淘宝网

Glibc 内存管理

Ptmalloc2 源代码分析

华庭（庄明强）

2011/4/17

由于华庭水平有限，因此也不能完全保证对 ptmalloc 的所有理解都准备无误，所以如果读者发现其中有什么错误，请勿见怪，如果方便请来信告之，并欢迎来信讨论。
（mqzhuang@gmail.com）

目录

1.问题

项目组正在研发的一个类似数据库的 NoSql 系统，遇到了 Glibc 的内存暴增问题。现象如下：在我们的 NoSql 系统中实现了一个简单的内存管理模块，在高压力高并发环境下长时间运行，当内存管理模块的内存释放给 C 运行时库以后，C 运行时库并没有立即把内存归还给操作系统，比如内存管理模块占用的内存为 10GB，释放内存以后，通过 TOP 命令或者/proc/pid/status 查看进程占用的内存有时仍然为 10G，有时为 5G，有时为 3G，etc，内存释放的行为不确定。

我们的 NoSql 系统中的内存管理方式比较简单，使用全局的定长内存池，内存管理模块每次分配/释放 2MB 内存，然后分成 64KB 为单位的一个个小内存块用 hash 加链表的方式进行管理。如果申请的内存小于等于 64KB 时，直接从内存池的空闲链表中获取一个内存块，内存释放时归还空闲链表；如果申请的内存大于 64KB，直接通过 C 运行时库的 malloc 和 free 获取。某些数据结构涉及到很多小对象的管理，比如 Hash 表，B-Tree，这些数据结构从全局内存池获取内存后再根据数据结构的特点进行组织。为了提高内存申请/释放的效率，减少锁冲突，为每一个线程单独保留 8MB 的内存块，每个线程优先从线程专属的 8MB 内存块获取内存，专属内存不足时才从全局的内存池获取。

系统中使用的网络库有独立的内存管理方式，并不从全局内存池中分配内存，该网络库在处理网络请求时也是按 2M 内存为单位向 C 运行时库申请内存，一次请求完成以后，释放分配的内存到 C 运行时库。

在弄清楚了系统的内存分配位置以后，对整个系统进行了内存泄露的排查，在解决了数个内存泄露的潜在问题以后，发现系统在高压力高并发环境下长时间运行仍然会发生内存暴增的现象，最终进程因 OOM 被操作系统杀掉。

为了便于跟踪分析问题，在全局的内存池中加入对每个子模块的内存统计功能：每个子模块申请内存时都将子模块编号传给全局的内存池，全局的内存池进行统计。复现问题后发现全局的内存池的统计结果符合预期，同样对网络模块也做了类似的内存使用统计，仍然符合预期。由于内存管理不外乎三个层面，用户管理层，C 运行时库层，操作系统层，在操作系统层发现进程的内存暴增，同时又确认了用户管理层没有内存泄露，因此怀疑是 C 运行时库的问题，也就是 Glibc 的内存管理方式导致了进程的内存暴增。

问题范围缩小了，但有如下的问题还没有搞清楚，搞不清楚这些问题，我们系统的中的问题就无法根本性解决。

1. Glibc 在什么情况下不会将内存归还给操作系统？
2. Glibc 的内存管理方式有哪些约束?适合什么样的内存分配场景？
3. 我们的系统中的内存管理方式是与 Glibc 的内存管理的约束相悖的？
4. Glibc 是如何管理内存的？

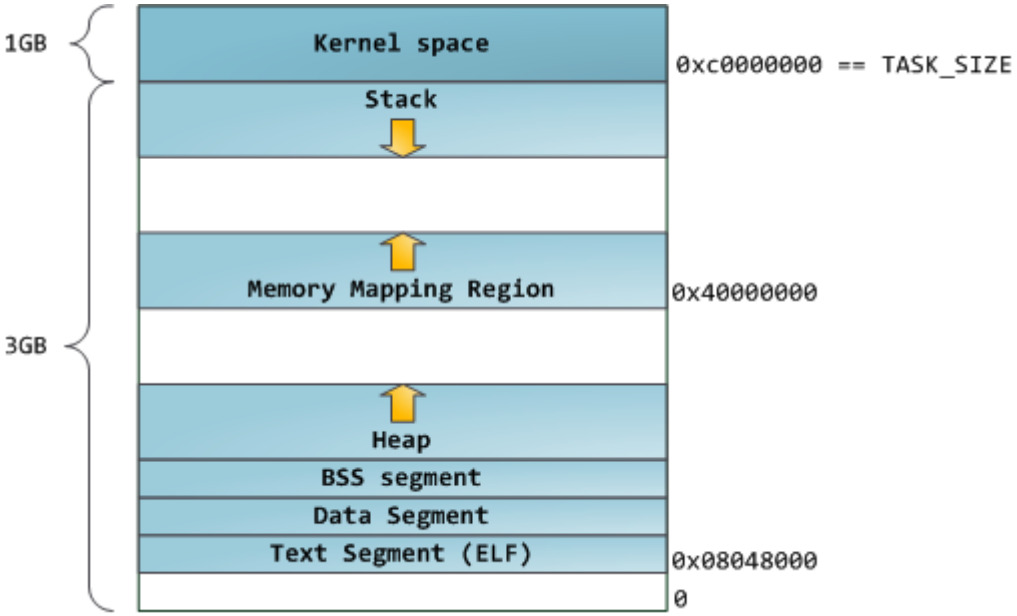
带着这些问题，决定对 Glibc 的 ptmalloc2 源代码进行一番研究，希望能找到这些问题的答案，并解决我们系统中遇到的问题。我研究的对象是当前最新版的 glibc-2.12.1 中的内存管理的相关代码。

2.基础知识

2.1 X86 平台 Linux 进程内存布局

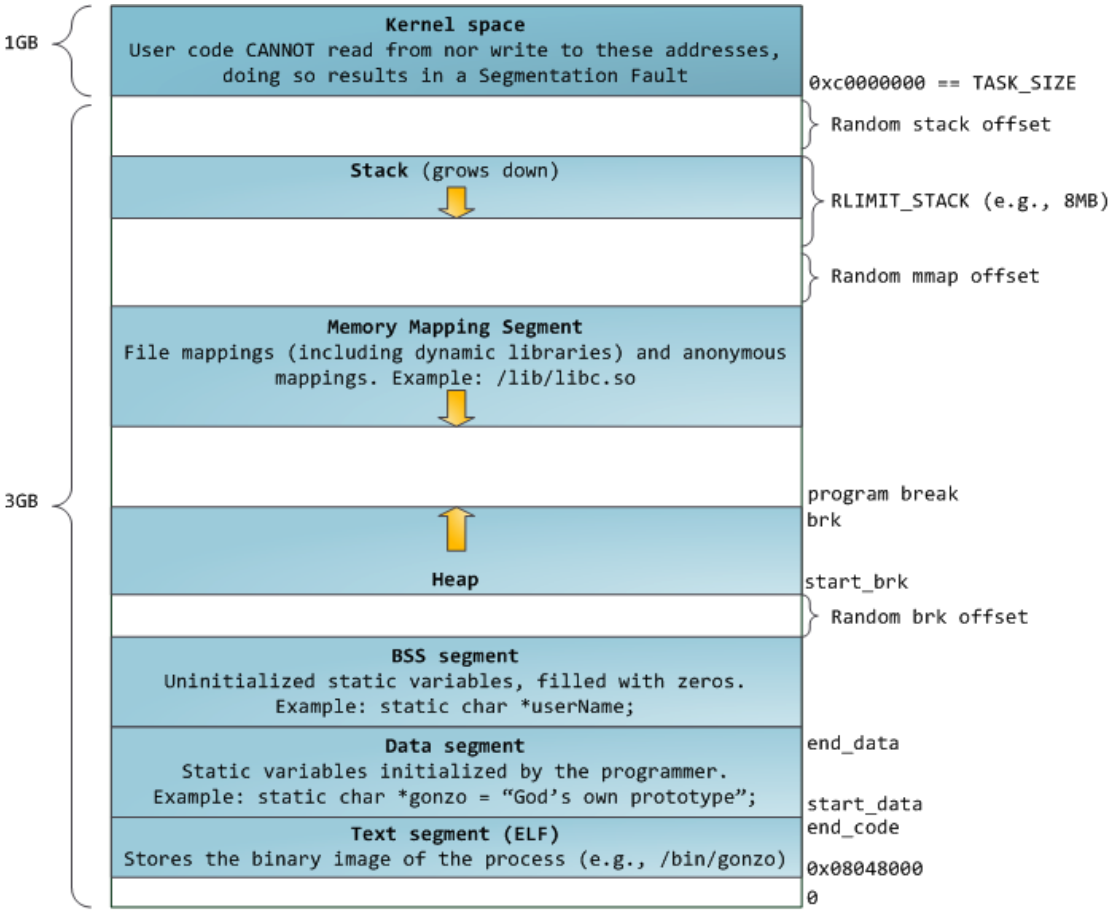
Linux 系统在装载 elf 格式的程序文件时，会调用 loader 把可执行文件中的各个段依次载入到从某一地址开始的空间中（载入地址取决 link editor(ld)和机器地址位数，在 32 位机器上是 0x8048000，即 128M 处）。如下图所示，以 32 位机器为例，首先被载入的是.text 段，然后是.data 段，最后是.bss 段。这可以看作是程序的开始空间。程序所能访问的最后的地址是 0xbfffffff，也就是到 3G 地址处，3G 以上的 1G 空间是内核使用的，应用程序不可以直接访问。应用程序的堆栈从最高地址处开始向下生长，.bss 段与堆栈之间的空间是空闲的，空闲空间被分成两部分，一部分为 heap，一部分为 mmap 映射区域，mmap 映射区域一般从 TASK_SIZE/3 的地方开始，但在不同的 Linux 内核和机器上，mmap 区域的开始位置一般是不同的。Heap 和 mmap 区域都可以供用户自由使用，但是它在刚开始的时候并没有映射到内存空间内，是不可访问的。在向内核请求分配该空间之前，对这个空间的访问会导致 segmentation fault。用户程序可以直接使用系统调用来管理 heap 和 mmap 映射区域，但更多的时候程序都是使用 C 语言提供的 malloc()和 free()函数来动态的分配和释放内存。Stack 区域是唯一不需要映射，用户却可以访问的内存区域，这也是利用堆栈溢出进行攻击的基础。

2.1.1 32 位模式下进程内存经典布局



这种布局是 Linux 内核 2.6.7 以前的默认进程内存布局形式，mmap 区域与栈区域相对增长，这意味着堆只有 1GB 的虚拟地址空间可以使用，继续增长就会进入 mmap 映射区域，这显然不是我们想要的。这是由于 32 模式地址空间限制造成的，所以内核引入了另一种虚拟地址空间的布局形式，将在后面介绍。但对于 64 位系统，提供了巨大的虚拟地址空间，这种布局就相当好。

2.1.2 32 位模式下进程默认内存布局



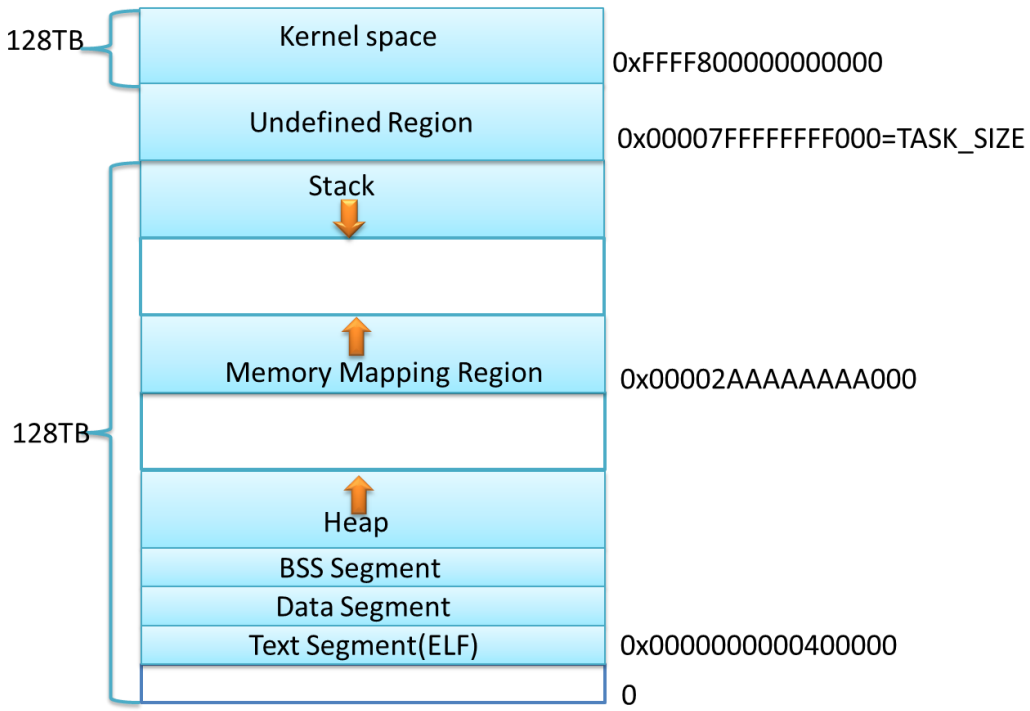
从上图可以看到，栈至顶向下扩展，并且栈是有界的。堆至底向上扩展，mmap 映射区域至顶向下扩展，mmap 映射区域和堆相对扩展，直至耗尽虚拟地址空间中的剩余区域，这种结构便于 C 运行时库使用 mmap 映射区域和堆进行内存分配。上图的布局形式是在内核 2.6.7 以后才引入的，这是 32 位模式下进程的默认内存布局形式。

2.1.3 64 位模式下进程内存布局

在 64 位模式下各个区域的起始位置是什么呢？对于 AMD64 系统，内存布局采用经典内存布局，text 的起始地址为 0x0000000000400000，堆紧接着 BSS 段向上增长，mmap 映射区域开始位置一般设为 TASK_SIZE/3。

```
#define TASK_SIZE_MAX          ((1UL << 47) - PAGE_SIZE)
#define TASK_SIZE              (test_thread_flag(TIF_IA32) ? \
                                    IA32_PAGE_OFFSET : TASK_SIZE_MAX)
#define STACK_TOP              TASK_SIZE
#define TASK_UNMAPPED_BASE     (PAGE_ALIGN(TASK_SIZE / 3))
```

计算一下可知，mmap 的开始区域地址为 0x00002AAAAAAAA000，栈顶地址为 0x00007FFFFFFF000



上图是 X86_64 下 Linux 进程的默认内存布局形式，这只是一个示意图，当前内核默认配置下，进程的栈和 mmap 映射区域并不是从一个固定地址开始，并且每次启动时的值都不一样，这是程序在启动时随机改变这些值的设置，使得使用缓冲区溢出进行攻击更加困难。当然也可以让进程的栈和 mmap 映射区域从一个固定位置开始，只需要设置全局变量 randomize_va_space 值为 0，这个变量默认值为 1。用户可以通过设置 /proc/sys/kernel/randomize_va_space 来停用该特性，也可以用如下命令：

sudo sysctl -w kernel.randomize_va_space=0

2.2 操作系统内存分配的相关函数

上节提到 heap 和 mmap 映射区域是可以提供给用户程序使用的虚拟内存空间，如何获得该区域的内存呢？操作系统提供了相关的系统调用来完成相关工作。对 heap 的操作，操作系统提供了 brk()函数，C 运行时库提供了 sbrk()函数；对 mmap 映射区域的操作，操作系统提供了 mmap()和 munmap()函数。sbrk()，brk() 或者 mmap() 都可以用来向我们的进程添加额外的虚拟内存。Glibc 同样是使用这些函数向操作系统申请虚拟内存。

这里要提到一个很重要的概念，内存的延迟分配，只有在真正访问一个地址的时候才建立这个地址的物理映射，这是 Linux 内存管理的基本思想之一。Linux 内核在用户申请内存的时候，只是给它分配了一个线性区（也就是虚拟内存），并没有分配实际物理内存；只有当用户使用这块内存的时候，内核才会分配具体的物理页面给用户，这时候才占用宝贵的物理内存。内核释放物理页面是通过释放线性区，找到其所对应的物理页面，将其全部释放的过程。

2.2.1 Heap 操作相关函数

Heap 操作函数主要有两个，brk()为系统调用，sbrk()为 C 库函数。系统调用通常提供一

种最小功能，而库函数通常提供比较复杂的功能。Glibc 的 malloc 函数族（realloc，calloc 等）就调用 sbrk()函数将数据段的下界移动，sbrk()函数在内核的管理下将虚拟地址空间映射到内存，供 malloc()函数使用。

内核数据结构 mm_struct 中的成员变量 start_code 和 end_code 是进程代码段的起始和终止地址，start_data 和 end_data 是进程数据段的起始和终止地址，start_stack 是进程堆栈段起始地址，start_brk 是进程动态内存分配起始地址（堆的起始地址），还有一个 brk（堆的当前最后地址），就是动态内存分配当前的终止地址。C 语言的动态内存分配基本函数是 malloc()，在 Linux 上的实现是通过内核的 brk 系统调用。brk()是一个非常简单的系统调用，只是简单地改变 mm_struct 结构的成员变量 brk 的值。

这两个函数的定义如下：

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

需要说明的是，但 sbrk()的参数 increment 为 0 时，sbrk()返回的是进程的当前 brk 值，increment 为正数时扩展 brk 值，当 increment 为负值时收缩 brk 值。

2.2.2 Mmap 映射区域操作相关函数

mmap()函数将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零。munmap 执行相反的操作，删除特定地址区域的对象映射。函数的定义如下：

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

在这里不准备对这两个函数做详细介绍，只是对 ptmalloc 中用到的功能做一下介绍，其他的用法请参看相关资料。

参数：

start：映射区的开始地址。

length：映射区的长度。

prot：期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过 or 运算合理地组合在一起。Ptmalloc 中主要使用了如下的几个标志：

PROT_EXEC //页内容可以被执行，ptmalloc 中没有使用

PROT_READ //页内容可以被读取，ptmalloc 直接用 mmap 分配内存并立即返回给用户时设置该标志

PROT_WRITE //页可以被写入，ptmalloc 直接用 mmap 分配内存并立即返回给用户时设置该标志

PROT_NONE //页不可访问，ptmalloc 用 mmap 向系统“批发”一块内存进行管理时设置该标志

flags：指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体

MAP_FIXED //使用指定的映射起始地址，如果由 start 和 len 参数指定的内存区重叠于现存的映射空间，重叠部分将会被丢弃。如果指定的起始地址不可用，操作将会失败。并且起

始地址必须落在页的边界上。Ptmalloc 在回收从系统中“批发”的内存时设置该标志。

MAP_PRIVATE //建立一个写入时拷贝的私有映射。内存区域的写入不会影响到原文件。这个标志和以上标志是互斥的，只能使用其中一个。Ptmalloc 每次调用 mmap 都设置该标志。

MAP_NORESERVE //不要为这个映射保留交换空间。当交换空间被保留，对映射区修改的可能会得到保证。当交换空间不被保留，同时内存不足，对映射区的修改会引起段违例信号。Ptmalloc 向系统“批发”内存块时设置该标志。

MAP_ANONYMOUS //匿名映射，映射区不与任何文件关联。Ptmalloc 每次调用 mmap 都设置该标志。

fd：有效的文件描述词。如果 MAP_ANONYMOUS 被设定，为了兼容问题，其值应为-1。

offset：被映射对象内容的起点。

3.概述

3.1 内存管理一般性描述

当不知道程序的每个部分将需要多少内存时，系统内存空间有限，而内存需求又是变化的，这时就需要内存管理程序来负责分配和回收内存。程序的动态性越强，内存管理就越重要，内存分配程序的选择也就更重要。

3.1.1 内存管理的方法

可用于内存管理的方法有许多种，它们各有好处与不足，不同的内存管理方法有最适用的情形。

1. C 风格的内存管理程序

C 风格的内存管理程序主要实现 malloc()和 free()函数。内存管理程序主要通过调用 brk()或者 mmap()进程添加额外的虚拟内存。Doug Lea Malloc，ptmalloc，BSD malloc，Hoard，TCMalloc 都属于这一类内存管理程序。

基于 malloc()的内存管理器仍然有很多缺点，不管使用的是哪个分配程序。对于那些需要保持长期存储的程序使用 malloc()来管理内存可能会非常令人失望。如果有大量的不固定的内存引用，经常难以知道它们何时被释放。生存期局限于当前函数的内存非常容易管理，但是对于生存期超出该范围的内存来说，管理内存则困难得多。因为管理内存的问题，很多程序倾向于使用它们自己的内存管理规则。

2. 池式内存管理

内存池是一种半内存管理方法。内存池帮助某些程序进行自动内存管理，这些程序会经历一些特定的阶段，而且每个阶段中都有分配给进程的特定阶段的内存。例如，很多网络服务器进程都会分配很多针对每个连接的内存——内存的最大生存期限为当前连接的存在期。Apache 使用了池式内存（pooled memory），将其连接拆分为各个阶段，每个阶段都有自己的内存池。在结束每个阶段时，会一次释放所有内存。

在池式内存管理中，每次内存分配都会指定内存池，从中分配内存。每个内存池都有不

同的生存期限。在 Apache 中，有一个持续时间为服务器存在期的内存池，还有一个持续时间为连接的存在期的内存池，以及一个持续时间为请求的存在期的池，另外还有其他一些内存池。因此，如果我的一系列函数不会生成比连接持续时间更长的数据，那么我就可以完全从连接池中分配内存，并知道在连接结束时，这些内存会被自动释放。另外，有一些实现允许注册清除函数（cleanup functions），在清除内存池之前，恰好可以调用它，来完成在内存被清理前需要完成的其他所有任务（类似于面向对象中的析构函数）。

使用池式内存分配的优点如下所示：

- 应用程序可以简单地管理内存。
- 内存分配和回收更快，因为每次都是在一个池中完成的。分配可以在 O(1)时间内完成，释放内存池所需时间也差不多（实际上是 O(n)时间，不过在大部分情况下会除以一个大的因数，使其变成 O(1)）。
- 可以预先分配错误处理池（Error-handling pools），以便程序在常规内存被耗尽时仍可以恢复。
- 有非常易于使用的标准实现。

池式内存的缺点是：

- 内存池只适用于操作可以分阶段的程序。
- 内存池通常不能与第三方库很好地合作。
- 如果程序的结构发生变化，则不得不修改内存池，这可能会导致内存管理系统的重新设计。
- 您必须记住需要从哪个池进行分配。另外，如果在这里出错，就很难捕获该内存池。

3. 引用计数

在引用计数中，所有共享的数据结构都有一个域来包含当前活动“引用”结构的次数。当向一个程序传递一个指向某个数据结构指针时，该程序会将引用计数增加 1。实质上，是在告诉数据结构，它正在被存储在多少个位置上。然后，当进程完成对它的使用后，该程序就会将引用计数减少 1。结束这个动作之后，它还会检查计数是否已经减到零。如果是，那么它将释放内存。

在 Java，Perl 等高级语言中，进行内存管理时使用引用计数非常广泛。在这些语言中，引用计数由语言自动地处理，所以您根本不必担心它，除非要编写扩展模块。由于所有内容都必须进行引用计数，所以这会对速度产生一些影响，但它极大地提高了编程的安全性和方便性。

以下是引用计数的好处：

- 实现简单。
- 易于使用。
- 由于引用是数据结构的一部分，所以它有一个好的缓存位置。

不过，它也有其不足之处：

- 要求您永远不要忘记调用引用计数函数。
- 无法释放作为循环数据结构的一部分的结构。
- 减缓几乎每一个指针的分配。
- 尽管所使用的对象采用了引用计数，但是当使用异常处理（比如 try 或 setjmp()/longjmp()）时，您必须采取其他方法。
- 需要额外的内存来处理引用。
- 引用计数占用了结构中的第一个位置，在大部分机器中最快可以访问到的就是这个位置。

- 在多线程环境中更慢也更难以使用。

4. 垃圾收集

垃圾收集（Garbage collection）是全自动地检测并移除不再使用的数据对象。垃圾收集器通常会在当可用内存减少到少于一个具体的阈值时运行。通常，它们以程序所知的可用的一组“基本”数据——栈数据、全局变量、寄存器——作为出发点。然后它们尝试去追踪通过这些数据连接到每一块数据。收集器找到的都是有用的数据；它没有找到的就是垃圾，可以被销毁并重新使用这些无用的数据。为了有效地管理内存，很多类型的垃圾收集器都需要知道数据结构内部指针的规划，所以，为了正确运行垃圾收集器，它们必须是语言本身的一部分。

垃圾收集的一些优点：

- 永远不必担心内存的双重释放或者对象的生命周期。
- 使用某些收集器，您可以使用与常规分配相同的 API。

其缺点包括：

- 使用大部分收集器时，您都无法干涉何时释放内存。
- 在多数情况下，垃圾收集比其他形式的内存管理更慢。
- 垃圾收集错误引发的缺陷难于调试。
- 如果您忘记将不再使用的指针设置为 null，那么仍然会有内存泄漏。

3.1.2 内存管理器的设计目标

内存管理器为什么难写？在设计内存管理算法时，要考虑什么因素？管理内存这是内存管理器的功能需求。正如设计其它软件一样，质量需求一样占有重要的地位。分析内存管理算法之前，我们先看看对内存管理算法的质量需求有哪些：

1. 最大化兼容性

要实现内存管理器时，先要定义出分配器的接口函数。接口函数没有必要标新立异，而是要遵循现有标准（如 POSIX 或者 Win32），让使用者可以平滑的过度到新的内存管理器上。

2. 最大化可移植性

通常情况下，内存管理器要向 OS 申请内存，然后进行二次分配。所以，在适当的时候要扩展内存或释放多余的内存，这要调用 OS 提供的函数才行。OS 提供的函数则是因平台而异，尽量抽象出平台相关的代码，保证内存管理器的可移植性。

3. 浪费最小的空间

内存管理器要管理内存，必然要使用自己一些数据结构，这些数据结构本身也要占内存空间。在用户眼中，这些内存空间毫无疑问是浪费掉了，如果浪费在内存管理器身的内存太多，显然是不可以接受的。

内存碎片也是浪费空间的罪魁祸首，若内存管理器中有大量的内存碎片，它们是一些不连续的小块内存，它们总量可能很大，但无法使用，这也是不可以接受的。

4. 最快的速度

内存分配/释放是常用的操作。按着 2/8 原则，常用的操作就是性能热点，热点函数的性能对系统的整体性能尤为重要。

5. 最大化可调性（以适应于不同的情况）

内存管理算法设计的难点就在于要适应用不同的情况。事实上，如果缺乏应用的上下文，是无法评估内存管理算法的好坏的。可以说在任何情况下，专用算法都比通用算法在时/空性能上的表现更优。

为每种情况都写一套内存管理算法，显然是不太合适的。我们不需要追求最优算法，那样代价太高，能达到次优就行了。设计一套通用内存管理算法，通过一些参数对它进行配置，可以让它在特定情况也有相当出色的表现，这就是可调性。

6. 最大化局部性（Locality）

大家都知道，使用 cache 可以提高程度的速度，但很多人未必知道 cache 使程序速度提高的真正原因。拿 CPU 内部的 cache 和 RAM 的访问速度相比，速度可能相差一个数量级。两者的速度上的差异固然重要，但这并不是提高速度的充分条件，只是必要条件。

另外一个条件是程序访问内存的局部性（Locality）。大多数情况下，程序总访问一块内存附近的内存，把附近的内存先加入到 cache 中，下次访问 cache 中的数据，速度就会提高。否则，如果程序一会儿访问这里，一会儿访问另外一块相隔十万八千里的内存，这只会使数据在内存与 cache 之间来回搬运，不但于提高速度无益，反而会大大降低程序的速度。

因此，内存管理算法要考虑这一因素，减少 cache miss 和 page fault。

7. 最大化调试功能

作为一个 C/C++程序员，内存错误可以说是我们的噩梦，上一次的内存错误一定还让你记忆犹新。内存管理器提供的调试功能，强大易用，特别对于嵌入式环境来说，内存错误检测工具缺乏，内存管理器提供的调试功能就更是不可或缺了。

8. 最大化适应性

前面说了最大化可调性，以便让内存管理器适用于不同的情况。但是，对于不同情况都要去调设置，无疑太麻烦，是非用户友好的。要尽量让内存管理器适用于很广的情况，只有极少情况下才去调设置。

设计是一个多目标优化的过程，有些目标之间存在着竞争。如何平衡这些竞争力是设计的难点之一。在不同的情况下，这些目标的重要性又不一样，所以根本不存在一个最好的内存分配算法。

一切都需要折衷：性能、易用、易于实现、支持线程的能力等，为了满足项目的要求，有很多内存管理模式可供使用。每种模式都有大量的实现，各有其优缺点。内存管理的设计目标中，有些目标是相互冲突的，比如最快的分配、释放速度与内存的利用率，也就是内存碎片问题。不同的内存管理算法在两者之间取不同的平衡点。

为了提高分配、释放的速度，多核计算机上，主要做的工作是避免所有核同时在竞争内存，常用的做法是内存池，简单来说就是批量申请内存，然后切割成各种长度，各种长度都有一个链表，申请、释放都只要在链表上操作，可以认为是 O(1)的。不可能所有的长度都对应一个链表。很多内存池是假设，A 释放掉一块内存后，B 会申请类似大小的内存，但是 A 释放的内存跟 B 需要的内存不一定完全相等，可能有一个小的误差，如果严格按大小分配，会导致复用率很低，这样各个链表上都会有很多释放了，但是没有复用的内存，导致利用率很低。这个问题也是可以解决的，可以回收这些空闲的内存，这就是传统的内存管理，不停地对内存块作切割和合并，会导致效率低下。所以通常的做法是只分配有限种类的长度。一般的内存池只提供几十种选择。

3.1.3 常见 C 内存管理程序

本文主要关注的是 C 内存管理程序，比较著名的几个 C 内存管理程序，其中包括：

- **Doug Lea Malloc：**Doug Lea Malloc 实际上是完整的一组分配程序，其中包括 Doug Lea 的原始分配程序，GNU libc 分配程序和 ptmalloc。Doug Lea 的分配程序加入了索引，这使得搜索速度更快，并且可以将多个没有被使用的块组合为一个大的块。它还支持缓存，以便更快地再次使用最近释放的内存。ptmalloc 是 Doug Lea Malloc 的一个扩展版本，支持多线程。在本文后面的部分详细分析 ptamlloc2 的源代码实现。
- **BSD Malloc：**BSD Malloc 是随 4.2 BSD 发行的实现，包含在 FreeBSD 之中，这个分配程序可以从预先确实大小的对象构成的池中分配对象。它有一些用于对象大小的 size 类，这些对象的大小为 2 的若干次幂减去某一常数。所以，如果您请求给定大小的一个对象，它就简单地分配一个与之匹配的 size 类。这样就提供了一个快速的实现，但是可能会浪费内存。
- **Hoard：**编写 Hoard 的目标是使内存分配在多线程环境中进行得非常快。因此，它的构造以锁的使用为中心，从而使所有进程不必等待分配内存。它可以显著地加快那些进行很多分配和回收的多线程进程的速度。
- **TCMalloc：**（Thread-Caching Malloc）是 google 开发的开源工具——“google-perftools”中的成员。与标准的 Glibc 库的 malloc 相比，TCMalloc 在内存的分配上效率和速度要高得多。TCMalloc 是一种通用内存管理程序，集成了内存池和垃圾回收的优点，对于小内存，按 8 的整数次倍分配，对于大内存，按 4K 的整数次倍分配。这样做有两个好处，一是分配的时候比较快，那种提供几十种选择的内存池，往往要遍历一遍各种长度，才能选出合适的种类，而 TCMalloc 则可以简单地做几个运算就行了。二是短期的收益比较大，分配的小内存至多浪费 7 个字节，大内存则 4K。但是长远来说，TCMalloc 分配的种类还是比别的内存池要多很多的，可能会导致复用率很低。TCMalloc 还有一套高效的机制回收这些空闲的内存。当一个线程的空闲内存比较多的时候，会交还给进程，进程可以把它调配给其他线程使用；如果某种长度交还给进程后，其他线程并没有需求，进程则把这些长度合并成内存页，然后切割成其他长度。如果进程占据的资源比较多呢，据说不会交回给操作系统。周期性的内存回收，避免可能出现的内存爆炸式增长的问题。TCMalloc 有比较高的空间利用率，只额外花费 1%的空间。尽量避免加锁（一次加锁解锁约浪费 100ns），使用更高效的 spinlock，采用更合理的粒度。小块内存和打开内存分配采取不同的策略：小于 32K 的被定义为小块内存，小块内存按大小被分为 8Bytes，16Bytes，。。。，236Bytes 进行分级。不是某个级别整数倍的大小都会被分配向上取整。如 13Bytes 的会按 16Bytes 分配，分配时，首先在本线程相应大小级别的空闲链表里面找，如果找到的话可以避免加锁操作（本线程的 cache 只有本线程自己使用）。如果找不到的话，则尝试从中心内存区的相应级别的空闲链表里搬一些对象到本线程的链表。如果中心内存区相应链表也为空的话，则向中心页分配器请求内存页面，然后分割成该级别的对象存储。大块内存处理方式：按页分配，每页大小是 4K，然后内存按 1 页，2 页，……，255 页的大小分类，相同大小的内存块也用链表连接。

各种 C 内存管理程序实现的对比

策略	分配速度	回收速度	局部缓存	易用性	通用性	SMP 线程友好度
GNU Malloc	中	快	中	容易	高	中

Hoard	中	中	中	容易	高	高
TCMalloc	快	快	中	容易	高	高

从上表可以看出，TCMalloc 的优势还是比较大的，TCMalloc 的优势体现在：

- 分配内存页的时候，直接跟 OS 打交道，而常用的内存池一般是基于别的内存管理器上分配，如果完全一样的内存管理策略，明显 TCMalloc 在性能及内存利用率上要省掉第三方内存管理的开销。之所以会出现这种情况，是因为大部分写内存池的 coder 都不太了解 OS
- 大部分的内存池只负责分配，不管回收。当然了，没有回收策略，也有别的方法解决问题。比如线程之间协调资源，模索模块一般是一写多读，也就是只有一个线程申请、释放内存，就不存在线程之间协调资源；为了避免某些块大量空闲，常用的做法是减少内存块的种类，提高复用率，这可能会造成内部碎片比较多，如果空闲的内存实在太多了，还可以直接重启。

作为一个通用的内存管理库，TCMalloc 也未必能超过专用的比较粗糙的内存池。比如应用中主要用到 7 种长度的块，专用的内存池，可以只分配这 7 种长度，使得没有内部碎片。或者利用统计信息设置内存池的长度，也可以使得内部碎片比较少。

所以 TCMalloc 的意义在于，不需要增加任何开发代价，就能使得内存的开销比较少，而且可以从理论上证明，最优的分配不会比 TCMalloc 的分配好很多。

对比 Glibc 可以发现，两者的思想其实是差不多的，差别只是在细节上，细节上的差别，对工程项目来说也是很重要的，至少在性能与内存使用率上 TCMalloc 是领先很多的。Glibc 在内存回收方面做得不太好，常见的一个问题，申请很多内存，然后又释放，只是有一小块没释放，这时候 Glibc 就必须要等待这一小块也释放了，也把整个大块释放，极端情况下，可能会造成几个 G 的浪费。

3.2 Ptmalloc 内存管理概述

3.2.1 简介

Linux 中 malloc 的早期版本是由 Doug Lea 实现的，它有一个重要问题就是在并行处理时多个线程共享进程的内存空间，各线程可能并发请求内存，在这种情况下应该如何保证分配和回收的正确和高效。Wolfram Gloger 在 Doug Lea 的基础上改进使得 Glibc 的 *malloc* 可以支持多线程——ptmalloc，在 glibc-2.3.x.中已经集成了 ptmalloc2，这就是我们平时使用的 malloc，目前 ptmalloc 的最新版本 ptmalloc3。ptmalloc2 的性能略微比 ptmalloc3 要高一点点。

ptmalloc 实现了 malloc()，free()以及一组其它的函数. 以提供动态内存管理的支持。分配器处在用户程序和内核之间，它响应用户的分配请求，向操作系统申请内存，然后将其返回给用户程序，为了保持高效的分配，分配器一般都会预先分配一块大于用户请求的内存，并通过某种算法管理这块内存。来满足用户的内存分配要求，用户释放掉的内存也并不是立即就返回给操作系统，相反，分配器会管理这些被释放掉的空闲空间，以应对用户以后的内存分配要求。也就是说，分配器不但要管理已分配的内存块，还需要管理空闲的内存块，当响应用户分配要求时，分配器会首先在空闲空间中寻找一块合适的内存给用户，在空闲空间中找不到的情况下才分配一块新的内存。为实现一个高效的分配器，需要考虑很多的因素。比如，分配器本身管理内存块所占用的内存空间必须很小，分配算法必须要足够的快。

