理解 glibc malloc

> 译者：猫科龙@csdn

> 来源：http://blog.csdn.net/maokelong95/article/details/51989081

> 本篇文章主要完成了对「Understanding glibc malloc」的翻译工作。限于本人翻译水平与专业技术水平（纯粹为了了解内存分配而翻），本文章必定会有很多不足之处，请大家见谅，也欢迎大家的指正！

> 联系邮箱：974985526@qq.com。

> 2017/03/17

> 优化排版

堆内存是一个很有意思的领域，这样的问题：

> 堆内存是如何从内核中分配的？ 内存管理效率怎样？ 它是由内核、库函数，还是应用本身管理的？ 堆内存可以开发吗?

我也困惑了很久，但是直到最近我才有时间去了解它。下面就让我来谈谈我的研究成果。开源社区提供了很多现成的内存分配器(memory allocators)：

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

每一种分配器都宣称自己快(fast)、可拓展(scalable)、效率高(memory efficient)！但是并非所有的分配器都适用于我们的应用。内存吞吐量大(memory hungry)的应用程序的性能很大程度上取决于内存分配器的性能。

在这篇文章中，我将只谈论「glibc malloc」内存分配器。为了更好地理解「glibc malloc」，我会联系最近的源代码。

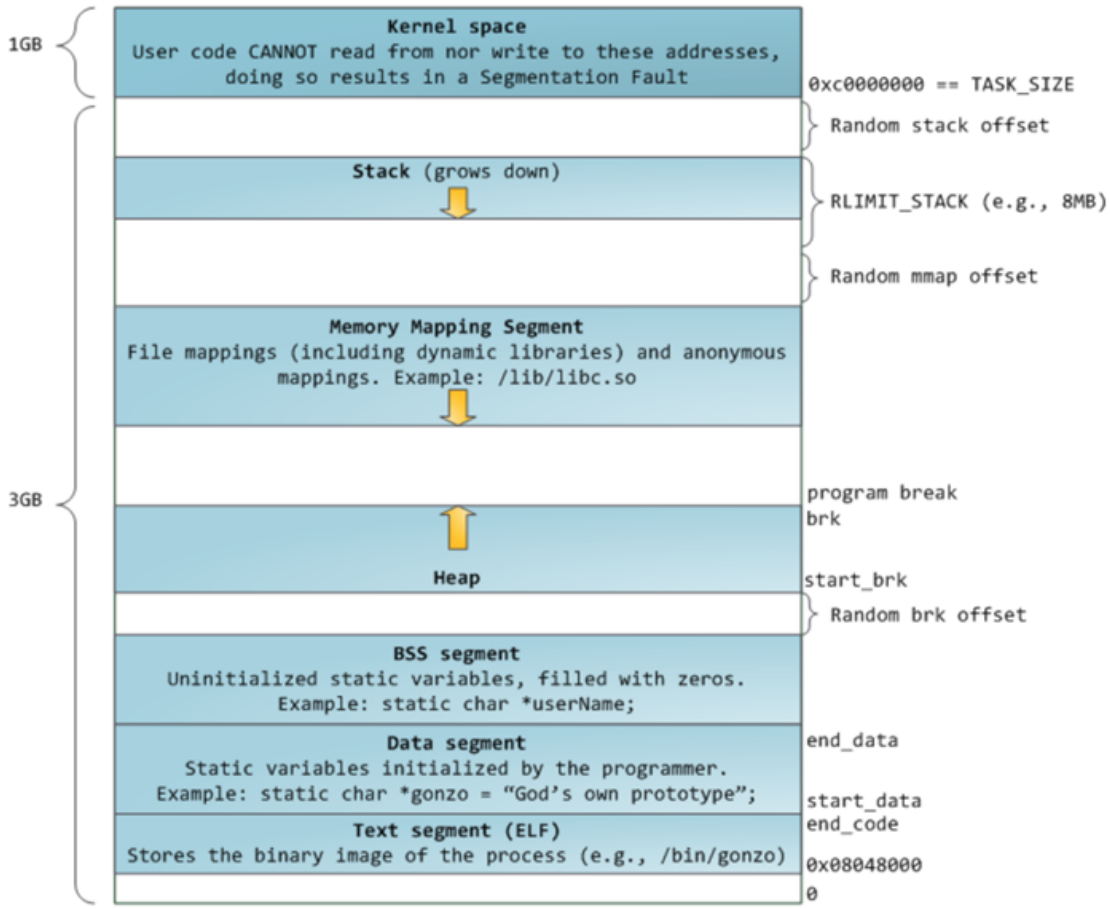
> **历史**：ptmalloc2 基于 dlmalloc 开发，并添加了对多线程的支持，于 2006 年公布。在公布之后，ptmalloc2 被整合到 glibc 源代码中，此后 ptmalloc2 所有的修改都直接提交到 glibc 的 malloc 部分去了。因此，ptmalloc2 的源码和 glibc 的 malloc源码有很多不一致的地方。（译者注：1996 年出现的 dlmalloc 只有一个主分配区，为所有线程所争用，1997 年发布的 ptmalloc 在 dlmalloc 的基础上引入了非主分配区的支持。）

        - Multiple Arena
        - Multiple Heaps
    - Chunk
        - Allocated chunk
        - Free chunk
    - Bins
        - Fast Bin
        - Unsorted Bin
        - mall Bin
        - Large Bin
        - Top Chunk
        - Last Remainder Chunk

