深入理解 malloc

2019年8月19日 | 操作系统 |657 阅读

本文是基于英文博客 Understanding glibc malloc ，对内容做了大量的补充和修改，主要阐释了 `malloc` 分配内存的底层实现原理。

我一直在执着于堆的一些问题。比如以下问题：

- 堆的内存怎样从内核中申请的?
- 怎样有效地进行内存管理?
- 堆内存是通过内核，库还是堆本身进行管理?
- 堆的一些相关问题能被利用吗?

虽然之前经常在想这些问题，但是光想并没有什么用。正好，最近我找到了点时间来好好思考这些问题。所以现在我就来分享一下这些知识的总结。此外，还有很多可用的内存分配器：

- `dlmalloc` – 通用分配器
- `ptmalloc2` – glibc
- `jemalloc` – FreeBSD and Firefox
- `tcmalloc`– Google

- `libumem` – Solaris

每种内存分配器都声称自己速度快、可扩展、空间利用高效！！但是并非所有的分配器都适合我们的程序。内存消耗大的应用，其性能很大程度依赖于内存分配器的性能。本文仅讨论 `glibc malloc` 内存分配器。

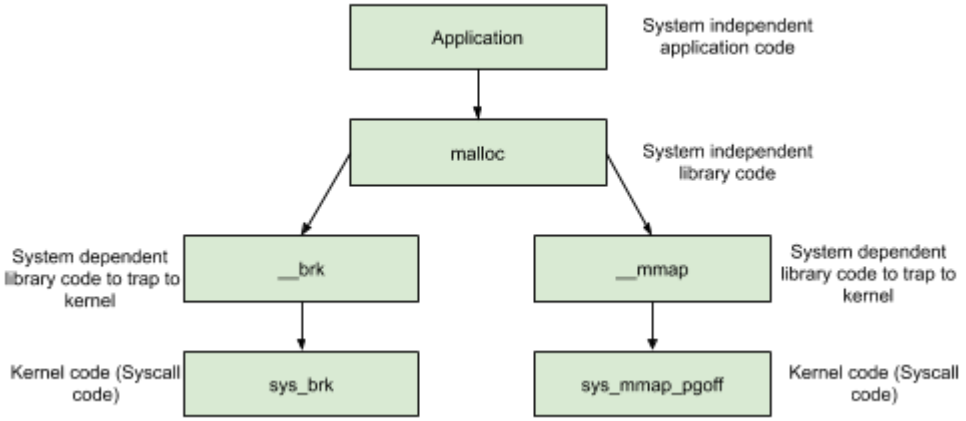
简介

`ptmalloc2` 来自于 `dlmalloc` 的分支。在其基础上添加线程支持并于 2006 年发布。正式发布后， `patmalloc2` 集成到 `glibc` 源码中。随着源码集成，代码修改便直接在 `glibc malloc` 源码里进行。因此 `ptmalloc2` 的实现与 `glibc malloc` 有很多不同。

在早期的 `Linux` 里， `dlmalloc` 被用做默认的内存分配器。但之后因为 `ptmalloc2` 添加了线程支持， `ptmalloc2` 成为了 `Linux` 默认内存分配器。线程支持可帮助提升内存分配器以及应用程序的性能。在 `dlmalloc` 里，当两个线程同时调用 `malloc` 时，只有一个线程能进入到临界段，因为这里的空闲列表是所有可用线程共用的。因此内存分配器要在多线程应用里耗费时间，从而导致性能降低。然而在 `ptmalloc2` 里，当两个线程同时调用 `malloc` 时，会立即分配内存。因为每个线程维护一个单独的堆分段，因此空闲列表维护的这些堆也是独立的。这种维护独立堆以及每一个线程享有空闲列表数据结构的行为被称为 `Per Thread Arena` 。

malloc 的实现

`malloc` 有两种方式获取内存，分别为 `sbrk` 和 `mmap` ，以下为示意图：



我们来看一下关于这两个系统调用的官方解释：

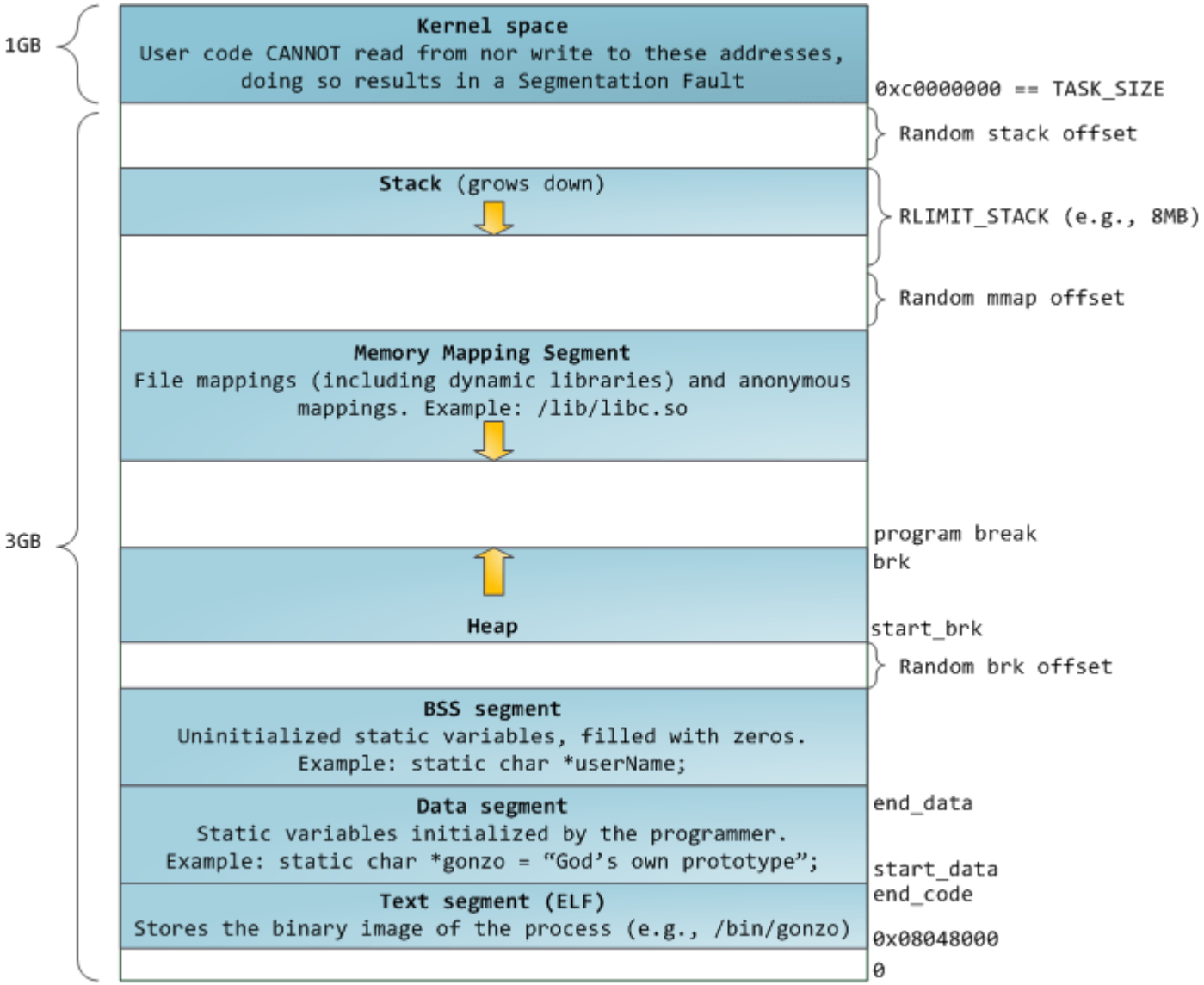
`sbrk`：

> The brk() function sets the break or lowest address of a process's data segment (uninitialized data) to addr (immediately above bss). Data addressing is restricted between addr and the lowest stack pointer to the stack segment.

