深入理解 Glibc 的内存管理

---

renyl 2016/3/24

目录

1 介绍

Linux 操作系统中的内存管理分为两个部分：

1） 物理内存的管理：该部分由 Linux 内核的 buddy 系统和 slab 分配器合作管理，管理着物理内存资源的分配及回收。

2） 虚拟内存的管理：该部分由 Glibc 进行管理，给用户层提供了多个关于内存管理方面的函数，其中 malloc()和 free()最为常用。

本文主要讨论 Glibc 对虚拟内存的管理，分为如下几个部分：

1) 首先，通过对 Glibc 的内存管理机制进行研究，认识到某些情况下不适合用 Glibc 来管理内存，因为其设计的管理机制在某些情况下可能导致内存泄露、大量的内存碎片、内存暴增等问题。

2) 然后，通过一些案例来说 Glibc 管理机制的特性以及如何运用适当的方法来规避 Glibc 在某些情况下对内存管理的缺陷。

3) 最后，介绍一些方法和工具来检测或解决内存泄露、内存无法释放等问题。

注：本文在如下平台下进行研究及测试。

-	描述
os	RHEL7.0_x86_64
kernel	kernel-3.10.0-110.el7.x86_64
cpu	Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
glibc	glibc-2.17-52.el7.x86_64。

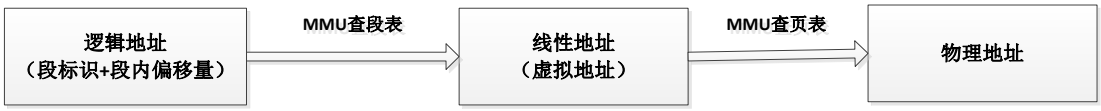
2 内存基础

2.1 内存地址

1) 内存地址可分为逻辑地址、线性地址、虚拟地址和物理地址。

2) 程序的各种数据操作是通过虚拟地址找到物理内存从而进行操作的。

3) 逻辑地址到物理地址的转换过程：

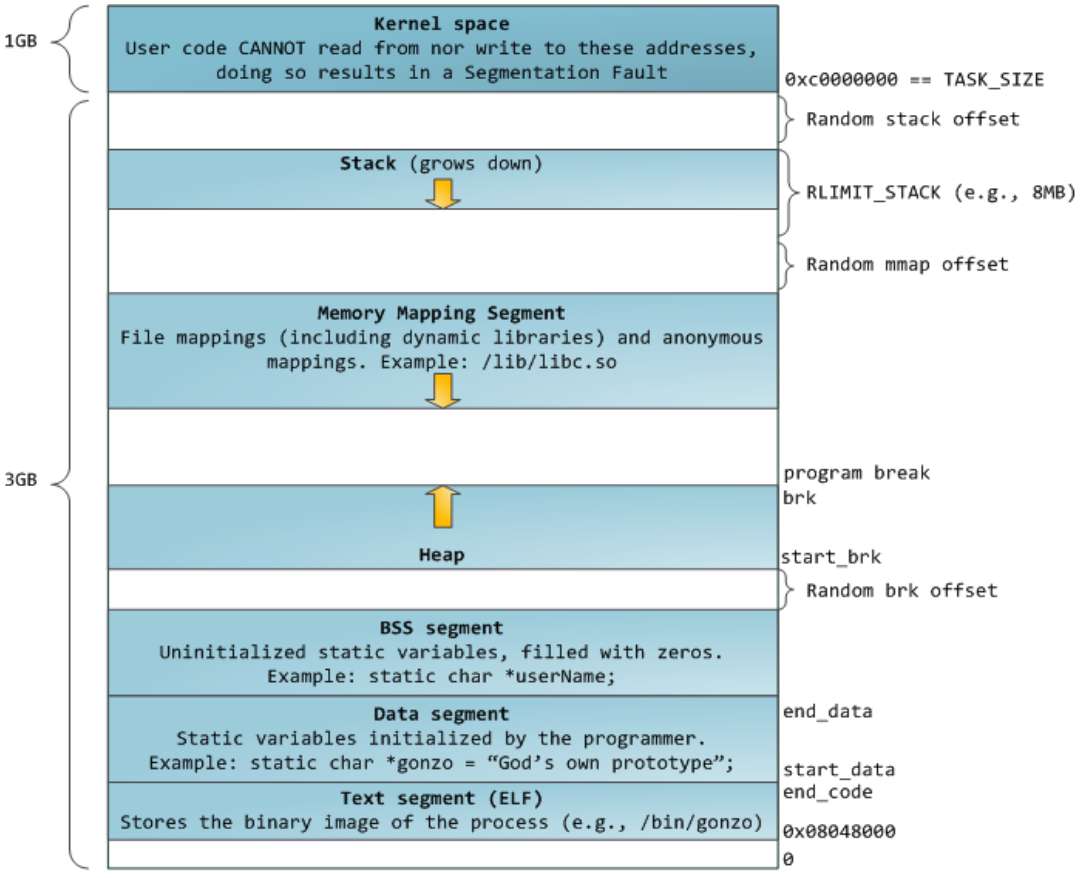


说明：

地址转换过程是与体系结构相关的，X86 的 CPU 支持分段和分页两种方式。如果 CPU 没有开启分段功能，就不需要 MMU 查段表，如果 CPU 没有开启分页功能，那么线性地址就是物理地址，不再需要查页表。

2.2 内存分布

1) x86_32 位系统下，进程内存分布如下所示：

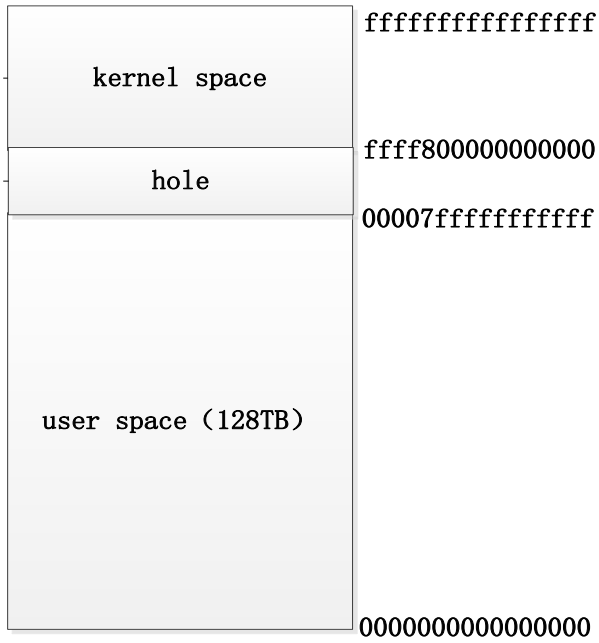


说明：
Linux 系统在运行 elf 格式的程序文件时，会调用加载器（loader）把可执行文件中的各个段依次载入到从某一地址开始的空间（不同系统起始地址不同），分为如下几段：

a) 代码段：该部分空间只能读不能写，存储程序的代码。
b) 数据段：该部分可读可写，存储程序中（已初始化过的）全局变量和静态变量。
c) bbs 段：该部分可读可写，存储程序中（未初始化过的）全局变量和静态变量。
d) 堆：该部分可读可写，malloc/calloc/realloc 函数动态申请的内存大都在此（不是全部，后面分析中会有说明），其中堆顶的位置可以通过函数 brk 和 sbrk 来进行动态调整。
e) mapping 区：由动态库、共享内存和文件映射内存等组成，一般是由 mmap 函数调用来分配虚拟地址空闲的。
f) 栈：用于维护函数调用的上下文空间，一般为 8M，可以通过 ulimit -s 来查看（唯一不需要映射，可以直接访问的内存区域，也是利用栈溢出进行攻击的基础）。
g) 内核空间：由内核管理，用户代码不可见的内存区域（如页表就存放在内核空间）。

需要注意的是，上图中的“Random offset”使得相同程序在每次启动时 栈区、mapping 区和堆区 的起始地址都是不同的。 这是为了安全起见，在程序启动时随机改变这些值的设置，使得 hacker 很难利用缓冲区溢出进行攻击。不过，用户可以通过设置文件 /proc/sys/kernel/randomize_va_space 为 0 来关闭该特性。

2) x86_64 位系统下，进程内存分布如下所示：



说明：
用户空间：与 x86_32 位下的用户空间布局基本一致，但是其虚拟内存空间达到 128TB。

内核空间：详细信息如下（参考内核文件 root/Documentation/x86/x86_64/mm.txt）

```
...
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
```

```
hole caused by [48:63] sign extension
ffff800000000000 - ffff80ffffffffff (=40 bits) guard hole
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
…
```

需要注意的是，目前 x86_64 系统的虚拟地址空间一般使用 48 位来表示虚拟地址空间，可以通过命令/proc/cpuinfo 来查看，如下所示：

```
[root@localhost ~]# cat /proc/cpuinfo
…
address sizes   : 46 bits physical, 48 bits virtual
…
```

2.3 延迟分配

1) 用户在动态申请内存的时候，系统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。

2) 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，如下所示：

```
…
char *p = malloc(8*1024)   //只是分配 8KB 的虚拟内存，并不占用实际物理内存
strcpy(p,”123”) //分配 1 页（4KB）大小的物理内存（内核以页为单位进行分配）
…
```

需要注意的是，malloc 函数返回的地址为非空时并不保证物理内存可获取，如果此时系统物理内存已满，系统将要启动 OOM(out of memory)killer 进程杀死一些进程来释放一些物理内存。

3) 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，将其释放的过程，如下所示：

```
…
free(p) //通过虚拟地址，找到其所对应的物理页面，释放物理页面
…
```

需要注意的是，调用 free 函数并一定释放物理内存（后面分析中会有说明）。

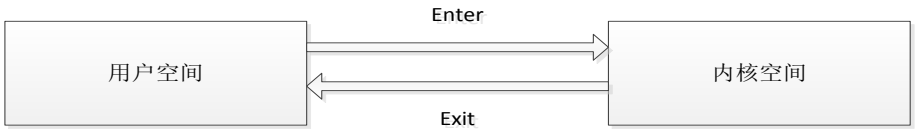
3 Glibc 内存管理

3.1 背景知识

1) 用户的进程和内核是运行在不同的级别，进程与内核之间的通信是通过系统调用来完成的。其中，进程在申请和释放内存方面主要通过 brk、sbrk、mmap、munmap 这几个系统调用来完成。