系统调用

在之前的文章中提到过malloc的内部调用为 brk 或 mmap 。

> **译者注**：其中有一张关于虚拟地址空间分布的图片，我觉得很有助于本篇文章的理解，因此把它放在此处。



线程处理

Linux 的早期版本使用 dlmalloc 为默认内存分配器，但是因为 ptmalloc2 提供了多线程支持，所以 Linux 后来采用 ptmalloc2 作为默认内存分配器。多线程支持可以提升内存分配器的性能，进而间接提升应用的性能。

在 dlmalloc 中，当有两个线程同时调用 malloc 时，只有一个线程能够访问临界区(critical section)——因为「空闲列表数据结构」(freelist data structure)被所有可用线程**共享**。正如此，使用 dlmalloc 的多线程应用会在内存分配上耗费过多时间，导致整个应用性能的下降。

而在 ptmalloc2 中，当有两个线程同时调用 malloc 时，内存均会得到立即分配——因为每个线程都维护着一个独立的「堆段」(heap segment)，因此维护这些堆的「空闲列表数据结构」也是独立的。这种为每个线程独立地维护堆和「空闲列表数据结构」的行为就称为 **per thread arena**。

Example:

```
/* Per thread arena example. */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

void* threadFunc(void* arg) {
        printf("Before malloc in thread 1\n");
        getchar();
        char* addr = (char*) malloc(1000);
        printf("After malloc and before free in thread 1\n");
        getchar();
        free(addr);
        printf("After free in thread 1\n");
        getchar();
}

int main() {
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

        printf("Welcome to per thread arena example::%d\n",getpid());
        printf("Before malloc in main thread\n");
        getchar();
        addr = (char*) malloc(1000);
        printf("After malloc and before free in main thread\n");
        getchar();
        free(addr);
        printf("After free in main thread\n");
        getchar();
        ret = pthread_create(&t1, NULL, threadFunc, NULL);
        if(ret)
        {
                printf("Thread creation error\n");
                return -1;
        }
        ret = pthread_join(t1, &s);
        if(ret)
        {
                printf("Thread join error\n");
                return -1;
        }
        return 0;
}
```

输出分析

在主线程 malloc 之前

在如下的输出里我们可以看到，这里还 **没有**「 堆段」 **也没有** 「每线程栈」(per-thread stack)，因为 thread1 还没有创建！

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

在主线程 malloc 之后

在如下的输出中我们可以看到堆段产生在数据段(0804b000 - 0806c000)之上，这表明堆内存是通过更高级别的中断产生（也即 brk 中断）。此外，请注意，尽管用户只申请了 1000 字节的内存，但是实际产生了 132KB 的堆内存。这个连续的堆内存区域被称为 「**arena**」 。因为这个「arena」是被主线程建立的，因此称为「**main arena**」。接下来的申请会继续分配这个「arena」的132KB中剩余的部分，直到用尽。当用尽时，它可以通过更高级别的中断扩容，在扩容之后，「top chunk」的大小也随之调整以圈进这块额外的空间。相应地，「arena」也可以在「top chunk」空间过大时缩小。

> **注意**：top chunk是一个arena中最顶层的chunk。有关top chunk的更多信息详见下述"top chunk"部分。

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

在主线程 free 之后

在如下的输出结果中我们可以看出当分配的内存区域free掉时，其后的内存并不会立即释放给操作系统。分配的内存区域（1000B）仅仅是移交给了「glibc malloc」，把这段free掉的区域添加在了「main arenas bin」中（在「glibc malloc」中，空闲列表数据结构被称为「bin」）。随后当用户请求内存时，「glibc malloc」就不再从内核中申请新的堆区了，而是尝试在「bin」中找到空闲区块，除非实在找不到。

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

在thread1 malloc 之前

在如下的输出中我们可以看见此时并没有 thread1 的堆段，但是其每个线程栈都已建立。

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

在thread1 malloc 之后

在如下的输出结果中我们可以看出 thread1 的堆段建立在内存映射段区域(b7500000 - b7521000，132KB)，这也表明了堆内存是使用 mmap 系统调用产生的，而非同主线程一样使用 sbrk 系统调用。同样地，尽管用户只请求了 1000B，1MB 的堆内存还是被映射到了进程地址空间。在这 1MB，只有 132KB 被设置了读写权限并成为该线程的堆内存。这段连续内存（132KB）被称为「thread arena」。

> **注意**：当用户请求超过 128KB 大小并且此时 「arena」 中没有足够的空间来满足用户的请求时，内存将通过使用 mmap 系统调用（不再是 sbrk）来分配而不论请求是发自「main arena」还是 「thread arena」。

```
ploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

在thread1 free 之后

在如下的输出结果中我们可以看出 free 掉的分配的内存区域这一过程并不会把堆内存归还给操作系统，而是仅仅是移交给了「glibc malloc」，然后添加在了「thread arenas bin」中。

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
```

```
Before malloc in thread 1
After malloc and before free in thread 1
After free in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

Arena

Arena的数量

在如下的例子中，我们可以看见主线程包含「main arena」而thread 1包含它自有的「thread arena」。所以若不计线程的数量，在线程和「arena」之间是否存在一对一映射关系？当然不存在，部分极端的应用甚至运行比处理器核心的数量还多的线程，在这种情况下，每个线程都拥有一个「arena」开销过高且意义不大。因此，应用的「arena」数量限制是基于系统的核心数的。