3.2.2 内存管理的设计假设

Ptmalloc 在设计时折中了高效率，高空间利用率，高可用性等设计目标。在其实现代码中，隐藏着内存管理中的一些设计假设，由于某些设计假设，导致了在某些情况下 ptmalloc 的行为很诡异。这些设计假设包括：

1. 具有长生命周期的大内存分配使用 mmap。
2. 特别大的内存分配总是使用 mmap。
3. 具有短生命周期的内存分配使用 brk，因为用 mmap 映射匿名页，当发生缺页异常时，linux 内核为缺页分配一个新物理页，并将该物理页清 0，一个 mmap 的内存块需要映射多个物理页，导致多次清 0 操作，很浪费系统资源，所以引入了 mmap 分配阈值动态调整机制，保证在必要的情况下才使用 mmap 分配内存。
4. 尽量只缓存临时使用的空闲小内存块，对大内存块或是长生命周期的大内存块在释放时都直接归还给操作系统。
5. 对空闲的小内存块只会在 malloc 和 free 的时候进行合并，free 时空闲内存块可能放入 pool 中，不一定归还给操作系统。
6. 收缩堆的条件是当前 free 的块大小加上前后能合并 chunk 的大小大于 64KB、，并且堆顶的大小达到阈值，才有可能收缩堆，把堆最顶端的空闲内存返回给操作系统。
7. 需要保持长期存储的程序不适合用 ptmalloc 来管理内存。
8. 为了支持多线程，多个线程可以从同一个分配区（arena）中分配内存，ptmalloc 假设线程 A 释放掉一块内存后，线程 B 会申请类似大小的内存，但是 A 释放的内存跟 B 需要的内存不一定完全相等，可能有一个小的误差，就需要不停地对内存块作切割和合并，这个过程中可能产生内存碎片。

由于学习 ptmalloc 源代码的目的是要解决项目中遇到的内存暴增问题，所以主要关注了 ptmalloc 可能造成内存管理问题的假设，所以不免有点鸡蛋里挑骨头的味道。

3.2.3 内存管理数据结构概述

3.2.3.1 Main_arena 与 non_main_arena

在 Doug Lea 实现的内存分配器中只有一个主分配区（main arena），每次分配内存都必须对主分配区加锁，分配完成后释放锁，在 SMP 多线程环境下，对主分配区的锁的争用很激烈，严重影响了 malloc 的分配效率。于是 Wolfram Gloger 在 Doug Lea 的基础上改进使得 Glibc 的 malloc 可以支持多线程，增加了非主分配区（non main arena）支持，主分配区与非主分配区用环形链表进行管理。每一个分配区利用互斥锁（mutex）使线程对于该分配区的访问互斥。

每个进程只有一个主分配区，但可能存在多个非主分配区，ptmalloc 根据系统对分配区的争用情况动态增加非主分配区的数量，分配区的数量一旦增加，就不会再减少了。主分配区可以访问进程的 heap 区域和 mmap 映射区域，也就是说主分配区可以使用 sbrk 和 mmap 向操作系统申请虚拟内存。而非主分配区只能访问进程的 mmap 映射区域，非主分配区每次使用 mmap()向操作系统“批发”HEAP_MAX_SIZE（32 位系统上默认为 1MB，64 位系统默认为 64MB）大小的虚拟内存，当用户向非主分配区请求分配内存时再切割成小块“零售”出去，毕竟系统调用是相对低效的，直接从用户空间分配内存快多了。所以 ptmalloc 在必要

的情况下才会调用 mmap()函数向操作系统申请虚拟内存。

主分配区可以访问 heap 区域，如果用户不调用 brk()或是 sbrk()函数，分配程序就可以保证分配到连续的虚拟地址空间，因为每个进程只有一个主分配区使用 sbrk()分配 heap 区域的虚拟内存。内核对 brk 的实现可以看着是 mmap 的一个精简版，相对高效一些。如果主分配区的内存是通过 mmap()向系统分配的，当 free 该内存时，主分配区会直接调用 munmap()将该内存归还给系统。

当某一线程需要调用 malloc()分配内存空间时，该线程先查看线程私有变量中是否已经存在一个分配区，如果存在，尝试对该分配区加锁，如果加锁成功，使用该分配区分配内存，如果失败，该线程搜索循环链表试图获得一个没有加锁的分配区。如果所有的分配区都已经加锁，那么 malloc()会开辟一个新的分配区，把该分配区加入到全局分配区循环链表并加锁，然后使用该分配区进行分配内存操作。在释放操作中，线程同样试图获得待释放内存块所在分配区的锁，如果该分配区正在被别的线程使用，则需要等待直到其他线程释放该分配区的互斥锁之后才可以进行释放操作。

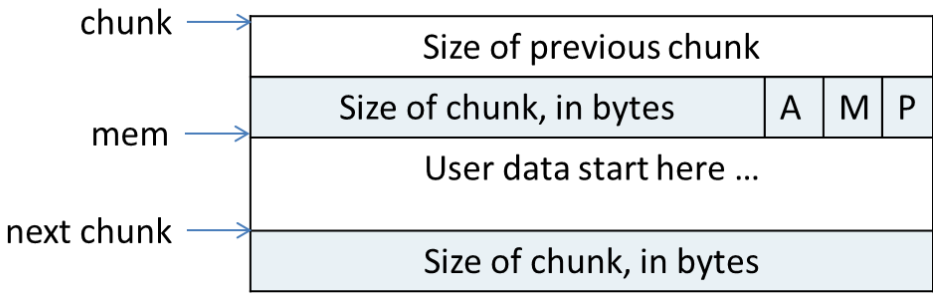
申请小块内存时会产生很多内存碎片，ptmalloc 在整理时也需要对分配区做加锁操作。每个加锁操作大概需要 5～10 个 cpu 指令，而且程序线程很多的情况下，锁等待的时间就会延长，导致 malloc 性能下降。一次加锁操作需要消耗 100ns 左右，正是锁的缘故，导致 ptmalloc 在多线程竞争情况下性能远远落后于 tcmalloc。最新版的 ptmalloc 对锁进行了优化，加入了 PER_THREAD 和 ATOMIC_FASTBINS 优化，但默认编译不会启用该优化，这两个对锁的优化应该能够提升多线程内存的分配的效率。

3.2.3.2 chunk 的组织

不管内存是在哪里被分配的，用什么方法分配，用户请求分配的空间在 ptmalloc 中都使用一个 chunk 来表示。用户调用 free()函数释放掉的内存也并不是立即就归还给操作系统，相反，它们也会被表示为一个 chunk，ptmalloc 使用特定的数据结构来管理这些空闲的 chunk。

1．Chunk 格式

ptmalloc 在给用户分配的空间的前后加上了一些控制信息，用这样的方法来记录分配的信息，以便完成分配和释放工作。一个使用中的 chunk（使用中，就是指还没有被 free 掉）在内存中的样子如图所示：



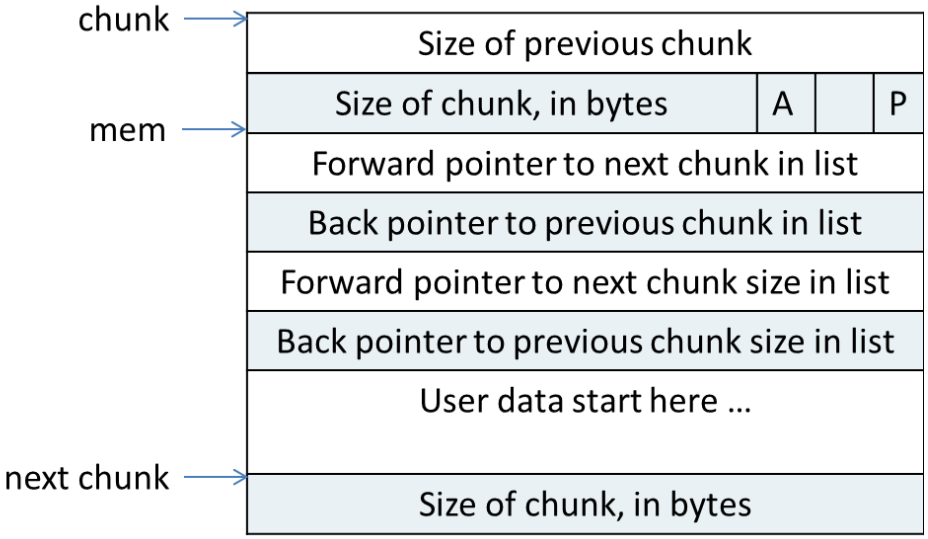
在图中，chunk 指针指向一个 chunk 的开始，一个 chunk 中包含了用户请求的内存区域和相关的控制信息。图中的 mem 指针才是真正返回给用户的内存指针。chunk 的第二个域的最低一位为 P，它表示前一个块是否在使用中，P 为 0 则表示前一个 chunk 为空闲，这时 chunk 的第一个域 prev_size 才有效，prev_size 表示前一个 chunk 的 size，程序可以使用这个值来找到前一个 chunk 的开始地址。当 P 为 1 时，表示前一个 chunk 正在使用中，prev_size

无效，程序也就不可以得到前一个 chunk 的大小。不能对前一个 chunk 进行任何操作。ptmalloc 分配的第一个块总是将 P 设为 1，以防止程序引用到不存在的区域。

Chunk 的第二个域的倒数第二个位为 M，他表示当前 chunk 是从哪个内存区域获得的虚拟内存。M 为 1 表示该 chunk 是从 mmap 映射区域分配的，否则是从 heap 区域分配的。

Chunk 的第二个域倒数第三个位为 A，表示该 chunk 属于主分配区或者非主分配区，如果属于非主分配区，将该位置为 1，否则置为 0。

空闲 chunk 在内存中的结构如图所示：



当 chunk 空闲时，其 M 状态不存在，只有 AP 状态，原本是用户数据区的地方存储了四个指针，指针 fd 指向后一个空闲的 chunk，而 bk 指向前一个空闲的 chunk，ptmalloc 通过这两个指针将大小相近的 chunk 连成一个双向链表。对于 large bin 中的空闲 chunk，还有两个指针，fd_nextsize 和 bk_nextsize，这两个指针用于加快在 large bin 中查找最近匹配的空闲 chunk。不同的 chunk 链表又是通过 bins 或者 fastbins 来组织的（bins 和 fastbins 在 3.2.3.3 中介绍）。

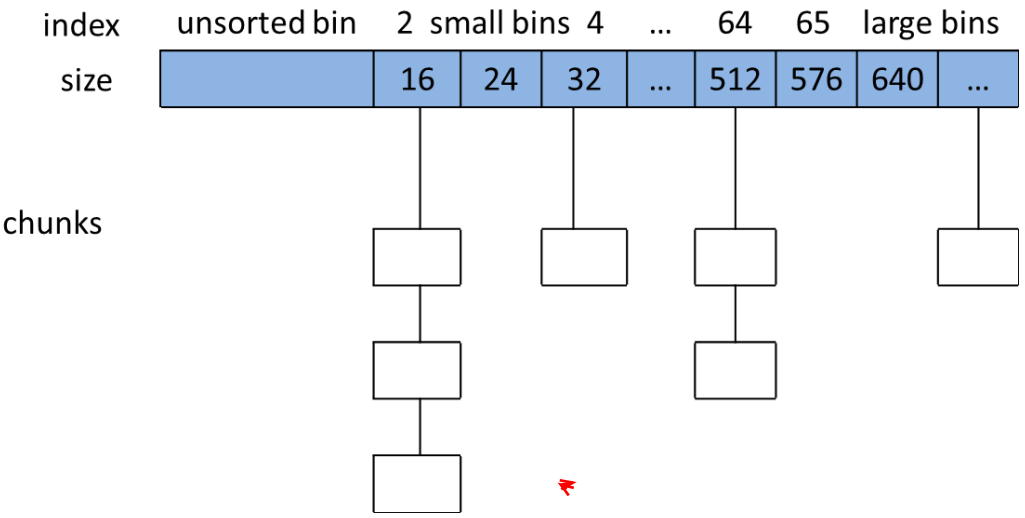
2．chunk 中的空间复用

为了使得 chunk 所占用的空间最小，ptmalloc 使用了空间复用，一个 chunk 或者正在被使用，或者已经被 free 掉，所以 chunk 的中的一些域可以在使用状态和空闲状态表示不同的意义，来达到空间复用的效果。以 32 位系统为例，空闲时，一个 chunk 中至少需要 4 个 size_t（4B）大小的空间，用来存储 prev_size，size，fd 和 bk（见上图），也就是 16B，chunk 的大小要对齐到 8B。当一个 chunk 处于使用状态时，它的下一个 chunk 的 prev_size 域肯定是无效的。所以实际上，这个空间也可以被当前 chunk 使用。这听起来有点不可思议，但确实是合理空间复用的例子。故而实际上，一个使用中的 chunk 的大小的计算公式应该是：in_use_size = (用户请求大小+ 8 - 4) align to 8B，这里加 8 是因为需要存储 prev_size 和 size，但又因为向下一个 chunk“借”了 4B，所以要减去 4。最后，因为空闲的 chunk 和使用中的 chunk 使用的是同一块空间。所以肯定要取其中最大者作为实际的分配空间。即最终的分配空间 chunk_size = max(in_use_size, 16)。这就是当用户请求内存分配时，ptmalloc 实际需要分配的内存大小，在后面的介绍中。如果不是特别指明的地方，指的都是这个经过转换的实际需要分配的内存大小，而不是用户请求的内存分配大小。

3.2.3.3 空闲 chunk 容器

1. Bins

用户 free 掉的内存并不是都会马上归还给系统，ptmalloc 会统一管理 heap 和 mmap 映射区域中的空闲的 chunk，当用户进行下一次分配请求时，ptmalloc 会首先试图在空闲的 chunk 中挑选一块给用户，这样就避免了频繁的系统调用，降低了内存分配的开销。ptmalloc 将相似大小的 chunk 用双向链表链接起来，这样的一个链表被称为一个 bin。Ptmalloc 一共维护了 128 个 bin，并使用一个数组来存储这些 bin（如下图所示）。



数组中的第一个为 unsorted bin，数组中从 2 开始编号的前 64 个 bin 称为 small bins，同一个 small bin 中的 chunk 具有相同的大小。两个相邻的 small bin 中的 chunk 大小相差 8bytes。small bins 中的 chunk 按照最近使用顺序进行排列，最后释放的 chunk 被链接到链表的头部，而申请 chunk 是从链表尾部开始，这样，每一个 chunk 都有相同的机会被 ptmalloc 选中。Small bins 后面的 bin 被称作 large bins。large bins 中的每一个 bin 分别包含了一个给定范围内的 chunk，其中的 chunk 按大小序排列。相同大小的 chunk 同样按照最近使用顺序排列。ptmalloc 使用“smallest-first，best-fit”原则在空闲 large bins 中查找合适的 chunk。

当空闲的 chunk 被链接到 bin 中的时候，ptmalloc 会把表示该 chunk 是否处于使用中的标志 P 设为 0（注意，这个标志实际上处在下一个 chunk 中），同时 ptmalloc 还会检查它前后的 chunk 是否也是空闲的，如果是的话，ptmalloc 会首先把它们合并为一个大的 chunk，然后将合并后的 chunk 放到 unstored bin 中。要注意的是，并不是所有的 chunk 被释放后就立即被放到 bin 中。ptmalloc 为了提高分配的速度，会把一些小的的 chunk 先放到一个叫做 fast bins 的容器内。

2. Fast Bins

一般的情况是，程序在运行时会经常需要申请和释放一些较小的内存空间。当分配器合并了相邻的几个小的 chunk 之后，也许马上就会有另一个小块内存的请求，这样分配器又需要从大的空闲内存中切分出一块，这样无疑是比较低效的，故而，ptmalloc 中在分配过程中引入了 fast bins，不大于 max_fast（默认值为 64B）的 chunk 被释放后，首先会被放到 fast bins 中，fast bins 中的 chunk 并不改变它的使用标志 P。这样也就无法将它们合并，当需要给用户分配的 chunk 小于或等于 max_fast 时，ptmalloc 首先会在 fast bins 中查找相应的空闲块，然后才会去查找 bins 中的空闲 chunk。在某个特定的时候，ptmalloc 会遍历 fast bins 中的 chunk，

将相邻的空闲 chunk 进行合并，并将合并后的 chunk 加入 unsorted bin 中，然后再将 usorted bin 里的 chunk 加入 bins 中。

3．Unsorted Bin

unsorted bin 的队列使用 bins 数组的第一个，如果被用户释放的 chunk 大于 max_fast，或者 fast bins 中的空闲 chunk 合并后，这些 chunk 首先会被放到 unsorted bin 队列中，在进行 malloc 操作的时候，如果在 fast bins 中没有找到合适的 chunk，则 ptmalloc 会先在 unsorted bin 中查找合适的空闲 chunk，然后才查找 bins。如果 unsorted bin 不能满足分配要求。malloc 便会将 unsorted bin 中的 chunk 加入 bins 中。然后再从 bins 中继续进行查找和分配过程。从这个过程可以看出来，unsorted bin 可以看做是 bins 的一个缓冲区，增加它只是为了加快分配的速度。

4．Top chunk

并不是所有的 chunk 都按照上面的方式来组织，实际上，有三种例外情况。Top chunk，mmaped chunk 和 last remainder，下面会分别介绍这三类特殊的 chunk。top chunk 对于主分配区和非主分配区是不一样的。

对于非主分配区会预先从 mmap 区域分配一块较大的空闲内存模拟 sub-heap，通过管理 sub-heap 来响应用户的需求，因为内存是按地址从低向高进行分配的，在空闲内存的最高处，必然存在着一块空闲 chunk，叫做 top chunk。当 bins 和 fast bins 都不能满足分配需要的时候，ptmalloc 会设法在 top chunk 中分出一块内存给用户，如果 top chunk 本身不够大，分配程序会重新分配一个 sub-heap，并将 top chunk 迁移到新的 sub-heap 上，新的 sub-heap 与已有的 sub-heap 用单向链表连接起来，然后在新的 top chunk 上分配所需的内存以满足分配的需要，实际上，top chunk 在分配时总是在 fast bins 和 bins 之后被考虑，所以，不论 top chunk 有多大，它都不会被放到 fast bins 或者是 bins 中。Top chunk 的大小是随着分配和回收不停变换的，如果从 top chunk 分配内存会导致 top chunk 减小，如果回收的 chunk 恰好与 top chunk 相邻，那么这两个 chunk 就会合并成新的 top chunk，从而使 top chunk 变大。如果在 free 时回收的内存大于某个阈值，并且 top chunk 的大小也超过了收缩阈值，ptmalloc 会收缩 sub-heap，如果 top-chunk 包含了整个 sub-heap，ptmalloc 会调用 munmap 把整个 sub-heap 的内存返回给操作系统。

由于主分配区是唯一能够映射进程 heap 区域的分配区，它可以通过 sbrk()来增大或是收缩进程 heap 的大小，ptmalloc 在开始时会预先分配一块较大的空闲内存（也就是所谓的 heap），主分配区的 top chunk 在第一次调用 malloc 时会分配一块(chunk_size + 128KB) align 4KB 大小的空间作为初始的 heap，用户从 top chunk 分配内存时，可以直接取出一块内存给用户。在回收内存时，回收的内存恰好与 top chunk 相邻则合并成新的 top chunk，当该次回收的空闲内存大小达到某个阈值，并且 top chunk 的大小也超过了收缩阈值，会执行内存收缩，减小 top chunk 的大小，但至少要保留一个页大小的空闲内存，从而把内存归还给操作系统。如果向主分配区的 top chunk 申请内存，而 top chunk 中没有空闲内存，ptmalloc 会调用 sbrk()将的进程 heap 的边界 brk 上移，然后修改 top chunk 的大小。

5．mmaped chunk

当需要分配的 chunk 足够大，而且 fast bins 和 bins 都不能满足要求，甚至 top chunk 本身也不能满足分配需求时，ptmalloc 会使用 mmap 来直接使用内存映射来将页映射到进程空间。这样分配的 chunk 在被 free 时将直接解除映射，于是就将内存归还给了操作系统，再次对这样的内存区的引用将导致 segmentation fault 错误。这样的 chunk 也不会包含在任何

bin 中。

6. Last remainder

Last remainder 是另外一种特殊的 chunk，就像 top chunk 和 mmaped chunk 一样，不会在任何 bins 中找到这种 chunk。当需要分配一个 small chunk，但在 small bins 中找不到合适的 chunk，如果 last remainder chunk 的大小大于所需的 small chunk 大小，last remainder chunk 被分裂成两个 chunk，其中一个 chunk 返回给用户，另一个 chunk 变成新的 last remainder chuk。

3.2.3.4 sbrk 与 mmap

从进程的内存布局可知，.bss 段之上的这块分配给用户程序的空间被称为 heap（堆）。start_brk 指向 heap 的开始，而 brk 指向 heap 的顶部。可以使用系统调用 brk()和 sbrk()来增加标识 heap 顶部的 brk 值，从而线性的增加分配给用户的 heap 空间。在使 malloc 之前，brk 的值等于 start_brk，也就是说 heap 大小为 0。ptmalloc 在开始时，若请求的空间小于 mmap 分配阈值（mmap threshold，默认值为 128KB）时，主分配区会调用 sbrk()增加一块大小为 (128 KB + chunk_size) align 4KB 的空间作为 heap。非主分配区会调用 mmap 映射一块大小为 HEAP_MAX_SIZE（32 位系统上默认为 1MB，64 位系统上默认为 64MB）的空间作为 sub-heap。这就是前面所说的 ptmalloc 所维护的分配空间，当用户请求内存分配时，首先会在这个区域内找一块合适的 chunk 给用户。当用户释放了 heap 中的 chunk 时，ptmalloc 又会使用 fast bins 和 bins 来组织空闲 chunk。以备用户的下一次分配。若需要分配的 chunk 大小小于 mmap 分配阈值，而 heap 空间又不够，则此时主分配区会通过 sbrk()调用来增加 heap 大小，非主分配区会调用 mmap 映射一块新的 sub-heap，也就是增加 top chunk 的大小，每次 heap 增加的值都会对齐到 4KB。

当用户的请求超过 mmap 分配阈值，并且主分配区使用 sbrk()分配失败的时候，或是非主分配区在 top chunk 中不能分配到需要的内存时，ptmalloc 会尝试使用 mmap()直接映射一块内存到进程内存空间。使用 mmap()直接映射的 chunk 在释放时直接解除映射，而不再属于进程的内存空间。任何对该内存的访问都会产生段错误。而在 heap 中或是 sub-heap 中分配的空间则可能会留在进程内存空间内，还可以再次引用（当然是很危险的）。

当 ptmalloc munmap chunk 时，如果回收的 chunk 空间大小大于 mmap 分配阈值的当前值，并且小于 DEFAULT_MMAP_THRESHOLD_MAX（32 位系统默认为 512KB，64 位系统默认为 32MB），ptmalloc 会把 mmap 分配阈值调整为当前回收的 chunk 的大小，并将 mmap 收缩阈值（mmap trim threshold）设置为 mmap 分配阈值的 2 倍。这就是 ptmalloc 的对 mmap 分配阈值的动态调整机制，该机制是默认开启的，当然也可以用 mallopt()关闭该机制（将在 3.2.6 节介绍如何关闭该机制）。

3.2.4 内存分配概述

1. 分配算法概述，以 32 系统为例，64 位系统类似。
   - 小于等于 64 字节：用 pool 算法分配。
   - 64 到 512 字节之间：在最佳匹配算法分配和 pool 算法分配中取一种合适的。
   - 大于等于 512 字节：用最佳匹配算法分配。
   - 大于等于 mmap 分配阈值（默认值 128KB）：根据设置的 mmap 的分配策略进行分配，如果没有开启 mmap 分配阈值的动态调整机制，大于等于 128KB 就直接调用 mmap

分配。否则，大于等于 mmap 分配阈值时才直接调用 mmap()分配。

2. ptmalloc 的响应用户内存分配要求的具体步骤为:
   1) 获取分配区的锁，为了防止多个线程同时访问同一个分配区，在进行分配之前需要取得分配区域的锁。线程先查看线程私有实例中是否已经存在一个分配区，如果存在尝试对该分配区加锁，如果加锁成功，使用该分配区分配内存，否则，该线程搜索分配区循环链表试图获得一个空闲（没有加锁）的分配区。如果所有的分配区都已经加锁，那么 ptmalloc 会开辟一个新的分配区，把该分配区加入到全局分配区循环链表和线程的私有实例中并加锁，然后使用该分配区进行分配操作。开辟出来的新分配区一定为非主分配区，因为主分配区是从父进程那里继承来的。开辟非主分配区时会调用 mmap()创建一个 sub-heap，并设置好 top chunk。
   2) 将用户的请求大小转换为实际需要分配的 chunk 空间大小。
   3) 判断所需分配 chunk 的大小是否满足 chunk_size <= max_fast (max_fast 默认为 64B)，如果是的话，则转下一步，否则跳到第 5 步。
   4) 首先尝试在 fast bins 中取一个所需大小的 chunk 分配给用户。如果可以找到，则分配结束。否则转到下一步。
   5) 判断所需大小是否处在 small bins 中，即判断 chunk_size < 512B 是否成立。如果 chunk 大小处在 small bins 中，则转下一步，否则转到第 6 步。
   6) 根据所需分配的 chunk 的大小，找到具体所在的某个 small bin，从该 bin 的尾部摘取一个恰好满足大小的 chunk。若成功，则分配结束，否则，转到下一步。
   7) 到了这一步，说明需要分配的是一块大的内存，或者 small bins 中找不到合适的 chunk。于是，ptmalloc 首先会遍历 fast bins 中的 chunk，将相邻的 chunk 进行合并，并链接到 unsorted bin 中，然后遍历 unsorted bin 中的 chunk，如果 unsorted bin 只有一个 chunk，并且这个 chunk 在上次分配时被使用过，并且所需分配的 chunk 大小属于 small bins，并且 chunk 的大小大于等于需要分配的大小，这种情况下就直接将该 chunk 进行切割，分配结束，否则将根据 chunk 的空间大小将其放入 small bins 或是 large bins 中，遍历完成后，转入下一步。
   8) 到了这一步，说明需要分配的是一块大的内存，或者 small bins 和 unsorted bin 中都找不到合适的 chunk，并且 fast bins 和 unsorted bin 中所有的 chunk 都清除干净了。从 large bins 中按照“smallest-first，best-fit”原则，找一个合适的 chunk，从中划分一块所需大小的 chunk，并将剩下的部分链接回到 bins 中。若操作成功，则分配结束，否则转到下一步。
   9) 如果搜索 fast bins 和 bins 都没有找到合适的 chunk，那么就需要操作 top chunk 来进行分配了。判断 top chunk 大小是否满足所需 chunk 的大小，如果是，则从 top chunk 中分出一块来。否则转到下一步。
   10) 到了这一步，说明 top chunk 也不能满足分配要求，所以，于是就有了两个选择: 如果是主分配区，调用 sbrk()，增加 top chunk 大小；如果是非主分配区，调用 mmap 来分配一个新的 sub-heap，增加 top chunk 大小；或者使用 mmap()来直接分配。在这里，需要依靠 chunk 的大小来决定到底使用哪种方法。判断所需分配的 chunk 大小是否大于等于 mmap 分配阈值，如果是的话，则转下一步，调用 mmap 分配，否则跳到第 12 步，增加 top chunk 的大小。
   11) 使用 mmap 系统调用为程序的内存空间映射一块 chunk_size align 4kB 大小的空间。然后将内存指针返回给用户。
   12) 判断是否为第一次调用 malloc，若是主分配区，则需要进行一次初始化工作，分配

一块大小为(chunk_size + 128KB) align 4KB 大小的空间作为初始的 heap。若已经初始化过了，主分配区则调用 sbrk()增加 heap 空间，分主分配区则在 top chunk 中切割出一个 chunk，使之满足分配需求，并将内存指针返回给用户。

总结一下：根据用户请求分配的内存的大小，ptmalloc 有可能会在两个地方为用户分配内存空间。在第一次分配内存时，一般情况下只存在一个主分配区，但也有可能从父进程那里继承来了多个非主分配区，在这里主要讨论主分配区的情况，brk 值等于 start_brk，所以实际上 heap 大小为 0，top chunk 大小也是 0。这时，如果不增加 heap 大小，就不能满足任何分配要求。所以，若用户的请求的内存大小小于 mmap 分配阈值，则 ptmalloc 会初始 heap。然后在 heap 中分配空间给用户，以后的分配就基于这个 heap 进行。若第一次用户的请求就大于 mmap 分配阈值，则 ptmalloc 直接使用 mmap()分配一块内存给用户，而 heap 也就没有被初始化，直到用户第一次请求小于 mmap 分配阈值的内存分配。第一次以后的分配就比较复杂了，简单说来，ptmalloc 首先会查找 fast bins，如果不能找到匹配的 chunk，则查找 small bins。若还是不行，合并 fast bins，把 chunk 加入 unsorted bin，在 unsorted bin 中查找，若还是不行，把 unsorted bin 中的 chunk 全加入 large bins 中，并查找 large bins。在 fast bins 和 small bins 中的查找都需要精确匹配，而在 large bins 中查找时，则遵循“smallest-first，best-fit”的原则，不需要精确匹配。若以上方法都失败了，则 ptmalloc 会考虑使用 top chunk。若 top chunk 也不能满足分配要求。而且所需 chunk 大小大于 mmap 分配阈值，则使用 mmap 进行分配。否则增加 heap，增大 top chunk。以满足分配要求。

3.2.5 内存回收概述

free() 函数接受一个指向分配区域的指针作为参数，释放该指针所指向的 chunk。而具体的释放方法则看该 chunk 所处的位置和该 chunk 的大小。free()函数的工作步骤如下：

1) free()函数同样首先需要获取分配区的锁，来保证线程安全。
2) 判断传入的指针是否为 0，如果为 0，则什么都不做，直接 return。否则转下一步。
3) 判断所需释放的 chunk 是否为 mmaped chunk，如果是，则调用 munmap()释放 mmaped chunk，解除内存空间映射，该该空间不再有效。如果开启了 mmap 分配阈值的动态调整机制，并且当前回收的 chunk 大小大于 mmap 分配阈值，将 mmap 分配阈值设置为该 chunk 的大小，将 mmap 收缩阈值设定为 mmap 分配阈值的 2 倍，释放完成，否则跳到下一步。
4) 判断 chunk 的大小和所处的位置，若 chunk_size <= max_fast，并且 chunk 并不位于 heap 的顶部，也就是说并不与 top chunk 相邻，则转到下一步，否则跳到第 6 步。（因为与 top chunk 相邻的小 chunk 也和 top chunk 进行合并，所以这里不仅需要判断大小，还需要判断相邻情况）
5) 将 chunk 放到 fast bins 中，chunk 放入到 fast bins 中时，并不修改该 chunk 使用状态位 P。也不与相邻的 chunk 进行合并。只是放进去，如此而已。这一步做完之后释放便结束了，程序从 free()函数中返回。
6) 判断前一个 chunk 是否处在使用中，如果前一个块也是空闲块，则合并。并转下一步。
7) 判断当前释放 chunk 的下一个块是否为 top chunk，如果是，则转第 9 步，否则转下一步。
8) 判断下一个 chunk 是否处在使用中，如果下一个 chunk 也是空闲的，则合并，并将

合并后的 chunk 放到 unsorted bin 中。注意，这里在合并的过程中，要更新 chunk 的大小，以反映合并后的 chunk 的大小。并转到第 10 步。

9) 如果执行到这一步，说明释放了一个与 top chunk 相邻的 chunk。则无论它有多大，都将它与 top chunk 合并，并更新 top chunk 的大小等信息。转下一步。

10) 判断合并后的 chunk 的大小是否大于 FASTBIN_CONSOLIDATION_THRESHOLD（默认 64KB），如果是的话，则会触发进行 fast bins 的合并操作，fast bins 中的 chunk 将被遍历，并与相邻的空闲 chunk 进行合并，合并后的 chunk 会被放到 unsorted bin 中。fast bins 将变为空，操作完成之后转下一步。

11) 判断 top chunk 的大小是否大于 mmap 收缩阈值（默认为 128KB），如果是的话，对于主分配区，则会试图归还 top chunk 中的一部分给操作系统。但是最先分配的 128KB 空间是不会归还的，ptmalloc 会一直管理这部分内存，用于响应用户的分配请求；如果为非主分配区，会进行 sub-heap 收缩，将 top chunk 的一部分返回给操作系统，如果 top chunk 为整个 sub-heap，会把整个 sub-heap 还回给操作系统。做完这一步之后，释放结束，从 free() 函数退出。可以看出，收缩堆的条件是当前 free 的 chunk 大小加上前后能合并 chunk 的大小大于 64k，并且要 top chunk 的大小要达到 mmap 收缩阈值，才有可能收缩堆。

3.2.6 配置选项概述

Ptmalloc 主要提供以下几个配置选项用于调优，这些选项可以通过 mallopt()进行设置：

1. M_MXFAST

M_MXFAST 用于设置 fast bins 中保存的 chunk 的最大大小，默认值为 64B，fast bins 中保存的 chunk 在一段时间内不会被合并，分配小对象时可以首先查找 fast bins，如果 fast bins 找到了所需大小的 chunk，就直接返回该 chunk，大大提高小对象的分配速度，但这个值设置得过大，会导致大量内存碎片，并且会导致 ptmalloc 缓存了大量空闲内存，去不能归还给操作系统，导致内存暴增。

M_MXFAST 的最大值为 80B，不能设置比 80B 更大的值，因为设置为更大的值并不能提高分配的速度。Fast bins 是为需要分配许多小对象的程序设计的，比如需要分配许多小 struct，小对象，小的 string 等等。

如果设置该选项为 0，就会不使用 fast bins。

2. M_TRIM_THRESHOLD

M_TRIM_THRESHOLD 用于设置 mmap 收缩阈值，默认值为 128KB。自动收缩只会在 free 时才发生，如果当前 free 的 chunk 大小加上前后能合并 chunk 的大小大于 64KB，并且 top chunk 的大小达到 mmap 收缩阈值，对于主分配区，调用 malloc_trim()返回一部分内存给操作系统，对于非主分配区，调用 heap_trim()返回一部分内存给操作系统，在发生内存收缩时，还是从新设置 mmap 分配阈值和 mmap 收缩阈值。

这个选项一般与 M_MMAP_THRESHOLD 选项一起使用，M_MMAP_THRESHOLD 用于设置 mmap 分配阈值，对于长时间运行的程序，需要对这两个选项进行调优，尽量保证在 ptmalloc 中缓存的空闲 chunk 能够得到重用，尽量少用 mmap 分配临时用的内存。不停地使用系统调用 mmap 分配内存，然后很快又 free 掉该内存，这样是很浪费系统资源的，并且这样分配的内存的速度比从 ptmalloc 的空闲 chunk 中分配内存慢得多，由于需要页对齐导致空间利用率降低，并且操作系统调用 mmap()分配内存是串行的，在发生缺页异常时加载新的物理页，需要对新的物理页做清 0 操作，大大影响效率。

M_TRIM_THRESHOLD 的值必须设置为页大小对齐，设置为-1 会关闭内存收缩设置。

注意：试图在程序开始运行时分配一块大内存，并马上释放掉，以期望来触发内存收缩，这是不可能的，因为该内存马上就返回给操作系统了。

3. M_MMAP_THRESHOLD

M_MMAP_THRESHOLD 用于设置 mmap 分配阈值，默认值为 128KB，ptmalloc 默认开启动态调整 mmap 分配阈值和 mmap 收缩阈值。

当用户需要分配的内存大于 mmap 分配阈值，ptmalloc 的 malloc()函数其实相当于 mmap()的简单封装，free 函数相当于 munmap()的简单封装。相当于直接通过系统调用分配内存，回收的内存就直接返回给操作系统了。因为这些大块内存不能被 ptmalloc 缓存管理，不能重用，所以 ptmalloc 也只有在万不得已的情况下才使用该方式分配内存。

但使用 mmap 分配有如下的好处：

- Mmap 的空间可以独立从系统中分配和释放的系统，对于长时间运行的程序，申请长生命周期的大内存块就很适合有这种方式。
- Mmap 的空间不会被 ptmalloc 锁在缓存的 chunk 中，不会导致 ptmalloc 内存暴增的问题。
- 对有些系统的虚拟地址空间存在洞，只能用 mmap()进行分配内存，sbrk()不能运行。