`mmap`：

> The mmap() system call causes the pages starting at addr and continuing for at most len bytes to be mapped from the object described by fd, starting at byte offset offset.

两者一个明显区别在于，通过 `sbrk` 获得的新的堆的内存地址和之前的地址是连续的，而 `mmap` 获得的地址由参数设定。下图为内存分配的示意图：



内存分析方法

在进行进一步的分析之前，我们先要找到合适的方法，分析程序的内存分配情况。本文通过读取 `/proc/$pid/maps` ，该文件的具体内容可以通过 `man 5 proc` 来了解，其的部分的解释如下：

```
address           perms  offset    dev    inode       pathname
00400000-00452000 r-xp   00000000  08:02  173521      /usr/bin/dbus-daemon
00651000-00652000 r--p   00051000  08:02  173521      /usr/bin/dbus-daemon
00652000-00655000 rw-p   00052000  08:02  173521      /usr/bin/dbus-daemon
00e03000-00e24000 rw-p   00000000  00:00  0           [heap]
00e24000-011f7000 rw-p   00000000  00:00  0           [heap]
```

`perms` 代表了内存的权限，有5种格式：

- `r` = read
- `w` = write
- `x` = execute
- `s` = shared
- `p` = private (copy on write)

`offset` 字段是文件中的偏移量； `dev` 是设备（主要：次要）； `inode` 是该设备上的 `inode` 。 0表示没有 `inode` 与内存区域相关联，就像 `.BSS` （未初始化的数据存放的 `section` ）

`pathname` 字段指向映射的文件。同时会提供几种伪地址： - `[stack]` ：进程的栈 - `[stack:<tid>]` ：各个线程的栈 - `[vdso]` ：虚拟动态共享对象 - `[heap]` ：进程的堆

具体分析

我们用C语言编写一个简单的程序，在其中调用 `malloc` 来动态分配内存，并使用 `getchar` 使程序暂停，帮助我们查看程序的内存分配具体情况，源代码如下：

```c
1 /* Per thread arena example. */
```

```c
#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 * sizeof(char));
    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(6553500 * sizeof(char));
    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;
}
```

我们运行程序，检测输出的内容：

```
$ gcc main.c -lpthread -o mthread
$ ./mthread
Welcome to per thread arena example::13201
Before malloc in main thread
```

我们根据程序 `PID` 查看程序的内存分配情况：

```
$ cat /proc/13140/maps
56424b4c2000-56424b4c3000 r-xp 00000000 08:01 673794                     /home/zhf/mthread
56424b6c2000-56424b6c3000 r--p 00000000 08:01 673794                     /home/zhf/mthread
56424b6c3000-56424b6c4000 rw-p 00001000 08:01 673794                     /home/zhf/mthread
56424c8db000-56424c8fc000 rw-p 00000000 00:00 0                          [heap]
...
```

此时主进程并没有调用 `malloc` ，但进程已经初始化部分内存空间作为进程的堆，地址为 `56424c8db000-56424c8fc000` ，大小为132KB。这个堆内存的连续区域被称为 `arena` 。这个 `arena` 是由主线程创建，则被称为 `main arena` 。进一步的分配请求会继续使用这个 `arena` 直到 `arena 空闲空间耗尽` 。：

```
...
After malloc and before free in main thread
...
$ cat /proc/13201/maps
56424b4c2000-56424b4c3000 r-xp 00000000 08:01 673794                     /home/zhf/mthread
56424b6c2000-56424b6c3000 r--p 00000000 08:01 673794                     /home/zhf/mthread
56424b6c3000-56424b6c4000 rw-p 00001000 08:01 673794                     /home/zhf/mthread
56424c8db000-56424c8fc000 rw-p 00000000 00:00 0                          [heap]
7f6075cd0000-7f6076310000 rw-p 00000000 00:00 0
...
```

由于我们在使用 `malloc` 申请了一块较大的地址，原有的堆空间无法满足需求，因此会使用 `mmap` 系统调用，进一步扩张 `arena` 的区域，新申请的内存区域与之前的堆相连，地址为 `7f6075cd0000-7f6076310000` 。接着主线程 `free` 内存块。

```
...
After free in main thread
...
$ cat /proc/13201/maps
56424b4c2000-56424b4c3000 r-xp 00000000 08:01 673794                     /home/zhf/mthread
56424b6c2000-56424b6c3000 r--p 00000000 08:01 673794                     /home/zhf/mthread
56424b6c3000-56424b6c4000 rw-p 00001000 08:01 673794                     /home/zhf/mthread
56424c8db000-56424c8fc000 rw-p 00000000 00:00 0                          [heap]
...
```

在主线程 `free` 之后： 在上面的输出里我们可以看到，当分配内存区域被释放时，其后内存不会被立即释放给操作系统。分配内存区域（1000 bytes 大小）只释放给 `glibc malloc` 库，在这里的释放掉的 `Chunk` 会被添加到 `main arenas` 中（在 `glibc malloc` 里， `freelist` 被称为 `bins` ）。此后当用户申请内存时， `glibc malloc` 不会从内核中获得新的堆内存，而是尽量在 `bins` 里找到一个空闲块(`Free Chunk`)。只有当没有空闲块存在时， `glibc malloc` 才会从继续内核中申请内存。

```
...
Before malloc in thread 1
...
$ cat /proc/13201/maps
56424b4c2000-56424b4c3000 r-xp 00000000 08:01 673794                     /home/zhf/mthread
56424b6c2000-56424b6c3000 r--p 00000000 08:01 673794                     /home/zhf/mthread
56424b6c3000-56424b6c4000 rw-p 00001000 08:01 673794                     /home/zhf/mthread
56424c8db000-56424c8fc000 rw-p 00000000 00:00 0                          [heap]
7f6075b0f000-7f6075b10000 ---p 00000000 00:00 0
7f6075b10000-7f6076310000 rw-p 00000000 00:00 0
...
```

接着我们创建线程，我们可以看到，在线程调用 `malloc` 之前，已经为线程分配好了线程堆，其地址为 `7f6075b10000-7f6076310000` 。我们可以看到线程堆的地址进程堆的地址并不连续，这表明堆内存通过使用 `mmap` 系统调用而不是主线程（使用 `sbrk` ）创建。

```
...
After malloc and before free in thread 1
...
$ cat /proc/13201/maps
56424b4c2000-56424b4c3000 r-xp 00000000 08:01 673794                     /home/zhf/mthread
56424b6c2000-56424b6c3000 r--p 00000000 08:01 673794                     /home/zhf/mthread
56424b6c3000-56424b6c4000 rw-p 00001000 08:01 673794                     /home/zhf/mthread
56424c8db000-56424c8fc000 rw-p 00000000 00:00 0                          [heap]
7f6070000000-7f6070031000 rw-p 00000000 00:00 0
7f6070031000-7f6074000000 ---p 00000000 00:00 0
7f6075b0f000-7f6075b10000 ---p 00000000 00:00 0
7f6075b10000-7f6076310000 rw-p 00000000 00:00 0
...
```