2) Glibc 库提供的进程申请、释放内存函数（如 malloc、free）在底层最终也是通过系统调用 brk、sbrk、mmap、munmap 来实现的。

3) 系统调用开销大，因为其每次都要经历如下过程：



因此，Glibc 为了避免每次申请、释放内存都要调用系统调用，使用了一套机制来对内存的申请和释放进行管理。

3.2 内存分配器（Allocator）

1) Glibc 的内存管理是通过一个叫 ptmalloc 的内存分配器来实现的，ptmalloc 通过实现 malloc、calloc、realloc、free 等函数来提供动态内存管理的支持。

2) Allocator 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。

3) 用户通过 malloc 申请内存时，Allocator 一般都会分配一块大于用户请求的内存并对剩余部分进行管理。用户通过 free 释放内存时并不是立即就返回给操作系统，Allocator 会管理这些被 free 掉的空闲空间，以应对用户以后的内存分配要求。

4) Allocator 不但要管理已分配的内存块，还需要管理空闲的内存块。当用户申请内存时，Allocator 会首先在空闲空间中寻找一块合适的内存给用户，只有在空闲空间中找不到的情况下才向系统申请分配一块新的内存。

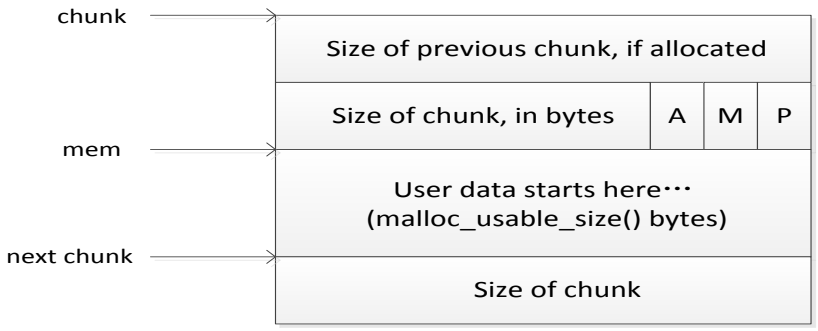
3.3 Chunk 结构

1) Allocator 使用 malloc_chunk 结构体对内存进行管理，如下所示：
（参考 glibc 源码文件/malloc/malloc.c）

```
…
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;
};
…
```

2) 用户调用 malloc 函数申请内存时，Allocator 对分配的内存都使用一个 chunk 来表示，如下所示：



注：在 x86_64 系统下，一个 chunk 的大小最小为 32B，且 chunk 对齐到 8B、mem 对齐到 16B。

说明：

一个 chunk 中包含了用户请求的内存区域和相关的控制信息：

a) chunk 指针指向一个 chunk 的开始

b) mem 指针才是真正返回给用户的内存指针

c) 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。

i. 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。

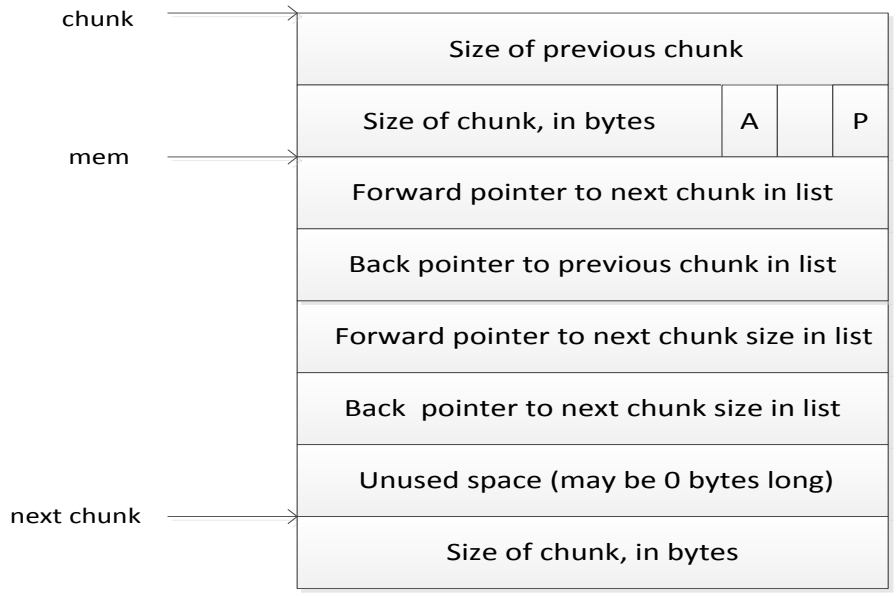
ii. 当 p 为 1 时, 表示前一个 chunk 正在使用中, prev_size 无效, 程序也就无法得到前一个 chunk 的大小，那么就不能对前一个 chunk 进行任何操作。

iii. Allocator 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。

d) 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。

e) 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。(关于主分配区和非主分配区后面会有介绍)。

3) 用户调用 free 函数释放掉内存时，由于内存不立即归还给操作系统，Allocator 也使用一个 Chunk 表示，如下所示：



说明：

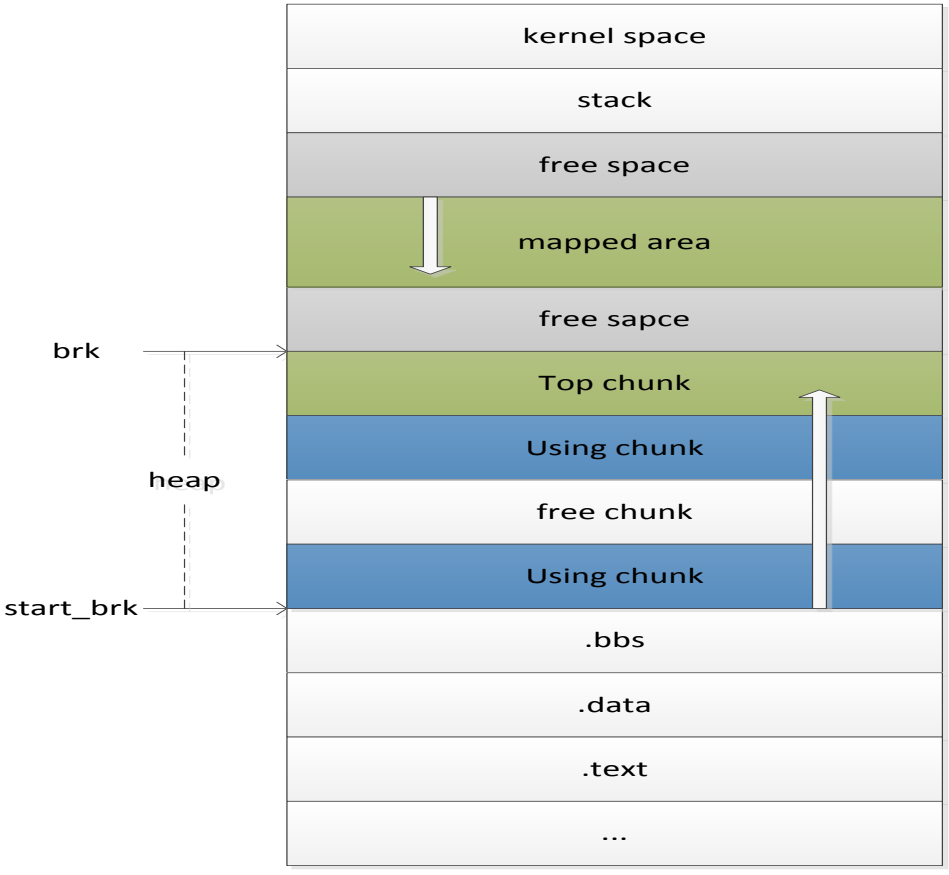
a) 当 chunk 空闲时，其 M 状态不存在，只有 AP 状态。

b) 原本是用户数据区的地方现在存储了 4 个指针，其中后两个指针只在是大数据块的时候才使用。

需要注意的是，有两种情况下 chunk 不使用上面这种方式来表示：

a）通过系统调用 mmap 进行分配的 mapped chunk

b）Top Chunk 空闲内存



说明：

a) mapped chunk：

用户使用 malloc 申请内存（request_size）的时候，Allocator 首先会在 heap 中查找合适的内存，如果 heap 中没有合适大小的内存块，Allocator 会根据申请内存的大小来做出不同的选择：

i. request_size < M_MMAP_THRESHOLD 时，Allocator 会调用函数 sbrk 来移动堆顶指针，扩大 heap 的大小，然后在 heap 上进行分配。

ii. request_size >= M_MMAP_THRESHOLD 时，Allocator 会调用函数 mmap 来进行映射内存。

注：

1:M_MMAP_THRESHOLD 大小默认为 128KB，可以使用 mallopt(M_MMAP_THRESHOLD，-)来进行设置。

2:M_MMAP_THRESHOLD 最小为 0，最大的话：32 位系统下为 512KB，64 位系统下 32MB。

mapped chunk 由于是通过 mmap 函数来获取内存的，当调用 free 函数时，系统会自动调用 munmap 函数来解除内存映射，从而释放这段物理内存给系统。

注：mmap 函数在申请内存时有如下两个特性：

1:内存起始地址需要以 PAGESIZE 大小（4KB）对齐

2:以匿名方式分配内存的内容都要被清零

b) Top chunk：

i. Allocator 会预先分配一块较大的空闲内存(也就是所为的 heap)，通过管理这块空闲内存来响应用户的需求。

ii. 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”。

iii. 当 fast bins、unsorted bins 和 bins（bin 将在后面介绍）都不能满足分配需求的时候，ptmalloc 会设法在“Top chunk”中分配出一块内存给用户。

4) chunk 中的空间复用：

a) 一个 chunk 可以正在被使用或者已经被 free，因此可以利用 chunk 的中的一些域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。

b) 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，那么这个空间也可以被当前 chunk 使用。

c) chunk 的大小要对齐到 8B，当用户请求分配内存时，最终分配的空间大小为：chunk_size = (用户请求大小+ 8) align to 8B。（这里加 8 是因为需要存储 prev_size 和 size）