```
For 32 bit systems:
     Number of arena = 2 * number of cores + 1.
For 64 bit systems:
     Number of arena = 8 * number of cores + 1.
```

Multiple Arena

举例而言：让我们来看一个运行在单核计算机上的32位操作系统上的多线程应用（4线程 = 主线程 + 3个用户线程）的例子。这里线程数量(4)大于核心数的二倍加一，因此在这种条件下，「glibc malloc」认定「multiple arenas」会被所有可用线程共享。那么它是如何共享的呢?

- 当主线程第一次调用 malloc 时，已经建立的 「main arena」会被没有任何竞争地使用。
- 当 thread 1 和 thread 2 第一次调用malloc时，一块新的「arena」就被创建且会被没有任何竞争地使用。此时线程和「arena」之间有着一对一的映射关系。
- 当 thread3 第一次调用 malloc 时，「arena」的数量限制被计算出来。这里超过了「arena」的数量限制，因此尝试复用已经存在的「arena」(「Main arena 」或 Arena 1 或 Arena 2)。
- 复用:
    - 一旦遍历出可用arena」，就开始自旋申请该「arena」的锁。
    - 如果上锁成功（比如说「main arena」上锁成功），就将该「arena」返回用户。
    - 如果查无可用「arena」，thread 3 的 malloc 操作阻塞，直到有可用的「arena」为止。
- 当thread 3 第二次调用 malloc 时，malloc 会尝试使用上一次使用的「arena」（『main arena』）。当「main arena」可用时就用，否则 thread 3 就一直阻塞直至「main arena」被 free 掉。因此现在「main arena」实际上是被「main thread」和 thread 3 所共享。

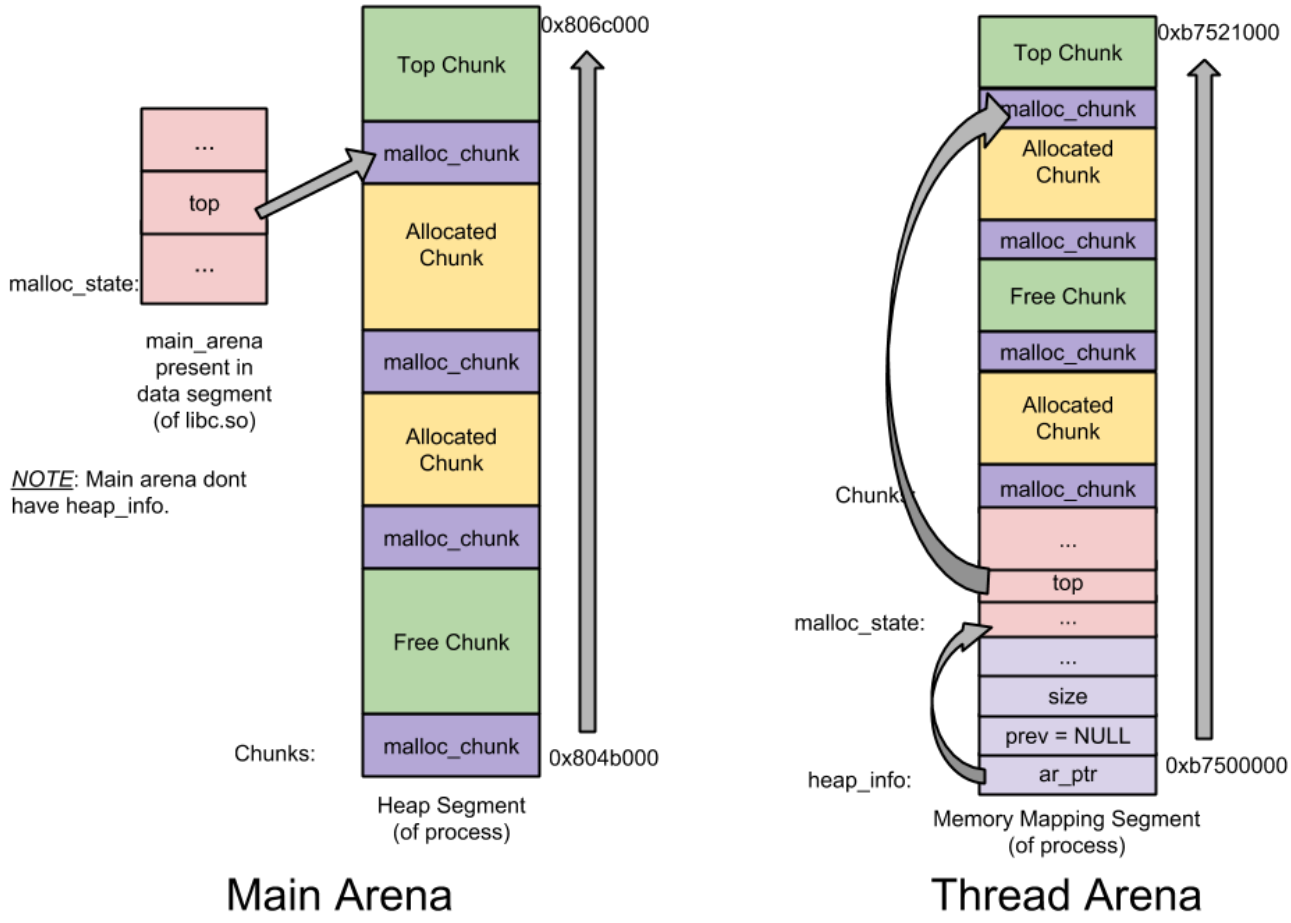
Multiple Heaps

在「glibc malloc」中主要发现了3种数据结构： heap_info ——Heap Header—— 一个「 thread arena 」可以有多个堆。每个堆都有自己的堆 Header。为什么需要多个堆？ 每个「 thread arena 」都只包含一个堆，但是当这个堆段空间

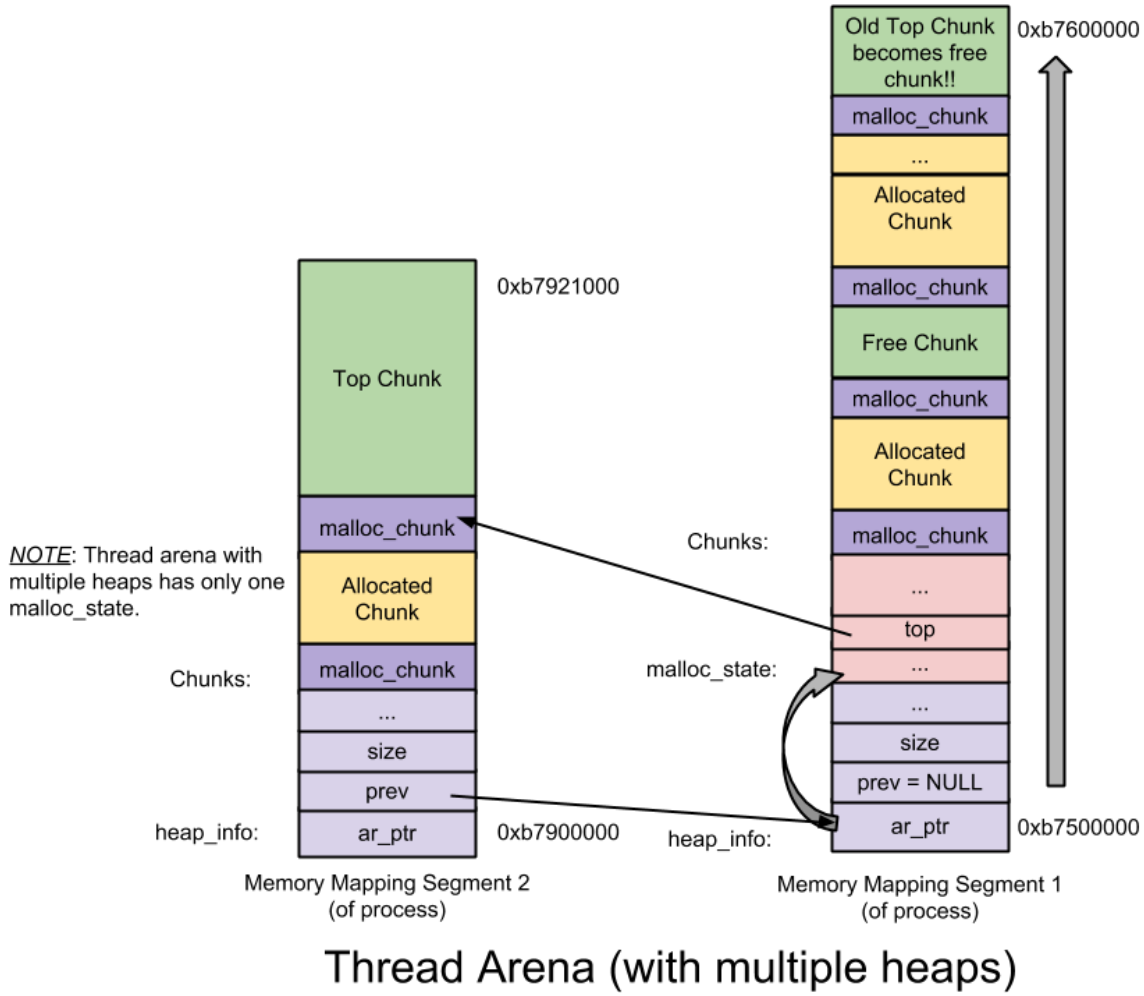
耗尽时，新的堆（非连续区域）就会被 mmap 到这个 「aerna」。 malloc_state ——Arena header—— 一个 「thread arena 」可以有多个堆，但是所有这些堆只存在 「arena」 header 。「arena header 」包括的信息有：「bins」、「top chunk」、「last remainder chunk」..... malloc_chunk ——Chunk header—— 一个堆根据用户请求被分为若干「chunk」。每个这样的「chunk」都有自己的「chunk」 header。