在线程 `malloc` 之后，出现一个新的线程堆地址，其中 `7f6070000000-7f6070031000` 有读写权限， `7f6070031000-7f6074000000` 仅支持写时拷贝(`copy on write`)。 `7f6070000000-7f6070031000` 这块内存区域被称为 `thread arena` 。

注意：当用户申请的内存大小超过 128KB （ `malloc(132*1024)` ）并且当一个 `arena` 里没有足够的空间来满足用户的请求时，内存是使用 `mmap` 系统调用来分配的（不使用 `sbrk` ） 无论这个请求是来自于 `main arena` 还是 `thread arena` 。

```
...
After free in thread 1
...
$ cat /proc/13201/maps
56424b4c2000-56424b4c3000 r-xp 00000000 08:01 673794                     /home/zhf/mthread
56424b6c2000-56424b6c3000 r--p 00000000 08:01 673794                     /home/zhf/mthread
56424b6c3000-56424b6c4000 rw-p 00001000 08:01 673794                     /home/zhf/mthread
56424c8db000-56424c8fc000 rw-p 00000000 00:00 0                          [heap]
7f6070000000-7f6070031000 rw-p 00000000 00:00 0
7f6070031000-7f6074000000 ---p 00000000 00:00 0
7f6075b0f000-7f6075b10000 ---p 00000000 00:00 0
7f6075b10000-7f6076310000 rw-p 00000000 00:00 0
...
```

在线程 `free` 之后后：在上面的输出我们可以看到，释放的堆内存并不会给操作系统。相反，这块内存区域还给了 `glibc malloc` 里，并将这个释放块添加到 `thread arenas` 的 `bins` 里，由 `freelist` 维护。

Arena 共享

在以上示例中，我们看到了主线程包含 `main arena` 同时每个线程包含了它自己的 `thread arena` 。那么是不是无论多少线程，每个线程都有自己独立的 `arena` 呢？显然不是，一个进程可以包含比 `CPU` 核数量更多的线程数量，在这样的情况下，每个线程单独有一个 `arena` ，其代价十分昂贵。因此，应用程序的arena 数量的限制是基于系统里现有的 CPU 核的数量。

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

假设我们有一个多线程的程序（4线程 - 主线程 + 3个用户线程），在一个单核 32 位系统上运行。这里线程数量 > 2 * 核心数量。因此，`glibc malloc` 认定 `Multiple Arena` 被所有可用进程共享。但它是怎样共享的呢?

- 当主线程第一次调用 `malloc` 时，glibc malloc 会直接为它分配一个 main arena，不需要任何的附加条件
- 当用户线程1和用户线程2第一次调用 `malloc` 时，会为这些线程创建一个新的 arena 。此时，各个线程与 `arena`是一一对应的。
- 当户线程3第一次调用 `malloc` 时， 此时 `glibc malloc` 能维持的 `arena` 数量已到达上限，因此尝试重用 现存的 `arena` （`main arena`、 `arena 1` 或 `arena 2`）。
    - 遍历了所有可用的 `arena`，尽量去锁定可用的 `arena`。
    - 如果锁定成功（我们假设说 `main arena` 被锁定成功），就向用户返回该 `arena`。
    - 如果没有 `arena` 是空闲的，那么就将线程3的 `malloc`操作 阻塞，直到有可用的arena为止。。
- 现在当用户线程3第二次调用 `malloc` 时， `malloc` 会尽量使用上次访问的 arena （ `main arena`）。如果 `main arena` 是空闲的， 用户线程3会一直使用该 `arena` 并屏蔽其他线程的申请直到 `main arena` 被释放。 `main arena` 就是这样在主线程和用户线程3间共享。

堆结构

在 `glibc malloc` 源代码里主要发现以下三种数据结构：

- `heap_info` —即 `heap header`，因为一个 `thread arena`(注意：不包含 `main thread`) 可以包含多个heaps，所以为了便于管理，就给每个heap分配一个 `heap header`。那么在什么情况下一个 `thread arena` 会包含多个heaps呢。在当前heap不够用的时候， `malloc` 会通过系统调用 `mmap` 申请新的堆空间，新的堆空间会被添加到当前 `thread arena` 中，便于管理。