注：在 x86_64 系统下，chunk_size 最小为 32B，且以 16B 为基数增加。

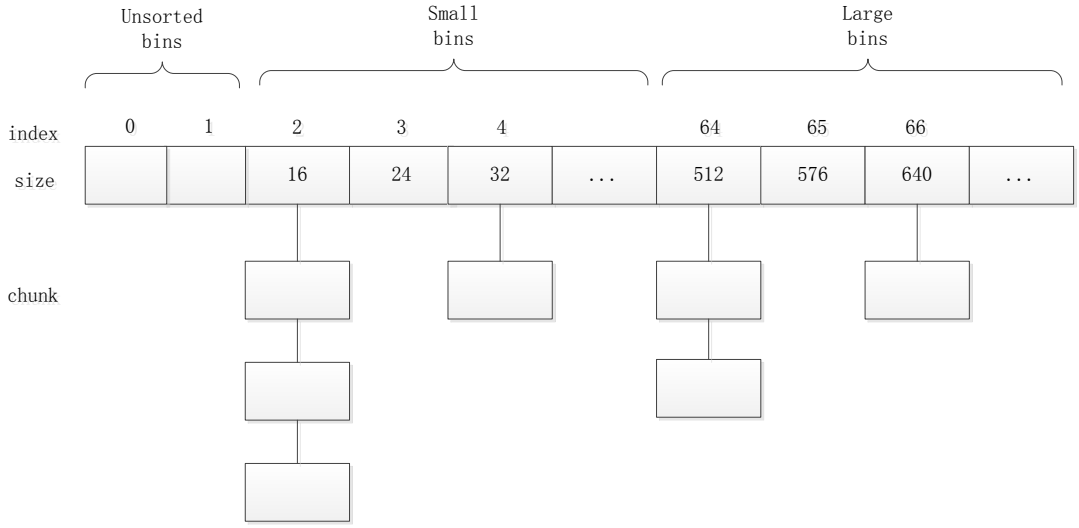
3.4 空闲 chunk 容器

3.4.1 bins

1) 用户 free 掉的内存并不是都会马上归还给操作系统，ptmalloc 会统一管理 heap 中的空闲的 chunk。

2) 当用户进行下一次分配请求时，ptmalloc 会首先试图在 heap 中的空闲的 chunk 中选择合适的块给用户，这样就避免了频繁的系统调用，降低了内存分配的开销。

3) ptmalloc 将 heap 中相似大小的 chunk 用双向链表连接起来，这样的一个链表被称为一个 bin。ptmalloc 共维护了 128 个 bin，并使用一个数组来存储这些 bin，如下所示：



说明：

a) 数组中编号从 0 到 1 的 bin 称为 unsorted bin（后面将会详细介绍）。

b) 数组编号从 2 到 63 的 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 8B。

c) 数组编号从 64 到 128 的 bin 称为 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。

关于不同 bin 所指定的 chunk 大小如下所示：

index 范围	chunk 范围（byte）	公差（byte）	个数
[2,63]	[16,504]	8	62
[64,94]	[512,2488]	64	31
[95,111]	[2496,10744]	512	17
[112,120]	[10752,45048]	4096	9
[121,123]	[45056,163832]	32768	3
[124,126]	[163840,786424]	262144	3
[127]	[786432,2^64]	-	1

4) 当空闲 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，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，然后再将其链接到 bin 中。

3.4.2 fast bins

1) 不是所有的 chunk 被释放后就立即被放 bin 中。ptmalloc 为了提高分配的速度，会把一些小的 chunk 先放到一个叫做 fast bins 的容器内。

2) 释放的 chunk 小于 max_fast（默认值为 64*sizeof（size_t）/4，即 x86_64 下为 128B）的 chunk 被释放后，首先会被放到 fast bins 容器中，fast bins 中的 chunk 并不改变它的使用标志 P，这样释放的块也就无法被合并。

   注：

   a）max_fast 的大小可以通过 mallopt（M_MXFAST，-）来进行设置。

   b) 在某些情况下 fast bin 可能会造成大量的碎片。

3) 当需要给用户分配的 chunk 小于或等于 max_fast 时，ptmalloc 会首先在 fast bins 中查找相应的空闲块，然后才会去查找 bins 中的空闲 chunk。

4) 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 unsorted bin 里的 chunk 加入 bins 中。

3.4.3 unsorted bins

1) unsorted bins 相当于一个不固定大小的缓存，回收的 chunk 要优先放在此处。

2) 当用户释放的 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 链表中。

3) ptmalloc 在进行分配 chunk 时，如果在 fast bin 中没有找到合适的 chunk，则会在 unsorted bin 中查找合适的空闲 chunk，如果 unsorted bin 不能满足分配要求，ptmalloc 便会将 unsorted bin 中的 chunk 加入到 bins 中，然后再从 bins 继续查找。

3.5 分配区

1) ptmalloc 是以分配区为基础对内存进行管理的。

2) 由于每次分配内存都必须对分配区加锁，在分配完成后才释放锁。在 SMP 多线程环境下，对分配区的锁争用严重影响了 malloc 的分配效率。

3) 为了解决锁争用的情况，ptmalloc 采用了一个主分配区（main_arena）和多个非主分配区（non_main_arena）合作的方式来对内存进行管理。

4) 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）使线程对与该分配区的访问互斥。

5) 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 根据分配区锁的争用境况来动态增加非主分配区的数量，但分配区的数量一旦增加，就不能减少。

关于主分配区和非主分配区的详细信息，参看下表：

分配区	个数	位置	说明
主分配区	只能有一个	heap 、 mmap 区域	主要的内存分配区。如果不存在多线程环境，会大量使用主分配区的内存，这样可以减少 mmap 的系统调用，提高性能。
非主分配区	可以有多个	mmap 区域	为了避免多线程对锁的竞争而引入的分区。

说明：

a) 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。

b) 非主分配区只能访问进程的 mmap 映射区域，也就是说非主分配区只能使用 mmap 向操作系统申请内存。

c) 非主分配区每次使用 mmap 向操作系统申请 HEAP_MAX_SIZE（x86_64 下默认为 64MB）大小的虚拟内存，然后使用该段内存模拟 sub_heap。当用户向非主分配区请求分配内存时，将其切割成小块分配出去，直接从用户空间进行分配，不需要系统调用，效率高。

6) 分配区的创建过程：

a) 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。

b) 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。

c) 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。

d) 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。

3.6 malloc 过程

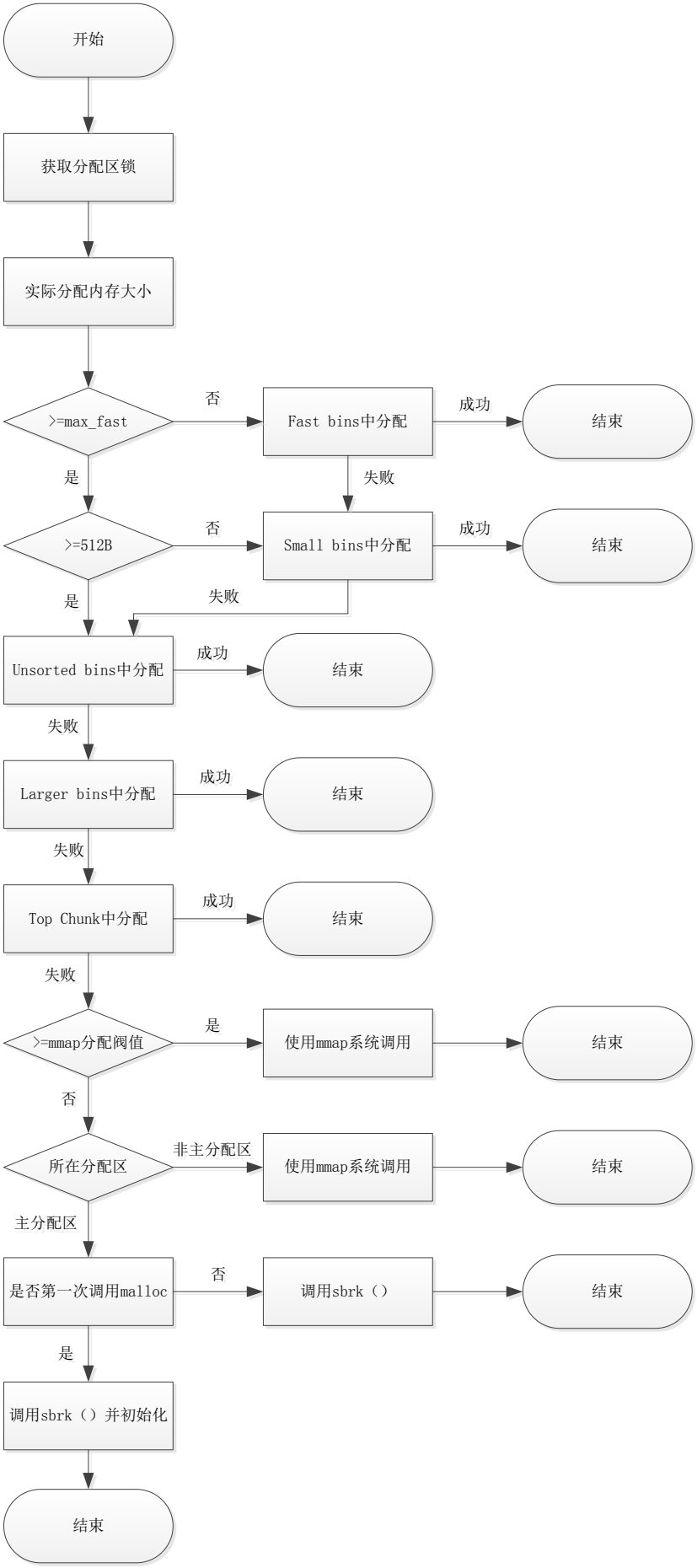
3.6.1 源代码

在 ptmalloc 中没有定义 malloc 函数的实现，而是定义了__libc_malloc(size_t bytes)函数，这样做是为了在不同平台下对 malloc 函数进行不同的实现。_libc_malloc 函数实现代码如下所示：（参考 Glibc 源码文件/malloc/malloc.c）

```
void *__libc_malloc (size_t bytes)
{
  mstate ar_ptr;  //分配区的结构体描述
  void *victim;   //返回给 user 的实际虚拟内存地址

  void *(*hook) (size_t, const void *)
    = atomic_forced_read (__malloc_hook);  //malloc 的 hook 函数
  if (__builtin_expect (hook != NULL, 0))
    return (*hook)(bytes, RETURN_ADDRESS (0));

  arena_lookup (ar_ptr);  //查看当前线程存储中是否有分配区的指针
  arena_lock (ar_ptr, bytes);  //获取分配区的锁
  if (!ar_ptr)
    return 0;

  victim = _int_malloc (ar_ptr, bytes);  //核心函数，用来从分配区中获取虚拟内存
  if (!victim)                           //分配失败，尝试到其它分配区获取虚拟内存
    {
      LIBC_PROBE (memory_malloc_retry, 1, bytes);
      ar_ptr = arena_get_retry (ar_ptr, bytes);
      if (__builtin_expect (ar_ptr != NULL, 1))
        {
          victim = _int_malloc (ar_ptr, bytes);
          (void) mutex_unlock (&ar_ptr->mutex);
        }
    }
  else
    (void) mutex_unlock (&ar_ptr->mutex);   //释放分配区锁
  assert (!victim || chunk_is_mmapped (mem2chunk (victim)) ||
          ar_ptr == arena_for_chunk (mem2chunk (victim)));
  return victim;
}
```

