。

有16个 bins ，它的 binlist 中内存块大小为 512 字节。

有 8 个 bins ，它们的 binlist 中内存块大小为 4096 字节

有 4 个 bins 的 binlist 大小为 32768 字节

有两个 bins ，它的 binlist 中内存块大小为 262144 字节

唯一有一个 bin ，它的内存块就是剩下的所有可用内存。

与 small bin 不同的是， large bin 中的内存块大小并不是完全一样的。降序存放，最大的内存块保存在 binlist 的 front 端，最小的保存在 rear 端。

合并 —— 两个空闲内存块不可相邻，会合并为单个空闲快。

Large bins 初始化为空，因此即使用户请求 large chunk，也不会走到 large bin 的代码，而是走到下一个 largest bin代码【这部分代码很多，按顺序排下来，在 _int_malloc 函数中，先转化用户请求size，根据size 判断是哪一种请求，接下来依次为 fast bin、small bin、 large bin、以及更大的 large request 等等】。

同样，在我们第一次调用 malloc 期间， malloc_state 结构中的 small bin 和 large bin 结构也会被初始化，比如指针指向它们自身以表明为空。

在这之后的请求，如果 large bin 不为空，且最大内存块大于用户请求的 size，这时候会从尾端到前端遍历 binlist，直到找到一块大小等于或者约等于用户请求的内存块。一旦找到，这个内存块会被分割为两块：

其一：用户块（用户请求的实际大小）——返回给用户

其二：剩余块（找到的内存块减去用户请求大小后剩下的部分）——添加到 unsorted bin。

如果最大内存块小于用户请求大小，malloc 会尝试下一个 largest bin（非空的）。下一块 largest bin 代码会扫描 binmaps，去寻找下一个非空的 largest bin，一旦找到 bin，就会检索这个 bin 中 binlist 的内存块，分割内存块并返回给用户。如果找不到的话，就会接着使用 top chunk，力争解决用户需求。

Free(large chunk) —— 这个释放的过程类似于 free(small chunk)。

Top Chunk:

Arena 中最顶部的内存块就是 top chunk。它不属于任何 bin。如果所有的 bins 中都没有空闲内存块，就会使用 top chunk 来服务用户。如果 top chunk 大于用户实际请求的 size，就会被分割为两块：

用户块（大小等于用户实际请求）

剩余块（剩下的那部分）

这样，剩余块就是新的 top chunk。如果 top chunk 小于用户请求块大小，sbrk (main arena)或者 mmap (thread arena)系统调用就会用来扩展 top chunk。

Last Remainder Chunk:

最近一次分割产生的剩余块就是 last reminder chunk。它可以用来提高访问局部性，比如，连续的小内存块 malloc 请求可能会使得这些分配的内存块离的很近。

但是 arena 中可用的内存块如此之多，哪块有资格可以成为 last reminder chunk 呢？

当用户请求小内存块时，如果无法从 small bin 和 unsorted bin 获取帮助，就会遍历 binmaps 找到下一个非空 largest bin。就像之前说的，找到了非空 bin 后，取得的内存块被一分为二，用户块返回给用户，剩余块到了 unsorted bin。这个剩余块就是新的 last reminder chunk。


如何实现访问局部性？

假设用户连续多次请求分配小内存块，而 unsorted bin 中只剩下 last reminder chunk，这个 chunk 会被一分为二。还是那样分割，返回一块给用户，留下剩余块塞给 unsorted bin。下一个请求又是这样，一次次操作下来，内存分配都在最开始的 last reminder chunk 上进行，最终使得分配出去的内存块都在连续区域中。

加入伯乐在线专栏作者。扩大知名度，还能得赞赏！详见《[招募专栏作者](#)》

 1 赞

 4 收藏





坐标上海，linux应用程序开发，家乡安徽巢湖。微博：[@饕餮巽离](http://weibo.com/carxus) [个人主页](#) · [我的文章](#) · [14](#)



相关文章

- [如何设计一个内存分配器？](#)
- [StackOverflow 这么大，它的架构是怎么样的？](#)
- [MySQL 读写分离介绍及搭建](#)
- [AlphaGo相关技术：蒙特卡罗方法简介](#)
- [蜕变成蝶：Linux设备驱动之字符设备驱动](#)
- [objc系列译文（2.4）：线程安全类的设计](#)
- [Go并发编程基础](#)
- [C#多线程之旅（2）：创建和开始线程](#)
- [MySQL 最佳实践：空间优化](#)
- [如果有人问你数据库的原理，叫他看这篇文章](#)

