深入理解 glibc malloc：内存分配器实现原理

CPP开发者 2021-12-07 19:55

前言

堆内存（Heap Memory）是一个很有意思的领域。你可能和我一样，也困惑于下述问题很久了：

- 如何从内核申请堆内存？
- 谁管理它？内核、库函数，还是应用本身？
- 内存管理效率怎么这么高？！
- 堆内存的管理效率可以进一步提高吗？

最近，我终于有时间去深入了解这些问题。下面就让我来谈谈我的调研成果。

开源社区公开了很多现成的内存分配器（Memory Allocators，以下简称为**分配器**）：

- dlmalloc – 第一个被广泛使用的通用动态内存分配器；
- ptmalloc2 – glibc 内置分配器的原型；
- jemalloc – FreeBSD & Firefox 所用分配器；
- tcmalloc – Google 贡献的分配器；
- libumem – Solaris 所用分配器；
- …

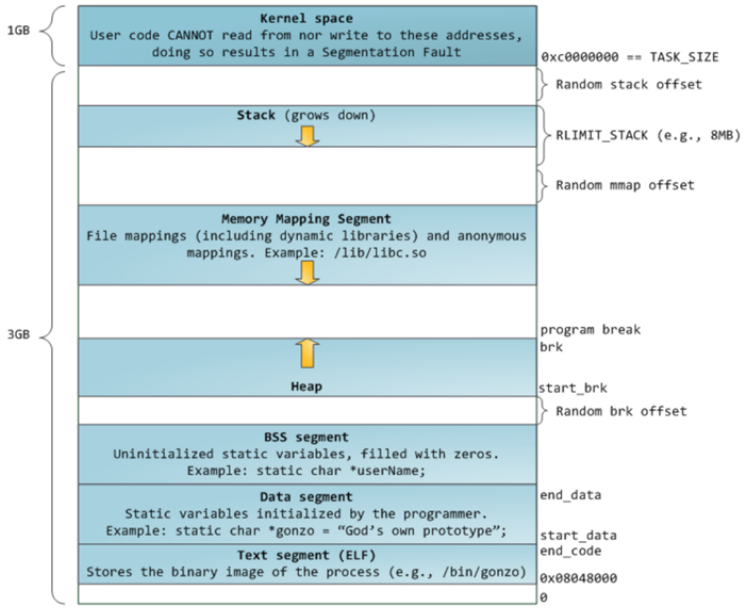
每一种分配器都宣称自己快（fast）、可拓展（scalable）、效率高（memory efficient）！但是并非所有的分配器都适用于我们的应用。内存吞吐量大（memory hungry）的应用程序，其性能很大程度上取决于分配器的性能。

在这篇文章中，我只谈「glibc malloc」分配器。为了方便大家理解「glibc malloc」，我会联系最新的源代码。

> **历史**：ptmalloc2 基于 dlmalloc 开发，其引入了多线程支持，于 2006 年发布。发布之后，ptmalloc2 整合进了 glibc 源码，此后其所有修改都直接提交到了 glibc malloc 里。因此，ptmalloc2 的源码和 glibc malloc 的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，该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概念。）

1. 申请堆的系统调用

我在之前的文章中提到过， `malloc` 内部通过 `brk` 或 `mmap` 系统调用向内核申请堆区。

> **译者注**：在内存管理领域，我们一般用「堆」指代用于分配动态内存的虚拟地址空间，而用「栈」指代用于分配静态内存的虚拟地址空间。具体到虚拟内存布局（Memory Layout），堆维护在通过 `brk` 系统调用申请的「Heap」及通过 `mmap` 系统调用申请的「Memory Mapping Segment」中；而栈维护在通过汇编栈指令动态调整的「Stack」中。在 Glibc 里，「Heap」用于分配较小的内存及主线程使用的内存。
>
> 下图为 Linux 内核 v2.6.7 之后，32 位模式下的虚拟内存布局方式。
>
> 
>