3.6.2 流程图

ptmalloc 响应 malloc 函数申请内存的过程如下所示：


开始
获取分配区锁
实际分配内存大小
>=max_fast
否
Fast bins中分配
成功
结束
失败
是
>=512B
否
Small bins中分配
成功
结束
失败
是
Unsorted bins中分配
成功
结束
失败
Larger bins中分配
成功
结束
失败
Top Chunk中分配
成功
结束
失败
>=mmap分配阀值
是
使用mmap系统调用
结束
否
所在分配区
非主分配区
使用mmap系统调用
结束
主分配区
是否第一次调用malloc
否
调用sbrk（）
结束
是
调用sbrk（）并初始化
结束

3.6.3 说明

ptmalloc 响应用户内存分配要求的具体步骤：

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。

2) 将用户的请求大小转换为实际需要分配的 chunk 空间大小 chunk_size。

3) 判断所需分配 chunk 的大小是否满足 chunk_size >= max_fast，如果是的话，则转到第 5 步，否则，转到下一步。

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，转到第 6 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。如果不成立，说明 chunk 大小处在 small bins 中，则转入下一步。否则，转到第 7 步。

6) 根据所需分配的 chunk 的大小，找到具体所在的某个 small bins，从该 bin 的尾部摘去一个恰好满足大小的 chunk。如果可以找到，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在 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。如果可以找到合适的 chunk，则分配结束。否则，将根据 unsorted bins 中的 chunk 大小将其放入 small bins 或 large bins，接着转到下一步。

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 bins 中都找不到合适的 chunk。在 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。如果分配成功，则分配结束。否则，转入下一步。

10) 到了这一步，说明 Top chunk 也不能满足分区要求。于是，就需要判断所需分配的内存大小是否大于 mmap 阀值。如果大于 mmap 阀值的话，转入下一步。否则，转到第 12 步。

11) 使用 mmap 系统调用为程序的内存空间映射一块 chunk_size 对齐 4KB 大小的空间，然后将内存指针返回给用户。

12) 判断用户获取的是主分配区的锁还是非主分配区的锁。如果是非主分配区的锁，则转到下一步。否则，转到第 14 步。

13) 到了这一步，说明是在非主分配区。那么，就调用 mmap 来分配一个新的 sub_heap，来增加 Top Chunk 大小。

14) 到了这一步，说明是在主分配区。那么，需要判断是否为第一次调用 malloc。如果不是的话，转到下一步。否则，转到第 16 步。

15) 调用 sbrk()分配一块大小为 chunk_size2 对齐到 4KB 大小的空间增加 heap 空间的大小，然后在 Top chunk 中切割一个 chunk 来满足分配需求，并将内存指针返回给用户。

16) 调用 sbrk()分配一块大小为（chunk_size3 + 128KB）对齐到 4KB 大小的空间作为初始化的 heap，然后在 Top chunk 中切割一个 chunk 来满足分配需求，并将内存指针返回给用户。

注 1：

a) 如果（chunk_size - top_chunk_size）能够整除 4，那么 chunk_size2=chunk_size - top_chunk_size + 128KB。（如，chunk_size=20KB, top_chunk_size=16KB，那么 chunk_size2=132KB）

b) 如果（chunk_size - top_chunk_size）不能够整除 4，那么 chunk_size2 将等于（chunk_size - top_chunk_size）加上一个小于 4KB 的值对齐到 4KB。（如，chunk_size=20KB,top_chunk_size=2KB，那么 chunk_size2=148KB）

注 2：

a) 如果 chunk_size 能够整除 4KB，那么 chunk_size3=chunk_size + 4KB。(如，chunk_size=8KB,那么 chunk_size3=12KB)

b) 如果 chunk_size 不能够整除 4KB，那么 chunk_size3 将等于 chunk_size 加上一个小于 4KB 的值对齐到 4KB。(如，chunk_size=6KB，那么 chunk_size3=8KB)

需要注意的是：

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。

3.7 free 过程

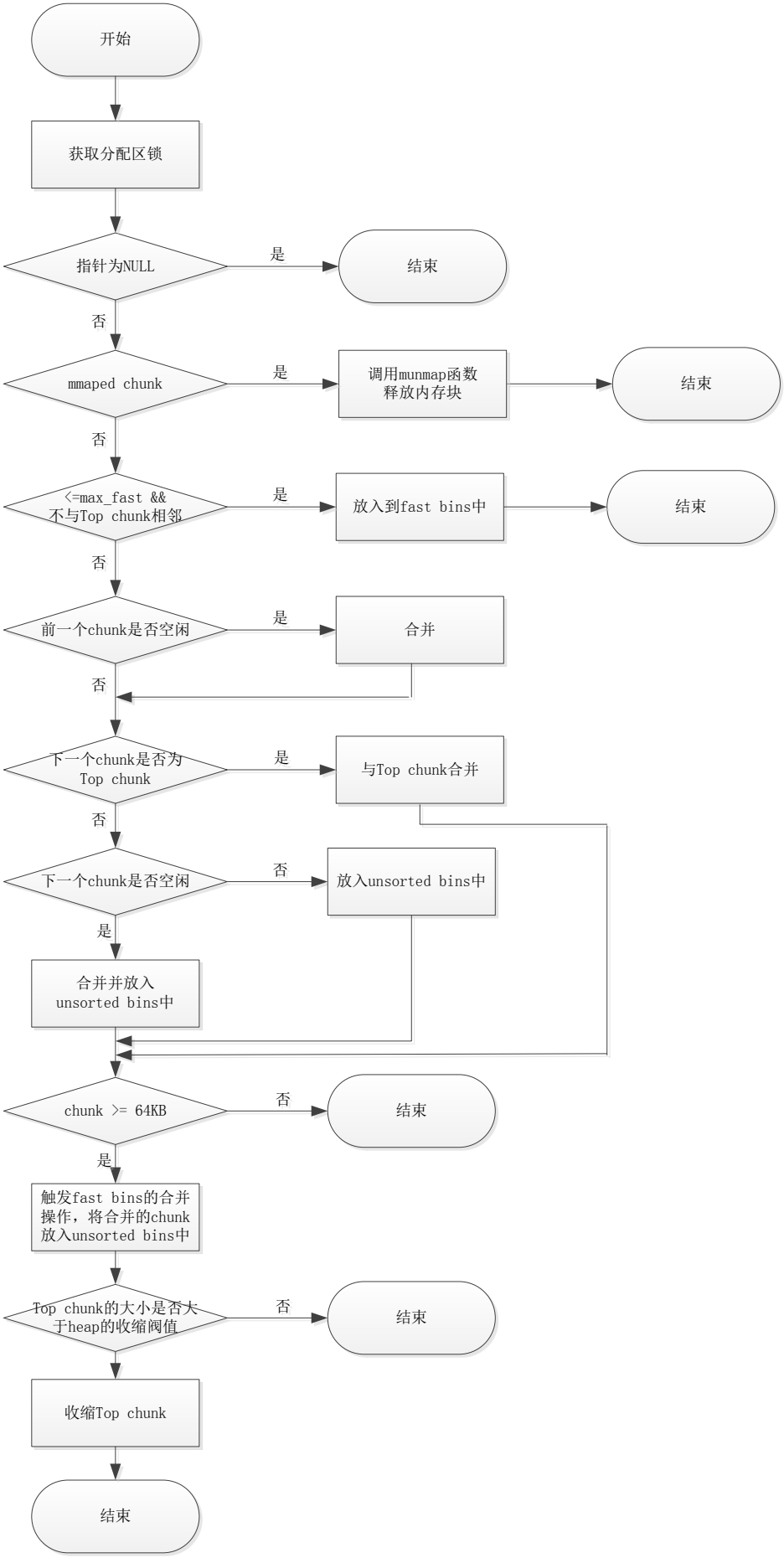
3.7.1 源代码

同 malloc 函数一样，在 ptmalloc 中没有定义 free 函数的实现，而是定义了__libc_free(void *mem)函数，这样做是为了在不同平台下对 free 函数进行不同的实现。_libc_free 函数实现代码如下所示：（参考 Glibc 源码文件/malloc/malloc.c）

```
void __libc_free (void *mem)
 {
   mstate ar_ptr;          //分配区的结构体描述
   mchunkptr p;           /* chunk corresponding to mem */
   void (*hook) (void *, const void *)
     = atomic_forced_read (__free_hook);      //free 的 hook 函数
   if (__builtin_expect (hook != NULL, 0))
     {
       (*hook)(mem, RETURN_ADDRESS (0));
       return;
     }
   if (mem == 0)                              /* free(0) has no effect */
     return;
   p = mem2chunk (mem);  //获取 mem 所指向的 chunk 指针
   if (chunk_is_mmapped (p))                  /* release mmapped memory. */
     {
       /* see if the dynamic brk/mmap threshold needs adjusting */
      //判断是否调整 mmap 分配阀值以及收缩阀值
       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;
           LIBC_PROBE (memory_mallopt_free_dyn_thresholds, 2,
                       mp_.mmap_threshold, mp_.trim_threshold);
         }
       munmap_chunk (p);   //释放内存给系统
       return;
     }
   ar_ptr = arena_for_chunk (p);
   _int_free (ar_ptr, p, 0);   //核心函数，释放非 mmaped chunk 内存块
 }
```