使用 mmap 分配内存的缺点：

- 该内存不能被 ptmalloc 回收再利用。
- 会导致更多的内存浪费，因为 mmap 需要按页对齐。
- 它的分配效率跟操作系统提供的 mmap()函数的效率密切相关，Linux 系统强制把匿名 mmap 的内存物理页清 0 是很低效的。

所以用 mmap 来分配长生命周期的大内存块就是最好的选择，其他情况下都不太高效。

4. M_MMAP_MAX

M_MMAP_MAX 用于设置进程中用 mmap 分配的内存块的最大限制，默认值为 64K，因为有些系统用 mmap 分配的内存块太多会导致系统的性能下降。

如果将 M_MMAP_MAX 设置为 0，ptmalloc 将不会使用 mmap 分配大块内存。

Ptmalloc 为优化锁的竞争开销，做了 PER_THREAD 的优化，也提供了两个选项，M_ARENA_TEST 和 M_ARENA_MAX，由于 PER_THREAD 的优化默认没有开启，这里暂不对这两个选项做介绍。

另外，ptmalloc 没有提供关闭 mmap 分配阈值动态调整机制的选项，mmap 分配阈值动态调整时默认开启的，如果要关闭 mmap 分配阈值动态调整机制，可以设置 M_TRIM_THRESHOLD，M_MMAP_THRESHOLD，M_TOP_PAD 和 M_MMAP_MAX 中的任意一个。但是强烈建议不要关闭该机制，该机制保证了 ptmalloc 尽量重用缓存中的空闲内存，不用每次对相对大一些的内存使用系统调用 mmap 去分配内存。

3.2.7 使用注意事项

为了避免 Glibc 内存暴增，使用时需要注意以下几点：

1. 后分配的内存先释放，因为 ptmalloc 收缩内存是从 top chunk 开始，如果与 top chunk 相邻的 chunk 不能释放，top chunk 以下的 chunk 都无法释放。
2. Ptmalloc 不适合用于管理长生命周期的内存，特别是持续不定期分配和释放长生命周期的内存，这将导致 ptmalloc 内存暴增。如果要用 ptmalloc 分配长周期内存，在 32 位系

统上，分配的内存块最好大于 1MB，64 位系统上，分配的内存块大小大于 32MB。这是由于 ptmalloc 默认开启 mmap 分配阈值动态调整功能，1MB 是 32 位系统 mmap 分配阈值的最大值，32MB 是 64 位系统 mmap 分配阈值的最大值，这样可以保证 ptmalloc 分配的内存一定是从 mmap 映射区域分配的，当 free 时，ptmalloc 会直接把该内存返回给操作系统，避免了被 ptmalloc 缓存。

3. 不要关闭 ptmalloc 的 mmap 分配阈值动态调整机制，因为这种机制保证了短生命周期的内存分配尽量从 ptmalloc 缓存的内存 chunk 中分配，更高效，浪费更少的内存。如果关闭了该机制，对大于 128KB 的内存分配就会使用系统调用 mmap 向操作系统分配内存，使用系统调用分配内存一般会比从 ptmalloc 缓存的 chunk 中分配内存慢，特别是在多线程同时分配大内存块时，操作系统会串行调用 mmap()，并为发生缺页异常的页加载新物理页时，默认强制清 0。频繁使用 mmap 向操作系统分配内存是相当低效的。使用 mmap 分配的内存只适合长生命周期的大内存块。
4. 多线程分阶段执行的程序不适合用 ptmalloc，这种程序的内存更适合用内存池管理，就像 Appach 那样，每个连接请求处理分为多个阶段，每个阶段都有自己的内存池，每个阶段完成后，将相关的内存就返回给相关的内存池。Google 的许多应用也是分阶段执行的，他们在使用 ptmalloc 也遇到了内存暴增的相关问题，于是他们实现了 TCMalloc 来代替 ptmalloc，TCMalloc 具有内存池的优点，又有垃圾回收的机制，并最大限度优化了锁的争用，并且空间利用率也高于 ptmalloc。Ptmalloc 假设了线程 A 释放的内存块能在线程 B 中得到重用，但 B 不一定会分配和 A 线程同样大小的内存块，于是就需要不断地做切割和合并，可能导致内存碎片。
5. 尽量减少程序的线程数量和避免频繁分配/释放内存，Ptmalloc 在多线程竞争激烈的情况下，首先查看线程私有变量是否存在分配区，如果存在则尝试加锁，如果加锁不成功会尝试其它分配区，如果所有的分配区的锁都被占用着，就会增加一个非主分配区供当前线程使用。由于在多个线程的私有变量中可能会保存同一个分配区，所以当线程较多时，加锁的代价就会上升，ptmalloc 分配和回收内存都要对分配区加锁，从而导致了多线程竞争环境下 ptmalloc 的效率降低。
6. 防止内存泄露，ptmalloc 对内存泄露是相当敏感的，根据它的内存收缩机制，如果与 top chunk 相邻的那个 chunk 没有回收，将导致 top chunk 一下很多的空闲内存都无法返回给操作系统。
7. 防止程序分配过多内存，或是由于 Glibc 内存暴增，导致系统内存耗尽，程序因 OOM 被系统杀掉。预估程序可以使用的最大物理内存大小，配置系统的 /proc/sys/vm/overcommit_memory，/proc/sys/vm/overcommit_ratio，以及使用 ulimt –v 限制程序能使用虚拟内存空间大小，防止程序因 OOM 被杀掉。

4.问题分析及解决

通过前面几节对 ptmalloc 实现的粗略分析，尝试去分析和解决我们遇到的问题，我们系统遇到的问题是 glibc 内存暴增，现象是程序已经把内存返回给了 Glibc 库，但 Glibc 库却没有把内存归还给操作系统，最终导致系统内存耗尽，程序因为 OOM 被系统杀掉。

请参考 3.2.2 节对 ptmalloc 的设计假设与 3.2.7 节对 ptmalloc 的使用注意事项，原因有如下几点：

1. 在 64 位系统上使用默认的系统配置，也就是说 ptmalloc 的 mmap 分配阈值动态调整机制是开启的。我们的 NoSql 系统经常分配内存为 2MB，并且这 2MB 的内存很快会被释

放，在 ptmalloc 回收 2MB 内存时，ptmalloc 的动态调整机制会认为 2MB 对我们的系统来说是一个临时的内存分配，每次都用系统调用 mmap()向操作系统分配内存，ptmalloc 认为这太低效了，于是把 mmap 的阈值设置成了 2MB+4K，当下次再分配 2MB 的内存时，尽量从 ptmalloc 缓存的 chunk 中分配，缓存的 chunk 不能满足要求，才考虑调用 mmap()进行分配，提高分配的效率。

2. 系统中分配 2M 内存的地方主要有两处，一处是全局的内存 cache，另一处是网络模块，网络模块每次分配 2MB 内存用于处理网络的请求，处理完成后就释放该内存。这可以看成是一个短生命周期的内存。内存 cache 每次分配 2MB，但不确定什么时候释放，也不确定下次会什么时候会再分配 2MB 内存，但有一点可以确定，每次分配的 2MB 内存，要经过比较长的一段时间才会释放，所以可以看成是长生命周期的内存块，对于这些 cache 中的多个 2M 内存块没有使用 free list 管理，每次都是先从 cache 中 free 调用一个 2M 内存块，再从 Glibc 中分配一块新的 2M 内存块。Ptmalloc 不擅长管理长生命周期的内存块，ptmalloc 设计的假设中就明确假设缓存的内存块都用于短生命周期的内存分配，因为 ptmalloc 的内存收缩是从 top chunk 开始，如果与 top chunk 相邻的那个 chunk 在我们 NoSql 的内存池中没有释放，top chunk 以下的空闲内存都无法返回给系统，即使这些空闲内存有几十个 G 也不行。
3. Glibc 内存暴增的问题我们定位为全局内存池中的内存块长时间没有释放，其中还有一个原因就是全局内存池会不定期的分配内存，可能下次分配的内存是在 top chunk 分配的，分配以后又短时间不释放，导致 top chunk 升到了一个更高的虚拟地址空间，从而使 ptmalloc 中缓存的内存块更多，但无法返回给操作系统。
4. 另一个原因就是进程的线程数越多，在高压力高并发环境下，频繁分配和释放内存，由于分配内存时锁争用更激烈，ptmalloc 会为进程创建更多的分配区，由于我们的全局内存池的长时间不释放内存的缘故，会导致 ptmalloc 缓存的 chunk 数量增长得更快，从而更容易重现 Glibc 内存暴增的问题。在我们的 ms 上这个问题最为突出，就是这个原因。
5. 内存池管理内存的方式导致 Glibc 大量的内存碎片。我们的内存池对于小于等于 64K 的内存分配，则从内存池中分配 64K 的内存块，如果内存池中没有，则调用 malloc()分配 64K 的内存块，释放时，该 64K 的内存块加入内存中，永不还回给操作系统，对于大于 64K 的内存分配，调用 malloc()分配，释放时调用 free()函数换回给 Glibc。这些大量的 64K 的内存块长时间存在于内存池中，导致了 Glibc 中缓存了大量的内存碎片不能释放回操作系统。比如:

64K	100K	64K

假如应用层分配内存的顺序是 64K，100K，64K，然后释放 100K 的内存块，Glibc 会缓存这个 100K 的内存块，其中的两个 64K 内存块都在 mempool 中，一直不释放，如果下次再分配 64K 的内存，就会将 100K 的内存块拆分成 64K 和 36K 的两个内存块，64K 的内存块返回给应用层，并被 mempool 缓存，但剩下的 36K 被 Glibc 缓存，再也不能被应用层分配了，因为应用层分配的最小内存为 64K，这个 36K 的内存块就是内存碎片，这也是内存暴增的原因之一。

问题找到了，解决的办法可以参考如下几种:

1. 禁用 ptmalloc 的 mmap 分配阈值动态调整机制。通过 mallopt()设置 M_TRIM_THRESHOLD，M_MMAP_THRESHOLD，M_TOP_PAD 和 M_MMAP_MAX 中的任意一个，关闭 mmap 分配阈值动态调整机制，同时需要将 mmap 分配阈值设置为 64K，大于 64K 的内存分配都使用 mmap 向系统分配，释放大于 64K 的内存将调用 munmap 释放回系统。但强烈建议不要这么做，这会大大降低 ptmalloc 的分配

释放效率。因为系统调用 mmap 是串行的，操作系统需要对 mmap 分配内存加锁，而且操作系统对 mmap 的物理页强制清 0 很慢，请参看 3.2.6 选项配置相关描述。由于最初我们的系统的预分配优化做得不够好，关闭 mmap 的动态阈值调整机制后，chunkserver 在 ssd 上的性能减少到原来的 1/3，这种性能结果是无法让人接受的。

2. 我们系统的关键问题出在全局内存池，它分配的内存是长生命周期的大内存块，通过前面的分析可知，对长生命周期的大内存块分配最好用 mmap 系统调用直接向操作系统分配，回收时用 munmap 返回给操作系统。比如内存池每次用 mmap 向操作系统分配 8M 或是更多的虚拟内存。如果非要用 ptmalloc 的 malloc 函数分配内存，就得绕过 ptmalloc 的 mmap 分配阈值动态调整机制，mmap 分配阈值在 64 位系统上的最大值为 32M，如果分配的内存大于 32M，可以保证 malloc 分配的内存肯定是用 mmap 向操作系统分配的，回收时 free 一定会返回给操作系统，而不会被 ptmalloc 缓存用于下一次分配。但是如果这样使用 malloc 分配的话，其实 malloc 就是 mmap 的简单封装，还不如直接使用 mmap 系统调用想操作系统分配内存来得简单，并且显式调用 munmap 回收分配的内存，根本不依赖 ptmalloc 的实现。
3. 改写内存 cache，使用 free list 管理所分配的内存块。使用预分配优化已有的代码，尽量在每个请求过程中少分配内存。并使用线程私有内存块来存放线程所使用的私有实例。这种解决办法也是暂时的。
4. 从长远的设计来看，我们的系统也是分阶段执行的，每次网络请求都会分配 2MB 为单位内存，请求完成后释放请求锁分配的内存，内存池最适合这种情景的操作。我们的线程池至少需要包含对 2MB 和几种系统中常用分配大小的支持，采用与 TCMalloc 类似的无锁设计，使用线程私用变量的形式尽量减少分配时线程对锁的争用。或者直接使用 TCMalloc，免去了很多的线程池设计考虑。

5.源代码分析

本部分主要对源代码实现技巧的细节做分析，希望能进一步理解 ptmalloc 的实现，做到终极无惑。主要分析的文件包括 arena.c 和 malloc.c，这两个文件包括了 ptmalloc 的核心实现，其中 arena.c 主要是对多线程支持的实现，malloc.c 定义了公用的 malloc()，free()等函数，实现了基于分配区的内存管理算法。本部分不会从头到尾分析 arena.c 和 malloc.c 整个文件，而是根据 ptmalloc 的实现原理，分成几个模块分别介绍，主要分析了 malloc()和 free()函数的实现，对其它的函数如 realloc()，calloc()等不作介绍。由于 ptmalloc 同时支持 32 位平台和 64 位平台，所以这里的分析尽量兼顾到这两类平台，但主要基于 Linux X86 平台。

5.1 边界标记法

Ptmalloc 使用 chunk 实现内存管理，对 chunk 的管理基于独特的边界标记法，第三节已经对 chunk 的管理做了概述，这里将详细分析 chunk 管理的源代码实现。

在不同的平台下，每个 chunk 的最小大小，地址对齐方式是不同的，ptmalloc 依赖平台定义的 size_t 长度，对于 32 位平台，size_t 长度为 4 字节，对 64 位平台，size_t 长度可能为 4 字节，也可能为 8 字节，在 Linux X86_64 上 size_t 为 8 字节，这里就以 size_t 为 4 字节和 8 字节的情况进行分析。先看一段源代码：

```
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size */
#define SIZE_SZ                (sizeof(INTERNAL_SIZE_T))

/*
  MALLOC_ALIGNMENT is the minimum alignment for malloc'ed chunks.
  It must be a power of two at least 2 * SIZE_SZ, even on machines
  for which smaller alignments would suffice. It may be defined as
  larger than this though. Note however that code and data structures
  are optimized for the case of 8-byte alignment.
*/

#ifndef MALLOC_ALIGNMENT
#define MALLOC_ALIGNMENT       (2 * SIZE_SZ)
#endif

/* The corresponding bit mask value */
#define MALLOC_ALIGN_MASK      (MALLOC_ALIGNMENT - 1)
```

Ptmalloc 使用宏来屏蔽不同平台的差异，将 INTERNAL_SIZE_T 定义为 size_t，SIZE_SZ 定义为 size_t 的大小，在 32 位平台下位 4 字节，在 64 位平台下位 4 字节或者 8 字节。另外分配 chunk 时必须以 2*SIZE_SZ 对齐，MALLOC_ALIGNMENT 和 MALLOC_ALIGN_MASK 是用来处理 chunk 地址对齐的宏，将在后面的源代码介绍中经常看到。这里只需要知道在 32 平台 chunk 地址按 8 字节对齐，64 位平台按 8 字节或是 16 字节对齐就可以了。

Ptmalloc 采用边界标记法将内存划分成很多块，从而对内存的分配与回收进行管理。在 ptmalloc 的实现源码中定义结构体 malloc_chunk 来描述这些块，并使用宏封装了对 chunk 中每个域的读取，修改，校验，遍历等等。malloc_chunk 定义如下：

```
struct malloc_chunk {
  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. */
  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;
};
```

chunk 的定义相当简单明了，对各个域做一下简单介绍:

prev_size:如果前一个 chunk 是空闲的，该域表示前一个 chunk 的大小，如果前一个 chunk 不空闲，该域无意义。

size：当前 chunk 的大小，并且记录了当前 chunk 和前一个 chunk 的一些属性，包括前

一个 chunk 是否在使用中，当前 chunk 是否是通过 mmap 获得的内存，当前 chunk 是否属于非主分配区。

fd 和 bk：指针 fd 和 bk 只有当该 chunk 块空闲时才存在，其作用是用于将对应的空闲 chunk 块加入到空闲 chunk 块链表中统一管理，如果该 chunk 块被分配给应用程序使用，那么这两个指针也就没有用（该 chunk 块已经从空闲链中拆出）了，所以也当作应用程序的使用空间，而不至于浪费。

fd_nextsize 和 bk_nextsize:当当前的 chunk 存在于 large bins 中时，large bins 中的空闲 chunk 是按照大小排序的，但同一个大小的 chunk 可能有多个，增加了这两个字段可以加快遍历空闲 chunk，并查找满足需要的空闲 chunk，fd_nextsize 指向下一个比当前 chunk 大小大的第一个空闲 chunk，bk_nextszie 指向前一个比当前 chunk 大小小的第一个空闲 chunk。如果该 chunk 块被分配给应用程序使用，那么这两个指针也就没有用（该 chunk 块已经从 size 链中拆出）了，所以也当作应用程序的使用空间，而不至于浪费。

```
/*
  malloc_chunk details:

   (The following includes lightly edited explanations by Colin Plumb.)

   Chunks of memory are maintained using a `boundary tag' method as
   described in e.g., Knuth or Standish.  (See the paper by Paul
   Wilson ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps for a
   survey of such techniques.)  Sizes of free chunks are stored both
   in the front of each chunk and at the end.  This makes
   consolidating fragmented chunks into bigger chunks very fast.  The
   size fields also hold bits representing whether chunks are free or
   in use.

   An allocated chunk looks like this:

   chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
           |             Size of previous chunk, if allocated            | |
           +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
           |             Size of chunk, in bytes                       |M|P|
     mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
           |             User data starts here...                          .
           .                                                               .
           .             (malloc_usable_size() bytes)                      .
           .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
           |             Size of chunk                                     |
           +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

   Where "chunk" is the front of the chunk for the purpose of most of
   the malloc code, but "mem" is the pointer that is returned to the
```

```
user.  "Nextchunk" is the beginning of the next contiguous chunk.

Chunks always begin on even word boundries, so the mem portion
(which is returned to the user) is also on an even word boundary, and
thus at least double-word aligned.

Free chunks are stored in circular doubly-linked lists, and look like this:

chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Size of previous chunk                            |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
`head:' |             Size of chunk, in bytes                         |P|
  mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Forward pointer to next chunk in list             |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Back pointer to previous chunk in list            |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             Unused space (may be 0 bytes long)                .
        .                                                               .
        .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

The P (PREV_INUSE) bit, stored in the unused low-order bit of the
chunk size (which is always a multiple of two words), is an in-use
bit for the *previous* chunk.  If that bit is *clear*, then the
word before the current chunk size contains the previous chunk
size, and can be used to find the front of the previous chunk.
The very first chunk allocated always has this bit set,
preventing access to non-existent (or non-owned) memory. If
prev_inuse is set for any given chunk, then you CANNOT determine
the size of the previous chunk, and might even get a memory
addressing fault when trying to do so.

Note that the `foot' of the current chunk is actually represented
as the prev_size of the NEXT chunk. This makes it easier to
deal with alignments etc but can be very confusing when trying
to extend or adapt this code.

The two exceptions to all this are
 1. The special chunk `top' doesn't bother using the
    trailing size field since there is no next contiguous chunk
    that would have to index off it. After initialization, `top'
```

```
    is forced to always exist.  If it would become less than
    MINSIZE bytes long, it is replenished.

  2. Chunks allocated via mmap, which have the second-lowest-order
     bit M (IS_MMAPPED) set in their size fields.  Because they are
     allocated one-by-one, each must contain its own trailing size field.
*/
```

上面这段注释详细描述了 chunk 的细节，已分配的 chunk 和空闲的 chunk 形式不一样，充分利用空间复用，设计相当的巧妙。在前面的 3.2.3.2 节描述了这两种 chunk 形式，请参考前文的描述。

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

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))
/* The smallest size we can malloc is an aligned minimal chunk */
#define MINSIZE  \
  (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK))

/* Check if m has acceptable alignment */
#define aligned_OK(m)  (((unsigned long)(m) & MALLOC_ALIGN_MASK) == 0)
#define misaligned_chunk(p) \
  ((uintptr_t)(MALLOC_ALIGNMENT == 2 * SIZE_SZ ? (p) : chunk2mem (p)) \
   & MALLOC_ALIGN_MASK)
```

对于已经分配的 chunk，通过 chunk2mem 宏根据 chunk 地址获得返回给用户的内存地址，反过来通过 mem2chunk 宏根据 mem 地址得到 chunk 地址，chunk 的地址是按 2*SIZE_SZ 对齐的，而 chunk 结构体的前两个域刚好也是 2*SIZE_SZ 大小，所以，mem 地址也是 2*SIZE_SZ 对齐的。宏 aligned_OK 和 misaligned_chunk(p)用于校验地址是否是按 2*SIZE_SZ 对齐的。

MIN_CHUNK_SIZE 定义了最小的 chunk 的大小，32 位平台上位 16 字节，64 位平台为 24 字节或是 32 字节。MINSIZE 定义了最小的分配的内存大小，是对 MIN_CHUNK_SIZE 进行了 2*SIZE_SZ 对齐，地址对齐后与 MIN_CHUNK_SIZE 的大小仍然是一样的。

```
/*
   Check if a request is so large that it would wrap around zero when
   padded and aligned. To simplify some other code, the bound is made
   low enough so that adding MINSIZE will also not wrap around zero.
*/
#define REQUEST_OUT_OF_RANGE(req)                                 \
  ((unsigned long)(req) >=                                        \
   (unsigned long)(INTERNAL_SIZE_T)(-2 * MINSIZE))

/* pad request bytes into a usable size -- internal version */
#define request2size(req)                                         \
```

```
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)

/*  Same, except also perform argument check */
#define checked_request2size(req, sz)                             \
  if (REQUEST_OUT_OF_RANGE(req)) {                                \
    MALLOC_FAILURE_ACTION;                                        \
    return 0;                                                     \
  }                                                               \
  (sz) = request2size(req);
```

这几个宏用于将用户请求的分配大小转换成内部需要分配的 chunk 大小，这里需要注意的在转换时不但考虑的地址对齐，还额外加上了 SIZE_SZ，这意味着 ptmalloc 分配内存需要一个额外的 overhead，为 SIZE_SZ 字节，通过 chunk 的空间复用，我们很容易得出这个 overhead 为 SIZE_SZ。

以 Linux X86_64 平台为例，假设 SIZE_SZ 为 8 字节，空闲时，一个 chunk 中至少要 4 个 size_t（8B）大小的空间，用来存储 prev_size，size，fd 和 bk，也就是 MINSIZE（32B），chunk 的大小要对齐到 2*SIZE_SZ（16B）。当一个 chunk 处于使用状态时，它的下一个 chunk 的 prev_size 域肯定是无效的。所以实际上，这个空间也可以被当前 chunk 使用。这听起来有点不可思议，但确实是合理空间复用的例子。故而实际上，一个使用中的 chunk 的大小的计算公式应该是：in_use_size = (用户请求大小+ 16 - 8) align to 8B，这里加 16 是因为需要存储 prev_size 和 size，但又因为向下一个 chunk“借”了 8B，所以要减去 8，每分配一个 chunk 的 overhead 为 8B，即 SIZE_SZ 的大小。最后，因为空闲的 chunk 和使用中的 chunk 使用的是同一块空间。所以肯定要取其中最大者作为实际的分配空间。即最终的分配空间 chunk_size = max(in_use_size, 32)。这就是当用户请求内存分配时，ptmalloc 实际需要分配的内存大小。

注意：如果 chunk 是由 mmap ()直接分配的，则该 chunk 不会有前一个 chunk 和后一个 chunk，所有本 chunk 没有下一个 chunk 的 prev_size 的空间可以“借”，所以对于直接 mmap() 分配内存的 overhead 为 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)->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)->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 non-main arena */
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

chunk 在分割时总是以地址对齐（默认是 8 字节，可以自由设置，但是 8 字节是最小值并且设置的值必须是 2 为底的幂函数值，即是 alignment = 2^n，n 为整数且 n>=3）的方式来进行的，所以用 chunk->size 来存储本 chunk 块大小字节数的话，其末 3bit 位总是 0，因此这三位可以用来存储其它信息，比如：

以第 0 位作为 P 状态位，标记前一 chunk 块是否在使用中，为 1 表示使用，为 0 表示空闲。

以第 1 位作为 M 状态位，标记本 chunk 块是否是使用 mmap()直接从进程的 mmap 映射区域分配的，为 1 表示是，为 0 表示否。

以第 2 位作为 A 状态位，标记本 chunk 是否属于非主分配区，为 1 表示是，为 0 表示否。

```
/*
  Bits to mask off when extracting size

  Note: IS_MMAPPED is intentionally not masked off from size field in
  macros for which mmapped chunks should never be seen. This should
  cause helpful core dumps to occur if it is tried by accident by
  people extending or adapting this malloc.
*/
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
/* Get size, ignoring use bits */
#define chunksize(p)         ((p)->size & ~(SIZE_BITS))
/* Ptr to next physical malloc_chunk. */
#define next_chunk(p) ((mchunkptr)( ((char*)(p)) + ((p)->size & ~SIZE_BITS) ))
/* Ptr to previous physical malloc_chunk */
#define prev_chunk(p) ((mchunkptr)( ((char*)(p)) - ((p)->prev_size) ))
/* Treat space at ptr + offset as a chunk */
#define chunk_at_offset(p, s)  ((mchunkptr)(((char*)(p)) + (s)))
```

prev_size 字段虽然在当前 chunk 块结构体内，记录的却是前一个邻接 chunk 块的信息，这样做的好处就是我们通过本块 chunk 结构体就可以直接获取到前一 chunk 块的信息，从而方便做进一步的处理操作。相对的，当前 chunk 块的 foot 信息就存在于下一个邻接 chunk 块的结构体内。字段 prev_size 记录的什么信息呢？有两种情况：

1）如果前一个邻接 chunk 块空闲，那么当前 chunk 块结构体内的 prev_size 字段记录的是前一个邻接 chunk 块的大小。这就是由当前 chunk 指针获得前一个空闲 chunk 地址的依据。宏 prev_chunk(p)就是依赖这个假设实现的。

2）如果前一个邻接 chunk 在使用中，则当前 chunk 的 prev_size 的空间被前一个 chunk 借用中，其中的值是前一个 chunk 的内存内容，对当前 chunk 没有任何意义。

字段 size 记录了本 chunk 的大小，无论下一个 chunk 是空闲状态或是被使用状态，都可

以通过本 chunk 的地址加上本 chunk 的大小，得到下一个 chunk 的地址，由于 size 的低 3 个 bit 记录了控制信息，需要屏蔽掉这些控制信息，取出实际的 size 在进行计算下一个 chunk 地址，这是 next_chunk(p)的实现原理。

宏 chunksize(p)用于获得 chunk 的实际大小，需要屏蔽掉 size 中的控制信息。

宏 chunk_at_offset(p, s)将 p+s 的地址强制看作一个 chunk。

注意：按照边界标记法，可以有多个连续的并且正在被使用中的 chunk 块，但是不会有多个连续的空闲 chunk 块，因为连续的多个空闲 chunk 块一定会合并成一个大的空闲 chunk 块。

```
/* extract p's inuse bit */
#define inuse(p)\
((((mchunkptr)(((char*)(p))+((p)->size & ~SIZE_BITS)))->size) & PREV_INUSE)
/* set/clear chunk as being inuse without otherwise disturbing */
#define set_inuse(p)\
((mchunkptr)(((char*)(p)) + ((p)->size & ~SIZE_BITS)))->size |= PREV_INUSE
#define clear_inuse(p)\
((mchunkptr)(((char*)(p)) + ((p)->size & ~SIZE_BITS)))->size &= ~(PREV_INUSE)
```

上面的这一组宏用于 check/set/clear 当前 chunk 使用标志位，有当前 chunk 的使用标志位存储在下一个 chunk 的 size 的第 0 bit（P 状态位），所以首先要获得下一个 chunk 的地址，然后 check/set/clear 下一个 chunk 的 size 域的第 0 bit。

```
/* check/set/clear inuse bits in known places */
#define inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size & PREV_INUSE)
#define set_inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size |= PREV_INUSE)
#define clear_inuse_bit_at_offset(p, s)\
 (((mchunkptr)(((char*)(p)) + (s)))->size &= ~(PREV_INUSE))
```

上面的三个宏用于 check/set/clear 指定 chunk 的 size 域中的使用标志位。

```
/* Set size at head, without disturbing its use bit */
#define set_head_size(p, s)  ((p)->size = (((p)->size & SIZE_BITS) | (s)))
/* Set size/use field */
#define set_head(p, s)       ((p)->size = (s))
/* Set size at footer (only when chunk is not in use) */
#define set_foot(p, s)       (((mchunkptr)((char*)(p) + (s)))->prev_size = (s))
```

宏 set_head_size(p, s)用于设置当前 chunk p 的 size 域并保留 size 域的控制信息。宏 set_head(p, s) 用于设置当前 chunk p 的 size 域并忽略已有的 size 域控制信息。宏 set_foot(p, s)用于设置当前 chunk p 的下一个 chunk 的 prev_size 为 s，s 为当前 chunk 的 size，只有当 chunk p 为空闲时才能使用这个宏，当前 chunk 的 foot 的内存空间存在于下一个 chunk，即下一个 chunk 的 prev_size。

5.2 分箱式内存管理

对于空闲的 chunk，ptmalloc 采用分箱式内存管理方式，根据空闲 chunk 的大小和处于的状态将其放在四个不同的 bin 中，这四个空闲 chunk 的容器包括 fast bins，unsorted bin，small bins 和 large bins。Fast bins 是小内存块的高速缓存，当一些大小小于 64 字节的 chunk 被回收时，首先会放入 fast bins 中，在分配小内存时，首先会查看 fast bins 中是否有合适的内存块，如果存在，则直接返回 fast bins 中的内存块，以加快分配速度。Usorted bin 只有一个，回收的 chunk 块必须先放到 unsorted bin 中，分配内存时会查看 unsorted bin 中是否有合适的 chunk，如果找到满足条件的 chunk，则直接返回给用户，否则将 unsorted bin 的所有 chunk 放入 small bins 或是 large bins 中。Small bins 用于存放固定大小的 chunk，共 64 个 bin，最小的 chunk 大小为 16 字节或 32 字节，每个 bin 的大小相差 8 字节或是 16 字节，当分配小内存块时，采用精确匹配的方式从 small bins 中查找合适的 chunk。Large bins 用于存储大于等于 512B 或 1024B 的空闲 chunk，这些 chunk 使用双向链表的形式按大小顺序排序，分配内存时按最近匹配方式从 large bins 中分配 chunk。

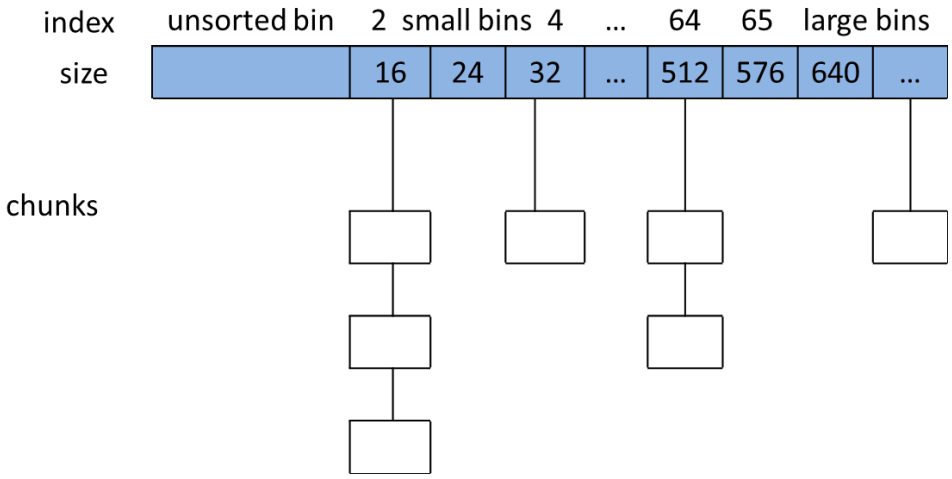
5.2.1 Small bins

ptmalloc 使用 small bins 管理空闲小 chunk，每个 small bin 中的 chunk 的大小与 bin 的 index 有如下关系：

Chunk_size=2 * SIZE_SZ * index

在 SIZE_SZ 为 4B 的平台上，small bins 中的 chunk 大小是以 8B 为公差的等差数列，最大的 chunk 大小为 504B，最小的 chunk 大小为 16B，所以实际共 62 个 bin。分别为 16B、24B、32B，……，504B。在 SIZE_SZ 为 8B 的平台上，small bins 中的 chunk 大小是以 16B 为公差的等差数列，最大的 chunk 大小为 1008B，最小的 chunk 大小为 32B，所以实际共 62 个 bin。分别为 32B、48B、64B，……，1008B。

ptmalloc 维护了 62 个双向环形链表（每个链表都具有链表头节点，加头节点的最大作用就是便于对链表内节点的统一处理，即简化编程），每一个链表内的各空闲 chunk 的大小一致，因此当应用程序需要分配某个字节大小的内存空间时直接在对应的链表内取就可以了，这样既可以很好的满足应用程序的内存空间申请请求而又不会出现太多的内存碎片。我们可以用如下图来表示在 SIZE_SZ 为 4B 的平台上 ptmalloc 对 512B 字节以下的空闲 chunk 组织方式（所谓的分箱机制）。



5.2.2 Large bins

在 SIZE_SZ 为 4B 的平台上，大于等于 512B 的空闲 chunk，或者，在 SIZE_SZ 为 8B 的平台上，大小大于等于 1024B 的空闲 chunk，由 sorted bins 管理。Large bins 一共包括 63 个 bin，每个 bin 中的 chunk 大小不是一个固定公差的等差数列，而是分成 6 组 bin，每组 bin 是一个固定公差的等差数列，每组的 bin 数量依次为 32、16、8、4、2、1，公差依次为 64B、512B、4096B、32768B、262144B 等。

以 SIZE_SZ 为 4B 的平台为例，第一个 large bin 的起始 chunk 大小为 512B，共 32 个 bin，公差为 64B，等差数列满足如下关系：

Chunk_size=512 + 64 * index

第二个 large bin 的起始 chunk 大小为第一组 bin 的结束 chunk 大小，满足如下关系：

Chunk_size=512 + 64 * 32 + 512 * index

同理，我们可计算出每个 bin 的起始 chunk 大小和结束 chunk 大小。这些 bin 都是很有规律的，其实 small bins 也是满足类似规律，small bins 可以看着是公差为 8 的等差数列，一共有 64 个 bin（第 0 和 1bin 不存在），所以我们可以将 small bins 和 large bins 存放在同一个包含 128 个 chunk 的数组上，数组的前一部分位 small bins，后一部分为 large bins，每个 bin 的 index 为 chunk 数组的下标，于是，我们可以根据数组下标计算出该 bin 的 chunk 大小（small bins）或是 chunk 大小范围（large bins），也可以根据需要分配内存块大小计算出所需 chunk 所属 bin 的 index，ptmalloc 使用了一组宏巧妙的实现了这种计算。

```
#define NBINS             128
#define NSMALLBINS         64
#define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
#define MIN_LARGE_SIZE    (NSMALLBINS * SMALLBIN_WIDTH)

#define in_smallbin_range(sz)  \
  ((unsigned long)(sz) < (unsigned long)MIN_LARGE_SIZE)

#define smallbin_index(sz) \
  (SMALLBIN_WIDTH == 16 ? (((unsigned)(sz)) >> 4) : (((unsigned)(sz)) >> 3))

#define largebin_index_32(sz)                                                \
(((((unsigned long)(sz)) >>  6) <= 38)?  56 + (((unsigned long)(sz)) >>  6): \
 ((((unsigned long)(sz)) >>  9) <= 20)?  91 + (((unsigned long)(sz)) >>  9): \
 ((((unsigned long)(sz)) >> 12) <= 10)? 110 + (((unsigned long)(sz)) >> 12): \
 ((((unsigned long)(sz)) >> 15) <=  4)? 119 + (((unsigned long)(sz)) >> 15): \
 ((((unsigned long)(sz)) >> 18) <=  2)? 124 + (((unsigned long)(sz)) >> 18): \
                                        126)

// XXX It remains to be seen whether it is good to keep the widths of
// XXX the buckets the same or whether it should be scaled by a factor
// XXX of two as well.
#define largebin_index_64(sz)                                                \
(((((unsigned long)(sz)) >>  6) <= 48)?  48 + (((unsigned long)(sz)) >>  6): \
```