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

- `malloc_state` — 即 `Arena Header`，每个线程只含有一个 `Arena Header`。 `Area Header` 包含bins信息，top chunk 以及 last remainder chunk 等。

```
struct malloc_state
{
    /* Serialize access.  */
    mutex_t 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.  */
    struct malloc_state *next_free;

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

- `malloc_chunk` — 即 `chunk header`。一个 header 被分为多个 chunk，至于每个chunk 的大小，由用户的请求决定，也就是说用户调用 `malloc(size)`传递的 `size`参数”就是”chunk的大小（这里给“就是”加上引号，说明这种表示并不正确，但是为了方便立即就暂时这么描述了）。每个chunk都由一个结构体 `malloc_chunk`表示。

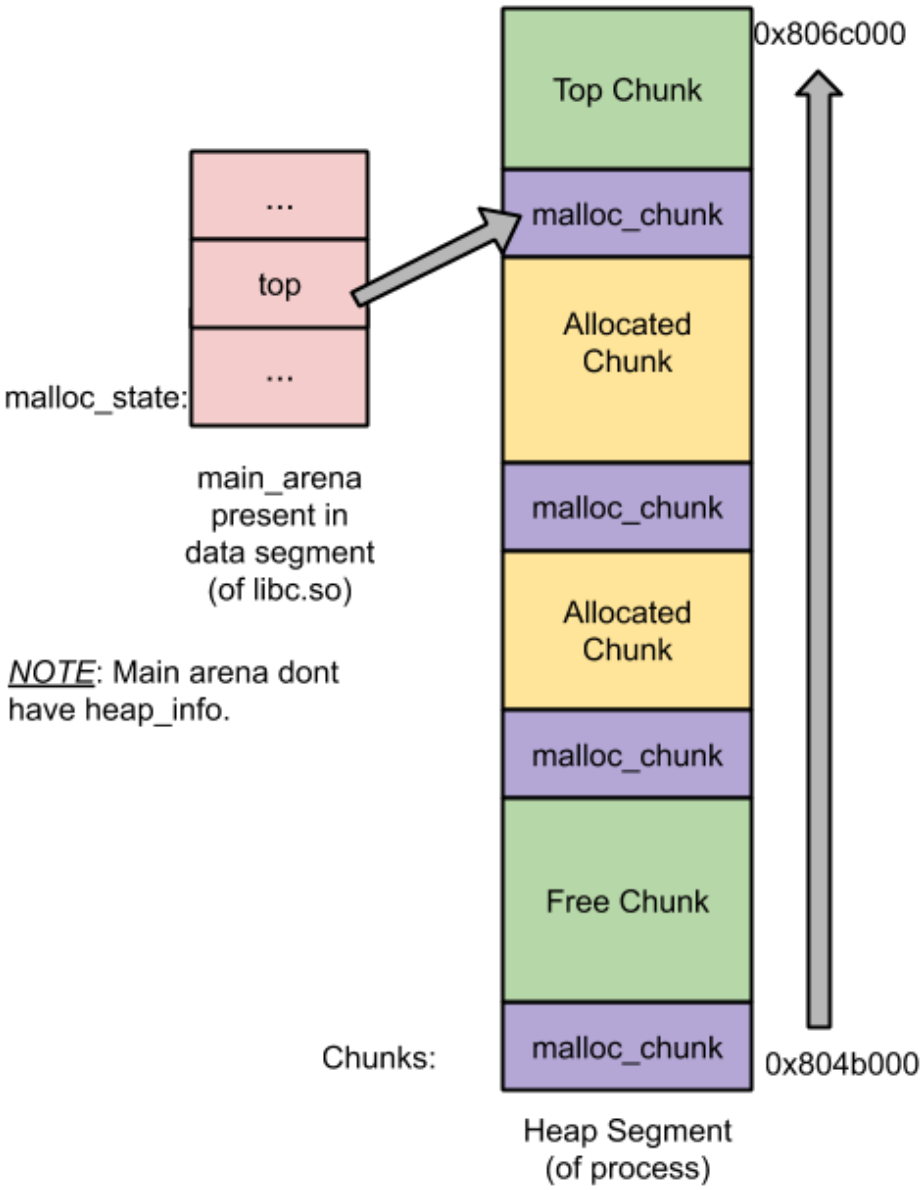
```
struct malloc_chunk {
    /* #define INTERNAL_SIZE_T size_t */
    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */
    struct malloc_chunk* fd;         /* double links -- used only if free. 这两个指针只在free chunk中存在*/
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size.  */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
```

注意：

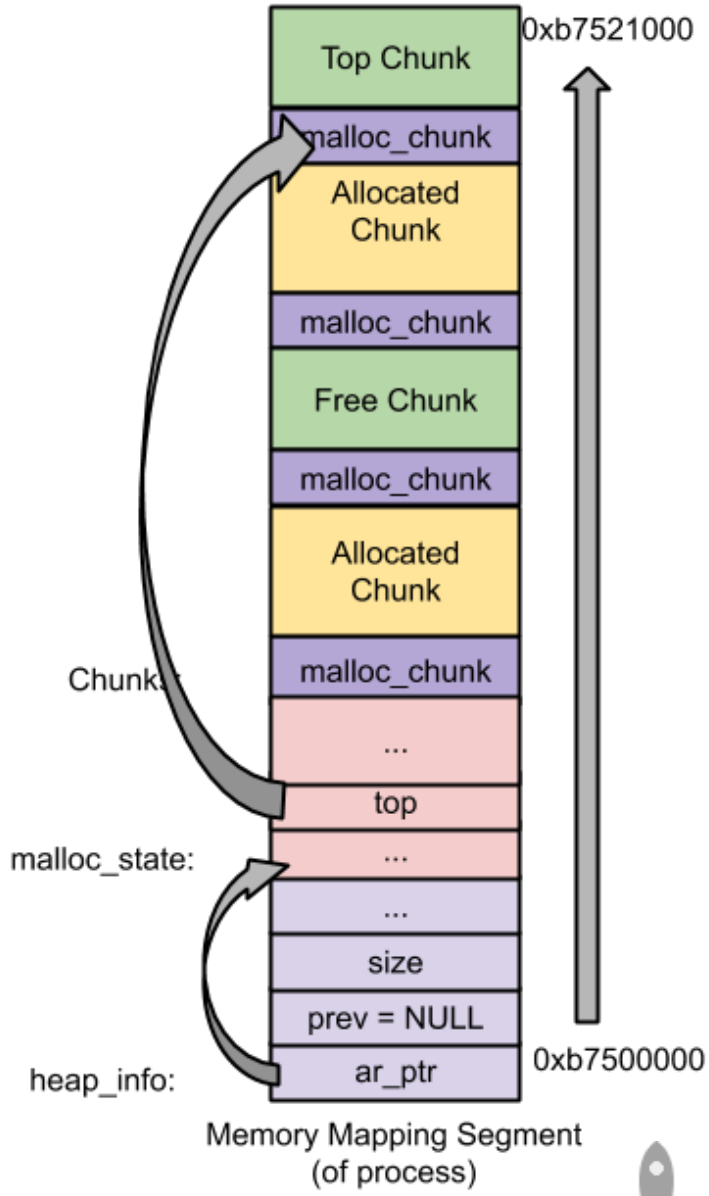
- `Main arena` 没有多重堆，因此没有 `heap_info` 结构体。当 `main arena` 耗尽空间时，就通过扩展 `sbrk` 的heap segment来获取更多的空间，直到它碰到内存mapping区域为止。
- 不同于 `thread arena` ， `main arena` 的 `Arena header` 不是 `sbrk sbrk heap segment` 堆分段的一部分。它是一个全局变量，因此它属于 lib.so 的 data segment。

首先，通过内存分布图解释清 `malloc_state` 与 `heap_info` 之间的组织关系。

下面是只有一个 heap segment 的 `main arena` 和 `thread arena` 的内存分布图：：

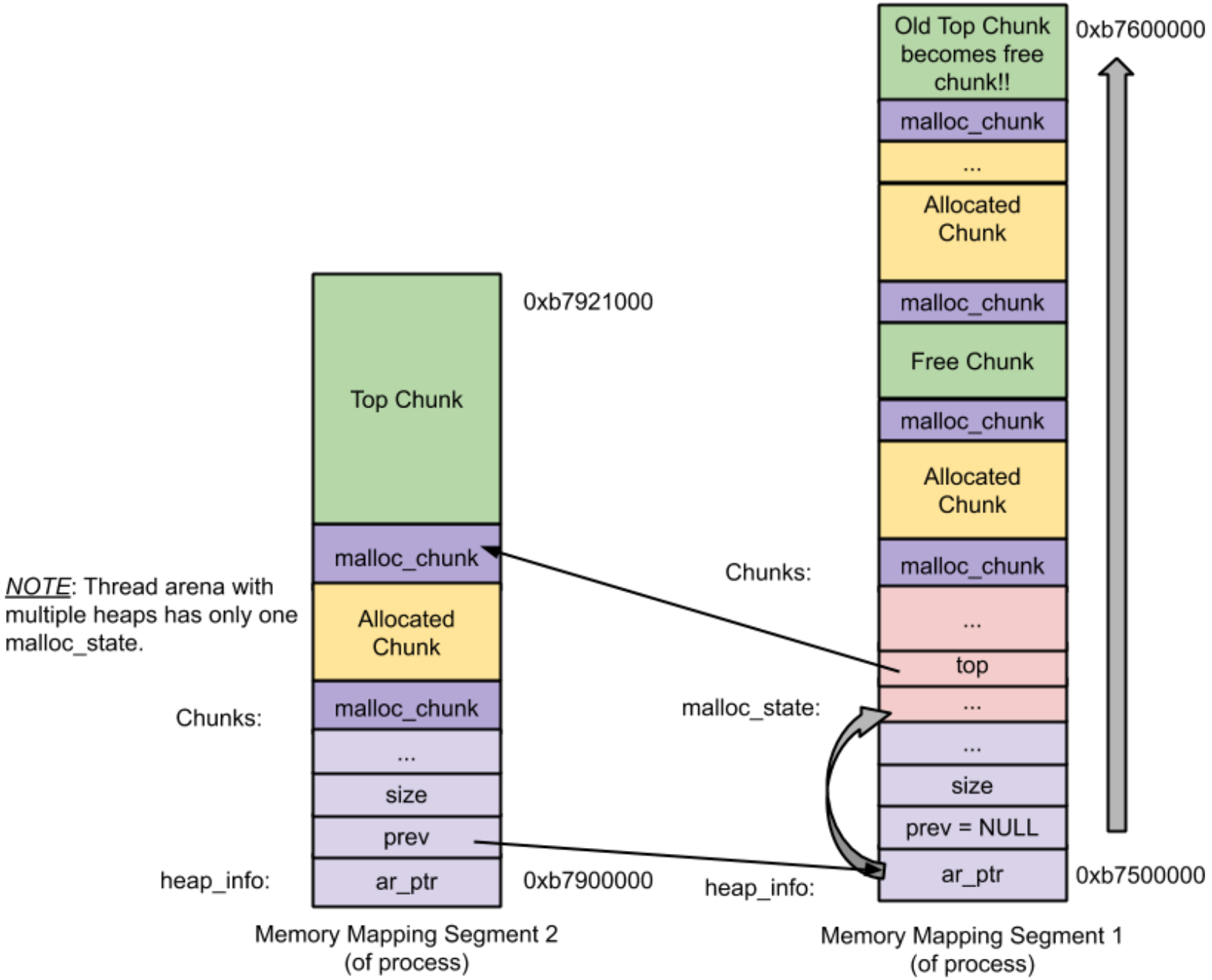


Main Arena



Thread Arena

下面是一个 `thread arena` 中含有多个 heap segments 的情况：



Thread Arena (with multiple heaps)