> **注意**：
>
> - 「Main arena」没有多个堆，因此没有「heap_info」结构。当「main arena」空间耗尽时，就拓展 sbrk 获得的堆段（拓展后是连续内存区域），直至"碰"到内存映射区为止。
> - 不像「thread arena」，「main arena」的「arena 」header不是 sbrk 获得的堆段的一部分，而是一个全局变量，因此它可以在 libc.so 的 数据段中被找到。

「main arena」和「thread arena」的图示如下（单堆段）：



「thread arena」的图示如下（多堆段）：

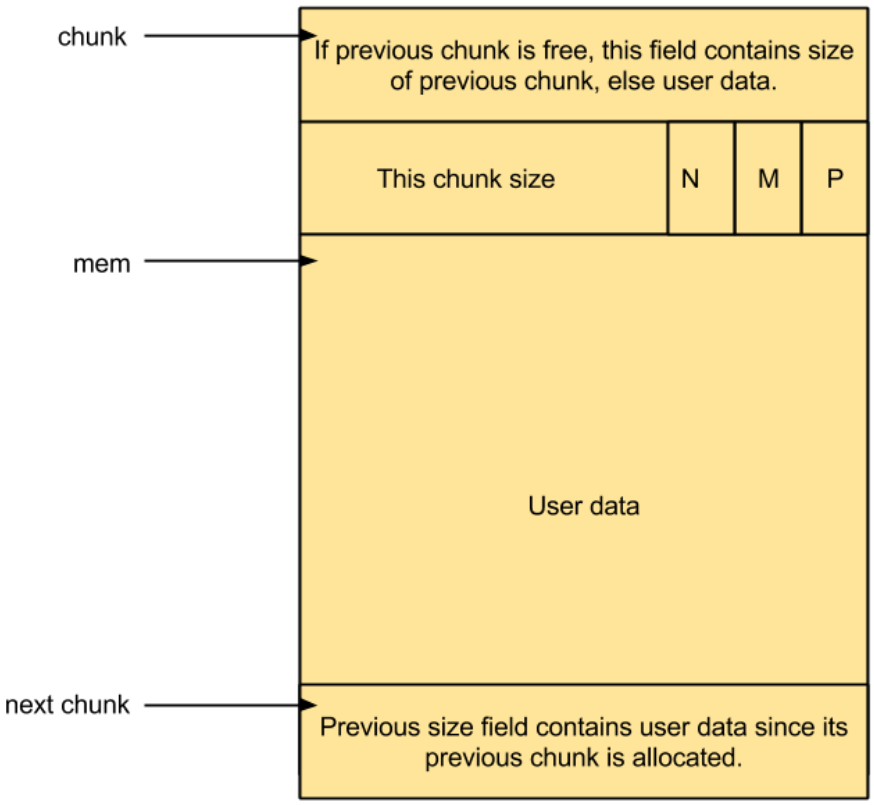


Chunk

堆段中能找到的「chunk」类型如下：

- Allocated chunk
- Free chunk
- Top chunk
- Last Remainder chunk

Allocated chunk



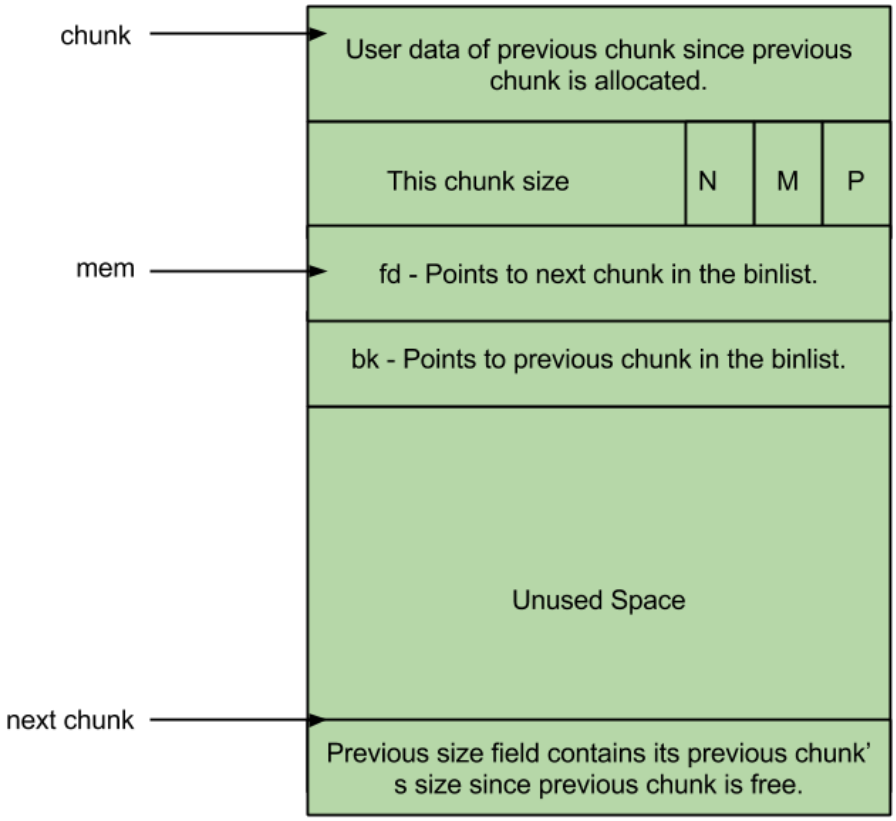
Allocated Chunk

prev_size：若上一个「chunk」可用，则此结构成员赋值为上一个「chunk」的大小；否则若上一个「chunk」被分配，此此结构成员赋值为上一个「chunk」的用户数据大小。 size：此结构成员赋值为已分配「chunk」大小，其最后三位包含标志(flag)信息。

- PREV_INUSE (P) – 置“1”表示上一个「chunk」被分配；
- IS_MMAPPED (M) – 置“1”表示这一个「chunk」是直接 mmap 申请的；
- NON_MAIN_ARENA (N) – 置“1”表示这一个「chunk」属于一个「thread arena」。

> **注意**：
>
> - malloc_chunk 中的其余结构成员，如 fd、 bk 不用于已分配的「chunk」，因此它们被拿来存储用户数据；
> - 用户请求的大小被转换为可用大小（内部显示大小），因为需要一些额外的空间存储 malloc_chunk，此外还需要考虑对齐的因素。

Free chunk



Free Chunk

prev_size：两个空闲「chunk」不能毗连，而应合并成一个。因此前一个「chunk」和这一个空闲「chunk」都会被分配，此时 prev_size 中保存上一个「chunk」的用户数据。 size：该结构成员保存本空闲「chunk」的大小。 fd：Forward pointer ——指向同一「bin」中的下一个「chunk」（而非物理内存中下一块）。 bk：Backward pointer ——指向同一「bin」中的上一个「chunk」（而非物理内存中上一块）。