2. 多线程支持

Linux 的早期版本采用 dlmalloc 作为它的默认分配器，但是因为 ptmalloc2 提供了多线程支持，所以 后来 Linux 就转而采用 ptmalloc2 了。多线程支持可以提升分配器的性能，进而间接提升应用的性能。

在 dlmalloc 中，当两个线程同时 `malloc` 时，只有一个线程能够访问临界区（critical section）——这是因为所有线程**共享**用以缓存已释放内存的「空闲列表数据结构」（freelist data structure），所以使用 dlmalloc 的多线程应用会在 `malloc` 上耗费过多时间，从而导致整个应用性能的下降。

在 ptmalloc2 中，当两个线程同时调用 `malloc` 时，内存均会得以立即分配——每个线程都维护着单独的堆，各个堆被独立的空闲列表数据结构管理，因此各个线程可以并发地从空闲列表数据结构中申请内存。这种为每个线程维护独立堆与空闲列表数据结构的行为就「per thread arena」。

2.1. 案例代码

```
/* 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;
}
```

2.2. 案例输出

2.2.1. 在主线程 malloc 之前

从如下的输出结果中我们可以看到，这里还**没有**堆段**也没有**每个线程的栈，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/mtl
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
```

```
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

2.2.2. 在主线程 malloc 之后

从如下的输出结果中我们可以看到，堆段已经产生，并且其地址区间正好在数据段（0x0804b000 - 0x0806c000）上面，这表明堆内存是移动「Program Break」的位置产生的（也即通过 `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。这个连续的堆区域被称为「**arena**」。因为这个 arena 是被主线程建立的，因此其被称为「**main arena**」。接下来的申请会继续分配这个 arena 的 132KB 中剩余的部分。当分配完毕时，它可以通过继续移动 Program Break 的位置扩容。扩容后，「top chunk」的大小也随之调整，以将这块新增的空间圈进去；相应地，arena 也可以在 top chunk 过大时缩小。

> **注意**：top chunk 是一个 arena 位于最顶层的 chunk。有关 top chunk 的更多信息详见后续章节「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/mtl
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

2.2.3. 在主线程 free 之后

从如下的输出结果中我们可以看到，当分配的内存区域 free 掉时，其并不会立即归还给操作系统，而仅仅是移交给了作为库函数的分配器。这块 free 掉的内存添加在了「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，分配器就不再向内核申请新堆了，而是先试着各个「bin」中查找空闲内存。只有当 bin 中不存在空闲内存时，分配器才会继续向内核申请内存。

```
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/mtl
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

2.2.4. 在 thread1 malloc 之前

从如下的输出结果中我们可以看到，此时 thread1 的堆尚不存在，但其栈已产生。

```
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/mtl
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
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$
```

2.2.5. 在 thread1 malloc 之后

从如下的输出结果中我们可以看到，thread1 的堆段(b7500000 - b7521000，132KB)建立在了内存映射段中，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。类似地，尽管用户只请求了 1000B，但是映射到程地址空间的堆内存足有 1MB。这 1MB 中，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(比如 `malloc(132*1024)`) 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 `mmap` 系统调用（不再是 `sbrk` ）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/mt
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mt
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mt
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$
```