上图可以看出，`thread arena` 只含有一个 `malloc_state` （即arena header），却有两个 `heap_info` (即heap header)。由于两个 heap segment 是通过 `mmap` 分配内存，两者 heap segments 是通过 `mmap` 分配的内存，两者在内存布局上并不相邻而是分属于不同的内存区间，所以为了便于管理，`libc malloc` 将第二个 `heap_info` 结构体的 `prev` 成员指向第一个 `heap_info` 结构体的起始位置（即 `ar_ptr` 成员），而第一个 `heap_info` 结构体的 `ar_ptr` 成员指向了 `malloc_state` ，这样就构成一个单链表，方便以后管理。

Chunk

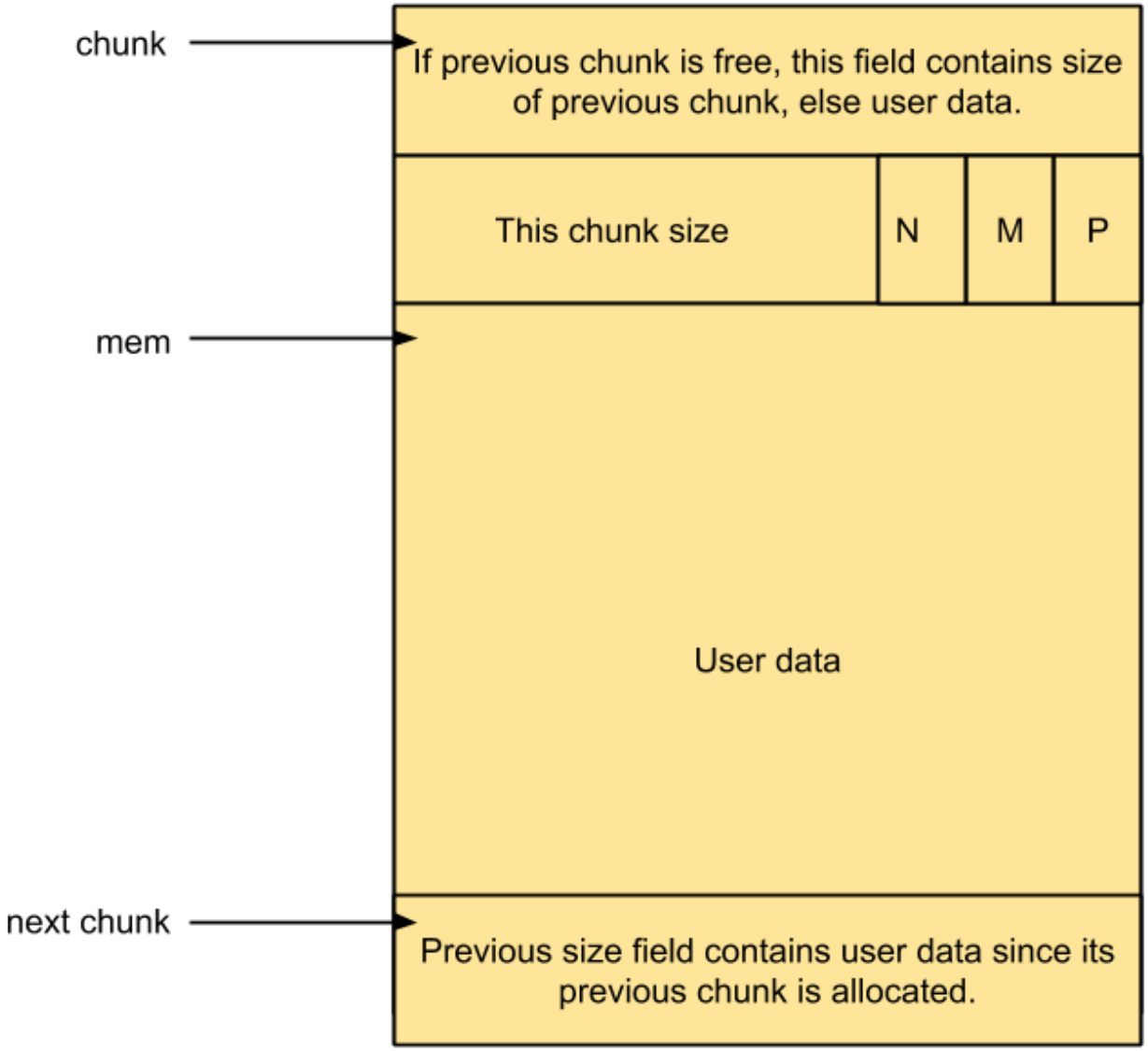
在堆里的块可以是以下几种类型中的一种：

- `Allocated chunk`—— 分配后的块
- `Free chunk` —— 空闲块
- `Top chunk` —— 开始块
- `Last Remainder chunk` —— 最后剩余块

从本质上来说，所有类型的 chunk 都是内存中一块连续的区域，只是通过该区域中特定位置的某些标识符加以区分。为了简便，我们先将这4类 chunk 简化为2类：`allocate chunk` 以及 `free chunk` ，前者标识已经分配给用户使用的 chunk，后者表示未使用的 chunk。

Allocated chunk

任何堆内存管理器都是以chunk为单位进行堆内存管理的，而这就需要一些数据结构来标志各个块的边界，以及区分已经分配块和空闲块。大多数堆内存管理都将这些边界信息作为 chunk 的一部分嵌入到 chunk 内部，下图为 allocated chunk 的结构：



Allocated Chunk

`size` ： 这部分包含了此处已分配的块的容量大小。

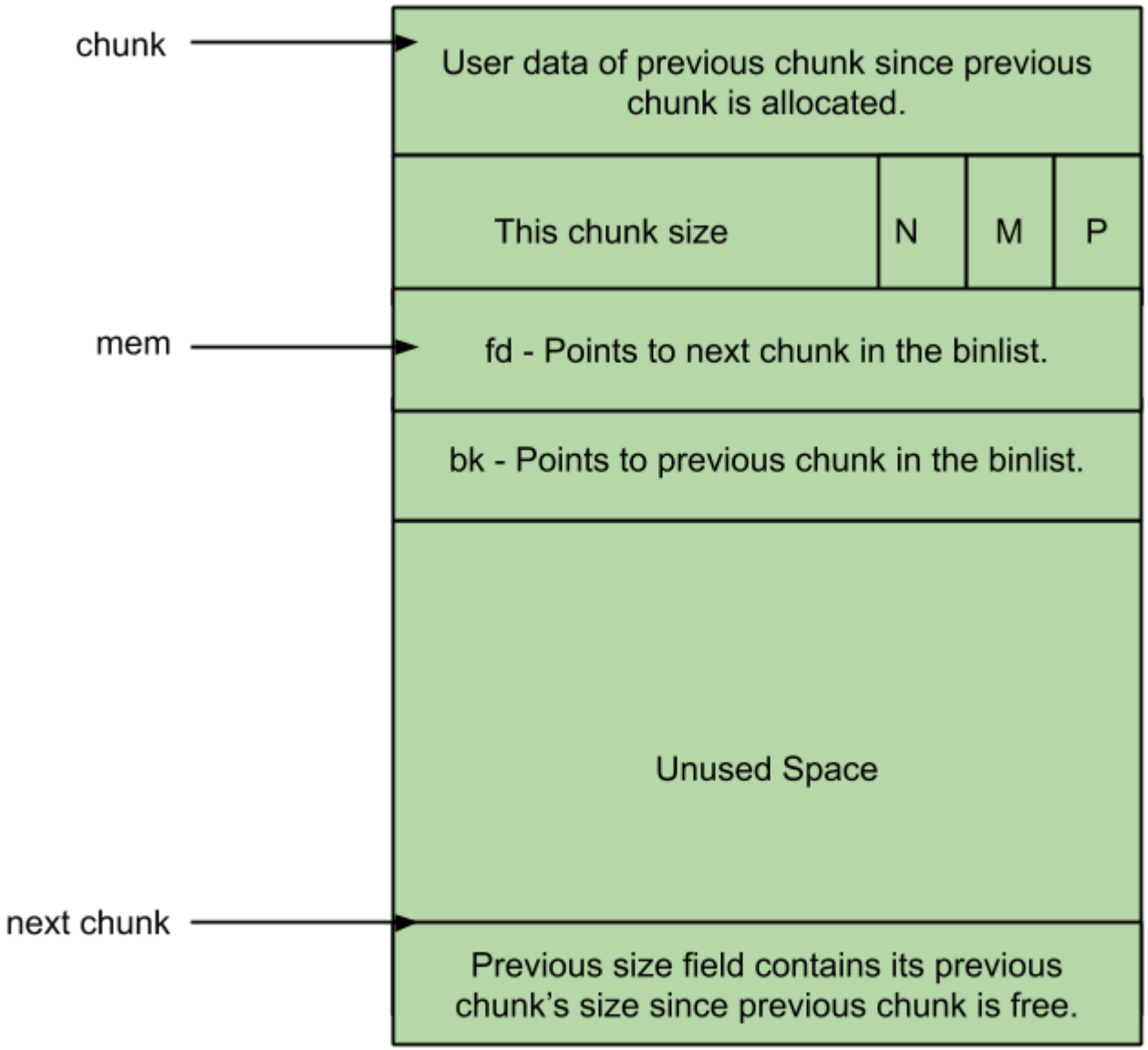
堆内存中要求每个chunk的大小必须为8的整数倍，因此chunk size的后3位是无效，为了充分利用内存，堆管理奖这3个bit比特位用作chunk的标志位，最后三位包含以下信息：

- `PREV_INUSE(P)` ——如果之前的块已经被分配，该位置1
- `IS_MMAPPED(M)` —— 如果块被映射，该位置1
- `NON_MAIN_ARENA(N)` —— 如果该快属于一个 `thread arena`，该位置1

请注意以下几点：

- allocated chunk 没有其他 `malloc_chunk`（比如 `fd – forward pointer，bk – back pointer`）。因此这部分区域会用来存储用户信息。
- 由于存储 `malloc_chunk` 需要一些额外的空间，用户请求的容量需要转换成实际需要的容量。转化不会改变最优 3 bits，因此它们用于存储关键信息。

Free Chunk



Free Chunk

以下是空闲块各个部分内容的说明：

- `prev_size`：不能同时调整两个空闲的块。当两个块都空闲时，它就会与一个单独的空闲块连接。因此前一个块及当前这个释放的块会被分配，同时 `prev_size` 包含前一个块的用户数据。
- `size`：这个部分包含有空闲块的 `size`
- `fd`：Forward pointer —— 同一 `bin`里指向下一块的指针（不是指向物理内存内的下一块）
- `bk`：Backward pointer —— 同一 `bin`里指向前一块的指针（不是指向物理内存内的前一块）

Bins