Bins

「bins」就是空闲列表数据结构。它们用以保存空闲「chunk」。基于「chunk」的大小，有下列几种可用「bins」：

- Fast bin
- Unsorted bin
- Small bin
- Large bin

保存这些「bins」的数据结构为：

fastbinsY：这个数组用以保存「fast bins」 。 bins：这个数组用以保存「unsorted bin」、「small bins」以及「large bins」，共计可容纳 126 个：

- Bin 1——「unsorted bin」
- Bin 2 到 Bin 63 ——「small bins」
- Bin 64 到 Bin 126 —— 「large bins」

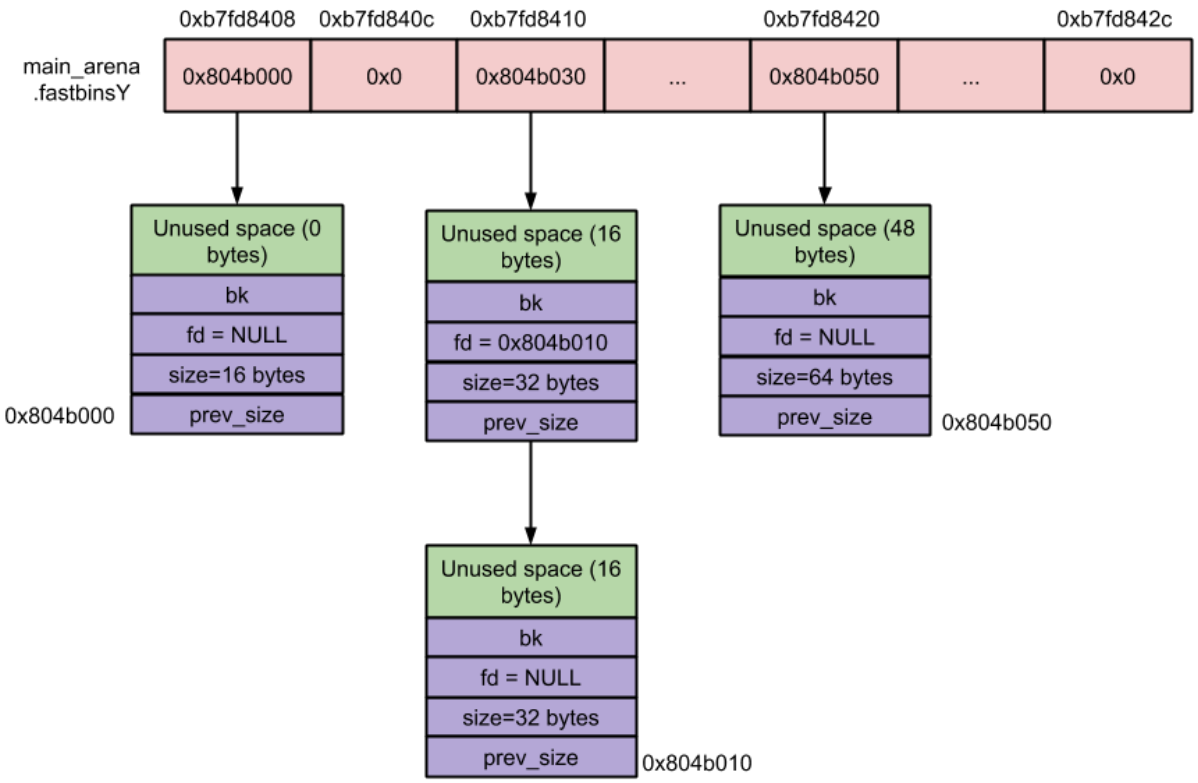
Fast Bin

大小为 16 ~ 80 字节的chunk被称为「fast chunk」。在所有的「bins」中，「fast bins」在内存分配以及释放上有更快的速度。

- 数量——10

    - 每个「fast bin」都记录着一条free 「chunk」的单链表（称为「binlist」，采用单链表是出于「fast bins」中链表中部的「chunk」不会被摘除的特点），增删「chunk」都发生在链表的前端。——LIFO

- 大小——8字节递增
    - 「fast bins」记录着大小以 8 字节递增的「binlist」。也即，「fast bin」(index 0)记录着大小为 16 字节的「chunk」的 「binlist」、「fast bin」(index 1) 记录着大小为 24 字节的「chunk」的 「binlist」依次类推.....
    - 指定「fast bin」中所有「chunk」大小均相同。
- 在 malloc 初始化过程中，最大的「fast bin」的大小被设置为 64 而非 80 字节。因为默认情况下只有大小 16 ~ 64 的「chunk」被分类为「fast chunk」。
- 不能合并—— 两个毗连的空闲「chunk」并不会被合并成一个空闲「chunk」。不合并可能会导致碎片化问题，但是却可以大大加速释放的过程！
- malloc(「fast chunk」)
    - 初始情况下「fast bin」最大内存容量以及指针域均未初始化，因此即使用户请求「fast chunk」，服务的也将是「small bin」 code而非「fast bin」 code。
    - 当它非空后，「fast bin」索引将被计算以检索对应「binlist」。
    - 「binlist」中被检索的第一个「chunk」将被摘除并返回给用户。
- free(「fast chunk」)
    - 「fast bin」索引被计算以索引相应「binlist」。
    - free 掉的「chunk」将被添加在索引到的「binlist」的前端。

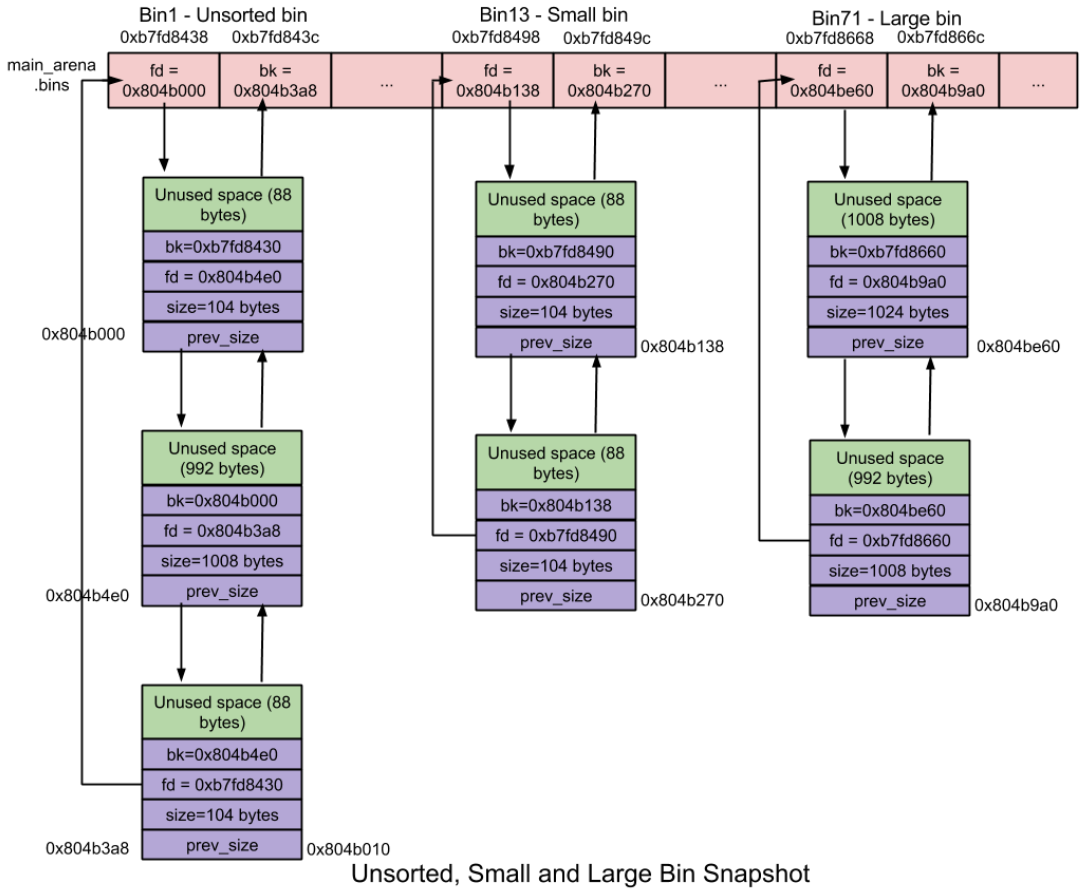