3.7.2 流程图

ptmalloc 响应 free 函数申请内存的过程如下所示：


开始
获取分配区锁
指针为NULL
是
结束
否
mmaped chunk
是
调用munmap函数
释放内存块
结束
否
<=max_fast &&
不与Top chunk相邻
是
放入到fast bins中
结束
否
前一个chunk是否空闲
是
合并
否
下一个chunk是否为
Top chunk
是
与Top chunk合并
否
下一个chunk是否空闲
否
放入unsorted bins中
是
合并并放入
unsorted bins中
chunk >= 64KB
否
结束
是
触发fast bins的合并
操作，将合并的chunk
放入unsorted bins中
Top chunk的大小是否大
于heap的收缩阀值
否
结束
收缩Top chunk
结束

3.7.3 说明

ptmalloc 响应用户释放内存请求的具体步骤：

1) 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 NULL，如果为 NULL，则什么都不做，直接 return。否则，转入下一步。

3) 判断所释放的 chunk 是否为 mmaped chunk，如果是，则调用 munmap()函数来释放 mmaped chunk，解除内存空间映射（如果开启了 mmap 分配阀值的动态调整机制，并且当前回收的 chunk 大小大于 mmap 分配阀值，则将 mmap 分配阀值设置为该 chunk 的大小，将堆的收缩阀值设置为 mmap 分配阀值的 2 倍），释放完成。否则，转入下一步。

4) 判断 chunk 的大小和所处的位置。若 chunk_size <= max_fast，并且 chunk 不与 top chunk 相邻，则转到下一步。否则，转入到第 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。由于 chunk 放入到 fast bins 中时并不修改该 chunk 使用状态位 P，因此，该 chunk 就不会与相邻的 chunk 进行合并。释放完成，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，则转到第 9 步。否则，转到下一步。

8) 判断当前释放 chunk 的下一个 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（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）。转到第 10 步。

9) 执行到这一步，说明释放了一个与 Top chunk 相邻的 chunk。那么无论该 chunk 有多大，都将它与 Top chunk 合并，并更新 Top chunk 的大小等信息。转到第 10 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），如果不是的话，释放完成，程序从 free()函数中返回。 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）；对于非主分配区，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，释放完成，程序从 free()函

数中返回。

需要注意的是：
收缩堆的条件是当前 free 的 chunk 在合并后的 chunk 大小大于 64KB，并且要 Top chunk 的大小要达到堆的收缩阈值，才有可能收缩堆。

4 内存分配的相关函数

4.1 库函数

Linux 操作系统关于动态内存分配提供了如下几个库函数：

函数原型	说明
void *malloc(size_t size);	1. 分配 size 个字节的内存并返回一个指针。 2. size 大小可以为 0 且返回的指针可以使用 free 来释放。
void *calloc(size_t nmemb, size_t size);	1. 同 malloc 基本一致，但是该函数会把申请的内存块清零。 2. nmemb 和 size 都可以为 0 且返回的指针可以使用 free 来释放。
void *realloc(void *ptr, size_t size);	1. 如果 size 比原来小，则前 size 个字节不变，后面的数据被截断。 2. 如果 size 比原来大，则原来的数据全部保留，后面增加的一块内存空间不被初始化。 3. 如果调用 realloc(NULL, size)，则相当于调用 malloc(size)。 4. 如果调用 realloc(ptr, 0)，ptr 不是 NULL，则相当于调用 free(ptr)。
void free(void *ptr);	1. ptr 必须为 malloc、calloc、realloc 返回的指针。 2. 如果 ptr 为 NULL，不执行任何操作，直接返回。 3. 不能对同一个 ptr 使用两次 free，否则可能会导致 heap 崩溃。

4.2 系统调用

Linux 操作系统关于动态内存分配提供了如下几个系统调用

函数原型	说明
int brk(void *addr);	1. addr 用来设置 program break（程序数据段的结尾位置）指向的位置，因此可以通过设置 addr 来增加或减少 heap 的大小。 2. addr 的设置不能使得数据段的大小超过 RLIMIT_DATA 的限制。
void *sbrk(intptr_t increment);	1. 同 brk 基本一致，是通过移动当前 program break 的位置来增加或减少 heap 的大小。 2. 如果 increment 为 0，则返回当前 program

	break 的位置。 3. increment 的设置不能使得数据段的大小超过 RLIMIT_DATA 的限制。
void *mmap(void *, size_t, int, int, int, off_t);	1. mmap 创建一个进程映射的虚拟内存空间 2. 内存起始地址需要以 PAGESIZE 大小（4KB）对齐 3. 以匿名方式分配内存的内容都要被清零
int munmap(void *addr, size_t length);	1. 释放由 mmap 函数返回的指针所指向的内存 2. 释放的内存直接返还给操作系统

注：sbrk 其实是通过 brk 系统调用来实现的一个库函数。

4.3 调优选项

ptmalloc 提供了多个配置选项用于调优，这些调优选项可以通过系统调用 mallopt()来进行设置。如下所示：

函数原型	说明
int mallopt(int param, int value);	1. 参数 param 指定调优选项，参数 value 设定调优数值。 2. 参数 param 可以指定：M_MMAP_MAX 和 M_MMAP_THRESHOLD 等选项。

注：

1) 这些参数选项都有对应的的环境变量可以设置，如参数选项 M_MMAP_MAX 对应的环境变量为 MALLOC_MMAP_MAX_，M_MMAP_THRESHOLD 对应的环境变量为 MALLOC_MMAP_THRESHOLD_。

2) 如果同样的参数选项在环境变量和 mallopt 函数都设置了，mallopt 函数的设置将覆盖环境变量中的值。

接下来，就对这些调优选项进行详细介绍。

4.3.1 M_MMAP_MAX

1) 该参数用来设置进程中使用 mmap 分配的内存块的最大数量。

2) 该参数默认值为 65536。

3) 如果将该参数设置为 0，ptmalloc 将不会使用 mmap 分配大块内存。

举个实例来说明该参数的应用，源码如下所示：

test1.c

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    char *p = malloc(1024*1024); //1MB
    memset(p,1,1024*1024);
```

test2.c

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>

int main(void)
{
    mallopt(M_MMAP_MAX,0);
```

```
    free(p);

    return 0;
}
```

```
    char *p = malloc(1024*1024);
    memset(p,1,1024*1024);
    free(p);

    return 0;
}
```

对上面的源码进行编译，然后使用 strace 工具进行跟踪，结果如下所示：

strace ./test1

```
…
mprotect(0x3de93b6000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)      = 0
mprotect(0x3de8a20000, 4096, PROT_READ) = 0
munmap(0x7fc88c8b5000, 160071)           = 0
mmap(NULL,      1052672,      PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS,      -1,      0)      =
0x7fc88c7b1000
munmap(0x7fc88c7b1000, 1052672)          = 0
exit_group(0)                            = ?
```

strace ./test2

```
…
mprotect(0x3de93b6000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)      = 0
mprotect(0x3de8a20000, 4096, PROT_READ) = 0
munmap(0x7ff34608b000, 160071)           = 0
brk(0)                                   = 0x16d3000
brk(0x17f4000)                           = 0x17f4000
brk(0)                                   = 0x17f4000
brk(0)                                   = 0x17f4000
brk(0)                                   = 0x17f4000
brk(0x16f4000)                           = 0x16f4000
brk(0)                                   = 0x16f4000
exit_group(0)                            = ?
```

说明：

1) test1：
   a) 由于 malloc 申请的内存大小为 1MB，大于 mmap 分配阀值（默认为 128KB），因此会调用 mmap 系统调用来分配内存空间。
   b) free 的时候会调用 munmap 系统调用直接把内存归还给操作系统。

2) test2：
   a) 由于使用 mallopt 函数将 M_MMAP_MAX 参数设置为 0，这样的话，即使申请的内存大小大于 mmap 分配阀值，也不会使用 mmap 系统调用来分配内存空间，而是使用 brk 系统调用在 heap 上分配内存空间。
   b) free 的时候将根据释放内存所处位置以及大小进行不同处理。

4.3.2 M_MMAP_THRESHOLD

1) 该参数用来设置 mmap 分配阀值。

2) 当用户需要分配的内存大于 mmap 分配阀值时，ptmalloc 的 malloc 函数将调用 mmap 分配内存空间，free 这块内存时将调用 munmap 直接释放内存空间给操作系统。

3) 当用户需要分配的内存小于 mmap 分配阀值时，ptmalloc 的 malloc 函数将调用 brk 分配内存空间，free 这块内存时将根据这块内存所处的位置及大小做不同处理。

4) 该参数默认值为 128KB。最小值为 0，最大值：在 32 位系统下为 512KB；在 64 位系统下为 32MB。
5) 默认情况下，glibc 采用了动态调整阀值策略。也就是说，当 free 的一块内存大于当前的阀值且小于最大值，那么分配阀值将被调整为当前释放内存块的大小，且 heap 的收缩阀值将调整为新的分配阀值的 2 倍。

注：动态调整阀值策略可以通过设置 M_TRIM_THRESHOLD、M_TOP_PAD、 M_MMAP_THRESHOLD、M_MMAP_MAX 其中的任意一个参数使之关闭。

6) 使用 mmap 分配内存的优点：
   a) mmap 分配内存的大小不受数据段大小（RLIMIT_DATA）的限制。
   b) mmap 分配的内存空间在释放时不会被 ptmalloc 缓存在 chunk 中，而是直接把内存归还给操作系统，这样可以使得被释放的内存及时归还给操作系统。