`bin` 是一种 `freelist` 数据结构，他们用来管理空闲的块。根据快的大小，以下为几不同的 `bins` ：

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

用来装载这些 `bins` 的数据结构有：

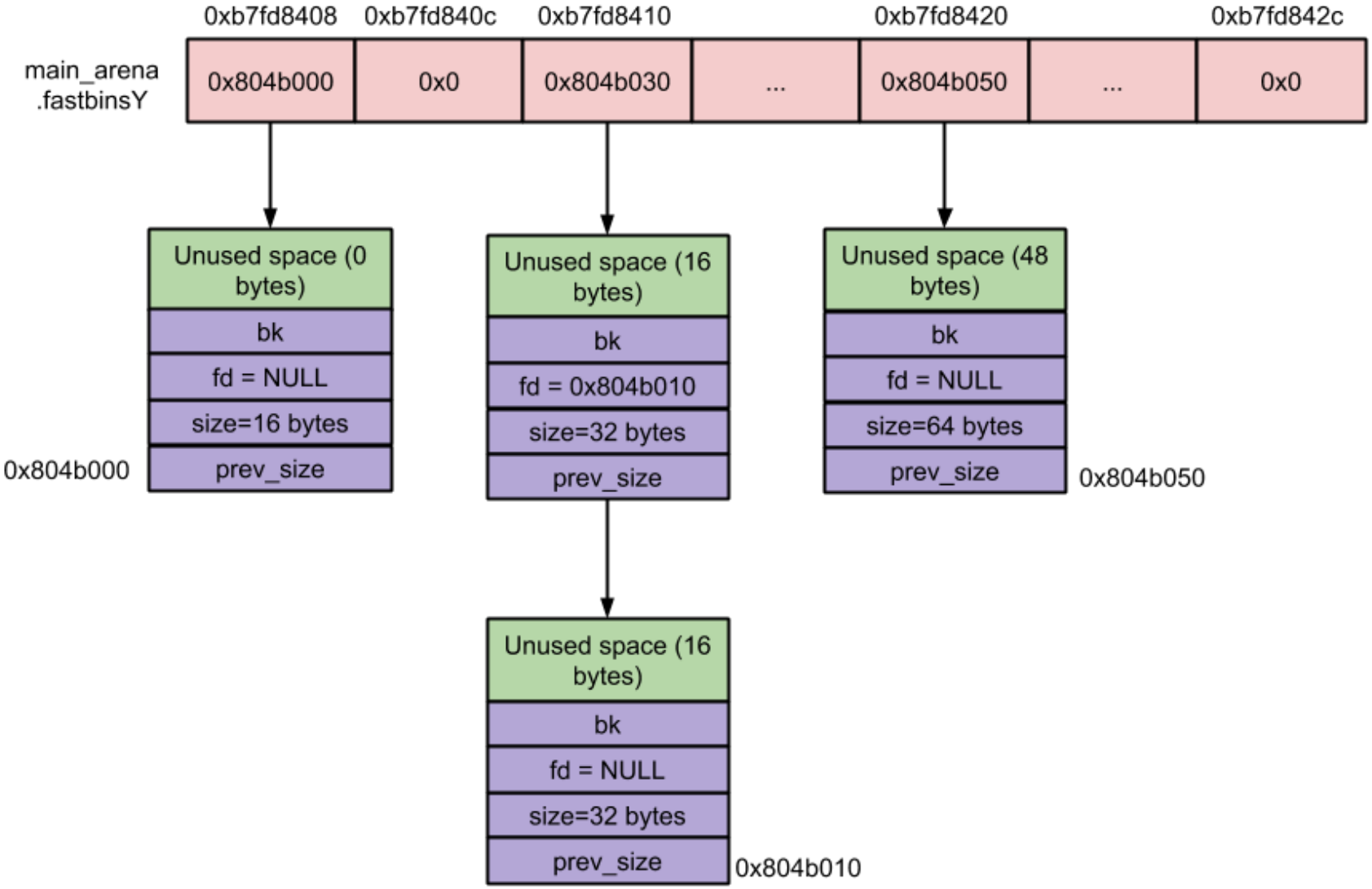
- `fastbinsY`：装有 fast bins 的数组
- `bins`：装有 unsorted, small, large bins 总共有126个，按照以下的规则被划分：
    - Bin 1 —— Unsorted bin
    - Bin 2 到 Bin 63 —— Small bin
    - Bin 64 到 Bin 126 —— Large bin

Fast Bin

大小在 16 到 80 bytes 的块叫做 `fast chunk` 。支持 `fast chunk` 的 bin 叫做 fast bins。在上述的这些 bin 里，fast bins 在内存分配和重新分配上更快。（译者注：只有 fast bin 是 LIFO 其他的 `bin` 都是 FIFO）

- Bins 的数量——10
    - 每个 fast bin 包含由空闲块组成的单向链表。由于在 fast bins 里，在列表中间块不会被移除，所以使用单向链表。添加和删除都在列表末端进行 —— LIFO。
- 块的大小 —— 以8 bytes累加
    - fast bin中的相邻的两个binlist中的块大小相差 8 bytes。举个例子，第一个 binlist 中块的大小为 16 bytes，第二 binlist 中块的大小为 24 bytes，依次类推。
    - 在同一个 binlist 中块的大小相同
- 在 malloc 初始化 过程中，最大的 fast bin 的大小设置为 64 (!80) bytes。因此通过块的默认大小设置为 16 到 64 就可以将其划分为 fast chunks 了。
- 不能合并 —— 空闲的两个块可以相邻，但不能合并为一个空闲块。不能合并导致产生外存储碎片，但它可以大大提速！！
- `malloc(fast chunk)`
    - 初始状态 fast bin 的最大容积以及 fast bin 目录是空的，因此即使用户请求一个 fast chunk，服务它的是 small bin code 而不是 fast bin code。
    - 之后当它不再为空时， 计算 fast bin 目录以检索其对应的 binlist。
    - binlist 中第一个被找到的块会被移除并返回给用户。
- `free(fast chunk)`

- 计算 Fast bin 目录以检索其对应的 binlist。
- 该空闲块被添加到以上检索的 binlist 的最前端。

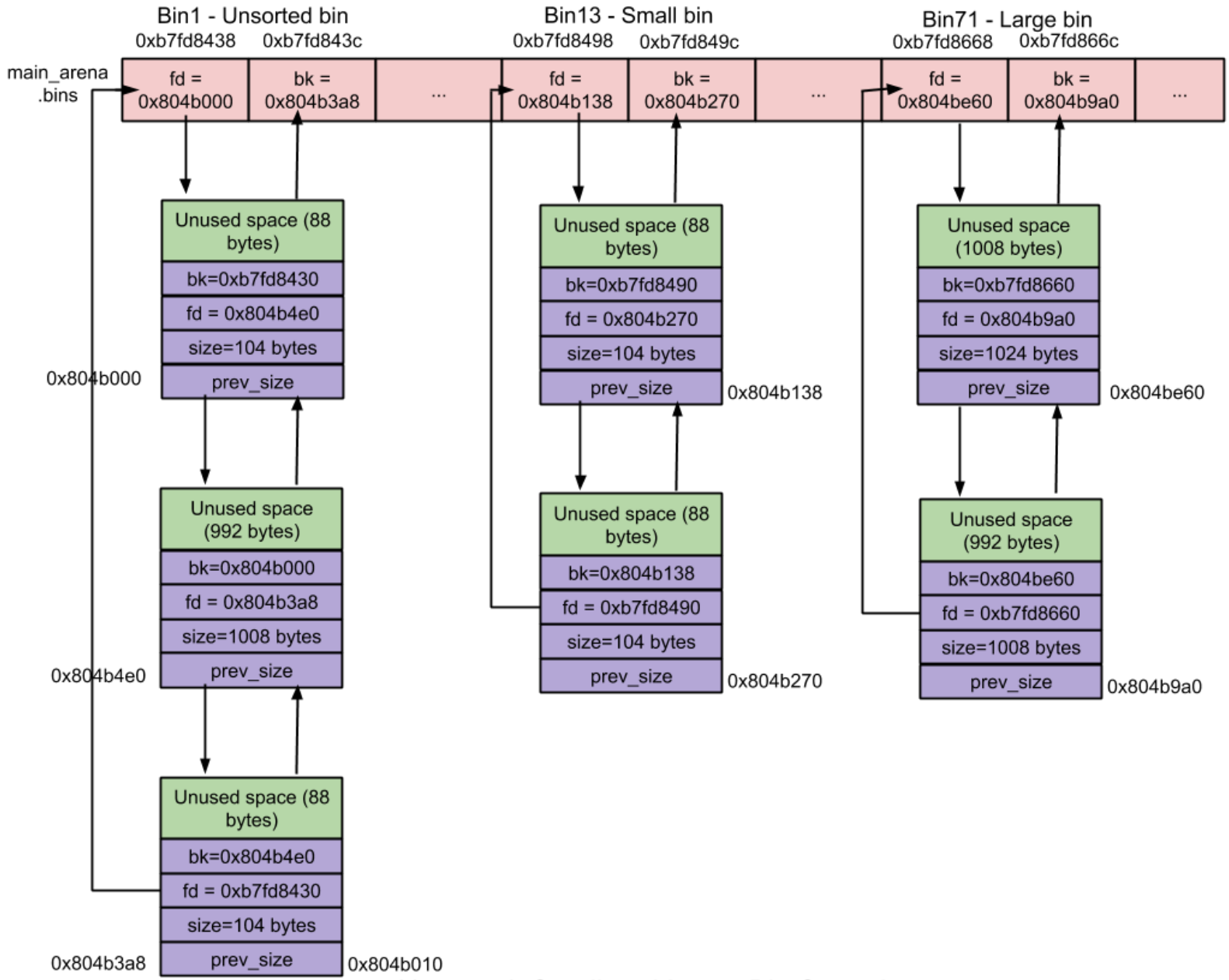


Fast Bin Snapshot

Unsorted Bin

当小块或大块被释放时，它会被添加到 unsorted bin 里。 这给了 `glibc malloc` 再次利用最近释放的块的机会。因此内存分配以及回收的速度都会有所提升（因为 unsorted bin）由于排除了用于查找合适容器的时间。

- Bins 的数量——1
    - Unsorted bin 包含空闲块的一个循环双向链表（也称为 binlist 容器列表）
- 块的大小 – 没有大小限制，任何大小的块都可以放入该 bin。



Unsorted, Small and Large Bin Snapshot