2.2.6. 在 thread1 free 之后

从如下的输出结果中我们可以看到， `free` 不会把内存归还给操作系统，而是移交给分配器，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/mtl
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun/ptmalloc.ppt/mthread/mtl
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$
```

3. Arena

3.1. Arena 的数量

在以上的例子中我们可以看到，主线程包含 main arena 而 thread 1 包含它自己的 thread arena。所以线程和 arena 之间是否存在一一映射关系，而不论线程的数量有多大？当然不是，部分极端的应用甚至运行比处理器核数还多的线程，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。所以，arena 数量其实是限于系统核数的。

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

3.2. 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 个线程）的例子。这里线程数量（4）> 2 * 核心数（1），所以分配器中可能有 Arena（也即标题所称「multiple arenas」）会被所有线程共享。那么是如何共享的呢？

- 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 将被创建，且将被没有任何竞争地使用。此时线程和 arena 之间存在一一映射关系；
- 当 thread3 第一次调用 `malloc` 时，arena 的数量限制被计算出来，结果显示已超出，因此尝试复用已经存在的 arena（也即 Main arena 或 Arena 1 或 Arena 2）；
- 复用：
  - 一旦遍历到可用 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；
  - 如果没找到可用 arena，thread 3 的 `malloc` 将被阻塞，直到有可用的 arena 为止。
- 当thread 3 调用 `malloc` 时(第二次了)，分配器会尝试使用上一次使用的 arena（也即，main arena），从而尽量提高缓存命中率。当 main arena 可用时就用，否则 thread 3 就一直阻塞，直至 main arena 空闲。因此现在 main arena 实际上是被 main thread 和 thread 3 所共享。

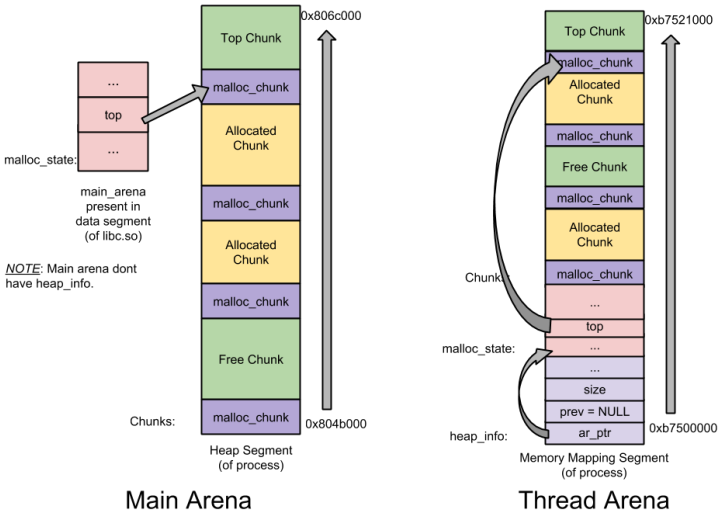
3.3. Multiple Heaps

在「glibc malloc」中主要有 3 种数据结构：

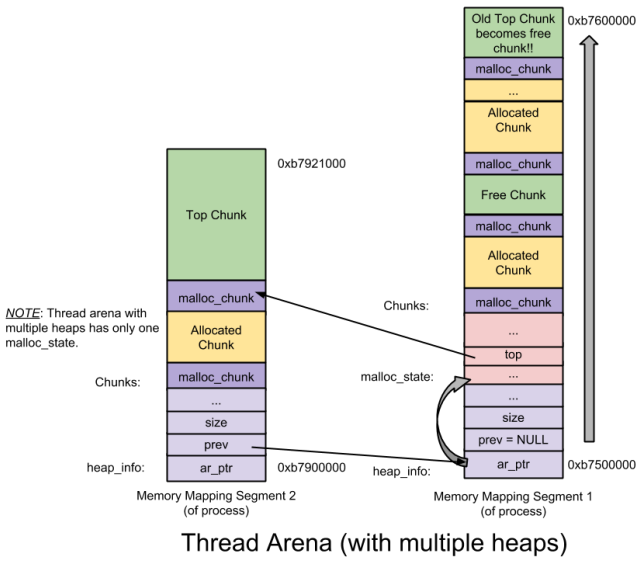
- heap_info ——Heap Header—— 一个 thread arena 可以维护多个堆。每个堆都有自己的堆 Header（注：也即头部元数据）。什么时候 Thread Arena 会维护多个堆呢？一般情况下，每个 thread arena 都只维护一个堆，但是当这个堆的空间耗尽时，新的堆（而非连续内存区域）就会被 `mmap` 到这个 aerna 里；
- malloc_state ——Arena header—— 一个 thread arena 可以维护多个堆，这些堆另外共享同一个 arena header。Arena header 描述的信息包括：bins、top chunk、last remainder chunk 等；