7) 使用 mmap 分配内存的缺点：
   a) 由于释放的内存不会被 ptmalloc 缓存在 chunk 中，这样再次申请内存空间时，需要再次调用系统调用 mmap，这样会导致开销增大了。
   b) 由于 mmap 分配内存需要按页对齐，会导致内存浪费。
   c) mmap 对匿名分配的内存的内容强制清零，这样会比较低效。

举个实例来说明该参数的应用，源码如下所示：

test1.c

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
	char		*p1		=
malloc(1024*1024);//1MB
	memset(p1,1,1024*1024);
	free(p1);

	char *p2 = malloc(1024*1024);
	memset(p2,1,1024*1024);
	free(p2);

	return 0;
}
```

test2.c

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>

int main(void)
{
    //设置分配阀值为 256KB
    mallopt(M_MMAP_THRESHOLD,256*1024);

    char *p1 = malloc(1024*1024);
    memset(p1,1,1024*1024);
    free(p1);

    char *p2 = malloc(1024*1024);
    memset(p2,1,1024*1024);
    free(p2);

    return 0;
}
```

对上面的源码进行编译，然后使用 strace 工具进行跟踪，结果如下所示：

strace ./test1

```
…
```

strace ./test2

```
…
```

```
mprotect(0x3de93b6000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)      = 0
mprotect(0x3de8a20000, 4096, PROT_READ) = 0
munmap(0x7fa4f4368000, 160071)           = 0
mmap(NULL,      1052672,      PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS,      -1,      0)      =
0x7fa4f4264000
munmap(0x7fa4f4264000, 1052672)          = 0
brk(0)                                   =
0x854000
brk(0x975000)                            =
0x975000
brk(0)                                   =
0x975000
exit_group(0)                            = ?
```

```
mprotect(0x3de93b6000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)      = 0
mprotect(0x3de8a20000, 4096, PROT_READ) = 0
munmap(0x7f0d67f25000, 160071)           = 0
mmap(NULL,      1052672,      PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0d67e21000
munmap(0x7f0d67e21000, 1052672)          = 0
mmap(NULL,      1052672,      PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0d67e21000
munmap(0x7f0d67e21000, 1052672)          = 0
exit_group(0)                            = ?
```

说明：

1) test1：
   a) 当第一次调用 malloc 申请 1MB 内存时，（大于 mmap 分配阀值，默认为 128KB），会调用 mmap 系统调用来分配内存空间，free 的时候发现内存大于分配阀值且小于最大值，那么分配阀值将被调整为当前释放内存块的大小，即 1MB。
   b) 当第二次调用 malloc 申请 1MB 内存时，由于没有超过分配阀值（此时分配阀值已经为 1MB），因此不会调用 mmap 来分配内存，而是调用 brk 在 heap 上分配内存。free 的时候将根据释放内存所处位置以及大小进行不同处理。

2) test2：
   a) 程序首先使用了 mallopt()函数设置了分配阀值为 256KB，关闭了动态调整分配阀值的机制。
   b) 当第一次调用 malloc 申请 1MB 内存时，（大于 mmap 分配阀值 256KB），会调用 mmap 系统调用来分配内存空间，free 的时候调用 munmap 直接将内存归还给操作系统。
   c) 当第二次调用 malloc 申请 1MB 内存时，（大于 mmap 分配阀值 256KB），会调用 mmap 系统调用来分配内存空间，free 的时候调用 munmap 直接将内存归还给操作系统。

4.3.3 M_MXFAST

1) 该参数用于设置 fast bins 中保存的 chunk 的最大大小。

2) 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。
3) 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，却不能归还给操作系统，导致内存暴增。

4) 该参数默认值为 64*sizeof(size_t)/4，最大值为 80*sizeof(size_t)/4。因此，在 64 位系统下，默认值为 128B，最大值为 160B。

5) 设置该参数为 0，将禁止使用 fast bins。

举个实例来说明该参数的应用，源码如下所示：

test1.c （test2.c 的源码把红色字体的注释去掉）

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <malloc.h>

void display_mallinfo(void)
{       struct mallinfo mi;         mi = mallinfo();
        printf("Total non-mmapped bytes (arena):       %d\n", mi.arena);
        printf("# of free chunks (ordblks):            %d\n", mi.ordblks);
        printf("# of free fastbin blocks (smblks):     %d\n", mi.smblks);
        printf("# of mapped regions (hblks):           %d\n", mi.hblks);
        printf("Bytes in mapped regions (hblkhd):      %d\n", mi.hblkhd);
        printf("Max. total allocated space (usmblks):  %d\n", mi.usmblks);
        printf("Free bytes held in fastbins (fsmblks): %d\n", mi.fsmblks);
        printf("Total allocated space (uordblks):      %d\n", mi.uordblks);
        printf("Total free space (fordblks):           %d\n", mi.fordblks);
        printf("Topmost releasable block (keepcost):   %d\n", mi.keepcost);
}

int main(void)
{
        //mallopt(M_MXFAST,0);       //禁用 fast bins

        printf("============== Before allocating blocks ==============\n");
        display_mallinfo();

        int *array[20];
        int i;

        for (i = 0; i < 20; i++)
                array[i] = malloc(32); //32B

        printf("============== After allocating blocks ==============\n");
        display_mallinfo();

        for (i = 0; i < 20; i++)
                free(array[i]);

       printf("============== After freeing blocks ==============\n");
       display_mallinfo();
       return 0;
}
```

注：mallinfo 函数只能获取主分配区的信息，要想同时获取非主分配区的信息使用 malloc_info 函数。

对上面的源码进行编译，然后执行，结果如下所示：

./test1		./test2	
========== Before allocating blocks ==========		========== Before allocating blocks ==========	
Total non-mmapped bytes (arena):	0	Total non-mmapped bytes (arena):	0
# of free chunks (ordblks):	1	# of free chunks (ordblks):	1
# of free fastbin blocks (smblks):	0	# of free fastbin blocks (smblks):	0
# of mapped regions (hblks):	0	# of mapped regions (hblks):	0
Bytes in mapped regions (hblkhd):	0	Bytes in mapped regions (hblkhd):	0
Max. total allocated space (usmblks):	0	Max. total allocated space (usmblks):	0
Free bytes held in fastbins (fsmblks):	0	Free bytes held in fastbins (fsmblks):	0
Total allocated space (uordblks):	0	Total allocated space (uordblks):	0
Total free space (fordblks):	0	Total free space (fordblks):	0
Topmost releasable block (keepcost):	0	Topmost releasable block (keepcost):	0
========== After allocating blocks ==========		========== After allocating blocks ==========	
Total non-mmapped bytes (arena):	135168	Total non-mmapped bytes (arena):	135168
# of free chunks (ordblks):	1	# of free chunks (ordblks):	1
# of free fastbin blocks (smblks):	0	# of free fastbin blocks (smblks):	0
# of mapped regions (hblks):	0	# of mapped regions (hblks):	0
Bytes in mapped regions (hblkhd):	0	Bytes in mapped regions (hblkhd):	0
Max. total allocated space (usmblks):	0	Max. total allocated space (usmblks):	0
Free bytes held in fastbins (fsmblks):	0	Free bytes held in fastbins (fsmblks):	0
Total allocated space (uordblks):	960	Total allocated space (uordblks):	960
Total free space (fordblks):	134208	Total free space (fordblks):	134208
Topmost releasable block (keepcost):	134208	Topmost releasable block (keepcost):	134208
========== After freeing blocks ==========		========== After freeing blocks ==========	
Total non-mmapped bytes (arena):	135168	Total non-mmapped bytes (arena):	135168
# of free chunks (ordblks):	1	# of free chunks (ordblks):	1
# of free fastbin blocks (smblks):	20	# of free fastbin blocks (smblks):	0
# of mapped regions (hblks):	0	# of mapped regions (hblks):	0
Bytes in mapped regions (hblkhd):	0	Bytes in mapped regions (hblkhd):	0
Max. total allocated space (usmblks):	0	Max. total allocated space (usmblks):	0
Free bytes held in fastbins (fsmblks):	960	Free bytes held in fastbins (fsmblks):	0
Total allocated space (uordblks):	0	Total allocated space (uordblks):	0
Total free space (fordblks):	135168	Total free space (fordblks):	135168
Topmost releasable block (keepcost):	134208	Topmost releasable block (keepcost):	135168

说明：

1) test1：由于 malloc 申请的内存大小为 32B（M_MXFAST 默认为 128B），因此释放的时候，这些小内存块被放入到 fast bins 中。
2) test2：由于程序在开始的位置调用 mallopt 函数禁用了 fast bins,因此不管 malloc 申请的内存块为多大，释放的时候都不会放入 fast bins 中。

4.3.4 M_TOP_PAD

1) 该参数用来设置 Top chunk 的最小大小，该参数默认值为 128KB。
   注：这里的最小大小是指 heap 收缩后 Top chunk 的最小大小。

2) 如果该参数设置的过小，由于可能频繁的申请内存和释放内存，那么可能会增加 brk 系统调用的次数。

3) 如果该参数设置的过大，由于内存申请和释放的频率非常小，那么可能会导致 Top chunk 过大，浪费内存空间。

举个实例来说明该参数的应用，源码如下所示：

<table>
<tr><th>test1.c</th><th>test2.c</th></tr>
<tr><td>

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>

int main(void)
{
   printf("current brk =%x\n",(int*)sbrk(0));
   printf("---------after malloc--------\n");

   char *p1 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   char *p2 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   char *p3 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   char *p4 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   char *p5 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   free(p1);
   free(p2);
   free(p3);
   free(p4);
   free(p5);

   printf("---------after free----------\n");
   printf("current brk =%x\n",(int*)sbrk(0));

   return 0;
  }
```

</td><td>

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>

int main(void)
{
   //首先申请的内存大小=400KB+chunk_size3
   mallopt(M_TOP_PAD,400*1024);

   printf("current brk =%x\n",(int*)sbrk(0));
   printf("---------after malloc--------\n");

   char *p1 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   char *p2 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   char *p3 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   char *p4 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   char *p5 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));

   free(p1);
   free(p2);
   free(p3);
   free(p4);
   free(p5);

   printf("---------after free----------\n");
   printf("current brk =%x\n",(int*)sbrk(0));
   return 0;
}
```

</td></tr>
</table>

对上面的源码进行编译，然后执行，结果如下所示：

./test1	./test2