Fast Bin Snapshot

Unsorted Bin

当「small chunk」和「large chunk」被 free 掉时，它们并非被添加到各自的bin中，而是被添加在「unsorted bin」中。这种途径给予 「glibc malloc」重新使用最近 free 掉的「chunk」的第二次机会，这样寻找合适「bin」的时间开销就被抹掉了，因此内存的分配和释放会更快一点。

- 数量 ——1
    - 「unsorted bin」包括一个用于保存空闲「chunk」的双向循环链表（又名「binlist」）。
- 「chunk」大小——无尺寸限制，任何大小「chunk」都可以添加进这里。



Unsorted, Small and Large Bin Snapshot

mall Bin

大小小于512字节的「chunk」被称为「small chunk」，而保存「small chunks」的「bin」被称为「small bin」。在内存分配回收的速度上，「small bin」比「large bin」更快。

- 数量 ——62
    - 每个「small bin」都包括一个空闲区块的双向循环链表（也称「binlist」）。free 掉的「chunk」添加在链表的前端，而所需「chunk」则从链表后端摘除。——FIFO
- 大小——8 字节递增
    - 「small bin」 记录着大小以 8 字节递增的「bin」链表。也即，第一个「small bin」(Bin 2)记录着大小为 16 字节的「chunk」的「binlist」、「small bin」(Bin 3)记录着大小为24字节的「chunk」的「binlist」依次类推.....
    - 指定「small bin」中所有「chunk」大小均相同，因此无需排序。
- 合并——两个毗连的空闲「chunk」会被合并成一个空闲「chunk」。合并消除了碎片化的影响但是减慢了 free 的速度。
- malloc（「small chunk」）

    - 初始情况下，「small bin」都会是 NULL，因此尽管用户请求「small chunk」，提供服务的将是 「unsorted bin」 code 而不是「small bin」code。
    - 同样地，在第一次调用 malloc 期间，在 malloc_state 找到的「small bin」和「large bin」数据结构被初始化，「bin」都会指向它们本身以表示「binlist」为空。
    - 此后当「small bin」非空后，相应的「bin」会摘除「binlist」中最后一个「chunk」并返回给用户。

- free（「small chunk」）

    - 在free一个「chunk」的时候，检查其前或其后的「chunk」是否空闲，若是则合并，也即把它们从所属的链表中摘除并合并成一个新的「chunk」，新「chunk」会添加在「unsorted bin」链表的前端。