- malloc_chunk ——Chunk header—— 根据用户请求，每个堆被分为若干 chunk。每个 chunk 都有自己的 chunk header。

> **注意**：
>
> - Main arena 无需维护多个堆，因此也无需 heap_info。当空间耗尽时，与 thread arena 不同，main arena 可以通过 `sbrk` 拓展**堆**段，直至堆段「碰」到内存映射段；
> - 与 thread arena 不同，main arena 的 arena header 不是保存在通过 `sbrk` 申请的堆段里，而是作为一个全局变量，可以在 libc.so 的数据段中找到。

main arena 和 thread arena 的图示如下（单堆段）：



thread arena 的图示如下（多堆段）：



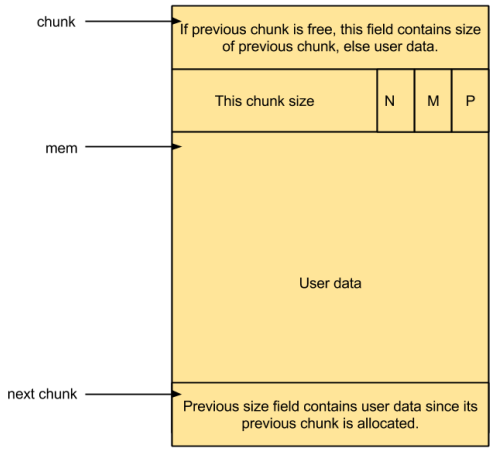
4. Chunk

堆段中存在的 chunk 类型如下：

- Allocated chunk;
- Free chunk;
- Top chunk;
- Last Remainder chunk.

4.1. Allocated chunk

「**Allocated chunck**」就是已经分配给用户的 chunk，其图示如下：



Allocated Chunk

图中左方三个箭头依次表示：

- chunk：该 Allocated chunk 的起始地址；
- mem：该 Allocated chunk 中用户可用区域的起始地址（`= chunk + sizeof(malloc_chunk)`）；
- next_chunk：下一个 chunck（无论类型）的起始地址。

图中结构体内部各字段的含义依次为：

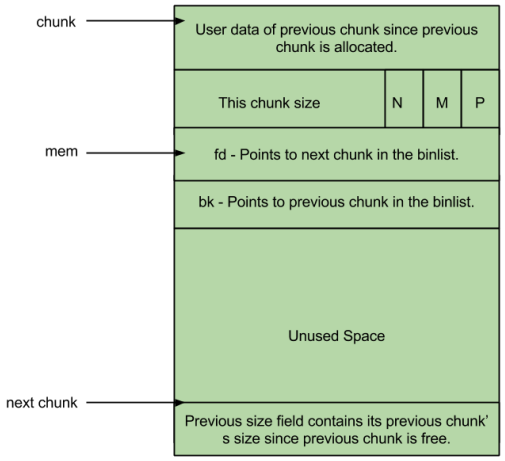
- prev_size：若上一个 chunk 可用，则此字段赋值为上一个 chunk 的大小；否则，此字段被用来存储上一个 chunk 的用户数据；
- size：此字段赋值本 chunk 的大小，其最后三位包含标志信息：
  - PREV_INUSE § – 置「1」表示上个 chunk 被分配；
  - IS_MMAPPED (M) – 置「1」表示这个 chunk 是通过 `mmap` 申请的（较大的内存）；

- NON_MAIN_ARENA (N) – 置「1」表示这个 chunk 属于一个 thread arena。

> **注意**：
>
> - malloc_chunk 中的其余结构成员，如 fd、 bk，没有使用的必要而拿来存储用户数据；
> - 用户请求的大小被转换为内部实际大小，因为需要额外空间存储 malloc_chunk，此外还需要考虑对齐。

4.2. Free chunk

「**Free chunck**」就是用户已释放的 chunk，其图示如下：



Free Chunk

图中结构体内部各字段的含义依次为：

- prev_size: 两个相邻 free chunk 会被合并成一个，因此该字段总是保存前一个 allocated chunk 的用户数据；
- size: 该字段保存本 free chunk 的大小；
- fd: Forward pointer —— 本字段指向同一 bin 中的下个 free chunk（free chunk 链表的前驱指针）；
- bk: Backward pointer —— 本字段指向同一 bin 中的上个 free chunk（free chunk 链表的后继指针）。

5. Bins

「**bins**」 就是空闲列表数据结构。它们用以保存 free chunks。根据其中 chunk 的大小，bins 被分为如下几种类型：