```
current brk =15b7000
--------after malloc--------
current brk =15f1000
current brk =15f1000
current brk =1623000
current brk =1623000
current brk =1655000
---------after free----------
current brk =15d8000
```

```
current brk =253d000
---------after malloc--------
current brk =25bb000
current brk =25bb000
current brk =25bb000
current brk =25bb000
current brk =25bb000
---------after free----------
current brk =25a2000
```

说明：

1) test1：分配前的 brk 为 0x15b7000,释放后的 brk 为 0x15d8000，因此释放后的 Top Chunk 大小为 0x21000 个字节，即 132KB。

2) test2：分配前的 brk 为 0x253d000,由于设置了 Top chunk 释放后的最小大小，释放后的 brk 为 0x25a2000，因此释放后的 Top Chunk 大小为 0x65000 个字节，即 404KB。

4.3.5 M_TRIM_THRESHOLD

1) 该参数用于设置 heap 的收缩阀值。如果 Top chunk 的大小大于收缩阀值，那么系统将会启动收缩阀值机制。(如，使用 mallopt 设置 Top chunk 的大小为 400KB，设置 heap 的收缩阀值为 800KB，当 Top chunk 达到 600KB 时，并不会启动收缩阀值)。

2) 如果该参数设置的过小，由于可能频繁的申请内存和释放内存，那么可能会增加 brk 系统调用的次数。

3) 如果该参数设置的过大，由于内存申请和释放的频率非常小，那么可能会导致 Top chunk 过大，浪费内存空间。

4) 该参数默认值为 128KB，设置该参数为-1，将禁止 heap 的收缩。

举个实例来说明该参数的应用，源码如下所示：

test1.c

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>

int main(void)
{
   printf("--------before malloc--------\n");
   printf("current brk =%x\n",(int*)sbrk(0));

   printf("---------after malloc--------\n");

   char *p1 = (char*)malloc(100*1024);
   printf("current brk =%x\n",(int*)sbrk(0));
```

test2.c

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <malloc.h>
int main(void)
{
   //设置收缩阀值为 800KB
   mallopt((M_TRIM_THRESHOLD,800*1024);

   printf("--------before malloc--------\n");
   printf("current brk =%x\n",(int*)sbrk(0));

   printf("---------after malloc--------\n");

   char *p1 = (char*)malloc(100*1024);
```

```
    char *p2 = (char*)malloc(100*1024);
    printf("current brk =%x\n",(int*)sbrk(0));

    char *p3 = (char*)malloc(100*1024);
    printf("current brk =%x\n",(int*)sbrk(0));

    char *p4 = (char*)malloc(100*1024);
    printf("current brk =%x\n",(int*)sbrk(0));

    char *p5 = (char*)malloc(100*1024);
    printf("current brk =%x\n",(int*)sbrk(0));

    free(p1);
    free(p2);
    free(p3);
    free(p4);
    free(p5);

    printf("---------after free----------\n");
    printf("current brk =%x\n",(int*)sbrk(0));

    return 0;
}
```

```
    printf("current brk =%x\n",(int*)sbrk(0));

    char *p2 = (char*)malloc(100*1024);
    printf("current brk =%x\n",(int*)sbrk(0));

    char *p3 = (char*)malloc(100*1024);
    printf("current brk =%x\n",(int*)sbrk(0));

    char *p4 = (char*)malloc(100*1024);
    printf("current brk =%x\n",(int*)sbrk(0));

    char *p5 = (char*)malloc(100*1024);
    printf("current brk =%x\n",(int*)sbrk(0));

    free(p1);
    free(p2);
    free(p3);
    free(p4);
    free(p5);

    printf("---------after free----------\n");
    printf("current brk =%x\n",(int*)sbrk(0));

    return 0;
}
```

对上面的源码进行编译，然后执行，结果如下所示：

./test1	./test2
--------before malloc--------	--------before malloc--------
current brk =d46000	current brk =1da1000
---------after malloc--------	---------after malloc--------
current brk =d80000	current brk =1ddb000
current brk =d80000	current brk =1ddb000
current brk =db2000	current brk =1e0d000
current brk =db2000	current brk =1e0d000
current brk =de4000	current brk =1e3f000
---------after free----------	---------after free----------
current brk =d67000	current brk =1e3f000

说明：

1) test1：分配前的 brk 为 0xd46000,释放后的 brk 为 0xd67000，因此释放后的 Top Chunk 大小为 0x21000 个字节，即 132KB。

2) test2：分配前的 brk 为 0x1da1000,由于设置了 heap 的收缩阀值为 800KB，因此，释放后的 brk 没有收缩，仍然为 0x1e3f000。

5 实例分析

5.1 指针数组

5.1.1 源代码

通过一段代码来说明指针数组在申请内存时，需要注意的细节，如下所示：

test1.c

```
#include <stdlib.h>
#include <stdio.h>
#define M 10
#define N 10

int main(void)
{
   char* a[M];//指针的数组
   int i;

   for (i=0; i<M; i++)
       a[i]  =  malloc(sizeof(char)  *
   N);

   printf("%d\n", sizeof(a));
   printf("%d\n", sizeof(a[0]));

   for(i=0;          i<M;          i++)
       free(a[i]);

   return 0;
}
```

test2.c

```
#include <stdlib.h>
#include <stdio.h>
#define M 10
#define N 10

int main(void)
{
    char* a[M];//指针的数组
    int i;

    a[0] = malloc(sizeof(char) * M *
    N);
    for(i=1; i<M; i++)
        a[i] = a[i-1] + n;

    printf("%d\n", sizeof(a));
    printf("%d\n", sizeof(a[0]));

    free(a[0]);

    return 0;
}
```

注：本段代码没有任何实际意义，仅仅作为说明。

5.1.2 代码分析

1) test1.c:（分别给每个数组元素申请内存块）
   a) 优点：能够灵活的给每个数组元素申请不同大小的内存块，且每个数组元素可能自由的选择释放内存块。
   b) 缺点：给个数组元素都要调用 malloc 函数，效率低。且每个内存块都需要用一个 chunk 结构来保存，浪费内存。

2) test2.c：（一次分配内存(保证内存的连续性)）
   a) 优点：只调用一次 malloc 函数，效率高。由于该段内存只用一个 chunk 结构来保存，内存利用率高。
   b) 缺点：不能够灵活的给每个数组元素申请不同大小的内存块，且每个数组元素指

向的内存不可释放，只有等所有数组元素指向的内存都不使用的时候，才可将整个内存块释放掉。

5.2 内存泄露 1

5.2.1 源代码

通过一段代码来说明一个典型的内存泄露问题，如下所示：

test1.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>

#define ARRAY_SIZE 5000
#define MALLOC_SIZE 100*1024   //100KB

int main(void)
{
    char command[100];
    sprintf(command,"%s/%d/%s",\
    "cat /proc",getpid(),"statm");

    printf("---------before malloc----------\n");
    printf("size\tResident\tShared\tTrs\tLrs\tDrs\t
dt\n");
    system(command);

    int *array[ARRAY_SIZE];
    int loop;

    for (loop = 0; loop < ARRAY_SIZE; loop++)
    {
        array[loop] = malloc(MALLOC_SIZE);
        memset(array[loop],0,MALLOC_SIZE);
     }

     printf("---------after malloc----------\n");
     printf("size\tResident\tShared\tTrs\tLrs\tDrs\
     tdt\n");
     system(command);

     for (loop =0 ; loop < ARRAY_SIZE; loop++)
         free(array[loop]);

     printf("--------after free----------\n");
     printf("size\tResident\tShared\tTrs\tLrs\tDrs\
```

test2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>

#define ARRAY_SIZE 5000
#define      MALLOC_SIZE      100*1024
//100KB

int main(void)
{
    char command[100];
    sprintf(command,"%s/%d/%s",\
    "cat /proc",getpid(),"statm");

    printf("---------before malloc---
  -------\n");
    printf("size\tResident\tShared\tT
rs\tLrs\tDrs\tdt\n");
     system(command);

    int *array[ARRAY_SIZE];
    int loop;

    for (loop = 0; loop < ARRAY_SIZE;
    loop++)
    {
        array[loop]                  =
    malloc(MALLOC_SIZE);

memset(array[loop],0,MALLOC_SIZE);
     }

     printf("---------after malloc---
  -------\n");
     printf("size\tResident\tShared\t
     Trs\tLrs\tDrs\tdt\n");
     system(command);
```

```
tdt\n");
    system(command);

    return 0;
}
```

```
    for (loop =0 ; loop < ARRAY_SIZE
- 1; loop++)
        free(array[loop]);

    printf("--------after free------
  ----\n");
    printf("size\tResident\tShared\t
Trs\tLrs\tDrs\tdt\n");
    system(command);

    free(array[ARRAY_SIZE - 1]);