Large Bin

大小大于等于 512 字节的「chunk」被称为「large chunk」，而保存「large chunks」的「bin」被称为「large bin」。在内存分配回收的速度上，「large bin」比「small bin」慢。

- 数量——63

    - 每个「large bin」都包括一个空闲区块的双向循环链表（也称「binlist」）。free 掉的「chunk」添加在链表的前端，而所需「chunk」则从链表后端摘除。——FIFO

    - 超过 63 个「bin」之后

        - 前 32 个「bin」记录着大小以 64 字节递增的「bin」链表，也即第一个「large chunk」(Bin 65)记录着大小为 512 字节~ 568 字节的「chunk」的「binlist」、第二个「large chunk」(Bin 66)记录着大小为 576 字节到 632 字节的「chunk」的「binlist」，依次类推……
        - 后 16 个「bin」记录着大小以 512 字节递增的「bin」链表。
        - 后 8 个「bin」记录着大小以 4096 字节递增的「bin」链表。
        - 后 4 个「bin」记录着大小以 32768 字节递增的「bin」链表。
        - 后 2 个「bin」记录着大小以 262144 字节递增的「bin」链表。
        - 最后 1 个「bin」记录着大小为剩余大小的「chunk」。

    - 不像「small bin」，large bin中所有「chunk」大小不一定相同，因此各「chunk」需要递减保存。最大的「chunk」保存在最前的位置，而最小的「chunk」保存在最后的位置。

- 合并——两个毗连的空闲chunk会被合并成一个空闲「chunk」。

- malloc（「large chunk」）

    - 初始情况下，large bin都会是NULL，因此尽管用户请求「large chunk」，提供服务的将是「next largetst bin code」 而不是 「large bin code」。

    - 同样地，在第一次调用 malloc 期间，在 malloc_state 找到的「small bin」和「large bin」数据结构被初始化，bin都会指向它们本身以表示「binlist」为空。

    - 此后当「small bin」非空后，当最大「chunk」大小（在相应「binlist」中的）大于用户所请求的大小时，「binlist」就从顶部遍历到底部以找到一个大小最接近用户需求的「chunk」。一旦找到，相应「chunk」就会分成两块：

        - 「User chunk」（用户请求大小）——返回给用户。
        - 「Remainder chunk」（剩余大小）——添加到「unsorted bin」。