- Fast bin;
- Unsorted bin;
- Small bin;
- Large bin.

保存这些 bins 的字段为：

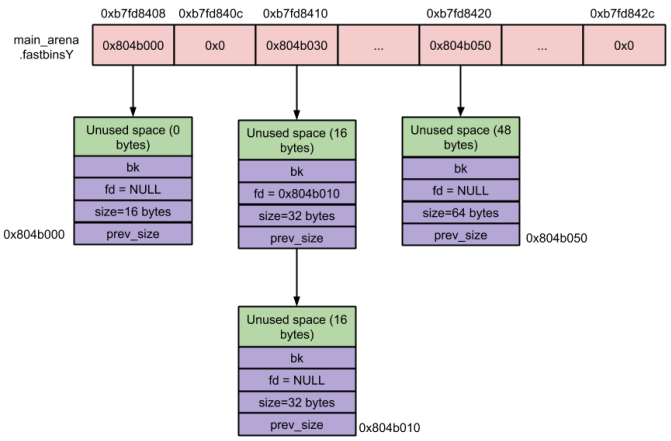
- fastbinsY: 这个数组用以保存 fast bins；
- bins: 这个数组用于保存 unsorted bin、small bins 以及 large bins，共计可容纳 126 个，其中：
  - Bin 1: unsorted bin;
  - Bin 2 - 63: small bins;
  - Bin 64 - 126: large bins.

5.1. Fast Bin

大小为 16 ~ 80 字节的 chunk 被称为「**fast chunk**」。在所有的 bins 中，fast bins 路径享有最快的内存分配及释放速度。

- **数量**：10
- 每个 fast bin 都维护着一条 free chunk 的单链表，采用单链表是因为链表中所有 chunk 的大小相等，增删 chunk 发生在链表顶端即可；—— LIFO
- **chunk 大小**：8 字节递增
- fast bins 由一系列所维护 chunk 大小以 8 字节递增的 bins 组成。也即， `fast bin[0]` 维护大小为 16 字节的 chunk、 `fast bin[1]` 维护大小为 24 字节的 chunk。依此类推……
- 指定 fast bin 中所有 chunk 大小相同；

- 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 被归为 fast chunk 。
- 无需合并 —— 两个相邻 chunk 不会被合并。虽然这可能会加剧内存碎片化，但也大大加速了内存释放的速度！
- `malloc(fast chunk)`
- 初始情况下 fast chunck 最大尺寸以及 fast bin 相应数据结构均未初始化，因此即使用户请求内存大小落在 fast chunk 相应区间，服务用户请求的也将是 small bin 路径而非 fast bin 路径；
- 初始化后，将在计算 fast bin 索引后检索相应 bin；
- 相应 bin 中被检索的第一个 chunk 将被摘除并返回给用户。
- `free(fast chunk)`
  - 计算 fast bin 索引以索引相应 bin；
  - `free` 掉的 chunk 将被添加到上述 bin 的顶端。



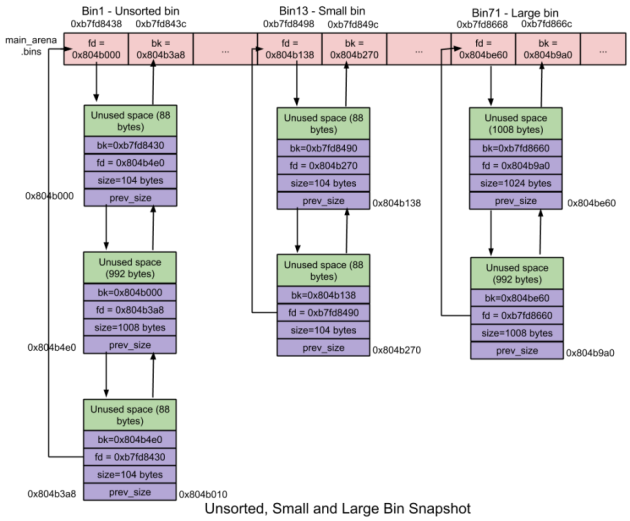
Fast Bin Snapshot

5.2. 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。这使得分配器可以重新使用最近 `free` 掉的 chunk，从而消除了寻找合适 bin 的时间开销，进而加速了内存分配及释放的效率。

> 译者注：经 @kwdecsdn 提醒，这里应补充说明「Unsorted Bin 中的 chunks 何时移至 small/large chunk 中」。在内存分配的时候，在前后检索 fast/small bins 未果之后，在特定条件下，会将 unsorted bin 中的 chunks 转移到合适的 bin 中去，small/large。

- **数量**：1
- unsorted bin 包括一个用于保存 free chunk 的双向循环链表（又名 binlist）；
- **chunk 大小**：无限制，任何大小的 chunk 均可添加到这里。



Unsorted, Small and Large Bin Snapshot