```
  ((((unsigned long)(sz)) >>  9) <= 20)?  91 + (((unsigned long)(sz)) >>  9): \
  ((((unsigned long)(sz)) >> 12) <= 10)? 110 + (((unsigned long)(sz)) >> 12): \
  ((((unsigned long)(sz)) >> 15) <=  4)? 119 + (((unsigned long)(sz)) >> 15): \
  ((((unsigned long)(sz)) >> 18) <=  2)? 124 + (((unsigned long)(sz)) >> 18): \
                                         126)

#define largebin_index(sz) \
  (SIZE_SZ == 8 ? largebin_index_64 (sz) : largebin_index_32 (sz))

#define bin_index(sz) \
 ((in_smallbin_range(sz)) ? smallbin_index(sz) : largebin_index(sz))
```

宏 bin_index(sz)根据所需内存大小计算出所需 bin 的 index，如果所需内存大小属于 small bins 的大小范围，调用 smallbin_index(sz)，否则调用 largebin_index(sz))。smallbin_index(sz)的计算相当简单，如果 SIZE_SZ 为 4B，则将 sz 除以 8，如果 SIZE_SZ 为 8B，则将 sz 除以 16，也就是除以 small bins 中等差数列的公差。largebin_index(sz)的计算相对复杂一些，可以用如下的表格直观的显示 chunk 的大小范围与 bin index 的关系。以 SIZE_SZ 为 4B 的平台为例，chunk 大小与 bin index 的对应关系如下表所示：

开始(字节)	结束（字节）	Bin index
0	7	不存在
8	15	不存在
16	23	2
24	31	3
32	39	4
40	47	5
48	55	6
56	63	7
64	71	8
72	79	9
80	87	10
88	95	11
96	103	12
104	111	13
112	119	14
120	127	15
128	135	16
136	143	17
144	151	18
152	159	19
160	167	20
168	175	21
176	183	22
184	191	23
192	199	24

200	207	25
208	215	26
216	223	27
224	231	28
232	239	29
240	247	30
248	255	31
256	263	32
264	271	33
272	279	34
280	287	35
288	295	36
296	303	37
304	311	38
312	319	39
320	327	40
328	335	41
336	343	42
344	351	43
352	359	44
360	367	45
368	375	46
376	383	47
384	391	48
392	399	49
400	407	50
408	415	51
416	423	52
424	431	53
432	439	54
440	447	55
448	455	56
456	463	57
464	471	58
472	479	59
480	487	60
488	495	61
496	503	62
504	511	63
512	575	64
576	639	65
640	703	66
704	767	67

768	831	68
832	895	69
896	959	70
960	1023	71
1024	1087	72
1088	1151	73
1152	1215	74
1216	1279	75
1280	1343	76
1344	1407	77
1408	1471	78
1472	1535	79
1536	1599	80
1600	1663	81
1664	1727	82
1728	1791	83
1792	1855	84
1856	1919	85
1920	1983	86
1984	2047	87
2048	2111	88
2112	2175	89
2176	2239	90
2240	2303	91
2304	2367	92
2368	2431	93
2432	2495	94
2496	2559	95
2560	3071	96
3072	3583	97
3584	4095	98
4096	4607	99
4608	5119	100
5120	5631	101
5632	6143	102
6144	6655	103
6656	7167	104
7168	7679	105
7680	8191	106
8192	8703	107
8704	9215	108
9216	9727	109
9728	10239	110

10240	10751	111
10752	14847	112
14848	18943	113
18944	23039	114
23040	27135	115
27136	31231	116
31232	35327	117
35328	39423	118
39424	43519	119
43520	76287	120
76288	109055	121
109056	141823	122
141824	174591	123
174592	436735	124
436736	698879	125
698880	2^32 或 2^64	126

注意：上表是 chunk 大小与 bin index 的对应关系，如果对于用户要分配的内存大小 size，必须先使用 checked_request2size(req, sz)计算出 chunk 的大小，再使用 bin_index(sz)计算出 chunk 所属的 bin index。

对于 SIZE_SZ 为 4B 的平台，bin[0]和 bin[1]是不存在的，因为最小的 chunk 为 16B，small bins 一共 62 个，large bins 一共 63 个，加起来一共 125 个 bin。而 NBINS 定义为 128，其实 bin[0]和 bin[127]都不存在，bin[1]为 unsorted bin 的 chunk 链表头。

```
typedef struct malloc_chunk* mbinptr;

/* addressing -- note that bin_at(0) does not exist */
#define bin_at(m, i) \
  (mbinptr) (((char *) &((m)->bins[((i) - 1) * 2]))                    \
             - offsetof (struct malloc_chunk, fd))

/* analog of ++bin */
#define next_bin(b)  ((mbinptr)((char*)(b) + (sizeof(mchunkptr)<<1)))

/* Reminders about list directionality within bins */
#define first(b)     ((b)->fd)
#define last(b)      ((b)->bk)

/* Take a chunk off a bin list */
#define unlink(P, BK, FD) {                                            \
  FD = P->fd;                                                          \
  BK = P->bk;                                                          \
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                \
    malloc_printerr (check_action, "corrupted double-linked list", P); \
  else {                                                               \
    FD->bk = BK;                                                       \
```

```
    BK->fd = FD;                                                          \
    if (!in_smallbin_range (P->size)                                      \
        && __builtin_expect (P->fd_nextsize != NULL, 0)) {                \
      assert (P->fd_nextsize->bk_nextsize == P);                          \
      assert (P->bk_nextsize->fd_nextsize == P);                          \
      if (FD->fd_nextsize == NULL) {                                      \
        if (P->fd_nextsize == P)                                          \
          FD->fd_nextsize = FD->bk_nextsize = FD;                         \
        else {                                                            \
          FD->fd_nextsize = P->fd_nextsize;                               \
          FD->bk_nextsize = P->bk_nextsize;                               \
          P->fd_nextsize->bk_nextsize = FD;                               \
          P->bk_nextsize->fd_nextsize = FD;                               \
        }                                                                 \
      } else {                                                            \
        P->fd_nextsize->bk_nextsize = P->bk_nextsize;                     \
        P->bk_nextsize->fd_nextsize = P->fd_nextsize;                     \
      }                                                                   \
    }                                                                     \
  }                                                                       \
}
```

宏 bin_at(m, i)通过 bin index 获得 bin 的链表头，chunk 中的 fb 和 bk 用于将空闲 chunk 链入链表中，而对于每个 bin 的链表头，只需要这两个域就可以了，prev_size 和 size 对链表都来说都没有意义，浪费空间，ptmalloc 为了节约这点内存空间，增大 cpu 高速缓存的命中率，在 bins 数组中只为每个 bin 预留了两个指针的内存空间用于存放 bin 的链表头的 fb 和 bk 指针。

从 bin_at(m, i)的定义可以看出，bin[0]不存在，以 SIZE_SZ 为 4B 的平台为例，bin[1]的前 4B 存储的是指针 fb，后 4B 存储的是指针 bk，而 bin_at 返回的是 malloc_chunk 的指针，由于 fb 在 malloc_chunk 的偏移地址为 offsetof (struct malloc_chunk, fd))=8，所以用 fb 的地址减去 8 就得到 malloc_chunk 的地址。但切记，对 bin 的链表头的 chunk，一定不能修改 prev_size 和 size 域，这两个域是与其他 bin 的链表头的 fb 和 bk 内存复用的。

宏 next_bin(b)用于获得下一个 bin 的地址，根据前面的分析，我们知道只需要将当前 bin 的地址向后移动两个指针的长度就得到下一个 bin 的链表头地址。

每个 bin 使用双向循环链表管理空闲 chunk，bin 的链表头的指针 fb 指向第一个可用的 chunk，指针 bk 指向最后一个可用的 chunk。宏 first(b)用于获得 bin 的第一个可用 chunk，宏 last(b)用于获得 bin 的最后一个可用的 chunk，这两个宏便于遍历 bin，而跳过 bin 的链表头。

宏 unlink(P, BK, FD)用于将 chunk 从所在的空闲链表中取出来，注意 large bins 中的空闲 chunk 可能处于两个双向循环链表中，unlink 时需要从两个链表中都删除。

5.2.3 Unsorted bin

Unsorted bin 可以看作是 small bins 和 large bins 的 cache，只有一个 unsorted bin，以双向链表管理空闲 chunk，空闲 chunk 不排序，所有的 chunk 在回收时都要先放到 unsorted bin

中，分配时，如果在 unsorted bin 中没有合适的 chunk，就会把 unsorted bin 中的所有 chunk 分别加入到所属的 bin 中，然后再在 bin 中分配合适的 chunk。Bins 数组中的元素 bin[1]用于存储 unsorted bin 的 chunk 链表头。

```
/*
  Unsorted chunks

    All remainders from chunk splits, as well as all returned chunks,
    are first placed in the "unsorted" bin. They are then placed
    in regular bins after malloc gives them ONE chance to be used before
    binning. So, basically, the unsorted_chunks list acts as a queue,
    with chunks being placed on it in free (and malloc_consolidate),
    and taken off (to be either used or placed in bins) in malloc.

    The NON_MAIN_ARENA flag is never set for unsorted chunks, so it
    does not have to be taken into account in size comparisons.
*/

/* The otherwise unindexable 1-bin is used to hold unsorted chunks. */
#define unsorted_chunks(M)          (bin_at(M, 1))

/*
  Top

    The top-most available chunk (i.e., the one bordering the end of
    available memory) is treated specially. It is never included in
    any bin, is used only if no other chunk is available, and is
    released back to the system if it is very large (see
    M_TRIM_THRESHOLD). Because top initially
    points to its own bin with initial zero size, thus forcing
    extension on the first malloc request, we avoid having any special
    code in malloc to check whether it even exists yet. But we still
    need to do so when getting memory from system, so we make
    initial_top treat the bin as a legal but unusable chunk during the
    interval between initialization and the first call to
    sYSMALLOc. (This is somewhat delicate, since it relies on
    the 2 preceding words to be zero during this interval as well.)
*/

/* Conveniently, the unsorted bin can be used as dummy top on first call */
#define initial_top(M)              (unsorted_chunks(M))
```

上面的宏的定义比较明显，把 bin[1]设置为 unsorted bin 的 chunk 链表头，对 top chunk 的初始化，也暂时把 top chunk 初始化为 unsort chunk，仅仅是初始化一个值而已，这个 chunk 的内容肯定不能用于 top chunk 来分配内存，主要原因是 top chunk 不属于任何 bin，但 ptmalloc 中的一些 check 代码，可能需要 top chunk 属于一个合法的 bin。

5.2.4 Fast bins

Fast bins 主要是用于提高小内存的分配效率，默认情况下，对于 SIZE_SZ 为 4B 的平台，小于 64B 的 chunk 分配请求，对于 SIZE_SZ 为 8B 的平台，小于 128B 的 chunk 分配请求，首先会查找 fast bins 中是否有所需大小的 chunk 存在（精确匹配），如果存在，就直接返回。

Fast bins 可以看着是 small bins 的一小部分 cache，默认情况下，fast bins 只 cache 了 small bins 的前 7 个大小的空闲 chunk，也就是说，对于 SIZE_SZ 为 4B 的平台，fast bins 有 7 个 chunk 空闲链表（bin），每个 bin 的 chunk 大小依次为 16B，24B，32B，40B，48B，56B，64B；对于 SIZE_SZ 为 8B 的平台，fast bins 有 7 个 chunk 空闲链表（bin），每个 bin 的 chunk 大小依次为 32B，48B，64B，80B，96B，112B，128B。以 32 为系统为例，分配的内存大小与 chunk 大小和 fast bins 的对应关系如下表所示：

Fast bin #	Holds chunk sizes	Read chunk size
0	00 – 12	16
1	13 – 20	24
2	21 – 28	32
3	29 – 36	40
4	37 – 44	48
5	45 – 52	56
6	53 – 60	64
7	61 – 68	72
8	69 – 76	80
9	77 - 80	88

Fast bins 可以看着是 LIFO 的栈，使用单向链表实现。

```
/*
  Fastbins

    An array of lists holding recently freed small chunks.  Fastbins
    are not doubly linked.  It is faster to single-link them, and
    since chunks are never removed from the middles of these lists,
    double linking is not necessary. Also, unlike regular bins, they
    are not even processed in FIFO order (they use faster LIFO) since
    ordering doesn't much matter in the transient contexts in which
    fastbins are normally used.

    Chunks in fastbins keep their inuse bit set, so they cannot
    be consolidated with other free chunks. malloc_consolidate
```

```
    releases all chunks in fastbins and consolidates them with
    other free chunks.
*/
typedef struct malloc_chunk* mfastbinptr;
#define fastbin(ar_ptr, idx) ((ar_ptr)->fastbinsY[idx])
```

根据 fast bin 的 index，获得 fast bin 的地址。Fast bins 的数字定义在 malloc_state 中，5.3 节会做详细介绍。

```
/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz) \
  ((((unsigned int)(sz)) >> (SIZE_SZ == 8 ? 4 : 3)) - 2)
```

宏 fastbin_index(sz)用于获得 fast bin 在 fast bins 数组中的 index，由于 bin[0]和 bin[1]中的 chunk 不存在，所以需要减 2，对于 SIZE_SZ 为 4B 的平台，将 sz 除以 8 减 2 得到 fast bin index，对于 SIZE_SZ 为 8B 的平台，将 sz 除以 16 减去 2 得到 fast bin index。

```
/* The maximum fastbin request size we support */
#define MAX_FAST_SIZE     (80 * SIZE_SZ / 4)
#define NFASTBINS  (fastbin_index(request2size(MAX_FAST_SIZE))+1)

/*
  FASTBIN_CONSOLIDATION_THRESHOLD is the size of a chunk in free()
  that triggers automatic consolidation of possibly-surrounding
  fastbin chunks. This is a heuristic, so the exact value should not
  matter too much. It is defined at half the default trim threshold as a
  compromise heuristic to only attempt consolidation if it is likely
  to lead to trimming. However, it is not dynamically tunable, since
  consolidation reduces fragmentation surrounding large chunks even
  if trimming is not used.
*/
#define FASTBIN_CONSOLIDATION_THRESHOLD  (65536UL)
```

根据 SIZE_SZ 的不同大小，定义 MAX_FAST_SIZE 为 80B 或是 160B，fast bins 数组的大小 NFASTBINS 为 10，FASTBIN_CONSOLIDATION_THRESHOLD 为 64k，当每次释放的 chunk 与该 chunk 相邻的空闲 chunk 合并后的大小大于 64k 时，就认为内存碎片可能比较多了，就需要把 fast bins 中的所有 chunk 都进行合并，以减少内存碎片对系统的影响。

```
#ifndef DEFAULT_MXFAST
#define DEFAULT_MXFAST     (64 * SIZE_SZ / 4)
#endif
/*
   Set value of max_fast.
   Use impossibly small value if 0.
   Precondition: there are no existing fastbin chunks.
   Setting the value clears fastchunk bit but preserves noncontiguous bit.
*/
```

```
#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
```

上面的宏 DEFAULT_MXFAST 定义了默认的 fast bins 中最大的 chunk 大小，对于 SIZE_SZ 为 4B 的平台，最大 chunk 为 64B，对于 SIZE_SZ 为 8B 的平台，最大 chunk 为 128B。ptmalloc 默认情况下调用 set_max_fast(s)将全局变量 global_max_fast 设置为 DEFAULT_MXFAST，也就是设置 fast bins 中 chunk 的最大值，get_max_fast()用于获得这个全局变量 global_max_fast 的值。

5.3 核心结构体分析

每个分配区是 struct malloc_state 的一个实例，ptmalloc 使用 malloc_state 来管理分配区，而参数管理使用 struct malloc_par，全局拥有一个唯一的 malloc_par 实例。

5.3.1 malloc_state

struct malloc_state 的定义如下：

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

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

#if THREAD_STATS
  /* Statistics for locking.  Only used if THREAD_STATS is defined.  */
  long stat_lock_direct, stat_lock_loop, stat_lock_wait;
#endif

  /* 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;

#ifdef PER_THREAD
  /* Linked list for free arenas.  */
  struct malloc_state *next_free;
#endif

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

Mutex 用于串行化访问分配区，当有多个线程访问同一个分配区时，第一个获得这个 mutex 的线程将使用该分配区分配内存，分配完成后，释放该分配区的 mutex，以便其它线程使用该分配区。

Flags 记录了分配区的一些标志，bit0 用于标识分配区是否包含至少一个 fast bin chunk，bit1 用于标识分配区是否能返回连续的虚拟地址空间。

```
/*
  FASTCHUNKS_BIT held in max_fast indicates that there are probably
  some fastbin chunks. It is set true on entering a chunk into any
  fastbin, and cleared only in malloc_consolidate.

  The truth value is inverted so that have_fastchunks will be true
  upon startup (since statics are zero-filled), simplifying
  initialization checks.
*/
#define FASTCHUNKS_BIT        (1U)
#define have_fastchunks(M)     (((M)->flags &  FASTCHUNKS_BIT) == 0)
#ifdef ATOMIC_FASTBINS
#define clear_fastchunks(M)    catomic_or (&(M)->flags, FASTCHUNKS_BIT)
#define set_fastchunks(M)      catomic_and (&(M)->flags, ~FASTCHUNKS_BIT)
#else
#define clear_fastchunks(M)    ((M)->flags |=  FASTCHUNKS_BIT)
#define set_fastchunks(M)      ((M)->flags &= ~FASTCHUNKS_BIT)
#endif
```

上面的宏用于设置或是置位 flags 中 fast chunk 的标志位 bit0，如果 bit0 为 0，表示分配区中有 fast chunk，如果为 1 表示没有 fast chunk，初始化完成后的 malloc_state 实例中，flags 值为 0，表示该分配区中有 fast chunk，但实际上没有，试图从 fast bins 中分配 chunk 都会返回 NULL，在第一次调用函数 malloc_consolidate()对 fast bins 进行 chunk 合并时，如果 max_fast 大于 0，会调用 clear_fastchunks 宏，标志该分配区中已经没有 fast chunk，因为函数 malloc_consolidate()会合并所有的 fast bins 中的 chunk。clear_fastchunks 宏只会在函数 malloc_consolidate()中调用。当有 fast chunk 加入 fast bins 时，就是调用 set_fastchunks 宏标

识分配区的 fast bins 中存在 fast chunk。

```
/*
  NONCONTIGUOUS_BIT indicates that MORECORE does not return contiguous
  regions.  Otherwise, contiguity is exploited in merging together,
  when possible, results from consecutive MORECORE calls.

  The initial value comes from MORECORE_CONTIGUOUS, but is
  changed dynamically if mmap is ever used as an sbrk substitute.
*/
#define NONCONTIGUOUS_BIT      (2U)
#define contiguous(M)           (((M)->flags &  NONCONTIGUOUS_BIT) == 0)
#define noncontiguous(M)        (((M)->flags &  NONCONTIGUOUS_BIT) != 0)
#define set_noncontiguous(M)    ((M)->flags |=  NONCONTIGUOUS_BIT)
#define set_contiguous(M)       ((M)->flags &= ~NONCONTIGUOUS_BIT)
```

Flags 的 bit1 如果为 0，表示 MORCORE 返回连续虚拟地址空间，bit1 为 1，表示 MORCORE 返回非连续虚拟地址空间，对于主分配区，MORECORE 其实为 sbr()，默认返回连续虚拟地址空间，对于非主分配区，使用 mmap()分配大块虚拟内存，然后进行切分来模拟主分配区的行为，而默认情况下 mmap 映射区域是不保证虚拟地址空间连续的，所以非住分配区默认分配非连续虚拟地址空间。

Malloc_state 中声明了几个对锁的统计变量，默认没有定义 THREAD_STATS，所以不会对锁的争用情况做统计。

fastbinsY 拥有 10（NFASTBINS）个元素的数组，用于存放每个 fast chunk 链表头指针，所以 fast bins 最多包含 10 个 fast chunk 的单向链表。

top 是一个 chunk 指针，指向分配区的 top chunk。

last_remainder 是一个 chunk 指针，分配区上次分配 small chunk 时，从一个 chunk 中分裂出一个 small chunk 返回给用户，分裂后的剩余部分形成一个 chunk，last_remainder 就是指向的这个 chunk。

bins 用于存储 unstored bin，small bins 和 large bins 的 chunk 链表头，small bins 一共 62 个，large bins 一共 63 个，加起来一共 125 个 bin。而 NBINS 定义为 128，其实 bin[0]和 bin[127]都不存在，bin[1]为 unsorted bin 的 chunk 链表头，所以实际只有 126bins。Bins 数组能存放了 254（NBINS*2 – 2）个 mchunkptr 指针，而我们实现需要存储 chunk 的实例，一般情况下，chunk 实例的大小为 6 个 mchunkptr 大小，这 254 个指针的大小怎么能存下 126 个 chunk 呢？这里使用了一个技巧，如果按照我们的常规想法，也许会申请 126 个 malloc_chunk 结构体指针元素的数组，然后再给链表申请一个头节点（即 126 个），再让每个指针元素正确指向而形成 126 个具有头节点的链表。事实上，对于 malloc_chunk 类型的链表“头节点”，其内的 prev_size 和 size 字段是没有任何实际作用的，fd_nextsize 和 bk_nextsize 字段只有 large bins 中的空闲 chunk 才会用到，而对于 large bins 的空闲 chunk 链表头不需要这两个字段，因此这四个字段所占空间如果不合理使用的话那就是白白的浪费。我们再来看一看 128 个 malloc_chunk 结构体指针元素的数组占了多少内存空间呢？假设 SIZE_SZ 的大小为 8B，则指针的大小也为 8B，结果为 126*2*8=2016 字节。而 126 个 malloc_chunk 类型的链表“头节点”需要多少内存呢？126*6*8=6048，真的是 6048B 么？不是，刚才不是说了，prev_size，size，fd_nextsize 和 bk_nextsize 这四个字段是没有任何实际作用的，因此完全可以被重用（覆盖），

因此实际需要内存为 126*2*8=2016。Bins 指针数组的大小为，（128*2-2）*8=2032,2032 大于 2016（事实上最后 16 个字节都被浪费掉了），那么这 254 个 malloc_chunk 结构体指针元素数组所占内存空间就可以存储这 126 个头节点了。

binmap 字段是一个 int 数组，ptmalloc 用一个 bit 来标识该 bit 对应的 bin 中是否包含空闲 chunk。

```
/*
  Binmap

    To help compensate for the large number of bins, a one-level index
    structure is used for bin-by-bin searching.  `binmap' is a
    bitvector recording whether bins are definitely empty so they can
    be skipped over during during traversals.  The bits are NOT always
    cleared as soon as bins are empty, but instead only
    when they are noticed to be empty during traversal in malloc.
*/

/* Conservatively use 32 bits per map word, even if on 64bit system */
#define BINMAPSHIFT      5
#define BITSPERMAP       (1U << BINMAPSHIFT)
#define BINMAPSIZE       (NBINS / BITSPERMAP)

#define idx2block(i)     ((i) >> BINMAPSHIFT)
#define idx2bit(i)       ((1U << ((i) & ((1U << BINMAPSHIFT)-1))))

#define mark_bin(m,i)    ((m)->binmap[idx2block(i)] |=  idx2bit(i))
#define unmark_bin(m,i)  ((m)->binmap[idx2block(i)] &= ~(idx2bit(i)))
#define get_binmap(m,i)  ((m)->binmap[idx2block(i)] &   idx2bit(i))
```

binmap 一共 128bit，16 字节，4 个 int 大小，binmap 按 int 分成 4 个 block，每个 block 有 32 个 bit，根据 bin indx 可以使用宏 idx2block 计算出该 bin 在 binmap 对应的 bit 属于哪个 block。idx2bit 宏取第 i 位为 1，其它位都为 0 的掩码，举个例子：idx2bit(3) 为 “0000 1000”（只显示 8 位）。mark_bin 设置第 i 个 bin 在 binmap 中对应的 bit 位为 1；unmark_bin 设置第 i 个 bin 在 binmap 中对应的 bit 位为 0；get_binmap 获取第 i 个 bin 在 binmap 中对应的 bit。

next 字段用于将分配区以单向链表链接起来。

next_free 字段空闲的分配区链接在单向链表中，只有在定义了 PER_THREAD 的情况下才定义该字段。

system_mem 字段记录了当前分配区已经分配的内存大小。

max_system_mem 记录了当前分配区最大能分配的内存大小。

5.3.2 Malloc_par

Struct malloc_par 的定义如下：

```
struct malloc_par {
  /* Tunable parameters */
  unsigned long    trim_threshold;
```

```
  INTERNAL_SIZE_T  top_pad;
  INTERNAL_SIZE_T  mmap_threshold;
#ifdef PER_THREAD
  INTERNAL_SIZE_T  arena_test;
  INTERNAL_SIZE_T  arena_max;
#endif

  /* Memory map support */
  int              n_mmaps;
  int              n_mmaps_max;
  int              max_n_mmaps;
  /* the mmap_threshold is dynamic, until the user sets
     it manually, at which point we need to disable any
     dynamic behavior. */
  int              no_dyn_threshold;

  /* Cache malloc_getpagesize */
  unsigned int     pagesize;

  /* Statistics */
  INTERNAL_SIZE_T  mmapped_mem;
  INTERNAL_SIZE_T  max_mmapped_mem;
  INTERNAL_SIZE_T  max_total_mem; /* only kept for NO_THREADS */

  /* First address handed out by MORECORE/sbrk.  */
  char*            sbrk_base;
};
```

trim_threshold 字段表示收缩阈值，默认为 128KB，当每个分配区的 top chunk 大小大于这个阈值时，在一定的条件下，调用 free 时会收缩内存，减小 top chunk 的大小。由于 mmap 分配阈值的动态调整，在 free 时可能将收缩阈值修改为 mmap 分配阈值的 2 倍，在 64 位系统上，mmap 分配阈值最大值为 32MB，所以收缩阈值的最大值为 64MB，在 32 位系统上，mmap 分配阈值最大值为 512KB，所以收缩阈值的最大值为 1MB。收缩阈值可以通过函数 mallopt()进行设置。

top_pad 字段表示在分配内存时是否添加额外的 pad，默认该字段为 0。

mmap_threshold 字段表示 mmap 分配阈值，默认值为 128KB，在 32 位系统上最大值为 512KB，64 位系统上的最大值为 32MB，由于默认开启 mmap 分配阈值动态调整，该字段的值会动态修改，但不会超过最大值。

arena_test 和 arena_max 用于 PER_THREAD 优化，在 32 位系统上 arena_test 默认值为 2，64 位系统上的默认值为 8，当每个进程的分配区数量小于等于 arena_test 时，不会重用已有的分配区。为了限制分配区的总数，用 arena_max 来保存分配区的最大数量，当系统中的分配区数量达到 arena_max，就不会再创建新的分配区，只会重用已有的分配区。这两个字段都可以使用 mallopt()函数设置。

n_mmaps 字段表示当前进程使用 mmap()函数分配的内存块的个数。

n_mmaps_max 字段表示进程使用 mmap()函数分配的内存块的最大数量，默认值为

65536，可以使用 mallopt()函数修改。

max_n_mmaps 字段表示当前进程使用 mmap()函数分配的内存块的数量的最大值，有关系 n_mmaps <= max_n_mmaps 成立。这个字段是由于 mstats()函数输出统计需要这个字段。

no_dyn_threshold 字段表示是否开启 mmap 分配阈值动态调整机制，默认值为 0，也就是默认开启 mmap 分配阈值动态调整机制。

pagesize 字段表示系统的页大小，默认为 4KB。

mmapped_mem 和 max_mmapped_mem 都用于统计 mmap 分配的内存大小，一般情况下两个字段的值相等，max_mmapped_mem 用于 mstats()函数。

max_total_mem 字段在单线程情况下用于统计进程分配的内存总数。

sbrk_base 字段表示堆的起始地址。

5.3.3 分配区的初始化

Ptmalloc 定义了如下几个全局变量：

```
/* There are several instances of this struct ("arenas") in this
   malloc.  If you are adapting this malloc in a way that does NOT use
   a static or mmapped malloc_state, you MUST explicitly zero-fill it
   before using. This malloc relies on the property that malloc_state
   is initialized to all zeroes (as is true of C statics).  */
static struct malloc_state main_arena;
/* There is only one instance of the malloc parameters.  */
static struct malloc_par mp_;
/* Maximum size of memory handled in fastbins.  */
static INTERNAL_SIZE_T global_max_fast;
```

main_arena 表示主分配区，任何进程有且仅有一个全局的主分配区，mp_是全局唯一的一个 malloc_par 实例，用于管理参数和统计信息，global_max_fast 全局变量表示 fast bins 中最大的 chunk 大小。

分配区 main_arena 初始化函数：

```
/*
  Initialize a malloc_state struct.

  This is called only from within malloc_consolidate, which needs
  be called in the same contexts anyway.  It is never called directly
  outside of malloc_consolidate because some optimizing compilers try
  to inline it at all call points, which turns out not to be an
  optimization at all. (Inlining it in malloc_consolidate is fine though.)
*/
#if __STD_C
static void malloc_init_state(mstate av)
#else
static void malloc_init_state(av) mstate av;
#endif
{
```

```
  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);
  av->flags |= FASTCHUNKS_BIT;

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

分配区的初始化函数默认分配区的实例 av 是全局静态变量或是已经将 av 中的所有字段都清 0 了。初始化函数做的工作比较简单，首先遍历所有的 bins，初始化每个 bin 的空闲链表为空，即将 bin 的 fb 和 bk 都指向 bin 本身。由于 av 中所有字段默认为 0，即默认分配连续的虚拟地址空间，但只有主分配区才能分配连续的虚拟地址空间，所以对于非主分配区，需要设置为分配非连续虚拟地址空间。如果初始化的是主分配区，需要设置 fast bins 中最大 chunk 大小，由于主分配区只有一个，并且一定是最先初始化，这就保证了对全局变量 global_max_fast 只初始化了一次，只要该全局变量的值非 0，也就意味着主分配区初始化了。最后初始化 top chunk。

Ptmalloc 参数初始化

```
/* Set up basic state so that _int_malloc et al can work.  */
static void
ptmalloc_init_minimal (void)
{
#if DEFAULT_TOP_PAD != 0
  mp_.top_pad        = DEFAULT_TOP_PAD;
#endif
  mp_.n_mmaps_max    = DEFAULT_MMAP_MAX;
  mp_.mmap_threshold = DEFAULT_MMAP_THRESHOLD;
  mp_.trim_threshold = DEFAULT_TRIM_THRESHOLD;
  mp_.pagesize       = malloc_getpagesize;
#ifdef PER_THREAD
# define NARENAS_FROM_NCORES(n) ((n) * (sizeof(long) == 4 ? 2 : 8))
  mp_.arena_test     = NARENAS_FROM_NCORES (1);
  narenas = 1;
```

```
#endif
}
```

主要是将全局变量 mp_的字段初始化为默认值，值得一提的是，如果定义了编译选项 PER_THREAD，会根据系统 cpu 的个数设置 arena_test 的值，默认 32 位系统是双核，64 位系统为 8 核，arena_test 也就设置为相应的值。

5.4 配置选项

Ptmalloc 的配置选项不多，在 3.2.6 节已经做过概要描述，这里给出 mallopt()函数的实现：

```
#if __STD_C
int mALLOPt(int param_number, int value)
#else
int mALLOPt(param_number, value) int param_number; int value;
#endif
{
  mstate av = &main_arena;
  int res = 1;

  if(__malloc_initialized < 0)
    ptmalloc_init ();
  (void)mutex_lock(&av->mutex);
  /* Ensure initialization/consolidation */
  malloc_consolidate(av);

  switch(param_number) {
  case M_MXFAST:
    if (value >= 0 && value <= MAX_FAST_SIZE) {
      set_max_fast(value);
    }
    else
      res = 0;
    break;

  case M_TRIM_THRESHOLD:
    mp_.trim_threshold = value;
    mp_.no_dyn_threshold = 1;
    break;

  case M_TOP_PAD:
    mp_.top_pad = value;
    mp_.no_dyn_threshold = 1;
    break;

  case M_MMAP_THRESHOLD:
```

```
#if USE_ARENAS
    /* Forbid setting the threshold too high. */
    if((unsigned long)value > HEAP_MAX_SIZE/2)
      res = 0;
    else
#endif
      mp_.mmap_threshold = value;
      mp_.no_dyn_threshold = 1;
    break;

  case M_MMAP_MAX:
#if !HAVE_MMAP
    if (value != 0)
      res = 0;
    else
#endif
      mp_.n_mmaps_max = value;
      mp_.no_dyn_threshold = 1;
    break;

  case M_CHECK_ACTION:
    check_action = value;
    break;

  case M_PERTURB:
    perturb_byte = value;
    break;

#ifdef PER_THREAD
  case M_ARENA_TEST:
    if (value > 0)
      mp_.arena_test = value;
    break;

  case M_ARENA_MAX:
    if (value > 0)
      mp_.arena_max = value;
    break;
#endif
  }
  (void)mutex_unlock(&av->mutex);
  return res;
}
```

Mallopt()所设置的字段在 5.3.2 节中做过详细介绍，这里不再作详细分析，需要提一下

的是，在 mallopt()函数配置前，需要检查主分配区是否初始化了，如果没有初始化，调用 ptmalloc_init()函数初始化 ptmalloc，然后获得主分配区的锁，调用 malloc_consolidate()函数，malloc_consolidate()函数会判断主分配区是否已经初始化，如果没有，则初始化主分配区。同时我们也看到，mp_都没有锁，对 mp_中参数字段的修改，是通过主分配区的锁来同步的。

5.5 Ptmalloc 的初始化

Ptmalloc 的初始化发生在进程的第一个内存分配请求，当 ptmalloc 的初始化一般都在用户的第一次调用 malloc()或 remalloc()之前，因为操作系统和 Glibc 库为进程的初始化做了不少工作，在用户分配内存以前，Glibc 已经分配了多次内存。

在 ptmalloc 中 malloc()函数的实际接口函数为 public_mALLOc()，这个函数最开始会执行如下的一段代码：

```
  __malloc_ptr_t (*hook) (size_t, __const __malloc_ptr_t)
    = force_reg (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));
```

在定义了__malloc_hook()全局函数的情况下，只是执行__malloc_hook()函数，在进程初始化时__malloc_hook 指向的函数为 malloc_hook_ini()。

```
    __malloc_ptr_t weak_variable (*__malloc_hook)
         (size_t __size, const __malloc_ptr_t) = malloc_hook_ini;
```

malloc_hook_ini()函数定义在 hooks.c 中，实现代码如下：

```
static Void_t*
#if __STD_C
malloc_hook_ini(size_t sz, const __malloc_ptr_t caller)
#else
malloc_hook_ini(sz, caller)
     size_t sz; const __malloc_ptr_t caller;
#endif
{
  __malloc_hook = NULL;
  ptmalloc_init();
  return public_mALLOc(sz);
}
```

malloc_hook_ini()函数处理很简单，就是调用 ptmalloc 的初始化函数 ptmalloc_init()，然后再重新调用 pbulit_mALLOc()函数分配内存。Ptmalloc_init()函数在初始化 ptmalloc 完成后，将全局变量__malloc_initialized 设置为 1，当 pbulit_mALLOc()函数再次执行时，先执行 malloc_hook_ini()函数，malloc_hook_ini()函数调用 ptmalloc_init()，ptmalloc_init()函数首先判断__malloc_initialized 是否为 1，如果是，则退出 ptmalloc_init()，不再执行 ptmalloc 初始化。

5.5.1 Ptmalloc 未初始化时分配/释放内存

当 ptmalloc 的初始化函数 ptmalloc_init()还没有调用之前，Glibc 中可能需要分配内存，比如线程私有实例的初始化需要分配内存，为了解决这一问题，ptmalloc 封装了内部的分配

释放函数供在这种情况下使用。Ptmalloc 提供了三个函数，malloc_starter()，memalign_starter()，free_starter()，但没有提供 realloc_starter()函数。这几个函数的实现如下：

```
static Void_t*
#if __STD_C
malloc_starter(size_t sz, const Void_t *caller)
#else
malloc_starter(sz, caller) size_t sz; const Void_t *caller;
#endif
{
  Void_t* victim;

  victim = _int_malloc(&main_arena, sz);

  return victim ? BOUNDED_N(victim, sz) : 0;
}

static Void_t*
#if __STD_C
memalign_starter(size_t align, size_t sz, const Void_t *caller)
#else
memalign_starter(align, sz, caller) size_t align, sz; const Void_t *caller;
#endif
{
  Void_t* victim;

  victim = _int_memalign(&main_arena, align, sz);

  return victim ? BOUNDED_N(victim, sz) : 0;
}

static void
#if __STD_C
free_starter(Void_t* mem, const Void_t *caller)
#else
free_starter(mem, caller) Void_t* mem; const Void_t *caller;
#endif
{
  mchunkptr p;

  if(!mem) return;
  p = mem2chunk(mem);
#if HAVE_MMAP
  if (chunk_is_mmapped(p)) {
    munmap_chunk(p);
```

```
    return;
  }
#endif
#ifdef ATOMIC_FASTBINS
  _int_free(&main_arena, p, 1);
#else
  _int_free(&main_arena, p);
#endif
}
```