Small bin

小于 512 bytes 的块叫做 small chunk。 支持 small chunks 的容器叫做 small bins。 Small bins 在内存分配和回收时比 large bins 快（比 fast bins 慢）。

- Bins 的数量——62
    - 每个 small bin 含有空闲块的一个双向循环链表。在这里使用双向链表的原因在于，在 small bins 里，需要从链表的中间取出块。在这里列表头部实现添加并在列表的后部删除 —— FIFO。
- 块的大小 —— 以8 bytes累加
    - Small bin 中的相邻的两个binlist 中的块大小相差 8 bytes。举个例子，第一个 binlist 中块的大小为 16 bytes，第二 binlist 中块的大小为 24 bytes，依次类推。
    - 在同一个 binlist 中块的大小相同
- 合并 —— 两个相邻的空闲块，会被合并为一个空闲块。合并消除了外存储碎片，但是影响运行速度。
- `malloc(small chunk)`
    - 初始状态下所有 small bins 都是 NULL，因此即使用户请求一个 small chunk，提供的是 unsorted bin code 而不是 small bin code。
    - 在第一次调用 `malloc` 期间，在 malloc_state 里发现的 small bin 和 large bin 数据结构（bins）被初始化（即，bins 会指向它自己表示他们是空的）。
    - 之后当 small bin 处于非空状态时，其对应 binlist 中的最后一个块会被移除并返回给用户。
- `free(small chunk)`

- 释放块时，查看其前一个或下一个块是否空闲，如果空闲就合并（即，从他们各自的链表里解除块的链接，然后在 unsorted bin 的链表最开端添加新的合并后的块）。

Large bin

大小大于等于 512 bytes 的块称为 large chunk。支持 large chunks 的 bins 叫作 large bins。Large bins 在内存分配和回收中比 small bins 要慢。

- Bins 的数量——63
  - 每个 large bin 是由空闲块组成的双向循环链表。在 large bins 里，使用双向循环链表的原因是，我们需要在任意位置增加或删除块。
  - 在这 63 个 bins之外的情况：
  - 32 个 bins 中，相邻的两个binlist 中的块大小相差 64 bytes 。
  - 16 个 bins 中，相邻的两个binlist 中的块大小相差 512 bytes 。
  - 8 个 bins 中，相邻的两个binlist 中的块大小相差 4096 bytes 。
  - 4 个 bins 中，相邻的两个binlist 中的块大小相差 32768 bytes 。
  - 2 个 bins 中，相邻的两个binlist 中的块大小相差 262144 bytes。
  - 1 个 bin，剩余的所有容量都在一个块中。
  - 不同于 small bin，在 large bin 里的块大小是不同的。因此他们以降序排列。最大的块在最前端而最小的块被排到 binlist 的末尾。
- 合并 —— 两个相邻的空闲块，会合并为一个空闲块。
- `malloc(large chunk)`

- 初始化状态下所有 large bins 都是 NULL，因此即使用户请求一个 large chunk，提供的是下一个最大的 bin code，而不是 large bin code。