5.3. Small 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 都维护着一条 free chunk 的双向循环链表。采用双向链表的原因是，small bins 中的 chunk 可能会从链表中部摘除。这里新增项放在链表的头部位置，而从链表的尾部位置移除项。—— FIFO
- **chunk 大小**：8 字节递增
  - Small bins 由一系列所维护 chunk 大小以 8 字节递增的 bins 组成。举例而言， `small bin[0]` （Bin 2）维护着大小为 16 字节的 chunks、 `small bin[1]` （Bin 3）维护着大小为 24 字节的 chunks ，依此类推……
  - 指定 small bin 中所有 chunk 大小均相同，因此无需排序；

- 合并 —— 相邻的 free chunk 将被合并，这减缓了内存碎片化，但是减慢了 `free` 的速度；

- `malloc(small chunk)`
  - 初始情况下，small bins 都是 NULL，因此尽管用户请求 small chunk ，提供服务的将是 unsorted bin 路径而不是 small bin 路径；
  - 第一次调用 `malloc` 时，维护在 malloc_state 中的 small bins 和 large bins 将被初始化，它们都会指向自身以表示其为空；
  - 此后当 small bin 非空，相应的 bin 会摘除其中最后一个 chunk 并返回给用户；

- `free(small chunk)`
  - `free` chunk 的时候，检查其前后的 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 的前端。

5.4. 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
  - 32 个 bins 所维护的 chunk 大小以 64B 递增，也即 `large chunk[0]` (Bin 65) 维护着大小为 512B ~ 568B 的 chunk 、 `large chunk[1]` (Bin 66) 维护着大小为 576B ~ 632B 的 chunk，依此类推……
  - 16 个 bins 所维护的 chunk 大小以 512 字节递增；
  - 8 个 bins 所维护的 chunk 大小以 4096 字节递增；
  - 4 个 bins 所维护的 chunk 大小以 32768 字节递增；
  - 2 个 bins 所维护的 chunk 大小以 262144 字节递增；
  - 1 个 bin 维护所有剩余 chunk 大小；
  - 每个 large bin 都维护着一条 free chunk 的双向循环链表。采用双向链表的原因是，large bins 中的 chunk 可能会从链表中的任意位置插入及删除。
  - 这 63 个 bins

    - 不像 small bin ，large bin 中所有 chunk 大小不一定相同，各 chunk 大小递减保存。最大的 chunk 保存顶端，而最小的 chunk 保存在尾端；

- 合并 —— 两个相邻的空闲 chunk 会被合并；

- `malloc(large chunk)`

    - User chunk（用户请求大小）—— 返回给用户；
    - Remainder chunk （剩余大小）—— 添加到 unsorted bin。
    - 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 路径而不是 large bin 路劲 。
    - 第一次调用 `malloc` 时，维护在 malloc_state 中的 small bin 和 large bin 将被初始化，它们都会指向自身以表示其为空；