这个函数的实现都很简单，只是调用 ptmalloc 的内部实现函数，这里就不详细介绍内部函数的实现了，在 5.7 和 5.8 节会详细介绍这些内部函数的实现细节。

5.5.2 ptmalloc_init()函数

ptmalloc_init()函数比较长，将分段对这个函数做介绍。

```
static void
ptmalloc_init (void)
{
#if __STD_C
  const char* s;
#else
  char* s;
#endif
  int secure = 0;

  if(__malloc_initialized >= 0) return;
  __malloc_initialized = 0;
```

首先检查全局变量__malloc_initialized 是否大于等于 0，如果该值大于 0，表示 ptmalloc 已经初始化，如果改值为 0，表示 ptmalloc 正在初始化，全局变量__malloc_initialized 用来保证全局只初始化 ptmalloc 一次。

```
#ifdef _LIBC
# if defined SHARED && !USE___THREAD
  /* ptmalloc_init_minimal may already have been called via
     __libc_malloc_pthread_startup, above.  */
  if (mp_.pagesize == 0)
# endif
#endif
    ptmalloc_init_minimal();

#ifndef NO_THREADS
# if defined _LIBC
  /* We know __pthread_initialize_minimal has already been called,
     and that is enough.  */
```

```
#   define NO_STARTER
# endif
# ifndef NO_STARTER
  /* With some threads implementations, creating thread-specific data
     or initializing a mutex may call malloc() itself.  Provide a
     simple starter version (realloc() won't work). */
  save_malloc_hook = __malloc_hook;
  save_memalign_hook = __memalign_hook;
  save_free_hook = __free_hook;
  __malloc_hook = malloc_starter;
  __memalign_hook = memalign_starter;
  __free_hook = free_starter;
#  ifdef _LIBC
  /* Initialize the pthreads interface. */
  if (__pthread_initialize != NULL)
    __pthread_initialize();
#  endif /* !defined _LIBC */
# endif /* !defined NO_STARTER */
#endif /* !defined NO_THREADS */
```

为多线程版本的 ptmalloc 的 pthread 初始化做准备，保存当前的 hooks 函数，并把 ptmalloc 为初始化时所有使用的分配/释放函数赋给 hooks 函数，因为在线程初始化一些私有实例时，ptmalloc 还没有初始化，所以需要做特殊处理。从这些 hooks 函数可以看出，在 ptmalloc 未初始化时，不能使用 remalloc 函数。在相关的 hooks 函数赋值以后，执行 pthread_initilaize()初始化 pthread。

```
  mutex_init(&main_arena.mutex);
  main_arena.next = &main_arena;
```

初始化主分配区的 mutex，并将主分配区的 next 指针指向自身组成环形链表。

```
#if defined _LIBC && defined SHARED
  /* In case this libc copy is in a non-default namespace, never use brk.
     Likewise if dlopened from statically linked program.  */
  Dl_info di;
  struct link_map *l;

  if (_dl_open_hook != NULL
      || (_dl_addr (ptmalloc_init, &di, &l, NULL) != 0
          && l->l_ns != LM_ID_BASE))
    __morecore = __failing_morecore;
#endif
```

Ptmalloc 需要保证只有主分配区才能使用 sbrk()分配连续虚拟内存空间，如果有多个分配区使用 sbrk()就不能获得连续的虚拟地址空间，大多数情况下 Glibc 库都是以动态链接库的形式加载的，处于默认命名空间，多个进程共用 Glibc 库，Glibc 库代码段在内存中只有一份拷贝，数据段在每个用户进程都有一份拷贝。但如果 Glibc 库不在默认名字空间，或是用

户程序是静态编译的并调用了 dlopen 函数加载 Glibc 库中的 ptamalloc_init()，这种情况下的 ptmalloc 不允许使用 sbrk()分配内存，只需修改__morecore 函数指针指向__failing_morecore 就可以禁止使用 sbrk()了，__morecore 默认指向 sbrk()。

```
  mutex_init(&list_lock);
  tsd_key_create(&arena_key, NULL);
  tsd_setspecific(arena_key, (Void_t *)&main_arena);
  thread_atfork(ptmalloc_lock_all, ptmalloc_unlock_all, ptmalloc_unlock_all2);
```

初始化全局锁 list_lock，list_lock 主要用于同步分配区的单向循环链表。然后创建线程私有实例 arena_key，该私有实例保存的是分配区（arena）的 malloc_state 实例指针。arena_key 指向的可能是主分配区的指针，也可能是非主分配区的指针，这里将调用 ptmalloc_init()的线程的 arena_key 绑定到主分配区上。意味着本线程首选从主分配区分配内存。

然后调用 thread_atfork()设置当前进程在 fork 子线程（linux 下线程是轻量级进程，使用类似 fork 进程的机制创建）时处理 mutex 的回调函数，在本进程 fork 子线程时，调用 ptmalloc_lock_all()获得所有分配区的锁，禁止所有分配区分配内存，当子线程创建完毕，父进程调用 ptmalloc_unlock_all()重新 unlock 每个分配区的锁 mutex，子线程调用 ptmalloc_unlock_all2()重新初始化每个分配区的锁 mutex。这几个回调函数将在 5.5.3 节介绍。

```
#ifndef NO_THREADS
# ifndef NO_STARTER
  __malloc_hook = save_malloc_hook;
  __memalign_hook = save_memalign_hook;
  __free_hook = save_free_hook;
# else
#  undef NO_STARTER
# endif
#endif
```

当 pthread 初始化完成后，将相应的 hooks 函数还原为原值。

```
#ifdef _LIBC
  secure = __libc_enable_secure;
  s = NULL;
  if (__builtin_expect (_environ != NULL, 1))
    {
      char **runp = _environ;
      char *envline;

      while (__builtin_expect ((envline = next_env_entry (&runp)) != NULL,
                               0))
        {
          size_t len = strcspn (envline, "=");

          if (envline[len] != '=')
            /* This is a "MALLOC_" variable at the end of the string
```

```
           without a '=' character.  Ignore it since otherwise we
           will access invalid memory below.  */
        continue;

      switch (len)
        {
        case 6:
          if (memcmp (envline, "CHECK_", 6) == 0)
            s = &envline[7];
          break;
        case 8:
          if (! secure)
            {
              if (memcmp (envline, "TOP_PAD_", 8) == 0)
                mALLOPt(M_TOP_PAD, atoi(&envline[9]));
              else if (memcmp (envline, "PERTURB_", 8) == 0)
                mALLOPt(M_PERTURB, atoi(&envline[9]));
            }
          break;
        case 9:
          if (! secure)
            {
              if (memcmp (envline, "MMAP_MAX_", 9) == 0)
                mALLOPt(M_MMAP_MAX, atoi(&envline[10]));
#ifdef PER_THREAD
              else if (memcmp (envline, "ARENA_MAX", 9) == 0)
                mALLOPt(M_ARENA_MAX, atoi(&envline[10]));
#endif
            }
          break;
#ifdef PER_THREAD
        case 10:
          if (! secure)
            {
              if (memcmp (envline, "ARENA_TEST", 10) == 0)
                mALLOPt(M_ARENA_TEST, atoi(&envline[11]));
            }
          break;
#endif
        case 15:
          if (! secure)
            {
              if (memcmp (envline, "TRIM_THRESHOLD_", 15) == 0)
                mALLOPt(M_TRIM_THRESHOLD, atoi(&envline[16]));
```

```
                else if (memcmp (envline, "MMAP_THRESHOLD_", 15) == 0)
                  mALLOPt(M_MMAP_THRESHOLD, atoi(&envline[16]));
              }
            break;
          default:
            break;
          }
      }
  }
#else
  if (! secure)
    {
      if((s = getenv("MALLOC_TRIM_THRESHOLD_")))
        mALLOPt(M_TRIM_THRESHOLD, atoi(s));
      if((s = getenv("MALLOC_TOP_PAD_")))
        mALLOPt(M_TOP_PAD, atoi(s));
      if((s = getenv("MALLOC_PERTURB_")))
        mALLOPt(M_PERTURB, atoi(s));
      if((s = getenv("MALLOC_MMAP_THRESHOLD_")))
        mALLOPt(M_MMAP_THRESHOLD, atoi(s));
      if((s = getenv("MALLOC_MMAP_MAX_")))
        mALLOPt(M_MMAP_MAX, atoi(s));
    }
  s = getenv("MALLOC_CHECK_");
#endif
  if(s && s[0]) {
    mALLOPt(M_CHECK_ACTION, (int)(s[0] - '0'));
    if (check_action != 0)
      __malloc_check_init();
  }
```

从环境变量中读取相应的配置参数值，这些参数包括 MALLOC_TRIM_THRESHOLD_，MALLOC_TOP_PAD_，MALLOC_PERTURB_，MALLOC_MMAP_THRESHOLD_，MALLOC_CHECK_，MALLOC_MMAP_MAX_，MALLOC_ ARENA_MAX,MALLOC_ ARENA_TEST,如果这些选项中的某些项存在，调用 mallopt()函数设置相应的选项。如果这段程序是在 Glibc 库初始化中执行的，会做更多的安全检查工作。

```
  void (*hook) (void) = force_reg (__malloc_initialize_hook);
  if (hook != NULL)
    (*hook)();
  __malloc_initialized = 1;
```

在 ptmalloc_init()函数结束处，查看是否存在__malloc_initialize_hook 函数，如果存在，执行该 hook 函数。最后将全局变量__malloc_initialized 设置为 1，表示 ptmalloc_init()已经初始化完成。

```
}
```

5.5.3 ptmalloc_lock_all(),ptmalloc_unlock_all(),ptmalloc_unlock_all2()

```
/* Magic value for the thread-specific arena pointer when
   malloc_atfork() is in use.  */
#define ATFORK_ARENA_PTR ((Void_t*)-1)

/* The following hooks are used while the `atfork' handling mechanism
   is active. */
static Void_t*
malloc_atfork(size_t sz, const Void_t *caller)
{
  Void_t *vptr = NULL;
  Void_t *victim;

  tsd_getspecific(arena_key, vptr);
  if(vptr == ATFORK_ARENA_PTR) {
    /* We are the only thread that may allocate at all.  */
    if(save_malloc_hook != malloc_check) {
      return _int_malloc(&main_arena, sz);
    } else {
      if(top_check()<0)
        return 0;
      victim = _int_malloc(&main_arena, sz+1);
      return mem2mem_check(victim, sz);
    }
  } else {
    /* Suspend the thread until the `atfork' handlers have completed.
       By that time, the hooks will have been reset as well, so that
       mALLOc() can be used again. */
    (void)mutex_lock(&list_lock);
    (void)mutex_unlock(&list_lock);
    return public_mALLOc(sz);
  }
}
```

当父进程中的某个线程使用 fork 的机制创建子线程时，如果进程中的线程需要分配内存，将使用 malloc_atfork()函数分配内存。malloc_atfork()函数首先查看自己的线程私有实例中的分配区指针，如果该指针为 ATFORK_ARENA_PTR，意味着本线程正在 fork 新线程，并锁住了全局锁 list_lock 和每个分配区，当前只有本线程可以分配内存，如果在 fork 线程前的分配函数不是处于 check 模式，直接调用内部分配函数_int_malloc()。否则在分配内存的同时做检查。如果线程私有实例中的指针不是 ATFORK_ARENA_PTR，意味着当前线程只是常规线程，有另外的线程在 fork 子线程，当前线程只能等待 fork 子线程的线程完成分配，于是等

待获得全局锁 list_lock，如果获得全局锁成功，表示 fork 子线程的线程已经完成 fork 操作，当前线程可以分配内存了，于是是释放全局所 list_lock，并调用 public_mALLOc()分配内存。

```
static void
free_atfork(Void_t* mem, const Void_t *caller)
{
  Void_t *vptr = NULL;
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  if (mem == 0)                              /* free(0) has no effect */
    return;

  p = mem2chunk(mem);         /* do not bother to replicate free_check here */

#if HAVE_MMAP
  if (chunk_is_mmapped(p))                       /* release mmapped memory. */
  {
    munmap_chunk(p);
    return;
  }
#endif

#ifdef ATOMIC_FASTBINS
  ar_ptr = arena_for_chunk(p);
  tsd_getspecific(arena_key, vptr);
  _int_free(ar_ptr, p, vptr == ATFORK_ARENA_PTR);
#else
  ar_ptr = arena_for_chunk(p);
  tsd_getspecific(arena_key, vptr);
  if(vptr != ATFORK_ARENA_PTR)
    (void)mutex_lock(&ar_ptr->mutex);
  _int_free(ar_ptr, p);
  if(vptr != ATFORK_ARENA_PTR)
    (void)mutex_unlock(&ar_ptr->mutex);
#endif
}
```

当父进程中的某个线程使用 fork 的机制创建子线程时，如果进程中的线程需要释放内存，将使用 free_atfork()函数释放内存。free_atfork()函数首先通过需 free 的内存块指针获得 chunk 的指针，如果该 chunk 是通过 mmap 分配的，调用 munmap()释放该 chunk，否则调用 _int_free()函数释放内存。在调用_int_free()函数前，先根据 chunk 指针获得分配区指针，并读取本线程私用实例的指针，如果开启了 ATOMIC_FASTBINS 优化，这个优化使用了 lock-free 的技术优化 fastbins 中单向链表操作。如果没有开启了 ATOMIC_FASTBINS 优化，并且当前线程没有正在 fork 新子线程，则对分配区加锁，然后调用_int_free()函数，然后对分配区解锁。

而对于正在 fork 子线程的线程来说，是不需要对分配区加锁的，因为该线程已经对所有的分配区加锁了。

```
/* Counter for number of times the list is locked by the same thread.  */
static unsigned int atfork_recursive_cntr;

/* The following two functions are registered via thread_atfork() to
   make sure that the mutexes remain in a consistent state in the
   fork()ed version of a thread.  Also adapt the malloc and free hooks
   temporarily, because the `atfork' handler mechanism may use
   malloc/free internally (e.g. in LinuxThreads). */
static void
ptmalloc_lock_all (void)
{
  mstate ar_ptr;

  if(__malloc_initialized < 1)
    return;
  if (mutex_trylock(&list_lock))
    {
      Void_t *my_arena;
      tsd_getspecific(arena_key, my_arena);
      if (my_arena == ATFORK_ARENA_PTR)
        /* This is the same thread which already locks the global list.
           Just bump the counter.  */
        goto out;

      /* This thread has to wait its turn.  */
      (void)mutex_lock(&list_lock);
    }
  for(ar_ptr = &main_arena;;) {
    (void)mutex_lock(&ar_ptr->mutex);
    ar_ptr = ar_ptr->next;
    if(ar_ptr == &main_arena) break;
  }
  save_malloc_hook = __malloc_hook;
  save_free_hook = __free_hook;
  __malloc_hook = malloc_atfork;
  __free_hook = free_atfork;
  /* Only the current thread may perform malloc/free calls now. */
  tsd_getspecific(arena_key, save_arena);
  tsd_setspecific(arena_key, ATFORK_ARENA_PTR);
 out:
  ++atfork_recursive_cntr;
```

```
}
```

当父进程中的某个线程使用 fork 的机制创建子线程时，首先调用 ptmalloc_lock_all()函数暂时对全局锁 list_lock 和所有的分配区加锁，从而保证分配区状态的一致性。ptmalloc_lock_all()函数首先检查 ptmalloc 是否已经初始化，如果没有初始化，退出，如果已经初始化，尝试对全局锁 list_lock 加锁，直到获得全局锁 list_lock，接着对所有的分配区加锁，接着保存原有的分配释放函数，将 malloc_atfork()和 free_atfork()函数作为 fork 子线程期间所使用的内存分配释放函数，然后保存当前线程的私有实例中的原有分配区指针，将 ATFORK_ARENA_PTR 存放到当前线程的私有实例中，用于标识当前现在正在 fork 子线程。为了保证父线程 fork 多个子线程工作正常，也就是说当前线程需要 fork 多个子线程，当一个子线程已经创建，当前线程继续创建其它子线程时，发现当前线程已经对 list_lock 和所有分配区加锁，于是对全局变量 atfork_recursive_cntr 加 1，表示递归 fork 子线程的层数，保证父线程在 fork 子线程过程中，调用 ptmalloc_unlock_all()函数加锁的次数与调用 ptmalloc_lock_all()函数解锁的次数保持一致，同时也保证保证所有的子线程调用 ptmalloc_unlock_all()函数加锁的次数与父线程调用 ptmalloc_lock_all()函数解锁的次数保持一致，防止没有释放锁。

```
static void
ptmalloc_unlock_all (void)
{
  mstate ar_ptr;

  if(__malloc_initialized < 1)
    return;
  if (--atfork_recursive_cntr != 0)
    return;
  tsd_setspecific(arena_key, save_arena);
  __malloc_hook = save_malloc_hook;
  __free_hook = save_free_hook;
  for(ar_ptr = &main_arena;;) {
    (void)mutex_unlock(&ar_ptr->mutex);
    ar_ptr = ar_ptr->next;
    if(ar_ptr == &main_arena) break;
  }
  (void)mutex_unlock(&list_lock);
}
```

当进程的某个线程完成 fork 子线程后，父线程和子线程都调用 ptmall_unlock_all()函数释放全局锁 list_lock，释放所有分配区的锁。ptmall_unlock_all()函数首先检查 ptmalloc 是否初始化，只有初始化后才能调用该函数，接着将全局变量 atfork_recursive_cntr 减 1，如果 atfork_recursive_cntr 为 0，才继续执行，这保证了递归 fork 子线程只会解锁一次。接着将当前线程的私有实例还原为原来的分配区，__malloc_hook 和__free_hook 还原为由来的 hook 函数。然后遍历所有分配区，依次解锁每个分配区，最后解锁 list_lock。

```
#ifdef __linux__
/* In NPTL, unlocking a mutex in the child process after a
```

```
   fork() is currently unsafe, whereas re-initializing it is safe and
   does not leak resources.  Therefore, a special atfork handler is
   installed for the child. */
static void
ptmalloc_unlock_all2 (void)
{
  mstate ar_ptr;

  if(__malloc_initialized < 1)
    return;
#if defined _LIBC || defined MALLOC_HOOKS
  tsd_setspecific(arena_key, save_arena);
  __malloc_hook = save_malloc_hook;
  __free_hook = save_free_hook;
#endif
#ifdef PER_THREAD
  free_list = NULL;
#endif
  for(ar_ptr = &main_arena;;) {
    mutex_init(&ar_ptr->mutex);
#ifdef PER_THREAD
    if (ar_ptr != save_arena) {
      ar_ptr->next_free = free_list;
      free_list = ar_ptr;
    }
#endif
    ar_ptr = ar_ptr->next;
    if(ar_ptr == &main_arena) break;
  }
  mutex_init(&list_lock);
  atfork_recursive_cntr = 0;
}
#else
#define ptmalloc_unlock_all2 ptmalloc_unlock_all
#endif
```

函数 ptmalloc_unlock_all2()被 fork 出的子线程调用，在 Linux 系统中，子线程（进程）unlock 从父线程（进程）中继承的 mutex 不安全，会导致资源泄漏，但重新初始化 mutex 是安全的，所有增加了这个特殊版本用于 Linux 下的 atfork handler。ptmalloc_unlock_all2()函数的处理流程跟 ptmalloc_unlock_all()函数差不多，使用 mutex_init()代替了 mutex_unlock()，如果开启了 PER_THREAD 的优化，将从父线程中继承来的分配区加入到 free_list 中，对于子线程来说，无论全局变量 atfork_recursive_cntr 的值是多少，都将该值设置为 0，因为 ptmalloc_unlock_all2()函数只会被子线程调用一次。

5.6 多分配区支持

由于只有一个主分配区从堆中分配小内存块，而稍大的内存块都必须从 mmap 映射区域分配，如果有多个线程都要分配小内存块，但多个线程是不能同时调用 sbrk()函数的，因为只有一个函数调用 sbrk()时才能保证分配的虚拟地址空间是连续的。如果多个线程都从主分配区中分配小内存块，效率很低效。为了解决这个问题，ptmalloc 使用非主分配区来模拟主分配区的功能，非主分配区同样可以分配小内存块，并且可以创建多个非主分配区，从而在线程分配内存竞争比较激烈的情况下，可以创建更多的非主分配区来完成分配任务，减少分配区的锁竞争，提高分配效率。

Ptmalloc 怎么用非主分配区来模拟主分配区的行为呢？首先创建一个新的非主分配区，非主分配区使用 mmap()函数分配一大块内存来模拟堆（sub-heap），所有的从该非主分配区总分配的小内存块都从 sub-heap 中切分出来，如果一个 sub-heap 的内存用光了，或是 sub-heap 中的内存不够用时，使用 mmap()分配一块新的内存块作为 sub-heap，并将新的 sub-heap 链接在非主分配区中 sub-heap 的单向链表中。

分主分配区中的 sub-heap 所占用的内存不会无限的增长下去，同样会像主分配区那样进行进行 sub-heap 收缩，将 sub-heap 中 top chunk 的一部分返回给操作系统，如果 top chunk 为整个 sub-heap，会把整个 sub-heap 还回给操作系统。收缩堆的条件是当前 free 的 chunk 大小加上前后能合并 chunk 的大小大于 64KB，并且 top chunk 的大小达到 mmap 收缩阈值，才有可能收缩堆。

一般情况下，进程中有多个线程，也有多个分配区，线程的数据一般会比分配区数量多，所以必能保证没有线程独享一个分配区，每个分配区都有可能被多个线程使用，为了保证分配区的线程安全，对分配区的访问需要锁保护，当线程获得分配区的锁时，可以使用该分配区分配内存，并将该分配区的指针保存在线程的私有实例中。

当某一线程需要调用 malloc 分配内存空间时，该线程先查看线程私有变量中是否已经存在一个分配区，如果存在，尝试对该分配区加锁，如果加锁成功，使用该分配区分配内存，如果失败，该线程搜分配区索循环链表试图获得一个空闲的分配区。如果所有的分配区都已经加锁，那么 malloc 会开辟一个新的分配区，把该分配区加入到分配区的全局分配区循环链表并加锁，然后使用该分配区进行分配操作。在回收操作中，线程同样试图获得待回收块所在分配区的锁，如果该分配区正在被别的线程使用，则需要等待直到其他线程释放该分配区的互斥锁之后才可以进行回收操作。

5.6.1 Heap_info

Struct heap_info 定义如下：

```
/* A heap is a single contiguous memory region holding (coalesceable)
   malloc_chunks.  It is allocated with mmap() and always starts at an
   address aligned to HEAP_MAX_SIZE.  Not used unless compiling with
   USE_ARENAS. */

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;
```

ar_ptr 是指向所属分配区的指针，mstate 的定义为：typedef struct malloc_state *mstate;

prev 字段用于将同一个分配区中的 sub_heap 用单向链表链接起来。prev 指向链表中的前一个 sub_heap。

size 字段表示当前 sub_heap 中的内存大小，以 page 对齐。

mprotect_size 字段表示当前 sub_heap 中被读写保护的内存大小，也就是说还没有被分配的内存大小。

Pad 字段用于保证 sizeof (heap_info) + 2 * SIZE_SZ 是按 MALLOC_ALIGNMENT 对齐的。MALLOC_ALIGNMENT_MASK 为 2 * SIZE_SZ - 1，无论 SIZE_SZ 为 4 或 8，-6 * SIZE_SZ & MALLOC_ALIGN_MASK 的值为 0，如果 sizeof (heap_info) + 2 * SIZE_SZ 不是按 MALLOC_ALIGNMENT 对齐，编译的时候就会报错，编译时会执行下面的宏。

```
/* Get a compile-time error if the heap_info padding is not correct
   to make alignment work as expected in sYSMALLOc.  */
extern int sanity_check_heap_info_alignment[(sizeof (heap_info)
                                             + 2 * SIZE_SZ) % MALLOC_ALIGNMENT
                                            ? -1 : 1];
```

为什么一定要保证对齐呢？作为分主分配区的第一个 sub_heap，heap_info 存放在 sub_heap 的头部，紧跟 heap_info 之后是该非主分配区的 malloc_state 实例，紧跟 malloc_state 实例后，是 sub_heap 中的第一个 chunk，但 chunk 的首地址必须按照 MALLOC_ALIGNMENT 对齐，所以在 malloc_state 实例和第一个 chunk 之间可能有几个字节的 pad，但如果 sub_heap 不是非主分配区的第一个 sub_heap，则紧跟 heap_info 后是第一个 chunk，但 sysmalloc()函数默认 heap_info 是按照 MALLOC_ALIGNMENT 对齐的，没有再做对齐的工作，直接将 heap_info 后的内存强制转换成一个 chunk。所以这里在编译时保证 sizeof (heap_info) + 2 * SIZE_SZ 是按 MALLOC_ALIGNMENT 对齐的，在运行时就不用再做检查了，也不必再做对齐。

5.6.2 获取分配区

为了支持多线程，ptmalloc 定义了如下的全局变量：

```
static tsd_key_t arena_key;
static mutex_t list_lock;
#ifdef PER_THREAD
static size_t narenas;
static mstate free_list;
#endif

/* Mapped memory in non-main arenas (reliable only for NO_THREADS). */
static unsigned long arena_mem;
```

```
/* Already initialized? */
int __malloc_initialized = -1;
```

arena_key 存放的是线程的私用实例，该私有实例保存的是分配区（arena）的 malloc_state 实例的指针。arena_key 指向的可能是主分配区的指针，也可能是非主分配区的指针。

list_lock 用于同步分配区的单向环形链表。

如果定义了 PRE_THREAD，narenas 全局变量表示当前分配区的数量，free_list 全局变量是空闲分配区的单向链表，这些空闲的分配区可能是从父进程那里继承来的。全局变量 narenas 和 free_list 都用锁 list_lock 同步。

arena_mem 只用于单线程的 ptmalloc 版本，记录了非主分配区所分配的内存大小。

__malloc_initializd 全局变量用来标识是否 ptmalloc 已经初始化了，其值大于 0 时表示已经初始化。

Ptmalloc 使用如下的宏来获得分配区：

```
/* arena_get() acquires an arena and locks the corresponding mutex.
   First, try the one last locked successfully by this thread.  (This
   is the common case and handled with a macro for speed.)  Then, loop
   once over the circularly linked list of arenas.  If no arena is
   readily available, create a new one.  In this latter case, `size'
   is just a hint as to how much memory will be required immediately
   in the new arena. */
#define arena_get(ptr, size) do { \
  arena_lookup(ptr); \
  arena_lock(ptr, size); \
} while(0)

#define arena_lookup(ptr) do { \
  Void_t *vptr = NULL; \
  ptr = (mstate)tsd_getspecific(arena_key, vptr); \
} while(0)

#ifdef PER_THREAD
#define arena_lock(ptr, size) do { \
  if(ptr) \
    (void)mutex_lock(&ptr->mutex); \
  else \
    ptr = arena_get2(ptr, (size)); \
} while(0)
#else
#define arena_lock(ptr, size) do { \
  if(ptr && !mutex_trylock(&ptr->mutex)) { \
    THREAD_STAT(++(ptr->stat_lock_direct)); \
  } else \
    ptr = arena_get2(ptr, (size)); \
} while(0)
```

```
#endif

/* find the heap and corresponding arena for a given ptr */
#define heap_for_ptr(ptr) \
 ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
#define arena_for_chunk(ptr) \
 (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr)->ar_ptr : &main_arena)
```

arena_get 首先调用 arena_lookup 查找本线程的私用实例中是否包含一个分配区的指针，返回该指针，调用 arena_lock 尝试对该分配区加锁，如果加锁成功，使用该分配区分配内存，如果对该分配区加锁失败，调用 arena_get2 获得一个分配区指针。如果定义了 PRE_THREAD，arena_lock 的处理有些不同，如果本线程拥有的私用实例中包含分配区的指针，则直接对该分配区加锁，否则，调用 arena_get2 获得分配区指针，PRE_THREAD 的优化保证了每个线程尽量从自己所属的分配区中分配内存，减少与其它线程因共享分配区带来的锁开销，但 PRE_THREAD 的优化并不能保证每个线程都有一个不同的分配区，当系统中的分配区数量达到配置的最大值时，不能再增加新的分配区，如果再增加新的线程，就会有多个线程共享同一个分配区。所以 ptmalloc 的 PRE_THREAD 优化，对线程少时可能会提升一些性能，但线程多时，提升性能并不明显。即使没有线程共享分配区的情况下，任然需要加锁，这是不必要的开销，每次加锁操作会消耗 100ns 左右的时间。

每个 sub_heap 的内存块使用 mmap()函数分配，并以 HEAP_MAX_SIZE 对齐，所以可以根据 chunk 的指针地址，获得这个 chunk 所属的 sub_heap 的地址。heap_for_ptr 根据 chunk 的地址获得 sub_heap 的地址。由于 sub_heap 的头部存放的是 heap_info 的实例，heap_info 中保存了分配区的指针，所以可以通过 chunk 的地址获得分配区的地址，前提是这个 chunk 属于非主分配区，arena_for_chunk 用来做这样的转换。

```
#define HEAP_MIN_SIZE (32*1024)
#ifndef HEAP_MAX_SIZE
# ifdef DEFAULT_MMAP_THRESHOLD_MAX
#  define HEAP_MAX_SIZE (2 * DEFAULT_MMAP_THRESHOLD_MAX)
# else
#  define HEAP_MAX_SIZE (1024*1024) /* must be a power of two */
# endif
#endif
```

HEAP_MIN_SIZE 定义了 sub_heap 内存块的最小值，32KB。HEAP_MAX_SIZE 定义了 sub_heap 内存块的最大值，在 32 位系统上，HEAP_MAX_SIZE 默认值为 1MB，64 为系统上，HEAP_MAX_SIZE 的默认值为 64MB。

5.6.3 Arena_get2()

在 5.6.2 节中提到，arena_get 宏尝试查看线程的私用实例中是否包含一个分配区，如果不存在分配区或是存在分配区，但对该分配区加锁失败，就会调用 arena_get2()函数获得一个分配区，下面将分析 arena_get2()函数的实现。

```
static mstate
internal_function
#if __STD_C
arena_get2(mstate a_tsd, size_t size)
```

```
#else
arena_get2(a_tsd, size) mstate a_tsd; size_t size;
#endif
{
  mstate a;

#ifdef PER_THREAD
  if ((a = get_free_list ()) == NULL
      && (a = reused_arena ()) == NULL)
    /* Nothing immediately available, so generate a new arena.  */
    a = _int_new_arena(size);
```

如果开启了 PER_THREAD 优化，首先尝试从分配区的 free list 中获得一个分配区，分配区的 free list 是从父线程（进程）中继承而来，如果 free list 中没有分配区，尝试重用已有的分配区，只有当分配区的数达到限制时才重用分配区，如果仍未获得可重用的分配区，创建一个新的分配区。

```
#else
  if(!a_tsd)
    a = a_tsd = &main_arena;
  else {
    a = a_tsd->next;
    if(!a) {
      /* This can only happen while initializing the new arena. */
      (void)mutex_lock(&main_arena.mutex);
      THREAD_STAT(++(main_arena.stat_lock_wait));
      return &main_arena;
    }
  }
```

如果线程的私有实例中没有分配区，将主分配区作为候选分配区，如果线程私有实例中存在分配区，但不能获得该分配区的锁，将该分配区的下一个分配区作为候选分配区，如果候选分配区为空，意味着当前线程私用实例中的分配区正在初始化，还没有加入到全局的分配区链表中，这种情况下，只有主分配区可选了，等待获得主分配区的锁，如果获得住分配区的锁成功，返回主分配区。

```
  /* Check the global, circularly linked list for available arenas. */
  bool retried = false;
 repeat:
  do {
    if(!mutex_trylock(&a->mutex)) {
      if (retried)
        (void)mutex_unlock(&list_lock);
      THREAD_STAT(++(a->stat_lock_loop));
      tsd_setspecific(arena_key, (Void_t *)a);
      return a;
```

```
    }
    a = a->next;
  } while(a != a_tsd);
```

遍历全局分配区链表，尝试对当前遍历中的分配区加锁，如果对分配区加锁成功，将该分配区加入线程私有实例中并返回该分配区。如果 retried 为 true，意味着这是第二次遍历全局分配区链表，并且获得了全局锁 list_lock，当对分配区加锁成功时，需要释放全局锁 list_lock。

```
  /* If not even the list_lock can be obtained, try again.  This can
     happen during `atfork', or for example on systems where thread
     creation makes it temporarily impossible to obtain _any_
     locks. */
  if(!retried && mutex_trylock(&list_lock)) {
    /* We will block to not run in a busy loop.  */
    (void)mutex_lock(&list_lock);

    /* Since we blocked there might be an arena available now.  */
    retried = true;
    a = a_tsd;
    goto repeat;
  }
```

由于在 atfork 时，父线程（进程）会对所有的分配区加锁，并对全局锁 list_lock 加锁，在有线程在创建子线程的情况下，当前线程是不能获得分配区的，所以在没有重试的情况下，先尝试获得全局锁 list_lock，如果不能获得全局锁 list_lock，阻塞在全局锁 list_lock 上，直到获得全局锁 list_lock，也就是说当前已没有线程在创建子线程，然后再重新遍历全局分配区链表，尝试对分配区加锁，如果经过第二次尝试仍然未能获得一个分配区，只能创建一个新的非主分配区了。

```
  /* Nothing immediately available, so generate a new arena.  */
  a = _int_new_arena(size);
  (void)mutex_unlock(&list_lock);
```

通过前面的所有尝试都未能获得一个可用的分配区，只能创建一个新的非主分配区，执行到这里，可以确保获得了全局锁 list_lock，在创建完新的分配区，并将分配区加入了全局分配区链表中以后，需要对全局锁 list_lock 解锁。

```
#endif

  return a;
}
```

5.6.4 _int_new_arena()

_int_new_arena()函数用于创建一个非主分配区，在 arena_get2()函数中被调用，该函数的实现代码如下：

```
static mstate
```

```
_int_new_arena(size_t size)
{
  mstate a;
  heap_info *h;
  char *ptr;
  unsigned long misalign;

  h = new_heap(size + (sizeof(*h) + sizeof(*a) + MALLOC_ALIGNMENT),
               mp_.top_pad);
  if(!h) {
    /* Maybe size is too large to fit in a single heap.  So, just try
       to create a minimally-sized arena and let _int_malloc() attempt
       to deal with the large request via mmap_chunk().  */
    h = new_heap(sizeof(*h) + sizeof(*a) + MALLOC_ALIGNMENT, mp_.top_pad);
    if(!h)
      return 0;
  }
```

对于一个新的非主分配区，至少包含一个 sub_heap，每个非主分配区中都有相应的管理数据结构，每个非主分配区都有一个 heap_info 实例和 malloc_state 的实例，这两个实例都位于非主分配区的第一个 sub_heap 的开始部分，malloc_state 实例紧接着 heap_info 实例。所以在创建非主分配区时，需要为管理数据结构分配额外的内存空间。New_heap()函数创建一个新的 sub_heap，并返回 sub_heap 的指针。

```
  a = h->ar_ptr = (mstate)(h+1);
  malloc_init_state(a);
  /*a->next = NULL;*/
  a->system_mem = a->max_system_mem = h->size;
  arena_mem += h->size;
```

在 heap_info 实例后紧接着 malloc_state 实例，初始化 malloc_state 实例，更新该分配区所分配的内存大小的统计值。

```
#ifdef NO_THREADS
  if((unsigned long)(mp_.mmapped_mem + arena_mem + main_arena.system_mem) >
     mp_.max_total_mem)
    mp_.max_total_mem = mp_.mmapped_mem + arena_mem + main_arena.system_mem;
#endif

  /* Set up the top chunk, with proper alignment. */
  ptr = (char *)(a + 1);
  misalign = (unsigned long)chunk2mem(ptr) & MALLOC_ALIGN_MASK;
  if (misalign > 0)
    ptr += MALLOC_ALIGNMENT - misalign;
  top(a) = (mchunkptr)ptr;
  set_head(top(a), (((char*)h + h->size) - ptr) | PREV_INUSE);
```

在 sub_heap 中 malloc_state 实例后的内存可以分配给用户使用，ptr 指向存储 malloc_state 实例后的空闲内存，对 ptr 按照 2*SZ_SIZE 对齐后，将 ptr 赋值给分配区的 top chunk，也就是说把 sub_heap 中整个空闲内存块作为 top chunk，然后设置 top chunk 的 size，并标识 top chunk 的前一个 chunk 为已处于分配状态。

```
  tsd_setspecific(arena_key, (Void_t *)a);
  mutex_init(&a->mutex);
  (void)mutex_lock(&a->mutex);
```