    - 当最大「chunk」大小（在相应「binlist」中的）小于用户所请求的大小时，尝试在「Next largest bin」中查到到所需的「chunk」以响应用户请求。「 next largetst bin code」会扫描「 binmaps」以找到下一个最大非空「bin」，如果这样的「bin「找到了，就从其中的「binlist」中检索到合适的「chunk」并返回给用户；反之就使用「top chunk」以响应用户请求。

- free（「large chunk」）——类似于「small chunk」。

Top Chunk

一个「arena」中最顶部的「chunk」被称为「top chunk」。它不属于任何「bin」。在所有「bin」中都没有合适空闲内存区块的时候，才会使用「top chunk」来响应用户请求。当「top chunk」大小比用户所请求大小还大的时候，「top chunk」会分为两个部分：

- 「User chunk」（用户请求大小）
- 「Remainder chunk」（剩余大小）

其中「Remainder chunk」成为新的「top chunk」。当「top chunk」大小小于用户所请求的大小时「top chunk」就通过sbrk（「main arena」）或mmap（「thread arena」）系统调用来扩容。

Last Remainder Chunk

最后一次 small request 中因分割而得到的 Remainder。「last remainder chunk」有助于改进引用的局部性，也即连续的对「small chunk」的 malloc 请求可能最终导致各「chunk」被分配得彼此贴近。

但是除了在一个 「arena」 里可用的的诸「chunk」，哪些「chunk」有资格成为「last remainder chunk」呢?

当一个用户请求「small chunk」而无法从「small bin」和「unsorted bin」得到服务时，「binmaps」就会扫描下一个最大非空「bin」（**译者注**：「top chunk」 不属于任何「bin」）。正如前文所提及的，如果这样的「bin」找到了，其中最适「chunk」就会分割为两部分：返回给用户的「User chunk」、添加到「unsorted bin」中的「Remainder chunk」。此外，这一「Remainder chunk」还会成为最新的「**last remainder chunk**」。

那么参考局部性是如何实现的呢?

现在当用户随后的请求是请求一块「small chunk」并且「last remainder chunk」是「unsorted bin」中唯一的「chunk」，「last remainder chunk」就分割成两部分：返回给用户的「User chunk」、添加到「unsorted bin」中的「Remainder chunk」。此外，这一「Remainder chunk」还会成为最新的 「last remainder chunk」。因此随后的内存分配最终导致各「chunk」被分配得彼此贴近。