    - 此后当 large bin 非空，如果相应 bin 中的最大 chunk 大小大于用户请求大小，分配器就从该 bin 顶端遍历到尾端，以找到一个大小最接近用户请求的 chunk。一旦找到，相应 chunk 就会被切分成两块：
    - 如果相应 bin 中的最大 chunk 大小小于用户请求大小，分配器就会扫描 binmaps，从而查找最小非空 bin。如果找到了这样的 bin，就从中选择合适的 chunk 并切割给用户；反之就使用 top chunk 响应用户请求。

- `free(large chunk)` —— 类似于 small chunk 。

5.5. Top Chunk

一个 arena 中最顶部的 chunk 被称为「**top chunk**」。它不属于任何 bin 。当所有 bin 中都没有合适空闲内存时，就会使用 top chunk 来响应用户请求。

当 top chunk 的大小比用户请求的大小大的时候，top chunk 会分割为两个部分：

- User chunk，返回给用户；
- Remainder chunk，剩余部分，将成为新的 top chunk。

当 top chunk 的大小比用户请求的大小小的时候，top chunk 就通过 `sbrk` （main arena）或 `mmap` （ thread arena）系统调用扩容。

5.6. Last Remainder Chunk

「**last remainder chunk**」即最后一次 small request 中因分割而得到的剩余部分，它有利于改进引用局部性，也即后续对 small chunk 的 `malloc` 请求可能最终被分配得彼此靠近。

那么 arena 中的若干 chunks，哪个有资格成为 last remainder chunk 呢?

当用户请求 small chunk 而无法从 small bin 和 unsorted bin 得到服务时，分配器就会通过扫描 binmaps 找到最小非空 bin。正如前文所提及的，如果这样的 bin 找到了，其中最合适的 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。这一 Remainder chunk 就将成为 last remainder chunk。

那么引用局部性是如何达成的呢?

当用户的后续请求 small chunk，并且 last remainder chunk 是 unsorted bin 中唯一的 chunk，该 last remainder chunk 就将分割成两部分：返回给用户的 User chunk、添加到 unsorted bin 中的 Remainder chunk（也是 last remainder chunk）。因此后续的请求的 chunk 最终将被分配得彼此靠近。

刘翔,童薇,刘景宁,冯丹,陈劲龙.动态内存分配器研究综述[J].计算机学报,2018,41(10):2359-2378.

> 译者：猫科龙
> https://blog.csdn.net/maokelong95/article/details/51989081

- EOF -

推荐阅读 — 点击标题可跳转

1、图解 Linux | 管道通信的原理?

2、深入理解虚拟化

3、C++ 按值返回对象那些事

公众号

点赞和在看就是最大的支持❤️

People who liked this content also liked

Python+Flask+MySQL实现的学生培养计划管理系统

Python联盟

在 Linux 上学习 C 语言的五种方式 | Linux 中国

Linux中国

一学就会：如何在 Linux 中挂载远程文件系统或目录

Linux公社