将创建好的非主分配区加入线程的私有实例中，然后对非主分配区的锁进行初始化，并获得该锁。

```
#ifdef PER_THREAD
  (void)mutex_lock(&list_lock);
#endif

  /* Add the new arena to the global list.  */
  a->next = main_arena.next;
  atomic_write_barrier ();
  main_arena.next = a;

#ifdef PER_THREAD
  ++narenas;

  (void)mutex_unlock(&list_lock);
#endif
```

将刚创建的非主分配区加入到分配区的全局链表中，如果开启了 PER_THREAD 优化，在 arena_get2()函数中没有对全局锁 list_lock 加锁，这里修改全局分配区链表时需要获得全局锁 list_lock。如果没有开启 PER_THREAD 优化，arene_get2()函数调用_int_new_arena()函数时已经获得了全局锁 list_lock，所以对全局分配区链表的修改不用再加锁。

```
  THREAD_STAT(++(a->stat_lock_loop));

  return a;
}
```

5.6.5 New_heap()

New_heap()函数负责从 mmap 区域映射一块内存来作为 sub_heap，在 32 位系统上，该函数每次映射 1M 内存，映射的内存块地址按 1M 对齐；在 64 为系统上，该函数映射 64M 内存，映射的内存块地址按 64M 对齐。New_heap()函数只是映射一块虚拟地址空间，该空间不可读写，不会被 swap。New_heap()函数的实现源代码如下：

```
/* If consecutive mmap (0, HEAP_MAX_SIZE << 1, ...) calls return decreasing
   addresses as opposed to increasing, new_heap would badly fragment the
   address space.  In that case remember the second HEAP_MAX_SIZE part
```

```
   aligned to HEAP_MAX_SIZE from last mmap (0, HEAP_MAX_SIZE << 1, ...)
   call (if it is already aligned) and try to reuse it next time.  We need
   no locking for it, as kernel ensures the atomicity for us - worst case
   we'll call mmap (addr, HEAP_MAX_SIZE, ...) for some value of addr in
   multiple threads, but only one will succeed.  */
static char *aligned_heap_area;

/* Create a new heap.  size is automatically rounded up to a multiple
   of the page size. */
static heap_info *
internal_function
#if __STD_C
new_heap(size_t size, size_t top_pad)
#else
new_heap(size, top_pad) size_t size, top_pad;
#endif
{
  size_t page_mask = malloc_getpagesize - 1;
  char *p1, *p2;
  unsigned long ul;
  heap_info *h;

  if(size+top_pad < HEAP_MIN_SIZE)
    size = HEAP_MIN_SIZE;
  else if(size+top_pad <= HEAP_MAX_SIZE)
    size += top_pad;
  else if(size > HEAP_MAX_SIZE)
    return 0;
  else
    size = HEAP_MAX_SIZE;
  size = (size + page_mask) & ~page_mask;
```

调整 size 的大小，size 的最小值为 32K,最大值 HEAP_MAX_SIZE 在不同的系统上不同，在 32 位系统为 1M，64 位系统为 64M，将 size 的大小调整到最小值与最大值之间，并以页对齐，如果 size 大于最大值，直接报错。

```
  /* A memory region aligned to a multiple of HEAP_MAX_SIZE is needed.
     No swap space needs to be reserved for the following large
     mapping (on Linux, this is the case for all non-writable mappings
     anyway). */
  p2 = MAP_FAILED;
  if(aligned_heap_area) {
    p2 = (char *)MMAP(aligned_heap_area, HEAP_MAX_SIZE, PROT_NONE,
                      MAP_PRIVATE|MAP_NORESERVE);
    aligned_heap_area = NULL;
```

```
    if (p2 != MAP_FAILED && ((unsigned long)p2 & (HEAP_MAX_SIZE-1))) {
      munmap(p2, HEAP_MAX_SIZE);
      p2 = MAP_FAILED;
    }
  }
```

全局变量 aligned_heap_area 是上一次调用 mmap 分配内存的结束虚拟地址，并已经按照 HEAP_MAX_SIZE 大小对齐。如果 aligned_heap_area 不为空，尝试从上次映射结束地址开始映射大小为 HEAP_MAX_SIZE 的内存块，由于全局变量 aligned_heap_area 没有锁保护，可能存在多个线程同时 mmap()函数从 aligned_heap_area 开始映射新的虚拟内存块，操作系统会保证只会有一个线程会成功，其它在同一地址映射新虚拟内存块都会失败。无论映射是否成功，都将全局变量 aligned_heap_area 设置为 NULL。如果映射成功，但返回的虚拟地址不是按 HEAP_MAX_SIZE 大小对齐的，取消该区域的映射，映射失败。

```
  if(p2 == MAP_FAILED) {
    p1 = (char *)MMAP(0, HEAP_MAX_SIZE<<1, PROT_NONE,
                      MAP_PRIVATE|MAP_NORESERVE);
```

全局变量 aligned_heap_area 为 NULL，或者从 aligned_heap_area 开始映射失败了，尝试映射 2 倍 HEAP_MAX_SIZE 大小的虚拟内存，便于地址对齐，因为在最坏可能情况下，需要映射 2 倍 HEAP_MAX_SIZE 大小的虚拟内存才能实现地址按照 HEAP_MAX_SIZE 大小对齐。

```
    if(p1 != MAP_FAILED) {
      p2 = (char *)(((unsigned long)p1 + (HEAP_MAX_SIZE-1))
                    & ~(HEAP_MAX_SIZE-1));
      ul = p2 - p1;
      if (ul)
        munmap(p1, ul);
      else
        aligned_heap_area = p2 + HEAP_MAX_SIZE;
      munmap(p2 + HEAP_MAX_SIZE, HEAP_MAX_SIZE - ul);
```

映射 2 倍 HEAP_MAX_SIZE 大小的虚拟内存成功，将大于等于 p1 并按 HEAP_MAX_SIZE 大小对齐的第一个虚拟地址赋值给 p2，p2 作为 sub_heap 的起始虚拟地址，p2+HEAP_MAX_SIZE 作为 sub_heap 的结束地址，并将 sub_heap 的结束地址赋值给全局变量 aligned_heap_area，最后还需要将多余的虚拟内存还回给操作系统。

```
    } else {
      /* Try to take the chance that an allocation of only HEAP_MAX_SIZE
         is already aligned. */
      p2 = (char *)MMAP(0, HEAP_MAX_SIZE, PROT_NONE, MAP_PRIVATE|MAP_NORESERVE);
      if(p2 == MAP_FAILED)
        return 0;
      if((unsigned long)p2 & (HEAP_MAX_SIZE-1)) {
        munmap(p2, HEAP_MAX_SIZE);
        return 0;
      }
```

映射 2 倍 HEAP_MAX_SIZE 大小的虚拟内存失败了，再尝试映射 HEAP_MAX_SIZE 大小的虚拟内存，如果失败，返回；如果成功，但该虚拟地址不是按照 HEAP_MAX_SIZE 大小对齐的，返回。

```
    }
  }
  if(mprotect(p2, size, PROT_READ|PROT_WRITE) != 0) {
    munmap(p2, HEAP_MAX_SIZE);
    return 0;
  }
  h = (heap_info *)p2;
  h->size = size;
  h->mprotect_size = size;
  THREAD_STAT(stat_n_heaps++);
```

调用 mprotect()函数将 size 大小的内存设置为可读可写，如果失败，解除整个 sub_heap 的映射。然后更新 heap_info 实例中的相关字段。

```
  return h;
}
```

5.6.6 get_free_list()和 reused_arena()

这两个函数在开启了 PER_THRAD 优化时用于获取分配区（arena），arena_get2 首先调用 get_free_list()尝试获得 arena，如果失败在尝试调用 reused_arena()获得 arena，如果仍然没有获得分配区，调用_int_new_arena()创建一个新的分配区。Get_free_list()函数的源代码如下：

```
static mstate
get_free_list (void)
{
  mstate result = free_list;
  if (result != NULL)
    {
      (void)mutex_lock(&list_lock);
      result = free_list;
      if (result != NULL)
        free_list = result->next_free;
      (void)mutex_unlock(&list_lock);

      if (result != NULL)
        {
          (void)mutex_lock(&result->mutex);
          tsd_setspecific(arena_key, (Void_t *)result);
          THREAD_STAT(++(result->stat_lock_loop));
        }
```

```
    }

  return result;
}
```

这个函数实现很简单，首先查看 arena 的 free_list 中是否为 NULL，如果不为 NULL，获得全局锁 list_lock，将 free_list 的第一个 arena 从单向链表中取出，解锁 list_lock。如果从 free_list 中获得一个 arena，对该 arena 加锁，并将该 arena 加入线程的私有实例中。

reused_arena()函数的源代码实现如下：

```
static mstate
reused_arena (void)
{
  if (narenas <= mp_.arena_test)
    return NULL;
```

首先判断全局分配区的总数是否小于分配区的个数的限定值（arena_test），在 32 位系统上 arena_test 默认值为 2，64 位系统上的默认值为 8，如果当前进程的分配区数量没有达到限定值，直接返回 NULL。

```
  static int narenas_limit;
  if (narenas_limit == 0)
    {
      if (mp_.arena_max != 0)
        narenas_limit = mp_.arena_max;
      else
        {
          int n  = __get_nprocs ();

          if (n >= 1)
            narenas_limit = NARENAS_FROM_NCORES (n);
          else
            /* We have no information about the system.  Assume two
               cores.  */
            narenas_limit = NARENAS_FROM_NCORES (2);
        }
    }

  if (narenas < narenas_limit)
    return NULL;
```

设定全局变量 narenas_limit，如果应用层设置了进程的最大分配区个数（arena_max），将 arena_max 赋值给 narenas_limit，否则根据系统的 cpu 个数和系统的字大小设定 narenas_limit 的大小，narenas_limit 的大小默认与 arena_test 大小相同。然后再次判断进程的当前分配区个数是否达到了分配区的限制个数，如果没有达到限定值，返回。

```
  mstate result;
```

```
  static mstate next_to_use;
  if (next_to_use == NULL)
    next_to_use = &main_arena;

  result = next_to_use;
  do
    {
      if (!mutex_trylock(&result->mutex))
        goto out;

      result = result->next;
    }
  while (result != next_to_use);

  /* No arena available.  Wait for the next in line.  */
  (void)mutex_lock(&result->mutex);

 out:
  tsd_setspecific(arena_key, (Void_t *)result);
  THREAD_STAT(++(result->stat_lock_loop));
  next_to_use = result->next;
```

全局变量 next_to_use 指向下一个可能可用的分配区，该全局变量没有锁保护，主要用于记录上次遍历分配区循环链表到达的位置，避免每次都从同一个分配区开始遍历，导致从某个分配区分配的内存过多。首先判断 next_to_use 是否为 NULL，如果是，将主分配区赋值给 next_to_use。然后从 next_to_use 开始遍历分配区链表，尝试对遍历的分配区加锁，如果加锁成功，退出循环，如果遍历分配区循环链表中的所有分配区，尝试加锁都失败了，等待获得 next_to_use 指向的分配区的锁。执行到 out 的代码，意味着已经获得一个分配区的锁，将该分配区加入线程私有实例，并将当前分配区的下一个分配区赋值给 next_to_use。

```
  return result;
}
```

5.6.7 grow_heap(),shrink_heap(),delete_heap(),heap_trim()

这几个函数实现 sub_heap 和增长和收缩，grow_heap()函数主要将 sub_heap 中可读可写区域扩大；shrink_heap()函数缩小 sub_heap 的虚拟内存区域，减小该 sub_heap 的虚拟内存占用量；delete_heap()为一个宏，如果 sub_heap 中所有的内存都空闲，使用该宏函数将 sub_heap 的虚拟内存还回给操作系统；heap_trim()函数根据 sub_heap 的 top chunk 大小调用 shrink_heap()函数收缩 sub_heap。

Grow_heap()函数的实现代码如下：

```
static int
#if __STD_C
grow_heap(heap_info *h, long diff)
```

```
#else
grow_heap(h, diff) heap_info *h; long diff;
#endif
{
  size_t page_mask = malloc_getpagesize - 1;
  long new_size;

  diff = (diff + page_mask) & ~page_mask;
  new_size = (long)h->size + diff;
  if((unsigned long) new_size > (unsigned long) HEAP_MAX_SIZE)
    return -1;
  if((unsigned long) new_size > h->mprotect_size) {
    if (mprotect((char *)h + h->mprotect_size,
                 (unsigned long) new_size - h->mprotect_size,
                 PROT_READ|PROT_WRITE) != 0)
      return -2;
    h->mprotect_size = new_size;
  }

  h->size = new_size;
  return 0;
}
```

Grow_heap()函数的实现比较简单，首先将要增加的可读可写的内存大小按照页对齐，然后计算 sub_heap 总的可读可写的内存大小 new_size，判断 new_size 是否大于 HEAP_MAX_SIZE，如果是，返回，否则判断 new_size 是否大于当前 sub_heap 的可读可写区域大小，如果否，调用 mprotect()设置新增的区域可读可写，并更新当前 sub_heap 的可读可写区域的大小为 new_size。最后将当前 sub_heap 的字段 size 更新为 new_size。

Shrink_heap()函数的实现源代码如下：

```
static int
#if __STD_C
shrink_heap(heap_info *h, long diff)
#else
shrink_heap(h, diff) heap_info *h; long diff;
#endif
{
  long new_size;

  new_size = (long)h->size - diff;
  if(new_size < (long)sizeof(*h))
    return -1;
  /* Try to re-map the extra heap space freshly to save memory, and
     make it inaccessible. */
#ifdef _LIBC
```

```
  if (__builtin_expect (__libc_enable_secure, 0))
#else
  if (1)
#endif
    {
      if((char *)MMAP((char *)h + new_size, diff, PROT_NONE,
                      MAP_PRIVATE|MAP_FIXED) == (char *) MAP_FAILED)
        return -2;
      h->mprotect_size = new_size;
    }
#ifdef _LIBC
  else
    madvise ((char *)h + new_size, diff, MADV_DONTNEED);
#endif
  /*fprintf(stderr, "shrink %p %08lx\n", h, new_size);*/

  h->size = new_size;
  return 0;
}
```

Shrink_heap()函数的参数 diff 已经页对齐，同时 sub_heap 的 size 也是安装页对齐的，所以计算 sub_heap 的 new_size 时不用再处理页对齐。如果 new_size 比 sub_heap 的首地址还小，报错退出，如果该函数运行在非 Glibc 中，则从 sub_heap 中切割出 diff 大小的虚拟内存，创建一个新的不可读写的映射区域，注意 mmap()函数这里使用了 MAP_FIXED 标志，然后更新 sub_heap 的可读可写内存大小。如果该函数运行在 Glibc 库中，则调用 madvise()函数，实际上 madvise()函数什么也不做，只是返回错误，这里并没有处理 madvise()函数的返回值。

```
#define delete_heap(heap) \
  do {                                                          \
    if ((char *)(heap) + HEAP_MAX_SIZE == aligned_heap_area)    \
      aligned_heap_area = NULL;                                 \
    munmap((char*)(heap), HEAP_MAX_SIZE);                       \
  } while (0)
```

Delete_heap()宏函数首先判断当前删除的 sub_heap 的结束地址是否与全局变量 aligned_heap_area 指向的地址相同，如果相同，则将全局变量 aligned_heap_area 设置为 NULL，因为当前 sub_heap 删除以后，就可以从当前 sub_heap 的起始地址或是更低的地址开始映射新的 sub_heap，这样可以尽量从地地址映射内存。然后调用 munmap()函数将整个 sub_heap 的虚拟内存区域释放掉。在调用 munmap()函数时，heap_trim()函数调用 shrink_heap()函数可能已将 sub_heap 切分成多个子区域，munmap()函数的第二个参数为 HEAP_MAX_SIZE，无论该 sub_heap（大小为 HEAP_MAX_SIZE）的内存区域被切分成多少个子区域，将整个 sub_heap 都释放掉了。

Heap_trim()函数的源代码如下：

```
static int
internal_function
```

```
#if __STD_C
heap_trim(heap_info *heap, size_t pad)
#else
heap_trim(heap, pad) heap_info *heap; size_t pad;
#endif
{
  mstate ar_ptr = heap->ar_ptr;
  unsigned long pagesz = mp_.pagesize;
  mchunkptr top_chunk = top(ar_ptr), p, bck, fwd;
  heap_info *prev_heap;
  long new_size, top_size, extra;

  /* Can this heap go away completely? */
  while(top_chunk == chunk_at_offset(heap, sizeof(*heap))) {
```

每个非主分配区至少有一个 sub_heap，每个非主分配区的第一个 sub_heap 中包含了一个 heap_info 的实例和 malloc_state 的实例，分主分配区中的其它 sub_heap 中只有一个 heap_info 实例，紧跟 heap_info 实例后，为可以用于分配的内存块。当当前非主分配区的 top chunk 与当前 sub_heap 的 heap_info 实例的结束地址相同时，意味着当前 sub_heap 中只有一个空闲 chunk，没有已分配的 chunk。所以可以将当前整个 sub_heap 都释放掉。

```
    prev_heap = heap->prev;
    p = chunk_at_offset(prev_heap, prev_heap->size - (MINSIZE-2*SIZE_SZ));
    assert(p->size == (0|PREV_INUSE)); /* must be fencepost */
    p = prev_chunk(p);
    new_size = chunksize(p) + (MINSIZE-2*SIZE_SZ);
    assert(new_size>0 && new_size<(long)(2*MINSIZE));
    if(!prev_inuse(p))
      new_size += p->prev_size;
    assert(new_size>0 && new_size<HEAP_MAX_SIZE);
    if(new_size + (HEAP_MAX_SIZE - prev_heap->size) < pad + MINSIZE + pagesz)
      break;
```

每个 sub_heap 的可读可写区域的末尾都有两个 chunk 用于 fencepost，以 64 位系统为例，最后一个 chunk 占用的空间为 MINSIZE-2*SIZE_SZ，为 16B，最后一个 chuk 的 size 字段记录的前一个 chunk 为 inuse 状态，并标识当前 chunk 大小为 0，倒数第二个 chunk 为 inuse 状态，这个 chunk 也是 fencepost 的一部分，这个 chunk 的大小为 2*SIZE_SZ，为 16B，所以用于 fencepost 的两个 chunk 的空间大小为 32B。fencepost 也有可能大于 32B，第二个 chunk 仍然为 16B，第一个 chunk 的大小大于 16B，这种情况发生在 top chunk 的空间小于 2*MINSIZE，大于 MINSIZE，但对于一个完全空闲的 sub_heap 来说，top chunk 的空间肯定大于 2*MINSIZE，所以在这里不考虑这种情况。用于 fencepost 的 chunk 空间其实都是被分配给应用层使用的，new_size 表示当前 sub_heap 中可读可写区域的可用空间，如果倒数第二个 chunk 的前一个 chunk 为空闲状态，当前 sub_heap 中可读可写区域的可用空间大小还需要加上这个空闲 chunk 的大小。如果 new_size 与 sub_heap 中剩余的不可读写的区域大小之和小于 32+4K（64 位系统），意味着前一个 sub_heap 的可用空间太少了，不能释放当前的 sub_heap。

```
    ar_ptr->system_mem -= heap->size;
    arena_mem -= heap->size;
    delete_heap(heap);
    heap = prev_heap;
    if(!prev_inuse(p)) { /* consolidate backward */
      p = prev_chunk(p);
      unlink(p, bck, fwd);
    }
    assert(((unsigned long)((char*)p + new_size) & (pagesz-1)) == 0);
    assert( ((char*)p + new_size) == ((char*)heap + heap->size) );
    top(ar_ptr) = top_chunk = p;
    set_head(top_chunk, new_size | PREV_INUSE);
    /*check_chunk(ar_ptr, top_chunk);*/
```

首先更新非主分配区的内存统计，然后调用 delete_heap()宏函数释放该 sub_heap，把当前 heap 设置为被释放 sub_heap 的前一个 sub_heap，p 指向的是被释放 sub_heap 的前一个 sub_heap 的倒数第二个 chunk，如果 p 的前一个 chunk 为空闲状态，由于不可能出现多个连续的空闲 chunk，所以将 p 设置为 p 的前一个 chunk，也就是 p 指向空闲 chunk，并将该空闲 chunk 从空闲 chunk 链表中移除，并将将该空闲 chunk 赋值给 sub_heap 的 top chunk，并设置 top chunk 的 size，标识 top chunk 的前一个 chunk 处于 inuse 状态。然后继续判断循环条件，如果循环条件不满足，退出循环，如果条件满足，继续对当前 sub_heap 进行收缩。

```
  }
  top_size = chunksize(top_chunk);
  extra = ((top_size - pad - MINSIZE + (pagesz-1))/pagesz - 1) * pagesz;
  if(extra < (long)pagesz)
    return 0;
  /* Try to shrink. */
  if(shrink_heap(heap, extra) != 0)
    return 0;
  ar_ptr->system_mem -= extra;
  arena_mem -= extra;

  /* Success. Adjust top accordingly. */
  set_head(top_chunk, (top_size - extra) | PREV_INUSE);
  /*check_chunk(ar_ptr, top_chunk);*/
```

首先查看 top chunk 的大小，如果 top chunk 的大小减去 pad 和 MINSIZE 小于一页大小，返回退出，否则调用 shrink_heap()函数对当前 sub_heap 进行收缩，将空闲的整数个页收缩掉，仅剩下不足一页的空闲内存，如果 shrink_heap()失败，返回退出，否则，更新内存使用统计，更新 top chunk 的大小。

```
  return 1;
}
```

5.7 内存分配 malloc

Ptmalloc2 主要的内存分配函数为 malloc()，但源代码中并不能找到该函数，该函数是用宏定义为 public_mALLOc()，因为该函数在不同的编译条件下，具有不同的名称。public_mALLOc()函数只是简单的封装_int_malloc()函数，_int_malloc()函数才是内存分配的核心实现。下面我们将分析 malloc 的实现。

5.7.1 public_mALLOc()

先给出源代码：

```
Void_t*
public_mALLOc(size_t bytes)
{
  mstate ar_ptr;
  Void_t *victim;

  __malloc_ptr_t (*hook) (size_t, __const __malloc_ptr_t)
    = force_reg (__malloc_hook);
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));
```

首先检查是否存在内存分配的 hook 函数，如果存在，调用 hook 函数，并返回，hook 函数主要用于进程在创建新线程过程中分配内存，或者支持用户提供的内存分配函数。请参考 5.3 节相关的描述。

```
  arena_lookup(ar_ptr);
  arena_lock(ar_ptr, bytes);
  if(!ar_ptr)
    return 0;
  victim = _int_malloc(ar_ptr, bytes);
```

获取分配区指针，如果获取分配区失败，返回退出，否则，调用_int_malloc()函数分配内存。

```
  if(!victim) {
    /* Maybe the failure is due to running out of mmapped areas. */
    if(ar_ptr != &main_arena) {
      (void)mutex_unlock(&ar_ptr->mutex);
      ar_ptr = &main_arena;
      (void)mutex_lock(&ar_ptr->mutex);
      victim = _int_malloc(ar_ptr, bytes);
      (void)mutex_unlock(&ar_ptr->mutex);
```

如果_int_malloc()函数分配内存失败，并且使用的分配区不是主分配区，这种情况可能是 mmap 区域的内存被用光了，当主分配区可以从堆中分配内存，所以需要再尝试从主分配区中分配内存。首先释放所使用分配区的锁，然后获得主分配区的锁，并调用_int_malloc()

函数分配内存，最后释放主分配区的锁。

```
    } else {
#if USE_ARENAS
      /* ... or sbrk() has failed and there is still a chance to mmap() */
      ar_ptr = arena_get2(ar_ptr->next ? ar_ptr : 0, bytes);
      (void)mutex_unlock(&main_arena.mutex);
      if(ar_ptr) {
        victim = _int_malloc(ar_ptr, bytes);
        (void)mutex_unlock(&ar_ptr->mutex);
      }
```

如果_int_malloc()函数分配内存失败，并且使用的分配区是主分配区，查看是否有非主分配区，如果有，调用 arena_get2()获取分配区，然后对主分配区解锁，如果 arena_get2()返回一个非主分配区，尝试调用_int_malloc()函数从该非主分配区分配内存，最后释放该非主分配区的锁。

```
#endif
    }
  } else
    (void)mutex_unlock(&ar_ptr->mutex);
```

如果_int_malloc()函数分配内存成功，释放所使用的分配区的锁。

```
  assert(!victim || chunk_is_mmapped(mem2chunk(victim)) ||
         ar_ptr == arena_for_chunk(mem2chunk(victim)));
  return victim;
}
```

5.7.2 _int_malloc()

_int_malloc()函数是内存分配的核心，根据分配的内存块的大小，该函数中实现了四种分配内存的路径，下面将分别分析这四种分配路径。

先给出_int_malloc()函数的函数定义及临时变量的定义：

```
static Void_t*
_int_malloc(mstate av, size_t bytes)
{
  INTERNAL_SIZE_T nb;               /* normalized request size */
  unsigned int    idx;              /* associated bin index */
  mbinptr         bin;              /* associated bin */

  mchunkptr       victim;           /* inspected/selected chunk */
  INTERNAL_SIZE_T size;             /* its size */
  int             victim_index;     /* its bin index */

  mchunkptr       remainder;        /* remainder from a split */
  unsigned long   remainder_size;   /* its size */
```

```
  unsigned int     block;            /* bit map traverser */
  unsigned int     bit;              /* bit map traverser */
  unsigned int     map;              /* current word of binmap */

  mchunkptr        fwd;              /* misc temp for linking */
  mchunkptr        bck;              /* misc temp for linking */

  const char *errstr = NULL;

  /*
    Convert request size to internal form by adding SIZE_SZ bytes
    overhead plus possibly more to obtain necessary alignment and/or
    to obtain a size of at least MINSIZE, the smallest allocatable
    size. Also, checked_request2size traps (returning 0) request sizes
    that are so large that they wrap around zero when padded and
    aligned.
  */
  checked_request2size(bytes, nb);
```

checked_request2size()函数将需要分配的内存大小 bytes 转换为需要分配的 chunk 大小 nb。Ptmalloc 内部分配都是以 chunk 为单位，根据 chunk 的大小，决定如何获得满足条件的 chunk。

5.7.2.1分配 fast bin chunk

如果所需的 chunk 大小小于等于 fast bins 中的最大 chunk 大小，首先尝试从 fast bins 中分配 chunk。源代码如下：

```
  /*
    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) <= (unsigned long)(get_max_fast ())) {
    idx = fastbin_index(nb);
    mfastbinptr* fb = &fastbin (av, idx);
#ifdef ATOMIC_FASTBINS
    mchunkptr pp = *fb;
    do
      {
        victim = pp;
        if (victim == NULL)
          break;
      }
```

```
    while ((pp = catomic_compare_and_exchange_val_acq (fb, victim->fd, victim))
           != victim);
#else
    victim = *fb;
#endif
    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));
          return NULL;
        }
#ifndef ATOMIC_FASTBINS
      *fb = victim->fd;
#endif
      check_remalloced_chunk(av, victim, nb);
      void *p = chunk2mem(victim);
      if (__builtin_expect (perturb_byte, 0))
        alloc_perturb (p, bytes);
      return p;
    }
  }
```

如果没有开启 ATOMIC_FASTBINS 优化，从 fast bins 中分配一个 chunk 相当简单，首先根据所需 chunk 的大小获得该 chunk 所属 fast bin 的 index，根据该 index 获得所需 fast bin 的空闲 chunk 链表的头指针，然后将头指针的下一个 chunk 作为空闲 chunk 链表的头部。为了加快从 fast bins 中分配 chunk，处于 fast bins 中 chunk 的状态仍然保持为 inuse 状态，避免被相邻的空闲 chunk 合并，从 fast bins 中分配 chunk，只需取出第一个 chunk，并调用 chunk2mem() 函数返回用户所需的内存块。

如果开启 ATOMIC_FASTBINS 优化，这里使用了 lock-free 的技术实现单向链表删除第一个 node 的操作。Lock-free 算法的基础是 CAS (Compareand-Swap) 原子操作。当某个地址的原始值等于某个比较值时，把值改成新值，无论有否修改，返回这个地址的原始值。目前的 cpu 支持最多 64 位的 CAS，并且指针 p 必须对齐。原子操作指一个 cpu 时钟周期内就可以完成的操作，不会被其他线程干扰。

一般的 CAS 使用方式是：假设有指针 p，它指向一个 32 位或者 64 位数，

1. 复制 p 的内容（*p）到比较量 cmp（原子操作）。
2. 基于这个比较量计算一个新值 xchg（非原子操作）。
3. 调用 CAS 比较当前*p 和 cmp，如果相等把*p 替换成 xchg（原子操作）。
4. 如果成功退出，否则回到第一步重新进行。

第 3 步的 CAS 操作保证了写入的同时 p 没有被其他线程更改。如果*p 已经被其他线程更改，那么第 2 步计算新值所使用的值（cmp）已经过期了，因此这个整个过程失败，重新来过。多线程环境下，由于 3 是一个原子操作，那么起码有一个线程（最快执行到 3）的 CAS 操作可以成功，这样整体上看，就保证了所有的线程上在“前进”，而不需要使用效率低下的锁来协调线程，更不会导致死锁之类的麻烦。

ABA 问题，当 A 线程执行 2 的时候，被 B 线程更改了*p 为 x，而 C 线程又把它改回了原始值，这时回到 A 线程，A 线程无法监测到原始值已经被更改过了，CAS 操作会成功（实际上应该失败）。ABA 大部分情况下会造成一些问题，因为 p 的内容一般不可能是独立的，其他内容已经更改，而 A 线程认为它没有更改就会带来不可预知的结果。

如果开启 ATOMIC_FASTBINS 优化，这里的实现会出现 ABA 问题吗？不会出现，如果开启了 ATOMIC_FASTBINS 优化，在 free 时，如果释放的 chunk 属于 fast bin，不需要对分配区加锁，可以通过 lock-free 技术将该 chunk 加入 fast bins 的链表中。当从分配区分配内存时，需要对分配区加锁，所以当 A 线程获得了分配区的锁，并从 fast bin 中分配内存执行 2 的时候，被 B 线程调用 free 函数向 fast bin 的链表中加入了一个新的 chunk，即更改了*fb 为 x，但不会存在 C 线程将*fb 改回原值，如果存在，意味着 C 线程先分配了*fb 所存的 chunk，并将该 chunk 释放回了 fast bin，但 C 线程分配*fb 所存的 chunk 需要获得分配区的锁，但分配区的锁被 A 线程持有，所以 C 线程不可能将*fb 改回原值，也就不会存在 ABA 问题。

5.7.2.2分配 small bin chunk

如果所需的 chunk 大小属于 small bin，则会执行如下的代码：

```
/*
  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 (__builtin_expect (bck->fd != victim, 0))
        {
          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)
        victim->size |= NON_MAIN_ARENA;
```

```
        check_malloced_chunk(av, victim, nb);
        void *p = chunk2mem(victim);
        if (__builtin_expect (perturb_byte, 0))
          alloc_perturb (p, bytes);
        return p;
      }
    }
  }
```

如果分配的 chunk 属于 small bin，首先查找 chunk 所对应 small bins 数组的 index，然后根据 index 获得某个 small bin 的空闲 chunk 双向循环链表表头，然后将最后一个 chunk 赋值给 victim，如果 victim 与表头相同，表示该链表为空，不能从 small bin 的空闲 chunk 链表中分配，这里不处理，等后面的步骤来处理。如果 victim 与表头不同，有两种情况，如果 victim 为 0，表示 small bin 还没有初始化为双向循环链表，调用 malloc_consolidate()函数将 fast bins 中的 chunk 合并。否则，将 victim 从 small bin 的双向循环链表中取出，设置 victim chunk 的 inuse 标志，该标志处于 victim chunk 的下一个相邻 chunk 的 size 字段的第一个 bit。从 small bin 中取出一个 chunk 也可以用 unlink()宏函数，只是这里没有使用。

接着判断当前分配区是否为非主分配区，如果是，将 victim chunk 的 size 字段中的表示非主分配区的标志 bit 清零，最后调用 chunk2mem()函数获得 chunk 的实际可用的内存指针，将该内存指针返回给应用层。到此从 small bins 中分配 chunk 的工作完成了，但我们看到，当对应的 small bin 中没有空闲 chunk，或是对应的 small bin 还没有初始化完成，并没有获取到 chunk，这两种情况都需要后面的步骤来处理。

5.7.2.3分配 large bin chunk

如果所需的 chunk 不属于 small bins，首先会执行如下的代码段：

```
  /*
     If this is a large request, consolidate fastbins before continuing.
     While it might look excessive to kill all fastbins before
     even seeing if there is space available, this avoids
     fragmentation problems normally associated with fastbins.
     Also, in practice, programs tend to have runs of either small or
     large requests, but less often mixtures, so consolidation is not
     invoked all that often in most programs. And the programs that
     it is called frequently in otherwise tend to fragment.
  */

  else {
    idx = largebin_index(nb);
    if (have_fastchunks(av))
      malloc_consolidate(av);
  }
```

所需 chunk 不属于 small bins，那么就一定属于 large bins，首先根据 chunk 的大小获得对应的 large bin 的 index，接着判断当前分配区的 fast bins 中是否包含 chunk，如果存在，调

用 malloc_consolidate()函数合并 fast bins 中的 chunk，并将这些空闲 chunk 加入 unsorted bin 中。

下面的源代码实现从 last remainder chunk，large bins 和 top chunk 中分配所需的 chunk，这里包含了多个多层循环，在这些循环中，主要工作是分配前两步都未分配成功的 small bin chunk，large bin chunk 和 large chunk。最外层的循环用于重新尝试分配 small bin chunk，因为如果在前一步分配 small bin chunk 不成功，并没有调用 malloc_consolidate()函数合并 fast bins 中的 chunk，将空闲 chunk 加入 unsorted bin 中，如果第一尝试从 last remainder chunk，top chunk 中分配 small bin chunk 都失败以后，如果 fast bins 中存在空闲 chunk，会调用 malloc_consolidate()函数，那么在 usorted bin 中就可能存在合适的 small bin chunk 供分配，所以需要再次尝试。

```
/*
  Process recently freed or remaindered chunks, taking one only if
  it is exact fit, or, if this a small request, the chunk is remainder from
  the most recent non-exact fit.  Place other traversed chunks in
  bins.  Note that this step is the only place in any routine where
  chunks are placed in bins.

  The outer loop here is needed because we might not realize until
  near the end of malloc that we should have consolidated, so must
  do so and retry. This happens at most once, and only when we would
  otherwise need to expand memory to service a "small" request.
*/
for(;;) {
  int iters = 0;
  while ( (victim = unsorted_chunks(av)->bk) != unsorted_chunks(av)) {
```

反向遍历 unsorted bin 的双向循环链表，遍历结束的条件是循环链表中只剩下一个头结点。

```
    bck = victim->bk;
    if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
        || __builtin_expect (victim->size > av->system_mem, 0))
      malloc_printerr (check_action, "malloc(): memory corruption",
                       chunk2mem (victim));
    size = chunksize(victim);
```

检查当前遍历的 chunk 是否合法，chunk 的大小不能小于等于 2 * SIZE_SZ，也不能超过该分配区总的内存分配量。然后获取 chunk 的大小并赋值给 size。这里的检查似乎有点小问题，直接使用了 victim->size，但 victim->size 中包含了相关的标志位信息，使用 chunksize(victim)才比较合理，但在 unsorted bin 中的空闲 chunk 的所有标志位都清零了，所以这里直接 victim->size 没有问题。

```
    /*
       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)) {
```

如果需要分配一个 small bin chunk，在 5.7.2.2 节中的 small bins 中没有匹配到合适的 chunk，并且 unsorted bin 中只有一个 chunk，并且这个 chunk 为 last remainder chunk，并且这个 chunk 的大小大于所需 chunk 的大小加上 MINSIZE，在满足这些条件的情况下，可以使用这个 chunk 切分出需要的 small bin chunk。这是唯一的从 unsorted bin 中分配 small bin chunk 的情况，这种优化利于 cpu 的高速缓存命中。

```
    /* 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;
      }
```

从该 chunk 中切分出所需大小的 chunk，计算切分后剩下 chunk 的大小，将剩下的 chunk 加入 unsorted bin 的链表中，并将剩下的 chunk 作为分配区的 last remainder chunk，若剩下的 chunk 属于 large bin chunk，将该 chunk 的 fd_nextsize 和 bk_nextsize 设置为 NULL，因为这个 chunk 仅仅存在于 unsorted bin 中，并且 unsorted bin 中有且仅有这一个 chunk。

```
    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);
    if (__builtin_expect (perturb_byte, 0))
      alloc_perturb (p, bytes);
    return p;
```