可能感兴趣的话题

- [请搞过ACM的大生请留下你的感想，拜托 · Q_11](#)
- [移动鼠标绘制光滑曲线 · Q_3](#)
- [在北京干了三年 Java 的程序员 需要懂什么技术呢 每门技术都需要... · Q_5](#)
- [解题：不用循环、递归，如何从 1 打印到 100？ · Q_20](#)
- [每天明明干了很多活，但是却不能按时下班 · Q_4](#)
- [做了3年网页美工，从公司出来发现我离ui设计和web前端的距离都好远，一下子... · Q_9](#)
- [把微信卸载一个月会是一种什么体验？有人敢试试并在小组分享体验么？ · Q_17](#)
- [我要去北京了码代码了，前辈们有没有什么建议？比如说吃的，玩的，工作的 · Q_68](#)
- [一个公司只有一个程序员是什么样的体验 · Q_41](#)
- [学习前端是结合ui一起学习好还是结合前端一起学习好呢？](#)

[« Netty5 HTTP协议栈浅析与实践](#)
[Linux 环境多线程编程基础设施 »](#)

登录后评论

新用户注册

直接登录

文章

输入搜索关键字

搜索



- [本周热门文章](#)
- [本月热门文章](#)
- [热门标签](#)

- 0 [开发者 MAC 电脑里的十八般兵器](#)
- 1 [为什么很多硅谷工程师偏爱 OS X](#)
- 2 [18 个锻炼编程技能的网站](#)

- 3 [你在用哪种编程字体？](#)
- 4 [NASA 的 10 大编程规则](#)
- 5 [四种框架分别实现百万websocket常连...](#)
- 6 [那些被岁月遗忘的 UNIX 经典著作](#)
- 7 [年轻人，你为啥使用 Linux](#)
- 8 [玩转 Windows 10 中的 Linux ...](#)
- 9 [“懒惰” Linux 管理员的 10 个...](#)



业界热点资讯

更多 »



[卡耐基梅隆大学赢得DARPA网络挑战赛](#)
3 小时前 · 2



[谷歌打造开源YOLO项目 Android APP登录将无需输...](#)
3 小时前 · 2



[小型可编程配置量子计算机问世](#)
2 天前 · 16



[研究人员利用旁路攻击窃取加密信息](#)
1 天前 · 2



[在浏览器中体验 Ubuntu](#)
2 天前 · 8

精选工具资源

更多资源 »



[jOpenDocument : 处理OpenDocument格式文档
文档处理工具](#)



[SpringSource Tool Suite : 基于Eclipse的Spring应...
IDE](#)



JDK 9 : JDK 9的早期访问版本 JVM与JDK



Jinja2 : 一个纯Python实现的模板引擎 Python, 模板引擎



Smack : 一个开源的XMPP用于即时通讯的客户端类库 Java, 消息传递

最新评论

-  Re: [程序员正在调 Bug : 15 张令人...
算术（不是算数）那个倒是真的，猿们会自己构造剩下的operations](#)
-  Re: [高效 Windows 工作环境 &&a...
最新更新，MacType 会导致 Win10 红石蓝屏，可参考 <http://bbs.themex...>](#)
-  Re: [你在用哪种编程字体？
自己混合 Glass_TTY_VT220+MSYH 逼格得不行](#)
-  Re: [有趣的机器学习：最简明入门指南
“generic algorithms ”这里更适合翻译成通用算法，图表1中相应的应翻译为通用机器学...](#)
-  Re: [你在用哪种编程字体？
我喜欢source code pro](#)
-  Re: [JDK8 Stream API中Collecto...
尝试过 map.get\(1\) 吗...](#)
-  Re: [你在用哪种编程字体？
都是大牛啊。。。](#)
-  Re: [一个高级PHP工程师所应该具备的
挺水的...](#)

关于伯乐在线博客

在这个信息爆炸的时代，人们已然被大量、快速并且简短的信息所包围。然而，我们相信：过多“快餐”式的阅读只会令人“虚胖”，缺乏实质的内涵。伯乐在线内容团队正试图以我们微薄的力量，把优秀的原创文章和译文分享给读者，为“快餐”添加一些“营养”元素。

快速链接

- [问题反馈与求助 »](#)
- [加入伯乐翻译小组 »](#)
- [加入专栏作者 »](#)

关注我们

新浪微博：[@伯乐在线官方微博](#)
RSS：[订阅地址](#)
推荐微信号



程序员的那些事

UI设计达人

极客范

合作联系

Email : bd@jobbole.com

QQ : 2302462408 (加好友请注明来意)

更多频道

- [小组](#) - 好的话题、有启发的回复、值得信赖的圈子
- [头条](#) - 分享和发现有价值的内容与观点
- [相亲](#) - 为IT单身男女服务的征婚传播平台
- [资源](#) - 优秀的工具资源导航
- [翻译](#) - 翻译传播优秀的外文文章
- [文章](#) - 国内外的精选文章
- [设计](#) - UI,网页,交互和用户体验
- [iOS](#) - 专注iOS技术分享
- [安卓](#) - 专注Android技术分享
- [前端](#) - JavaScript, HTML5, CSS
- [Java](#) - 专注Java技术分享
- [Python](#) - 专注Python技术分享