- 同样在第一次调用 `malloc` 期间，在 malloc_state 里发现的 small bin 和 large bin 被初始化。即，bins 会指向它自己表示它们是空的。
- 此后当 large bin 不为空时，如果最大块的大小（在它的容器列表里）比用户请求的容量还大， binlist 会从尾部到头部，来查找一个大小接近或等于用户请求大小的合适的块。一旦找到，这个块将分裂为两个块。
- 用户块（容量为用户请求的大小）—— 返回给用户。
- 剩余块（容量为剩余容量的大小）—— 添加到 unsorted bin。
- 如果最大块的大小（在它的容器列表里）小于用户请求的大小，那么尽量使用下一个最大（非空）bin 为用户的请求提供服务。下一个最大的 bin code 会扫描容器映射来查找下一个最大的非空 bin，如果找到任何一个，从 binlist 里检索到了一个合适的块，分裂并返回给用户。如果没找到，尝试使用 Top Chunk 为用户请求提供服务。

- `free(large chunk)` —— 过程与 `free(small chunk)` 类似

Top Chunk

处于 `arena` 顶部的块叫做 top chunk。它不属于任何 bin。而是在系统当前的所有 free chunk 都无法满足用户请求)的内存大小的时候，将此 chunk 分配给用户使用。如果 top chunk 比用户请求的容量要大，top chunk 将会分为两个块：

- 用户请求的 chunk
- 剩余的部分为新的 top chunk

如果内存空间不足，top chunk 使用 sbrk (`main arena`) 或 `mmap` (`thread arena`)系统调用来扩展内存空间。

Last Remainder Chunk

last remainder chunk 是由最近一次请求，对块进行分裂而产生的。last remainder chunk 加强了引用的局部性（即，连续的 `malloc small chunk` 获取的空间可能彼此相邻）。

但是除了在一个 `arena` 里可利用的的块，哪些块有资格成为 last remainder chunk?

当用户请求一个small chunk，且该请求无法被 small bin，unsorted bin 满足的时候，就通过 bin maps 遍历合适的 chunk，如果该 chunk 有剩余部分的话，就将剩余部分变成一个新的 chunk 加入到 unsorted bin 中，另外，再将该新的 chunk 变成新的last remainder chunk。

那么如何使引用存在局部性?

当用户请求一个small chunk，且该请求无法被 small bin满足，那么就转而交由 unsorted bin处理。同时，假设当前 unsorted bin 中只有一个 chunk 的话——就是last remainder chunk，那么就将 chunk 分成两部分：前者分配给用户，剩下的部分放到 unsorted bin 中，并成为新的 last remainder chunk。这个就保证了连续malloc(small chunk)中，各个small chunk在内存分布中是相邻的，即提高内存分配的局部性。

See Also

- 调试器工作原理：第三部分 调试信息
- 调试器工作原理：第二部分 断点
- 调试器工作原理：第一部分 基础
- 操作系统 & 编译原理 学习攻略
- Python代码性能优化方法总结

内存 翻译

© 2018 寒枫的博客 By Hanfeng Zhang. Powered by Hugo. Theme based on maupassant.