设置分配出的 chunk 和 last remainder chunk 的相关信息，如 chunk 的 size，状态标志位，对于 last remainder chunk，需要调用 set_foot 宏，因为只有处于空闲状态的 chunk 的 foot 信息（prev_size）才是有效的，处于 inuse 状态的 chunk 的 foot 无效，该 foot 是返回给应用层的内存块的一部分。设置完成 chunk 的相关信息，调用 chunk2mem()获得 chunk 中可用的内存指针，返回给应用层，退出。

```
}

/* remove from unsorted list */
unsorted_chunks(av)->bk = bck;
bck->fd = unsorted_chunks(av);
```

将双向循环链表中的最后一个 chunk 移除。

```
/* Take now instead of binning if exact fit */
if (size == nb) {
  set_inuse_bit_at_offset(victim, size);
  if (av != &main_arena)
    victim->size |= NON_MAIN_ARENA;
  check_malloced_chunk(av, victim, nb);
  void *p = chunk2mem(victim);
  if (__builtin_expect (perturb_byte, 0))
    alloc_perturb (p, bytes);
  return p;
}
```

如果当前遍历的 chunk 与所需的 chunk 大小一致，将当前 chunk 返回。首先设置当前 chunk 处于 inuse 状态，该标志位处于相邻的下一个 chunk 的 size 中，如果当前分配区不是主分配区，设置当前 chunk 的非主分配区标志位，最后调用 chunk2mem()获得 chunk 中可用的内存指针，返回给应用层，退出。

```
/* place chunk in bin */
if (in_smallbin_range(size)) {
  victim_index = smallbin_index(size);
  bck = bin_at(av, victim_index);
  fwd = bck->fd;
```

如果当前 chunk 属于 small bins，获得当前 chunk 所属 small bin 的 index，并将该 small bin 的链表表头赋值给 bck，第一个 chunk 赋值给 fwd，也就是当前的 chunk 会插入到 bck 和 fwd 之间，作为 small bin 链表的第一个 chunk。

```
}
else {
  victim_index = largebin_index(size);
  bck = bin_at(av, victim_index);
  fwd = bck->fd;
```

如果当前 chunk 属于 large bins，获得当前 chunk 所属 large bin 的 index，并将该 large bin 的链表表头赋值给 bck，第一个 chunk 赋值给 fwd，也就是当前的 chunk 会插入到 bck 和 fwd 之间，作为 large bin 链表的第一个 chunk。

```
/* 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((bck->bk->size & NON_MAIN_ARENA) == 0);
```

如果 fwd 不等于 bck，意味着 large bin 中有空闲 chunk 存在，由于 large bin 中的空闲 chunk 是按照大小顺序排序的，需要将当前从 unsorted bin 中取出的 chunk 插入到 large bin 中合适的位置。将当前 chunk 的 size 的 inuse 标志 bit 置位，相当于加 1，便于加快 chunk 大小的比较，找到合适的地方插入当前 chunk。这里还做了一次检查，断言在 large bin 双向循环链表中的最后一个 chunk 的 size 字段中的非主分配区的标志 bit 没有置位，因为所有在 large bin 中的 chunk 都处于空闲状态，该标志位一定是清零的。

```
if ((unsigned long)(size) < (unsigned long)(bck->bk->size)) {
  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;
```

如果当前 chunk 比 large bin 的最后一个 chunk 的大小还小，那么当前 chunk 就插入到 large bin 的链表的最后，作为最后一个 chunk。可以看出 large bin 中的 chunk 是按照从大到小的顺序排序的，同时一个 chunk 存在于两个双向循环链表中，一个链表包含了 large bin 中所有的 chunk，另一个链表为 chunk size 链表，该链表从每个相同大小的 chunk 的取出第一个 chunk 按照大小顺序链接在一起，便于一次跨域多个相同大小的 chunk 遍历下一个不同大小的 chunk，这样可以加快在 large bin 链表中的遍历速度。

```
}
else {
  assert((fwd->size & NON_MAIN_ARENA) == 0);
  while ((unsigned long) size < fwd->size)
    {
      fwd = fwd->fd_nextsize;
      assert((fwd->size & NON_MAIN_ARENA) == 0);
    }
```

正向遍历 chunk size 链表，直到找到第一个 chunk 大小小于等于当前 chunk 大小的 chunk 退出循环。

```
  if ((unsigned long) size == (unsigned long) fwd->size)
    /* Always insert in the second position.  */
    fwd = fwd->fd;
```

如果从 large bin 链表中找到了与当前 chunk 大小相同的 chunk，则同一大小的 chunk 已经存在，那么 chunk size 链表中一定包含了 fwd 所指向的 chunk，为了不修改 chunk size 链表，当前 chunk 只能插入 fwd 之后。

```
  else
    {
      victim->fd_nextsize = fwd;
```

```
                victim->bk_nextsize = fwd->bk_nextsize;
                fwd->bk_nextsize = victim;
                victim->bk_nextsize->fd_nextsize = victim;
```

如果 chunk size 链表中还没有包含当前 chunk 大小的 chunk，也就是说当前 chunk 的大小大于 fwd 的大小，则将当前 chunk 作为该 chunk size 的代表加入 chunk size 链表，chunk size 链表也是按照由大到小的顺序排序。

```
              }
            bck = fwd->bk;
          }
        } else
          victim->fd_nextsize = victim->bk_nextsize = victim;
```

如果 large bin 链表中没有 chunk，直接将当前 chunk 加入 chunk size 链表。

```
      }

      mark_bin(av, victim_index);
      victim->bk = bck;
      victim->fd = fwd;
      fwd->bk = victim;
      bck->fd = victim;
```

上面的代码将当前 chunk 插入到 large bin 的空闲 chunk 链表中，并将 large bin 所对应 binmap 的相应 bit 置位。

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

如果 unsorted bin 中的 chunk 超过了 10000 个，最多遍历 10000 个就退出，避免长时间处理 unsorted bin 影响内存分配的效率。

```
    }
```

当将 unsorted bin 中的空闲 chunk 加入到相应的 small bins 和 large bins 后，将使用最佳匹配法分配 large bin chunk。源代码如下：

```
    /*
      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)(victim->size) >= (unsigned long)(nb)) {
```

如果所需分配的 chunk 为 large bin chunk，查询对应的 large bin 链表，如果 large bin 链表为空，或者链表中最大的 chunk 也不能满足要求，则不能从 large bin 中分配。否则，遍历

large bin 链表，找到合适的 chunk。

```
victim = victim->bk_nextsize;
while (((unsigned long)(size = chunksize(victim)) <
        (unsigned long)(nb)))
  victim = victim->bk_nextsize;
```

反向遍历 chunk size 链表，直到找到第一个大于等于所需 chunk 大小的 chunk 退出循环。

```
/* Avoid removing the first entry for a size so that the skip
   list does not have to be rerouted.  */
if (victim != last(bin) && victim->size == victim->fd->size)
  victim = victim->fd;
```

如果从 large bin 链表中选取的 chunk victim 不是链表中的最后一个 chunk，并且与 victim 大小相同的 chunk 不止一个，那么意味着 victim 为 chunk size 链表中的节点，为了不调整 chunk size 链表，需要避免将 chunk size 链表中的节点取出，所以取 victim->fd 节点对应的 chunk 作为候选 chunk。由于 large bin 链表中的 chunk 也是按大小排序，同一大小的 chunk 有多个时，这些 chunk 必定排在一起，所以 victim->fd 节点对应的 chunk 的大小必定与 victim 的大小一样。

```
remainder_size = size - nb;
unlink(victim, bck, fwd);
```

计算将 victim 切分后剩余大小，并调用 unlink()宏函数将 victim 从 large bin 链表中取出。

```
/* Exhaust */
if (remainder_size < MINSIZE)  {
  set_inuse_bit_at_offset(victim, size);
  if (av != &main_arena)
    victim->size |= NON_MAIN_ARENA;
```

如果将 victim 切分后剩余大小小于 MINSIZE，则将真个 victim 分配给应用层，这种情况下，实际分配的 chunk 比所需的 chunk 要大一些。以 64 位系统为例，remainder_size 的可能大小为 0 和 16，如果为 0，表示 victim 的大小刚好等于所需 chunk 的大小，设置 victim 的 inuse 标志，inuse 标志位于下一个相邻的 chunk 的 size 字段中。如果 remainder_size 为 16，则这 16 字节就浪费掉了。如果当前分配区不是主分配区，将 victim 的 size 字段中的非主分配区标志置位。

```
}
/* Split */
else {
  remainder = chunk_at_offset(victim, nb);
  /* We cannot assume the unsorted list is empty and therefore
     have to perform a complete insert here.  */
  bck = unsorted_chunks(av);
  fwd = bck->fd;
  if (__builtin_expect (fwd->bk != bck, 0))
```

```
        {
          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;
        }
```

从 victim 中切分出所需的 chunk，剩余部分作为一个新的 chunk 加入到 unsorted bin 中。如果剩余部分 chunk 属于 large bins，将剩余部分 chunk 的 chunk size 链表指针设置为 NULL，因为 unsorted bin 中的 chunk 是不排序的，这两个指针无用，必须清零。

```
      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);
```

设置 victim 和 remainder 的状态，由于 remainder 为空闲 chunk，所以需要设置该 chunk 的 foot。

```
    }
    check_malloced_chunk(av, victim, nb);
    void *p = chunk2mem(victim);
    if (__builtin_expect (perturb_byte, 0))
      alloc_perturb (p, bytes);
    return p;
```

从 large bin 中使用最佳匹配法找到了合适的 chunk，调用 chunk2mem()获得 chunk 中可用的内存指针，返回给应用层，退出。

```
  }
}
```

如果通过上面的方式从最合适的 small bin 或 large bin 中都没有分配到需要的 chunk，则查看比当前 bin 的 index 大的 small bin 或 large bin 是否有空闲 chunk 可利用来分配所需的 chunk。源代码实现如下：

```
/*
  Search for a chunk by scanning bins, starting with next largest
  bin. This search is strictly by best-fit; i.e., the smallest
  (with ties going to approximately the least recently used) chunk
  that fits is selected.
```

```
     The bitmap avoids needing to check that most blocks are nonempty.
     The particular case of skipping all bins during warm-up phases
     when no chunks have been returned yet is faster than it might look.
   */
   ++idx;
   bin = bin_at(av,idx);
   block = idx2block(idx);
   map = av->binmap[block];
   bit = idx2bit(idx);
```

获取下一个相邻bin的空闲chunk链表，并获取该bin对于binmap中的bit位的值。Binmap中的标识了相应的 bin 中是否有空闲 chunk 存在。Binmap 按 block 管理，每个 block 为一个 int，共 32 个 bit，可以表示 32 个 bin 中是否有空闲 chunk 存在。使用 binmap 可以加快查找 bin 是否包含空闲 chunk。这里只查询比所需 chunk 大的 bin 中是否有空闲 chunk 可用。

```
   for (;;) {
     /* Skip rest of block if there are no more set bits in this block.  */
     if (bit > map || bit == 0) {
       do {
         if (++block >= BINMAPSIZE)  /* out of bins */
           goto use_top;
       } while ( (map = av->binmap[block]) == 0);

       bin = bin_at(av, (block << BINMAPSHIFT));
       bit = 1;
     }
```

Idx2bit()宏将 idx 指定的位设置为 1，其它位清零，map 表示一个 block（unsigned int）值，如果 bit 大于 map，意味着 map 为 0，该 block 所对应的所有 bins 中都没有空闲 chunk，于是遍历 binmap 的下一个 block，直到找到一个不为 0 的 block 或者遍历完所有的 block。退出循环遍历后，设置 bin 指向 block 的第一个 bit 对应的 bin，并将 bit 置为 1，表示该 block 中 bit 1 对应的 bin，这个 bin 中如果有空闲 chunk，该 chunk 的大小一定满足要求。

```
     /* Advance to bin with set bit. There must be one. */
     while ((bit & map) == 0) {
       bin = next_bin(bin);
       bit <<= 1;
       assert(bit != 0);
     }
```

在一个 block 遍历对应的 bin，直到找到一个 bit 不为 0 退出遍历，则该 bit 对于的 bin 中有空闲 chunk 存在。

```
     /* Inspect the bin. It is likely to be non-empty */
     victim = last(bin);
```

将 bin 链表中的最后一个 chunk 赋值为 victim。

```
/*  If a false alarm (empty bin), clear the bit. */
if (victim == bin) {
  av->binmap[block] = map &= ~bit; /* Write through */
  bin = next_bin(bin);
  bit <<= 1;
```

如果 victim 与 bin 链表头指针相同，表示该 bin 中没有空闲 chunk，binmap 中的相应位设置不准确，将 binmap 的相应 bit 位清零，获取当前 bin 下一个 bin，将 bit 移到下一个 bit 位，即乘以 2。

```
}
else {
  size = chunksize(victim);
  /*  We know the first chunk in this bin is big enough to use. */
  assert((unsigned long)(size) >= (unsigned long)(nb));
  remainder_size = size - nb;
  /* unlink */
  unlink(victim, bck, fwd);
```

当前 bin 中的最后一个 chunk 满足要求，获取该 chunk 的大小，计算切分出所需 chunk 后剩余部分的大小，然后将 victim 从 bin 的链表中取出。

```
  /* Exhaust */
  if (remainder_size < MINSIZE) {
    set_inuse_bit_at_offset(victim, size);
    if (av != &main_arena)
      victim->size |= NON_MAIN_ARENA;
```

如果剩余部分的大小小于 MINSIZE，将整个 chunk 分配给应用层，设置 victim 的状态为 inuse，如果当前分配区为非主分配区，设置 victim 的非主分配区标志位。

```
  }
  /* Split */
  else {
    remainder = chunk_at_offset(victim, nb);

    /* We cannot assume the unsorted list is empty and therefore
       have to perform a complete insert here.  */
    bck = unsorted_chunks(av);
    fwd = bck->fd;
    if (__builtin_expect (fwd->bk != bck, 0))
      {
        errstr = "malloc(): corrupted unsorted chunks 2";
        goto errout;
      }
    remainder->bk = bck;
    remainder->fd = fwd;
```

```
        bck->fd = remainder;
        fwd->bk = remainder;

        /* advertise as last remainder */
        if (in_smallbin_range(nb))
          av->last_remainder = remainder;
        if (!in_smallbin_range(remainder_size))
          {
            remainder->fd_nextsize = NULL;
            remainder->bk_nextsize = NULL;
          }
```

从 victim 中切分出所需的 chunk，剩余部分作为一个新的 chunk 加入到 unsorted bin 中。如果剩余部分 chunk 属于 small bins，将分配区的 last remainder chunk 设置为剩余部分构成的 chunk；如果剩余部分 chunk 属于 large bins，将剩余部分 chunk 的 chunk size 链表指针设置为 NULL，因为 unsorted bin 中的 chunk 是不排序的，这两个指针无用，必须清零。

```
        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);
```

设置 victim 和 remainder 的状态，由于 remainder 为空闲 chunk，所以需要设置该 chunk 的 foot。

```
      }
      check_malloced_chunk(av, victim, nb);
      void *p = chunk2mem(victim);
      if (__builtin_expect (perturb_byte, 0))
        alloc_perturb (p, bytes);
      return p;
```

调用 chunk2mem()获得 chunk 中可用的内存指针，返回给应用层，退出。

```
    }
  }
```

如果从所有的 bins 中都没有获得所需的 chunk，可能的情况为 bins 中没有空闲 chunk，或者所需的 chunk 大小很大，下一步将尝试从 top chunk 中分配所需 chunk。源代码实现如下：

```
  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);
```

将当前分配区的 top chunk 赋值给 victim，并获得 victim 的大小。

```
    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);
      if (__builtin_expect (perturb_byte, 0))
        alloc_perturb (p, bytes);
      return p;
    }
```

由于 top chunk 切分出所需 chunk 后，还需要 MINSIZE 的空间来作为 fencepost，所需必须满足 top chunk 的大小大于所需 chunk 的大小加上 MINSIZE 这个条件，才能从 top chunk 中分配所需 chunk。从 top chunk 切分出所需 chunk 的处理过程跟前面的 chunk 切分类似，不同的是，原 top chunk 切分后的剩余部分将作为新的 top chunk，原 top chunk 的 fencepost 仍然作为新的 top chunk 的 fencepost，所以切分之后剩余的 chunk 不用 set_foot。

```
#ifdef ATOMIC_FASTBINS
    /* When we are using atomic ops to free fast chunks we can get
       here for all block sizes.  */
    else if (have_fastchunks(av)) {
      malloc_consolidate(av);
      /* restore original bin index */
      if (in_smallbin_range(nb))
        idx = smallbin_index(nb);
      else
        idx = largebin_index(nb);
    }
```

如果 top chunk 也不能满足要求，查看 fast bins 中是否有空闲 chunk 存在，由于开启了 ATOMIC_FASTBINS 优化情况下，free 属于 fast bins 的 chunk 时不需要获得分配区的锁，所以

在调用_int_malloc()函数时，有可能有其它线程已经向 fast bins 中加入了新的空闲 chunk，也有可能是所需的 chunk 属于 small bins，但通过前面的步骤都没有分配到所需的 chunk，由于分配 small bin chunk 时在前面的步骤都不会调用 malloc_consolidate()函数将 fast bins 中的 chunk 合并加入到 unsorted bin 中。所在这里如果 fast bin 中有 chunk 存在，调用 malloc_consolidate()函数，并重新设置当前 bin 的 index。并转到最外层的循环，尝试重新分配 small bin chunk 或是 large bin chunk。如果开启了 ATOMIC_FASTBINS 优化，有可能在由其它线程加入到 fast bins 中的 chunk 被合并后加入 unsorted bin 中，从 unsorted bin 中就可以分配出所需的 large bin chunk 了，所以对没有成功分配的 large bin chunk 也需要重试。

```
#else
    /*
      If there is space available in fastbins, consolidate and retry,
      to possibly avoid expanding memory. This can occur only if nb is
      in smallbin range so we didn't consolidate upon entry.
    */
    else if (have_fastchunks(av)) {
      assert(in_smallbin_range(nb));
      malloc_consolidate(av);
      idx = smallbin_index(nb); /* restore original bin index */
    }
```

如果 top chunk 也不能满足要求，查看 fast bins 中是否有空闲 chunk 存在，如果 fast bins 中有空闲 chunk 存在，在没有开启 ATOMIC_FASTBINS 优化的情况下，只有一种可能，那就是所需的 chunk 属于 small bins，但通过前面的步骤都没有分配到所需的 small bin chunk，由于分配 small bin chunk 时在前面的步骤都不会调用 malloc_consolidate()函数将 fast bins 中的空闲 chunk 合并加入到 unsorted bin 中。所在这里如果 fast bins 中有空闲 chunk 存在，调用 malloc_consolidate()函数，并重新设置当前 bin 的 index。并转到最外层的循环，尝试重新分配 small bin chunk。

```
#endif
    /*
       Otherwise, relay to handle system-dependent cases
    */
    else {
      void *p = sYSMALLOc(nb, av);
      if (p != NULL && __builtin_expect (perturb_byte, 0))
        alloc_perturb (p, bytes);
      return p;
```

山穷水尽了，只能想系统申请内存了。sYSMALLOc()函数可能分配的 chunk 包括 small bin chunk，large bin chunk 和 large chunk。将在下一节中介绍该函数的实现。

```
    }
  }
```

至此，_int_malloc()函数的代码就罗列完了，当还有两个关键函数没有分析，一个为 malloc_consolidate()，另一个为 sYSMALLOc()，将在下面的章节介绍其实现。

5.7.2.4 sYSMALLOc()

当_int_malloc()函数尝试从 fast bins，last remainder chunk，small bins，large bins 和 top chunk 都失败之后，就会使用 sYSMALLOc()函数直接向系统申请内存用于分配所需的 chunk。其实现源代码如下：

```
/*
  sysmalloc handles malloc cases requiring more memory from the system.
  On entry, it is assumed that av->top does not have enough
  space to service request for nb bytes, thus requiring that av->top
  be extended or replaced.
*/
#if __STD_C
static Void_t* sYSMALLOc(INTERNAL_SIZE_T nb, mstate av)
#else
static Void_t* sYSMALLOc(nb, av) INTERNAL_SIZE_T nb; mstate av;
#endif
{
  mchunkptr       old_top;        /* incoming value of av->top */
  INTERNAL_SIZE_T old_size;       /* its size */
  char*           old_end;        /* its end address */

  long            size;           /* arg to first MORECORE or mmap call */
  char*           brk;            /* return value from MORECORE */

  long            correction;     /* arg to 2nd MORECORE call */
  char*           snd_brk;        /* 2nd return val */

  INTERNAL_SIZE_T front_misalign; /* unusable bytes at front of new space */
  INTERNAL_SIZE_T end_misalign;   /* partial page left at end of new space */
  char*           aligned_brk;    /* aligned offset into brk */

  mchunkptr       p;              /* the allocated/returned chunk */
  mchunkptr       remainder;      /* remainder from allocation */
  unsigned long   remainder_size; /* its size */

  unsigned long   sum;            /* for updating stats */

  size_t          pagemask  = mp_.pagesize - 1;
  bool            tried_mmap = false;

#if HAVE_MMAP
  /*
    If have mmap, and the request size meets the mmap threshold, and
```

```
    the system supports mmap, and there are few enough currently
    allocated mmapped regions, try to directly map this request
    rather than expanding top.
  */
  if ((unsigned long)(nb) >= (unsigned long)(mp_.mmap_threshold) &&
      (mp_.n_mmaps < mp_.n_mmaps_max)) {
    char* mm;             /* return value from mmap call*/
```

如果所需分配的 chunk 大小大于 mmap 分配阈值，默认为 128K，并且当前进程使用 mmap()分配的内存块小于设定的最大值,将使用 mmap()系统调用直接向操作系统申请内存。

```
  try_mmap:
    /*
      Round up size to nearest page.  For mmapped chunks, the overhead
      is one SIZE_SZ unit larger than for normal chunks, because there
      is no following chunk whose prev_size field could be used.
    */
#if 1
    /* See the front_misalign handling below, for glibc there is no
       need for further alignments.  */
    size = (nb + SIZE_SZ + pagemask) & ~pagemask;
#else
    size = (nb + SIZE_SZ + MALLOC_ALIGN_MASK + pagemask) & ~pagemask;
#endif
    tried_mmap = true;
```

由于 nb 为所需 chunk 的大小，在_int_malloc()函数中已经将用户需要分配的大小转化为 chunk 大小，当如果这个 chunk 直接使用 mmap()分配的话，该 chunk 不存在下一个相邻的 chunk，也就没有 prev_size 的内存空间可以复用，所以还需要额外 SIZE_SZ 大小的内存。由于 mmap()分配的内存块必须页对齐。如果使用 mmap()分配内存，需要重新计算分配的内存大小 size。

```
    /* Don't try if size wraps around 0 */
    if ((unsigned long)(size) > (unsigned long)(nb)) {
      mm = (char*)(MMAP(0, size, PROT_READ|PROT_WRITE, MAP_PRIVATE));
      if (mm != MAP_FAILED) {
        /*
          The offset to the start of the mmapped region is stored
          in the prev_size field of the chunk. This allows us to adjust
          returned start address to meet alignment requirements here
          and in memalign(), and still be able to compute proper
          address argument for later munmap in free() and realloc().
        */
#if 1
        /* For glibc, chunk2mem increases the address by 2*SIZE_SZ and
           MALLOC_ALIGN_MASK is 2*SIZE_SZ-1.  Each mmap'ed area is page
```

```
           aligned and therefore definitely MALLOC_ALIGN_MASK-aligned.  */
        assert (((INTERNAL_SIZE_T)chunk2mem(mm) & MALLOC_ALIGN_MASK) == 0);
#else
        front_misalign = (INTERNAL_SIZE_T)chunk2mem(mm) & MALLOC_ALIGN_MASK;
        if (front_misalign > 0) {
          correction = MALLOC_ALIGNMENT - front_misalign;
          p = (mchunkptr)(mm + correction);
          p->prev_size = correction;
          set_head(p, (size - correction) |IS_MMAPPED);
        }
        else
#endif
          {
            p = (mchunkptr)mm;
            set_head(p, size|IS_MMAPPED);
          }
```

如果重新计算所需分配的 size 小于 nb，表示溢出了，不分配内存，否则，调用 mmap()分配所需大小的内存。如果 mmap()分配内存成功，将 mmap()返回的内存指针强制转换为 chunk 指针，并设置该 chunk 的大小为 size，同时设置该 chunk 的 IS_MMAPPED 标志位，表示本 chunk 是通过 mmap()函数直接从系统分配的。由于 mmap()返回的内存地址是按照页对齐的，也一定是按照 2*SIZE_SZ 对齐的，满足 chunk 的边界对齐规则，使用 chunk2mem()获取 chunk 中实际可用的内存也没有问题，所以这里不需要做额外的对齐操作。

```
        /* update statistics */
        if (++mp_.n_mmaps > mp_.max_n_mmaps)
          mp_.max_n_mmaps = mp_.n_mmaps;
        sum = mp_.mmapped_mem += size;
        if (sum > (unsigned long)(mp_.max_mmapped_mem))
          mp_.max_mmapped_mem = sum;
#ifdef NO_THREADS
        sum += av->system_mem;
        if (sum > (unsigned long)(mp_.max_total_mem))
          mp_.max_total_mem = sum;
#endif
```

更新相关统计值，首先将当前进程 mmap 分配内存块的计数加一，如果使用 mmap()分配的内存块数量大于设置的最大值，将最大值设置为最新值，这个判断不会成功，因为使用 mmap 分配内存的条件中包括了 mp_.n_mmaps < mp_.n_mmaps_max，所以++mp_.n_mmaps > mp_.max_n_mmaps 不会成立。然后更新 mmap 分配的内存总量，如果该值大于设置的最大值，将当前值赋值给 mp_.max_mmapped_mem。如果只支持单线程，还需要计数当前进程所分配的内存总数，如果总数大于设置的最大值 mp_.max_total_mem，修改 mp_.max_total_mem 为当前值。

```
        check_chunk(av, p);
        return chunk2mem(p);
```

```
      }
    }
  }
#endif

  /* Record incoming configuration of top */
  old_top  = av->top;
  old_size = chunksize(old_top);
  old_end  = (char*)(chunk_at_offset(old_top, old_size));
  brk = snd_brk = (char*)(MORECORE_FAILURE);
```

保存当前 top chunk 的指针，大小和结束地址到临时变量中。

```
  /*
     If not the first time through, we require old_size to be
     at least MINSIZE and to have prev_inuse set.
  */
  assert((old_top == initial_top(av) && old_size == 0) ||
         ((unsigned long) (old_size) >= MINSIZE &&
          prev_inuse(old_top) &&
          ((unsigned long)old_end & pagemask) == 0));
  /* Precondition: not enough current space to satisfy nb request */
  assert((unsigned long)(old_size) < (unsigned long)(nb + MINSIZE));
#ifndef ATOMIC_FASTBINS
  /* Precondition: all fastbins are consolidated */
  assert(!have_fastchunks(av));
#endif
```

检查 top chunk 的合法性，如果第一次调用本函数，top chunk 可能没有初始化，可能 old_size 为 0，如果 top chunk 已经初始化，则 top chunk 的大小必须大于等于 MINSIZE，因为 top chunk 中包含了 fencepost，fencepost 需要 MINSIZE 大小的内存。Top chunk 必须标识前一个 chunk 处于 inuse 状态，这是规定，并且 top chunk 的结束地址必定是页对齐的。另外 top chunk 的除去 fencepost 的大小必定小于所需 chunk 的大小，不然在_int_malloc()函数中就应该使用 top chunk 获得所需的 chunk。最后检查如果没有开启 ATOMIC_FASTBINS 优化，在使用_int_malloc()分配内存时，获得了分配区的锁，free 时也要获得分配区的锁才能向 fast bins 中加入新的 chunk，由于_int_malloc()在调用本函数前，已经将 fast bins 中的所有 chunk 都合并加入到 unsorted bin 中了，所以，本函数中 fast bins 中一定不会有空闲 chunk 存在。

```
  if (av != &main_arena) {
    heap_info *old_heap, *heap;
    size_t old_heap_size;

    /* First try to extend the current heap. */
    old_heap = heap_for_ptr(old_top);
    old_heap_size = old_heap->size;
    if ((long) (MINSIZE + nb - old_size) > 0
```

```
        && grow_heap(old_heap, MINSIZE + nb - old_size) == 0) {
      av->system_mem += old_heap->size - old_heap_size;
      arena_mem += old_heap->size - old_heap_size;
#if 0
      if(mmapped_mem + arena_mem + sbrked_mem > max_total_mem)
        max_total_mem = mmapped_mem + arena_mem + sbrked_mem;
#endif
      set_head(old_top, (((char *)old_heap + old_heap->size) - (char *)old_top)
               | PREV_INUSE);
```

如果当前分配区为非主分配区，根据 top chunk 的指针获得当前 sub_heap 的 heap_info 实例，如果 top chunk 的剩余有效空间不足以分配出所需的 chunk（前面已经断言，这个肯定成立），尝试增长 sub_heap 的可读可写区域大小，如果成功，修改过内存分配的统计信息，并更新新的 top chunk 的 size。

```
    }
    else if ((heap = new_heap(nb + (MINSIZE + sizeof(*heap)), mp_.top_pad))) {
```

调用 new_heap()函数创建一个新的 sub_heap，由于这个 sub_heap 中至少需要容下大小为 nb 的 chunk，大小为 MINSIZE 的 fencepost 和大小为 sizeof(*heap)的 heap_info 实例，所以传入 new_heap()函数的分配大小为 nb + (MINSIZE + sizeof(*heap))。

```
      /* Use a newly allocated heap.  */
      heap->ar_ptr = av;
      heap->prev = old_heap;
      av->system_mem += heap->size;
      arena_mem += heap->size;
#if 0
      if((unsigned long)(mmapped_mem + arena_mem + sbrked_mem) > max_total_mem)
        max_total_mem = mmapped_mem + arena_mem + sbrked_mem;
#endif
      /* Set up the new top.  */
      top(av) = chunk_at_offset(heap, sizeof(*heap));
      set_head(top(av), (heap->size - sizeof(*heap)) | PREV_INUSE);
```

使新创建的 sub_heap 保存当前的分配区指针，将该 sub_heap 加入当前分配区的 sub_heap 链表中，更新当前分配区内存分配统计，将新创建的 sub_heap 仅有的一个空闲 chunk 作为当前分配区的 top chunk，并设置 top chunk 的状态。

```
      /* Setup fencepost and free the old top chunk. */
      /* The fencepost takes at least MINSIZE bytes, because it might
         become the top chunk again later.  Note that a footer is set
         up, too, although the chunk is marked in use. */
      old_size -= MINSIZE;
      set_head(chunk_at_offset(old_top, old_size + 2*SIZE_SZ), 0|PREV_INUSE);
      if (old_size >= MINSIZE) {
        set_head(chunk_at_offset(old_top, old_size), (2*SIZE_SZ)|PREV_INUSE);
```

```
        set_foot(chunk_at_offset(old_top, old_size), (2*SIZE_SZ));
        set_head(old_top, old_size|PREV_INUSE|NON_MAIN_ARENA);
#ifdef ATOMIC_FASTBINS
        _int_free(av, old_top, 1);
#else
        _int_free(av, old_top);
#endif
      } else {
        set_head(old_top, (old_size + 2*SIZE_SZ)|PREV_INUSE);
        set_foot(old_top, (old_size + 2*SIZE_SZ));
      }
```

设置原 top chunk 的 fencepost，fencepost 需要 MINSIZE 大小的内存空间，将该 old_size 减去 MINSIZE 得到原 top chunk 的有效内存空间，首先设置 fencepost 的第二个 chunk 的 size 为 0，并标识前一个 chunk 处于 inuse 状态。接着判断原 top chunk 的有效内存空间上是否大于等于 MINSIZE，如果是，表示原 top chunk 可以分配出大于等于 MINSIZE 大小的 chunk，于是将原 top chunk 切分成空闲 chunk 和 fencepost 两部分，先设置 fencepost 的第一个 chunk 的大小为 2*SIZE_SZ，并标识前一个 chunk 处于 inuse 状态，fencepost 的第一个 chunk 还需要设置 foot，表示该 chunk 处于空闲状态，而 fencepost 的第二个 chunk 却标识第一个 chunk 处于 inuse 状态，因为不能有两个空闲 chunk 相邻，才会出现这么奇怪的 fencepost。另外其实 top chunk 切分出来的 chunk 也是处于空闲状态，但 fencepost 的第一个 chunk 却标识前一个 chunk 为 inuse 状态，然后强制将该处于 inuse 状态的 chunk 调用_int_free()函数释放掉。这样做完全是要遵循不能有两个空闲 chunk 相邻的约定。

如果原 top chunk 中有效空间不足 MINSIZE，则将整个原 top chunk 作为 fencepost，并设置 fencepost 的第一个 chunk 的相关状态。

```
    }
    else if (!tried_mmap)
      /* We can at least try to use to mmap memory.  */
      goto try_mmap;
```

如果增长 sub_heap 的可读可写区域大小和创建新 sub_heap 都失败了，尝试使用 mmap()函数直接从系统分配所需 chunk。

```
  } else { /* av == main_arena */
  /* Request enough space for nb + pad + overhead */
  size = nb + mp_.top_pad + MINSIZE;
```

如果为当前分配区为主分配区，重新计算需要分配的 size。

```
  /*
    If contiguous, we can subtract out existing space that we hope to
    combine with new space. We add it back later only if
    we don't actually get contiguous space.
  */
  if (contiguous(av))
    size -= old_size;
```

一般情况下，主分配区使用 sbrk()从 heap 中分配内存，sbrk()返回连续的虚拟内存，这

里调整需要分配的 size，减掉 top chunk 中已有空闲内存大小。

```
  /*
    Round to a multiple of page size.
    If MORECORE is not contiguous, this ensures that we only call it
    with whole-page arguments.  And if MORECORE is contiguous and
    this is not first time through, this preserves page-alignment of
    previous calls. Otherwise, we correct to page-align below.
  */
  size = (size + pagemask) & ~pagemask;
```

将 size 按照页对齐，sbrk()必须以页为单位分配连续虚拟内存。

```
  /*
    Don't try to call MORECORE if argument is so big as to appear
    negative. Note that since mmap takes size_t arg, it may succeed
    below even if we cannot call MORECORE.
  */
  if (size > 0)
    brk = (char*)(MORECORE(size));
```

使用 sbrk()从 heap 中分配 size 大小的虚拟内存块。

```
  if (brk != (char*)(MORECORE_FAILURE)) {
    /* Call the `morecore' hook if necessary.  */
    void (*hook) (void) = force_reg (__after_morecore_hook);
    if (__builtin_expect (hook != NULL, 0))
      (*hook) ();
```

如果 sbrk()分配成功，并且 morecore 的 hook 函数存在，调用 morecore 的 hook 函数。

```
  } else {
  /*
    If have mmap, try using it as a backup when MORECORE fails or
    cannot be used. This is worth doing on systems that have "holes" in
    address space, so sbrk cannot extend to give contiguous space, but
    space is available elsewhere.  Note that we ignore mmap max count
    and threshold limits, since the space will not be used as a
    segregated mmap region.
  */
#if HAVE_MMAP
    /* Cannot merge with old top, so add its size back in */
    if (contiguous(av))
      size = (size + old_size + pagemask) & ~pagemask;

    /* If we are relying on mmap as backup, then use larger units */
    if ((unsigned long)(size) < (unsigned long)(MMAP_AS_MORECORE_SIZE))
```

```
      size = MMAP_AS_MORECORE_SIZE;
```

如果 sbrk()返回失败，或是 sbrk()不可用，使用 mmap()代替，重新计算所需分配的内存大小并按页对齐，如果重新计算的 size 小于 1M，将 size 设为 1M，也就是说使用 mmap()作为 morecore 函数分配的最小内存块大小为 1M。

```
    /* Don't try if size wraps around 0 */
    if ((unsigned long)(size) > (unsigned long)(nb)) {
      char *mbrk = (char*)(MMAP(0, size, PROT_READ|PROT_WRITE, MAP_PRIVATE));
      if (mbrk != MAP_FAILED) {
        /* We do not need, and cannot use, another sbrk call to find end */
        brk = mbrk;
        snd_brk = brk + size;

        /*
           Record that we no longer have a contiguous sbrk region.
           After the first time mmap is used as backup, we do not
           ever rely on contiguous space since this could incorrectly
           bridge regions.
        */
        set_noncontiguous(av);
      }
```

如果所需分配的内存大小合法，使用 mmap()函数分配内存。如果分配成功，更新 brk 和 snd_brk，并将当前分配区属性设置为可分配不连续虚拟内存块。

```
    }
#endif
  }

  if (brk != (char*)(MORECORE_FAILURE)) {
    if (mp_.sbrk_base == 0)
      mp_.sbrk_base = brk;
    av->system_mem += size;
```

如果 brk 合法，即 sbrk()或 mmap()分配成功，如果 sbrk_base 还没有初始化，更新 sbrk_base 和当前分配区的内存分配总量。

```
    /*
      If MORECORE extends previous space, we can likewise extend top size.
    */
    if (brk == old_end && snd_brk == (char*)(MORECORE_FAILURE))
      set_head(old_top, (size + old_size) | PREV_INUSE);
    else if (contiguous(av) && old_size && brk < old_end) {
      /* Oops!  Someone else killed our space..  Can't touch anything.  */
      malloc_printerr (3, "break adjusted to free malloc space", brk);
    }
```

如果 sbrk()分配成功，更新 top chunk 的大小，并设定 top chunk 的前一个 chunk 处于 inuse 状态。如果当前分配区可分配连续虚拟内存，原 top chunk 的大小大于 0，但新的 brk 值小于原 top chunk 的结束地址，出错了。

```
/*
  Otherwise, make adjustments:
  * If the first time through or noncontiguous, we need to call sbrk
    just to find out where the end of memory lies.
  * We need to ensure that all returned chunks from malloc will meet
    MALLOC_ALIGNMENT
  * If there was an intervening foreign sbrk, we need to adjust sbrk
    request size to account for fact that we will not be able to
    combine new space with existing space in old_top.
  * Almost all systems internally allocate whole pages at a time, in
    which case we might as well use the whole last page of request.
    So we allocate enough more memory to hit a page boundary now,
    which in turn causes future contiguous calls to page-align.
*/
else {
  front_misalign = 0;
  end_misalign = 0;
  correction = 0;
  aligned_brk = brk;
```

执行到这个分支，意味着 sbrk()返回的 brk 值大于原 top chunk 的结束地址，那么新的地址与原 top chunk 的地址不连续，可能是由于外部其它地方调用 sbrk()函数，这里需要处理地址的重新对齐问题。

```
  /* handle contiguous cases */
  if (contiguous(av)) {
    /* Count foreign sbrk as system_mem.  */
    if (old_size)
      av->system_mem += brk - old_end;
```

如果本分配区可分配连续虚拟内存，并且有外部调用了 sbrk()函数，将外部调用 sbrk()分配的内存计入当前分配区所分配内存统计中。

```
    /* Guarantee alignment of first new chunk made from this space */
    front_misalign = (INTERNAL_SIZE_T)chunk2mem(brk) & MALLOC_ALIGN_MASK;
    if (front_misalign > 0) {

      /*
        Skip over some bytes to arrive at an aligned position.
        We don't need to specially mark these wasted front bytes.
        They will never be accessed anyway because
        prev_inuse of av->top (and any chunk created from its start)
```

```
      is always true after initialization.
    */
    correction = MALLOC_ALIGNMENT - front_misalign;
    aligned_brk += correction;
  }
```

计算当前的 brk 要矫正的字节数据，保证 brk 地址按 MALLOC_ALIGNMENT 对齐。

```
  /*
    If this isn't adjacent to existing space, then we will not
    be able to merge with old_top space, so must add to 2nd request.
  */
  correction += old_size;

  /* Extend the end address to hit a page boundary */
  end_misalign = (INTERNAL_SIZE_T)(brk + size + correction);
  correction += ((end_misalign + pagemask) & ~pagemask) - end_misalign;

  assert(correction >= 0);
  snd_brk = (char*)(MORECORE(correction));
```

由于原 top chunk 的地址与当前 brk 不相邻，也就不能再使用原 top chunk 的内存了，需要重新为所需 chunk 分配足够的内存，将原 top chunk 的大小加到矫正值中，从当前 brk 中分配所需 chunk，计算出未对齐的 chunk 结束地址 end_misalign，然后将 end_misalign 按照页对齐计算出需要矫正的字节数加到矫正值上。然后再调用 sbrk()分配矫正值大小的内存，如果 sbrk()分配成功，则当前的 top chunk 中可以分配出所需的连续内存的 chunk。

```
  /*
    If can't allocate correction, try to at least find out current
    brk.  It might be enough to proceed without failing.

    Note that if second sbrk did NOT fail, we assume that space
    is contiguous with first sbrk. This is a safe assumption unless
    program is multithreaded but doesn't use locks and a foreign sbrk
    occurred between our first and second calls.
  */
  if (snd_brk == (char*)(MORECORE_FAILURE)) {
    correction = 0;
    snd_brk = (char*)(MORECORE(0));
```

如果 sbrk()执行失败，更新当前 brk 的结束地址。

```
  } else {
    /* Call the `morecore' hook if necessary.  */
    void (*hook) (void) = force_reg (__after_morecore_hook);
    if (__builtin_expect (hook != NULL, 0))
      (*hook) ();
```

如果 sbrk()执行成功，并且有 morecore hook 函数存在，执行该 hook 函数。

```
    }
  }

  /* handle non-contiguous cases */
  else {
    /* MORECORE/mmap must correctly align */
    assert(((unsigned long)chunk2mem(brk) & MALLOC_ALIGN_MASK) == 0);

    /* Find out current end of memory */
    if (snd_brk == (char*)(MORECORE_FAILURE)) {
      snd_brk = (char*)(MORECORE(0));
    }
```

执行到这里，意味着 brk 是用 mmap()分配的，断言 brk 一定是按 MALLOC_ALIGNMENT 对齐的，因为 mmap()返回的地址按页对齐。如果 brk 的结束地址非法，使用 morecore 获得当前 brk 的结束地址。

```
  }

  /* Adjust top based on results of second sbrk */
  if (snd_brk != (char*)(MORECORE_FAILURE)) {
    av->top = (mchunkptr)aligned_brk;
    set_head(av->top, (snd_brk - aligned_brk + correction) | PREV_INUSE);
    av->system_mem += correction;
```

如果 brk 的结束地址合法，设置当前分配区的 top chunk 为 brk，设置 top chunk 的大小，并更新分配区的总分配内存量。

```
    /*
      If not the first time through, we either have a
      gap due to foreign sbrk or a non-contiguous region.  Insert a
      double fencepost at old_top to prevent consolidation with space
      we don't own. These fenceposts are artificial chunks that are
      marked as inuse and are in any case too small to use.  We need
      two to make sizes and alignments work out.
    */
    if (old_size != 0) {
      /*
         Shrink old_top to insert fenceposts, keeping size a
         multiple of MALLOC_ALIGNMENT. We know there is at least
         enough space in old_top to do this.
      */
      old_size = (old_size - 4*SIZE_SZ) & ~MALLOC_ALIGN_MASK;
      set_head(old_top, old_size | PREV_INUSE);
```

```
          /*
            Note that the following assignments completely overwrite
            old_top when old_size was previously MINSIZE.  This is
            intentional. We need the fencepost, even if old_top otherwise gets
            lost.
          */
          chunk_at_offset(old_top, old_size            )->size =
            (2*SIZE_SZ)|PREV_INUSE;

          chunk_at_offset(old_top, old_size + 2*SIZE_SZ)->size =
            (2*SIZE_SZ)|PREV_INUSE;

          /* If possible, release the rest. */
          if (old_size >= MINSIZE) {
#ifdef ATOMIC_FASTBINS
            _int_free(av, old_top, 1);
#else
            _int_free(av, old_top);
#endif
          }
```

设置原 top chunk 的 fencepost，fencepost 需要 MINSIZE 大小的内存空间，将该 old_size 减去 MINSIZE 得到原 top chunk 的有效内存空间，我们可以确信原 top chunk 的有效内存空间一定大于 MINSIZE，将原 top chunk 切分成空闲 chunk 和 fencepost 两部分，首先设置切分出来的 chunk 的大小为 old_size，并标识前一个 chunk 处于 inuse 状态，原 top chunk 切分出来的 chunk 本应处于空闲状态，但 fencepost 的第一个 chunk 却标识前一个 chunk 为 inuse 状态，然后强制将该处于 inuse 状态的 chunk 调用_int_free()函数释放掉。然后设置 fencepost 的第一个 chunk 的大小为 2*SIZE_SZ，并标识前一个 chunk 处于 inuse 状态，然后设置 fencepost 的第二个 chunk 的 size 为 2*SIZE_SZ，并标识前一个 chunk 处于 inuse 状态。这里的主分配区的 fencepost 与非主分配区的 fencepost 不同，主分配区 fencepost 的第二个 chunk 的大小设置为 2*SIZE_SZ，而非主分配区的 fencepost 的第二个 chunk 的大小设置为 0。

```
        }
      }
    }

    /* Update statistics */
#ifdef NO_THREADS
    sum = av->system_mem + mp_.mmapped_mem;
    if (sum > (unsigned long)(mp_.max_total_mem))
      mp_.max_total_mem = sum;
#endif

  }
```

到此为止，对主分配区的分配出来完毕。

```
  } /* if (av !=  &main_arena) */
```

```
  if ((unsigned long)av->system_mem > (unsigned long)(av->max_system_mem))
    av->max_system_mem = av->system_mem;
```

如果当前分配区所分配的内存量大于设置的最大值，更新当前分配区最大分配的内存量，

```
  check_malloc_state(av);

  /* finally, do the allocation */
  p = av->top;
  size = chunksize(p);

  /* check that one of the above allocation paths succeeded */
  if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
    remainder_size = size - nb;
    remainder = chunk_at_offset(p, nb);
    av->top = remainder;
    set_head(p, nb | PREV_INUSE | (av != &main_arena ? NON_MAIN_ARENA : 0));
    set_head(remainder, remainder_size | PREV_INUSE);
    check_malloced_chunk(av, p, nb);
    return chunk2mem(p);
  }
```

如果当前 top chunk 中已经有足够的内存来分配所需的 chunk，从当前的 top chunk 中分配所需的 chunk 并返回。

```
  /* catch all failure paths */
  MALLOC_FAILURE_ACTION;
  return 0;
}
```

5.7.2.5 malloc_consolidate()

malloc_consolidate()函数用于将 fast bins 中的 chunk 合并，并加入 unsorted bin 中，其实现源代码如下：

```
/*
  ------------------------- 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.
*/
#if __STD_C
static void malloc_consolidate(mstate av)
#else
static void malloc_consolidate(av) mstate av;
#endif
{
  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);
```

如果全局变量 global_max_fast 不为零，表示 ptmalloc 已经初始化，清除分配区 flag 中 fast bin 的标志位，该标志位表示分配区的 fast bins 中包含空闲 chunk。然后获得分配区的 unsorted bin。

```
    /*
      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.
    */

#if 0
    /* It is wrong to limit the fast bins to search using get_max_fast
       because, except for the main arena, all the others might have
       blocks in the high fast bins.  It's not worth it anyway, just
       search all bins all the time.  */
    maxfb = &fastbin (av, fastbin_index(get_max_fast ()));
#else
    maxfb = &fastbin (av, NFASTBINS - 1);
#endif
    fb = &fastbin (av, 0);
```

将分配区最大的一个 fast bin 赋值给 maxfb，第一个 fast bin 赋值给 fb，然后遍历 fast bins。

```
    do {
#ifdef ATOMIC_FASTBINS
      p = atomic_exchange_acq (fb, 0);
#else
      p = *fb;
#endif
      if (p != 0) {
#ifndef ATOMIC_FASTBINS
        *fb = 0;
#endif
```

获取当前遍历的 fast bin 中空闲 chunk 单向链表的头指针赋值给 p，如果 p 不为 0，将当前 fast bin 链表的头指针赋值为 0，即删除了该 fast bin 中的空闲 chunk 链表。

```
        do {
          check_inuse_chunk(av, p);
          nextp = p->fd;
```

将空闲 chunk 链表的下一个 chunk 赋值给 nextp。

```
          /* Slightly streamlined version of consolidation code in free() */
          size = p->size & ~(PREV_INUSE|NON_MAIN_ARENA);
          nextchunk = chunk_at_offset(p, size);
          nextsize = chunksize(nextchunk);
```

获得当前 chunk 的 size，需要去除 size 中的 PREV_INUSE 和 NON_MAIN_ARENA 标志，并获取相邻的下一个 chunk 和下一个 chunk 的大小。

```
          if (!prev_inuse(p)) {
            prevsize = p->prev_size;
            size += prevsize;
            p = chunk_at_offset(p, -((long) prevsize));
```

```
        unlink(p, bck, fwd);
      }
```

如果当前 chunk 的前一个 chunk 空闲，则将当前 chunk 与前一个 chunk 合并成一个空闲 chunk，由于前一个 chunk 空闲，则当前 chunk 的 prev_size 保存了前一个 chunk 的大小，计算出合并后的 chunk 大小，并获取前一个 chunk 的指针，将前一个 chunk 从空闲链表中删除。

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

如果与当前 chunk 相邻的下一个 chunk 不是分配区的 top chunk，查看与当前 chunk 相邻的下一个 chunk 是否处于 inuse 状态。

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

如果与当前 chunk 相邻的下一个 chunk 处于 inuse 状态，清除当前 chunk 的 inuse 状态，则当前 chunk 空闲了。否则，将相邻的下一个空闲 chunk 从空闲链表中删除，并计算当前 chunk 与下一个 chunk 合并后的 chunk 大小。

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

将合并后的 chunk 加入 unsorted bin 的双向循环链表中。

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

如果合并后的 chunk 属于 large bin，将 chunk 的 fd_nextsize 和 bk_nextsize 设置为 NULL，因为在 unsorted bin 中这两个字段无用。

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

设置合并后的空闲 chunk 大小，并标识前一个 chunk 处于 inuse 状态，因为必须保证不能有两个相邻的 chunk 都处于空闲状态。然后将合并后的 chunk 加入 unsorted bin 的双向循环链表中。最后设置合并后的空闲 chunk 的 foot，chunk 空闲时必须设置 foot，该 foot 处于下一个 chunk 的 prev_size 中，只有 chunk 空闲是 foot 才是有效的。

```
      }
      else {
        size += nextsize;
        set_head(p, size | PREV_INUSE);
```

```
            av->top = p;
          }
```

如果当前 chunk 的下一个 chunk 为 top chunk，则将当前 chunk 合并入 top chunk，修改 top chunk 的大小。

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

直到遍历完当前 fast bin 中的所有空闲 chunk。

```
      }
    } while (fb++ != maxfb);
```

直到遍历完所有的 fast bins。

```
  }
  else {
    malloc_init_state(av);
    check_malloc_state(av);
```

如果 ptmalloc 没有初始化，初始化 ptmalloc。

```
  }
}
```

5.8 内存释放 free

5.8.1 Public_fREe()

Public_fREe()函数的源代码如下：

```
void
public_fREe(Void_t* mem)
{
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  void (*hook) (__malloc_ptr_t, __const __malloc_ptr_t)
    = force_reg (__free_hook);
  if (__builtin_expect (hook != NULL, 0)) {
    (*hook)(mem, RETURN_ADDRESS (0));
    return;
  }
```

如果存在 free 的 hook 函数，执行该 hook 函数返回，free 的 hook 函数主要用于创建新线程使用或使用用户提供的 free 函数。

```
  if (mem == 0)                              /* free(0) has no effect */
    return;

  p = mem2chunk(mem);
```

free NULL 指针直接返回，然后根据内存指针获得 chunk 的指针。

```
#if HAVE_MMAP
  if (chunk_is_mmapped(p))                     /* release mmapped memory. */
  {
    /* see if the dynamic brk/mmap threshold needs adjusting */
    if (!mp_.no_dyn_threshold
        && p->size > mp_.mmap_threshold
        && p->size <= DEFAULT_MMAP_THRESHOLD_MAX)
      {
        mp_.mmap_threshold = chunksize (p);
        mp_.trim_threshold = 2 * mp_.mmap_threshold;
      }
    munmap_chunk(p);
    return;
  }
#endif
```

如果当前 free 的 chunk 是通过 mmap()分配的，调用 munmap_chunk()函数 unmap 本 chunk。munmap_chunk()函数调用 munmap()函数释放 mmap()分配的内存块。同时查看是否开启了 mmap 分配阈值动态调整机制，默认是开启的，如果当前 free 的 chunk 的大小大于设置的 mmap 分配阈值，小于 mmap 分配阈值的最大值，将当前 chunk 的大小赋值给 mmap 分配阈值，并修改 mmap 收缩阈值为 mmap 分配阈值的 2 倍。默认情况下 mmap 分配阈值与 mmap 收缩阈值相等，都为 128KB。

```
  ar_ptr = arena_for_chunk(p);
```

根据 chunk 指针获得分配区的指针。

```
#ifdef ATOMIC_FASTBINS
  _int_free(ar_ptr, p, 0);
```

如果开启了 ATOMIC_FASTBINS 优化，不需要对分配区加锁，调用_int_free()函数执行实际的释放工作。

```
#else
# if THREAD_STATS
  if(!mutex_trylock(&ar_ptr->mutex))
    ++(ar_ptr->stat_lock_direct);
  else {
    (void)mutex_lock(&ar_ptr->mutex);
    ++(ar_ptr->stat_lock_wait);
  }
# else
  (void)mutex_lock(&ar_ptr->mutex);
# endif
  _int_free(ar_ptr, p);
  (void)mutex_unlock(&ar_ptr->mutex);
#endif
```

如果没有开启了 ATOMIC_FASTBINS 优化，或去分配区的锁，调用_int_free()函数执行实

际的释放工作，然后对分配区解锁。
}

5.8.2 _int_free()

_int_free()函数的实现源代码如下：

```
static void
#ifdef ATOMIC_FASTBINS
_int_free(mstate av, mchunkptr p, int have_lock)
#else
_int_free(mstate av, mchunkptr p)
#endif
{
  INTERNAL_SIZE_T size;        /* its size */
  mfastbinptr*    fb;          /* associated fastbin */
  mchunkptr       nextchunk;   /* next contiguous chunk */
  INTERNAL_SIZE_T nextsize;    /* its size */
  int             nextinuse;   /* true if nextchunk is used */
  INTERNAL_SIZE_T prevsize;    /* size of previous contiguous chunk */
  mchunkptr       bck;         /* misc temp for linking */
  mchunkptr       fwd;         /* misc temp for linking */

  const char *errstr = NULL;
#ifdef ATOMIC_FASTBINS
  int locked = 0;
#endif

  size = chunksize(p);
```

获取需要释放的 chunk 的大小。

```
  /* Little security check which won't hurt performance: the
     allocator never wrapps around at the end of the address space.
     Therefore we can exclude some size values which might appear
     here by accident or by "design" from some intruder.  */
  if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
      || __builtin_expect (misaligned_chunk (p), 0))
    {
      errstr = "free(): invalid pointer";
    errout:
#ifdef ATOMIC_FASTBINS
      if (! have_lock && locked)
        (void)mutex_unlock(&av->mutex);
#endif
```

```
      malloc_printerr (check_action, errstr, chunk2mem(p));
      return;
    }
  /* We know that each chunk is at least MINSIZE bytes in size.  */
  if (__builtin_expect (size < MINSIZE, 0))
    {
      errstr = "free(): invalid size";
      goto errout;
    }

  check_inuse_chunk(av, p);
```

上面的代码用于安全检查，chunk 的指针地址不能溢出，chunk 的大小必须大于等于 MINSIZE。

```
  /*
    If eligible, place chunk on a fastbin so it can be found
    and used quickly in malloc.
  */
  if ((unsigned long)(size) <= (unsigned long)(get_max_fast ())

#if TRIM_FASTBINS
      /*
        If TRIM_FASTBINS set, don't place chunks
        bordering top into fastbins
      */
      && (chunk_at_offset(p, size) != av->top)
#endif
      ) {

    if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
        || __builtin_expect (chunksize (chunk_at_offset (p, size))
                             >= av->system_mem, 0))
      {
#ifdef ATOMIC_FASTBINS
        /* We might not have a lock at this point and concurrent modifications
           of system_mem might have let to a false positive.  Redo the test
           after getting the lock.  */
        if (have_lock
            || ({ assert (locked == 0);
                  mutex_lock(&av->mutex);
                  locked = 1;
                  chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
                    || chunksize (chunk_at_offset (p, size)) >= av->system_mem;
              }))
```

```
#endif
          {
            errstr = "free(): invalid next size (fast)";
            goto errout;
          }
#ifdef ATOMIC_FASTBINS
        if (! have_lock)
          {
            (void)mutex_unlock(&av->mutex);
            locked = 0;
          }
#endif
      }
```

如果当前 free 的 chunk 属于 fast bins，查看下一个相邻的 chunk 的大小是否小于等于 2*SIZE_SZ，下一个相邻 chunk 的大小是否大于分配区所分配的内存总量，如果是，报错。这里计算下一个相邻 chunk 的大小似乎有点问题，因为 chunk 的 size 字段中包含了一些标志位，正常情况下下一个相邻 chunk 的 size 中的 PREV_INUSE 标志位会置位，但这里就是要检出错的情况，也就是下一个相邻 chunk 的 size 中标志位都没有置位，并且该 chunk 大小为 2*SIZE_SZ 的错误情况。如果开启了 ATOMIC_FASTBINS 优化，并且调用本函数前没有对分配区加锁，所以读取分配区所分配的内存总量需要对分配区加锁，检查完以后，释放分配区的锁。

```
    if (__builtin_expect (perturb_byte, 0))
      free_perturb (chunk2mem(p), size - SIZE_SZ);

    set_fastchunks(av);
    unsigned int idx = fastbin_index(size);
    fb = &fastbin (av, idx);
```

设置当前分配区的 fast bin flag，表示当前分配区的 fast bins 中已有空闲 chunk。然后根据当前 free 的 chunk 大小获取所属的 fast bin。

```
#ifdef ATOMIC_FASTBINS
    mchunkptr fd;
    mchunkptr old = *fb;
    unsigned int old_idx = ~0u;
    do
      {
        /* Another simple check: make sure the top of the bin is not the
           record we are going to add (i.e., double free).  */
        if (__builtin_expect (old == p, 0))
          {
            errstr = "double free or corruption (fasttop)";
            goto errout;
          }
        if (old != NULL)
```

```
          old_idx = fastbin_index(chunksize(old));
        p->fd = fd = old;
      }
    while ((old = catomic_compare_and_exchange_val_rel (fb, p, fd)) != fd);

    if (fd != NULL && __builtin_expect (old_idx != idx, 0))
      {
        errstr = "invalid fastbin entry (free)";
        goto errout;
      }
```

如果开启了 ATOMIC_FASTBINS 优化，使用 lock-free 技术实现 fast bin 的单向链表插入操作。对 lock-free 的描述，请参见 5.7.2.1 节。这里也没有 ABA 问题，比如当前线程获取*fb 并保存到 old 中，在调用 cas 原子操作前，B 线程将*fb 修改为 x，如果 B 线程加入了新的 chunk，则 x->fb 指向 old，如果 B 线程删除了 old，则 x 为 old->fb。如果 C 线程将*fb 修改为 old，则可能将 B 线程加入的 chunk x 删除，或者 C 将 B 删除的 old 又重新加入。这两种情况，都不会导致链表出错，所以不会有 ABA 问题。

```
#else
    /* Another simple check: make sure the top of the bin is not the
       record we are going to add (i.e., double free).  */
    if (__builtin_expect (*fb == p, 0))
      {
        errstr = "double free or corruption (fasttop)";
        goto errout;
      }
    if (*fb != NULL
        && __builtin_expect (fastbin_index(chunksize(*fb)) != idx, 0))
      {
        errstr = "invalid fastbin entry (free)";
        goto errout;
      }

    p->fd = *fb;
    *fb = p;
```

如果没有开启了 ATOMIC_FASTBINS 优化，将 free 的 chunk 加入 fast bin 的单向链表中，修改过链表表头为当前 free 的 chunk。同时需要校验是否为 double free 错误，校验表头不为 NULL 情况下，保证表头 chunk 的所属的 fast bin 与当前 free 的 chunk 所属的 fast bin 相同。

```
#endif
  }

  /*
    Consolidate other non-mmapped chunks as they arrive.
  */
  else if (!chunk_is_mmapped(p)) {
#ifdef ATOMIC_FASTBINS
```

```
    if (! have_lock) {
# if THREAD_STATS
      if(!mutex_trylock(&av->mutex))
        ++(av->stat_lock_direct);
      else {
        (void)mutex_lock(&av->mutex);
        ++(av->stat_lock_wait);
      }
# else
      (void)mutex_lock(&av->mutex);
# endif
      locked = 1;
    }
#endif
```

如果当前 free 的 chunk 不是通过 mmap()分配的，并且当前还没有获得分配区的锁，获取分配区的锁。

```
    nextchunk = chunk_at_offset(p, size);
```

获取当前 free 的 chunk 的下一个相邻的 chunk。

```
    /* Lightweight tests: check whether the block is already the
       top block.  */
    if (__builtin_expect (p == av->top, 0))
      {
        errstr = "double free or corruption (top)";
        goto errout;
      }
    /* Or whether the next chunk is beyond the boundaries of the arena.  */
    if (__builtin_expect (contiguous (av)
                          && (char *) nextchunk
                          >= ((char *) av->top + chunksize(av->top)), 0))
      {
        errstr = "double free or corruption (out)";
        goto errout;
      }
    /* Or whether the block is actually not marked used.  */
    if (__builtin_expect (!prev_inuse(nextchunk), 0))
      {
        errstr = "double free or corruption (!prev)";
        goto errout;
      }
```

安全检查，当前 free 的 chunk 不能为 top chunk，因为 top chunk 为空闲 chunk，如果再次 free 就可能为 double free 错误了。如果当前 free 的 chunk 是通过 sbrk()分配的，并且下一个相邻的 chunk 的地址已经超过了 top chunk 的结束地址，超过了当前分配区的结束地址，

报错。如果当前 free 的 chunk 的下一个相邻 chunk 的 size 中标志位没有标识当前 free chunk 为 inuse 状态，可能为 double free 错误。

```
nextsize = chunksize(nextchunk);
if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0)
    || __builtin_expect (nextsize >= av->system_mem, 0))
  {
    errstr = "free(): invalid next size (normal)";
    goto errout;
  }

if (__builtin_expect (perturb_byte, 0))
  free_perturb (chunk2mem(p), size - SIZE_SZ);
```

计算当前 free 的 chunk 的下一个相邻 chunk 的大小，该大小如果小于等于 2*SIZE_SZ 或是大于了分配区所分配区的内存总量，报错。

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = p->prev_size;
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(p, bck, fwd);
}
```

如果当前 free 的 chunk 的前一个相邻 chunk 为空闲状态，与前一个空闲 chunk 合并。计算合并后的 chunk 大小，并将前一个相邻空闲 chunk 从空闲 chunk 链表中删除。

```
if (nextchunk != av->top) {
  /* get and clear inuse bit */
  nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
```

如果与当前 free 的 chunk 相邻的下一个 chunk 不是分配区的 top chunk，查看与当前 chunk 相邻的下一个 chunk 是否处于 inuse 状态。

```
  /* consolidate forward */
  if (!nextinuse) {
    unlink(nextchunk, bck, fwd);
    size += nextsize;
  } else
    clear_inuse_bit_at_offset(nextchunk, 0);
```

如果与当前 free 的 chunk 相邻的下一个 chunk 处于 inuse 状态，清除当前 chunk 的 inuse 状态，则当前 chunk 空闲了。否则，将相邻的下一个空闲 chunk 从空闲链表中删除，并计算当前 chunk 与下一个 chunk 合并后的 chunk 大小。

```
  /*
    Place the chunk in unsorted chunk list. Chunks are
```

```
    not placed into regular bins until after they have
    been given one chance to be used in malloc.
  */
  bck = unsorted_chunks(av);
  fwd = bck->fd;
  if (__builtin_expect (fwd->bk != bck, 0))
    {
      errstr = "free(): corrupted unsorted chunks";
      goto errout;
    }
  p->fd = fwd;
  p->bk = bck;
  if (!in_smallbin_range(size))
    {
      p->fd_nextsize = NULL;
      p->bk_nextsize = NULL;
    }
  bck->fd = p;
  fwd->bk = p;
```

将合并后的 chunk 加入 unsorted bin 的双向循环链表中。如果合并后的 chunk 属于 large bins，将 chunk 的 fd_nextsize 和 bk_nextsize 设置为 NULL，因为在 unsorted bin 中这两个字段无用。

```
  set_head(p, size | PREV_INUSE);
  set_foot(p, size);
```

设置合并后的空闲 chunk 大小，并标识前一个 chunk 处于 inuse 状态，因为必须保证不能有两个相邻的 chunk 都处于空闲状态。然后将合并后的 chunk 加入 unsorted bin 的双向循环链表中。最后设置合并后的空闲 chunk 的 foot，chunk 空闲时必须设置 foot，该 foot 处于下一个 chunk 的 prev_size 中，只有 chunk 空闲是 foot 才是有效的。

```
  check_free_chunk(av, p);
}

/*
  If the chunk borders the current high end of memory,
  consolidate into top
*/
else {
  size += nextsize;
  set_head(p, size | PREV_INUSE);
  av->top = p;
  check_chunk(av, p);
}
```

如果当前 free 的 chunk 下一个相邻的 chunk 为 top chunk，则将当前 chunk 合并入 top

chunk，修改 top chunk 的大小。

```
    /*
      If freeing a large space, consolidate possibly-surrounding
      chunks.  Then, if the total unused topmost memory exceeds trim
      threshold, ask malloc_trim to reduce top.

      Unless max_fast is 0, we don't know if there are fastbins
      bordering top, so we cannot tell for sure whether threshold
      has been reached unless fastbins are consolidated.  But we
      don't want to consolidate on each free.  As a compromise,
      consolidation is performed if FASTBIN_CONSOLIDATION_THRESHOLD
      is reached.
    */
    if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
      if (have_fastchunks(av))
        malloc_consolidate(av);
```

如果合并后的 chunk 大小大于 64KB，并且 fast bins 中存在空闲 chunk，调用 malloc_consolidate()函数合并 fast bins 中的空闲 chunk 到 unsorted bin 中。

```
      if (av == &main_arena) {
#ifndef MORECORE_CANNOT_TRIM
        if ((unsigned long)(chunksize(av->top)) >=
            (unsigned long)(mp_.trim_threshold))
          sYSTRIm(mp_.top_pad, av);
```

如果当前分配区为主分配区，并且 top chunk 的大小大于 heap 的收缩阈值，调用 sYSTRIm()函数首先 heap。

```
#endif
      } else {
        /* Always try heap_trim(), even if the top chunk is not
           large, because the corresponding heap might go away.  */
        heap_info *heap = heap_for_ptr(top(av));

        assert(heap->ar_ptr == av);
        heap_trim(heap, mp_.top_pad);
```

如果为非主分配区，调用 heap_trim()函数收缩非主分配区的 sub_heap。

```
      }
    }

#ifdef ATOMIC_FASTBINS
    if (! have_lock) {
      assert (locked);
      (void)mutex_unlock(&av->mutex);
    }
```

```
#endif
```
如果开启了 ATOMIC_FASTBINS 优化并获得分配区的锁，则对分配区解锁。
```
  }
  /*
    If the chunk was allocated via mmap, release via munmap(). Note
    that if HAVE_MMAP is false but chunk_is_mmapped is true, then
    user must have overwritten memory. There's nothing we can do to
    catch this error unless MALLOC_DEBUG is set, in which case
    check_inuse_chunk (above) will have triggered error.
  */
  else {
#if HAVE_MMAP
    munmap_chunk (p);
```
如果当前 free 的 chunk 是通过 mmap()分配的，调用 munma_chunk()释放内存。
```
#endif
  }
}
```

5.8.3 sYSTRIm()和 munmap_chunk()

sYSTRIm()函数源代码如下：

```
/*
  sYSTRIm is an inverse of sorts to sYSMALLOc.  It gives memory back
  to the system (via negative arguments to sbrk) if there is unused
  memory at the `high' end of the malloc pool. It is called
  automatically by free() when top space exceeds the trim
  threshold. It is also called by the public malloc_trim routine.  It
  returns 1 if it actually released any memory, else 0.
*/
#if __STD_C
static int sYSTRIm(size_t pad, mstate av)
#else
static int sYSTRIm(pad, av) size_t pad; mstate av;
#endif
{
  long  top_size;        /* Amount of top-most memory */
  long  extra;           /* Amount to release */
  long  released;        /* Amount actually released */
  char* current_brk;     /* address returned by pre-check sbrk call */
  char* new_brk;         /* address returned by post-check sbrk call */
  size_t pagesz;

  pagesz = mp_.pagesize;
  top_size = chunksize(av->top);
```

获取页大小也 top chunk 的大小。

```
/* Release in pagesize units, keeping at least one page */
extra = ((top_size - pad - MINSIZE + (pagesz-1)) / pagesz - 1) * pagesz;
```

计算 top chunk 中最大可释放的整数页大小，top chunk 中至少需要 MINSIZE 的内存保存 fencepost。

```
if (extra > 0) {
  /*
    Only proceed if end of memory is where we last set it.
    This avoids problems if there were foreign sbrk calls.
  */
  current_brk = (char*)(MORECORE(0));
  if (current_brk == (char*)(av->top) + top_size) {
```

获取当前 brk 值，如果当前 top chunk 的结束地址与当前的 brk 值相等，执行 heap 收缩。

```
    /*
      Attempt to release memory. We ignore MORECORE return value,
      and instead call again to find out where new end of memory is.
      This avoids problems if first call releases less than we asked,
      of if failure somehow altered brk value. (We could still
      encounter problems if it altered brk in some very bad way,
      but the only thing we can do is adjust anyway, which will cause
      some downstream failure.)
    */
    MORECORE(-extra);
```

调用 sbrk()释放指定大小的内存到 heap 中。

```
    /* Call the `morecore' hook if necessary.  */
    void (*hook) (void) = force_reg (__after_morecore_hook);
    if (__builtin_expect (hook != NULL, 0))
      (*hook) ();
    new_brk = (char*)(MORECORE(0));
```

如果 morecore hook 存在，执行 hook 函数，然后获得当前新的 brk 值。

```
    if (new_brk != (char*)MORECORE_FAILURE) {
      released = (long)(current_brk - new_brk);

      if (released != 0) {
        /* Success. Adjust top. */
        av->system_mem -= released;
        set_head(av->top, (top_size - released) | PREV_INUSE);
        check_malloc_state(av);
        return 1;
```

如果获取新的 brk 值成功，计算释放的内存大小，更新当前分配区所分配的内存总量，更新 top chunk 的大小。

```
        }
      }
    }
  }
  return 0;
}
```

Munmap_chunk()函数源代码如下：

```
static void
internal_function
#if __STD_C
munmap_chunk(mchunkptr p)
#else
munmap_chunk(p) mchunkptr p;
#endif
{
  INTERNAL_SIZE_T size = chunksize(p);

  assert (chunk_is_mmapped(p));
#if 0
  assert(! ((char*)p >= mp_.sbrk_base && (char*)p < mp_.sbrk_base +
mp_.sbrked_mem));
  assert((mp_.n_mmaps > 0));
#endif

  uintptr_t block = (uintptr_t) p - p->prev_size;
  size_t total_size = p->prev_size + size;
  /* Unfortunately we have to do the compilers job by hand here.  Normally
     we would test BLOCK and TOTAL-SIZE separately for compliance with the
     page size.  But gcc does not recognize the optimization possibility
     (in the moment at least) so we combine the two values into one before
     the bit test.  */
  if (__builtin_expect (((block | total_size) & (mp_.pagesize - 1)) != 0, 0))
    {
      malloc_printerr (check_action, "munmap_chunk(): invalid pointer",
                       chunk2mem (p));
      return;
    }

  mp_.n_mmaps--;
  mp_.mmapped_mem -= total_size;
```

```
  int ret __attribute__ ((unused)) = munmap((char *)block, total_size);

  /* munmap returns non-zero on failure */
  assert(ret == 0);
}
```

Munmap_chunk()函数实现相当简单，首先获取当前 free 的 chunk 的大小，断言当前 free 的 chunk 是通过 mmap()分配的，由于使用 mmap()分配的 chunk 的 prev_size 中记录的前一个相邻空闲 chunk 的大小，mmap()分配的内存是页对齐的，所以一般情况下 prev_size 为 0。然后计算当前 free 的 chunk 占用的总内存大小 total_size，再次校验内存块的起始地址是否是对齐的，更新分配区的 mmap 统计信息，最后调用 munmap()函数释放 chunk 的内存。