    return 0;
}
```

5.2.2 测定结果

对上面的源码进行编译，然后执行，结果如下所示：

./test1						
----------before malloc----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
1041	81	64	1	0	47	0
----------after malloc----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
126068	125138	101	1	0	125074	0
----------after free----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
1074	164	102	1	0	80	0

./test2						
----------before malloc----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
1041	81	64	1	0	47	0
----------after malloc----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
126068	125138	101	1	0	125074	0
----------after free----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
126068	125151	102	1	0	125074	0

注：（上面的数字是以页（4KB）为单位）

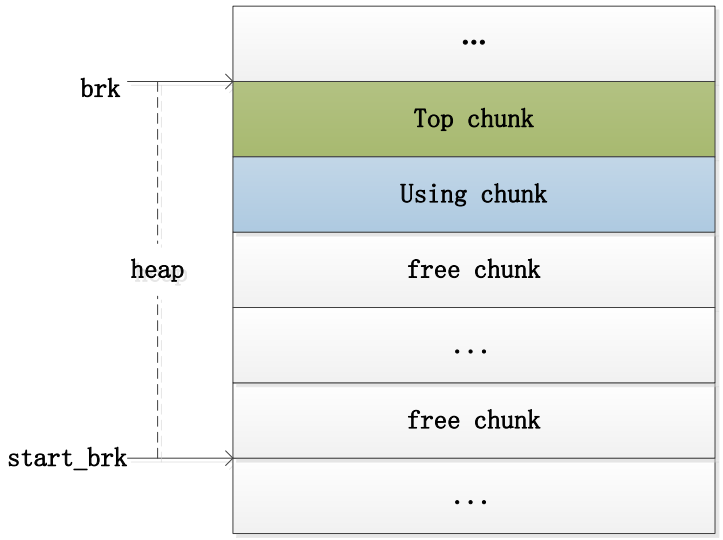
1) size：进程的虚拟地址空间大小

2) Resident：进程正在使用的物理内存大小

5.2.3 原因分析

3) test1.c：与期望一致，申请的内存在释放后都归还给操作系统了。

4) test2.c：与期望不一致，只有最后一次申请的内存没有调用 free，竟导致所有的内存都没有归还给操作系统。通过下图来说明，如下所示：



分析：

i. ptmalloc 管理 heap 空间内存是通过移动 brk 指针来增大或减小 heap 空间的大小。

ii. 由上图可以看到，与 Top Chunk 相邻的一块内存正在使用，那么 brk 指针就不能向下移动，因此也就不能把前面已释放的内存归还给操作系统。

iii. 这里的内存泄露是由于 ptmalloc 通过简单的移动 brk 指针来对 heap 空间进行管理导致的，非用户代码有误。

5.2.4 解决方法

1) 调用 mallopt 函数调整 mmap 分配阀值，使得 malloc 最终调用 mmap 系统调用分配内存，但是这种方法由于每次都要调用系统调用，效率差。

2) 针对特定的应用程序，编写一个特定的内存池来进行内存分配，效率既高又不会导致内存泄露。

5.3 内存泄露 2

5.3.1 源代码

通过一段代码来说明另一个典型的内存泄露问题，如下所示：

test1.c	test2.c
#include <stdio.h> #include <unistd.h> #include <stdlib.h> #include <list> #include <malloc.h> #define NUM 50000 #define SIZE 1024	#include <stdio.h> #include <unistd.h> #include <stdlib.h> #include <list> #include <malloc.h> #define NUM 50000 #define SIZE 1024

```
int main()
{
    char command[100];
    sprintf(command,    "%s/%d/%s",    "cat    /proc",
getpid(), "statm");

    printf("--------before malloc---------\n");
    printf("size\tResident\tShared\tTrs\tLrs\tDrs\tdt
\n");
    system(command);
    //display_mallinfo();

    char **ptrs = new char *[NUM];
    for (size_t i = 0; i < NUM; ++i)
        ptrs[i] = (char *)malloc(SIZE);

     printf("-------after malloc----------\n");
     printf("size\tResident\tShared\tTrs\tLrs\tDrs\td
t\n");
     system(command);
    //display_mallinfo();

     for (size_t i = 0; i < NUM; ++i)
          free(ptrs[i]);

     free(ptrs);

     printf("---------after free----------\n");
     printf("size\tResident\tShared\tTrs\tLrs\tDrs\td
t\n");
     system(command);
    //display_mallinfo();

        return 0;
}
```

```
int main()
{
    char command[100];
    sprintf(command,"%s/
%d/%s","cat/proc",
getpid(), "statm");

    printf("---------before malloc----------\n");
    printf("size\tResident\tShared\tTrs\tLrs\tDrs\t
dt\n");
    system(command);
    //display_mallinfo();

    std::list < char *>ptrs;
    for (size_t i = 0; i < NUM; ++i)
            ptrs.push_back((char *) malloc(SIZE));

     printf("---------after malloc----------\n");
     printf("size\tResident\tShared\tTrs\tLrs\tDrs\
tdt\n");
      system(command);
    //display_mallinfo();

      for (size_t i = 0; i < NUM; ++i) {
           free(ptrs.back());
           ptrs.pop_back();
      }

      ptrs.clear();

      printf("---------after free ----------\n");
      printf("size\tResident\tShared\tTrs\tLrs\tDrs
\tdt\n");
      system(command);
    //display_mallinfo();

      return 0;
}
```

注：display_mallinfo()函数的实现代码在 4.3.3 小节可以找到。

5.3.2 测定结果

对上面的源码进行编译，然后执行，结果如下所示：

./test1	./test2
----------before malloc----------	----------before malloc----------
size Resident Shared Trs Lrs Drs dt	size Resident Shared Trs Lrs Drs dt

3119	194	163	1	0	70	0
----------after malloc----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
15922	12985	208	1	0	12873	0
----------after free----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
3152	279	209	1	0	103	0

3120	195	163	2	0	70	0
----------after malloc----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
16221	13316	210	2	0	13171	0
----------after free----------						
size	Resident	Shared	Trs	Lrs	Drs	dt
16221	13336	212	2	0	13171	0

说明：

1) test1.c：与期望一致，申请的内存在释放后都归还给操作系统了。

2) test2.c：与期望不一致，仅仅只是使用了 STL（标准模板库）的 list 来对申请的内存进行管理，导致申请的内存在全部释放后却没有归还给操作系统。

5.3.3 原因分析

通过如下两种方法来对 test2.c 进行分析：

1) 将上述代码中的注释去掉，使程序调用 display_mallinfo()函数，输出结果如下：

./test1	
============ Before allocating blocks ============	
Total non-mmapped bytes (arena):	0
# of free chunks (ordblks):	1
# of free fastbin blocks (smblks):	0
# of mapped regions (hblks):	0
Bytes in mapped regions (hblkhd):	0
Max. total allocated space (usmblks):	0
Free bytes held in fastbins (fsmblks):	0
Total allocated space (uordblks):	0
Total free space (fordblks):	0
Topmost releasable block (keepcost):	0
============ After allocating blocks ============	
Total non-mmapped bytes (arena):	512892928
# of free chunks (ordblks):	1
# of free fastbin blocks (smblks):	0
# of mapped regions (hblks):	1
Bytes in mapped regions (hblkhd):	401408
Max. total allocated space (usmblks):	0
Free bytes held in fastbins (fsmblks):	0
Total allocated space (uordblks):	512800000
Total free space (fordblks):	92928
Topmost releasable block (keepcost):	92928
============== After freeing blocks ==============	
Total non-mmapped bytes (arena):	135168
# of free chunks (ordblks):	1

./test2	
============ Before allocating blocks ============	
Total non-mmapped bytes (arena):	0
# of free chunks (ordblks):	1
# of free fastbin blocks (smblks):	0
# of mapped regions (hblks):	0
Bytes in mapped regions (hblkhd):	0
Max. total allocated space (usmblks):	0
Free bytes held in fastbins (fsmblks):	0
Total allocated space (uordblks):	0
Total free space (fordblks):	0
Topmost releasable block (keepcost):	0
============ After allocating blocks ============	
Total non-mmapped bytes (arena):	53661696
# of free chunks (ordblks):	1
# of free fastbin blocks (smblks):	0
# of mapped regions (hblks):	0
Bytes in mapped regions (hblkhd):	0
Max. total allocated space (usmblks):	0
Free bytes held in fastbins (fsmblks):	0
Total allocated space (uordblks):	53600000
Total free space (fordblks):	61696
Topmost releasable block (keepcost):	61696
============== After freeing blocks ==============	
Total non-mmapped bytes (arena):	53661696
# of free chunks (ordblks):	50001

# of free fastbin blocks (smblks): 0 # of mapped regions (hblks): 0 Bytes in mapped regions (hblkhd): 0 Max. total allocated space (usmblks): 0 Free bytes held in fastbins (fsmblks): 0 Total allocated space (uordblks): 0 Total free space (fordblks): 135168 Topmost releasable block (keepcost): 135168	# of free fastbin blocks (smblks): 50000 # of mapped regions (hblks): 0 Bytes in mapped regions (hblkhd): 0 Max. total allocated space (usmblks): 0 Free bytes held in fastbins (fsmblks): 1600000 Total allocated space (uordblks): 0 Total free space (fordblks): 53661696 Topmost releasable block (keepcost): 61696

根据 test2 的输出结果，有以下几个发现：

a) fast bins 中有 50000 个 chunk，chunk 数量与代码中的循环次数 NUM 一致，且每个 chunk 的平均大小为 32B(通过 fsmblks/smblks 计算得出)。

b) 空闲 chunk（ordblks）的数量为 500001，且每个 chunk 的平均大小约为 1KB（通过 fordblks/ordblks 计算得出）。这个值与 malloc 申请的大小基本一致。

c) Top chunk（keepcost）的大小仅仅为 61696B。

根据上面几个发现，可以推断出：

a) 由于 heap 空间中存在大量的 chunk 放入 fast bins 中，而 fast bins 中的 chunk 由于不修改使用标志 P，使得其无法与相邻 chunk 进行合并，导致 Top chunk 仅仅为 61696B，从而使得 heap 空间不能够收缩，内存也就无法归还给操作系统。

b) 然后，由于 mallloc 每次申请的内存都为 1KB，远大于 M_MXFAST（默认值为 128B），那么释放的时候，该段内存是不应该放入 fast bins 中的。

接下来，就需要研究 fast bins 中的 chunk 是如何产生的。

2) 使用工具 gdb 对代码进行调式。为了调式方便，对源代码进行修改，如下所示：

test1.c

```
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <list>

#define NUM 5
#define SIZE 16

int main()
{
    char *ptrs[NUM];
    for (size_t i = 0; i < NUM; ++i)
    {
        ptrs[i] = (char *)malloc(SIZE);
        memset(ptrs[i],1,SIZE);
    }

    for (size_t i = 0; i < NUM; ++i)
        free(ptrs[i]);
```

test2.c

```
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <list>

#define NUM 5
#define SIZE 16

int main()
{
    char *p[NUM];
    std::list < char *>ptrs;
    for (size_t i = 0; i < NUM; ++i)
    {
        p[i] = (char*)malloc(SIZE);
        memset(p[i],1,SIZE);
        ptrs.push_back(p[i]);
    }
```

```
 return 0;
}
```

```
    for (size_t i = 0; i < NUM; ++i)
    {
        free(ptrs.back());
         ptrs.pop_back();
    }

    ptrs.clear();

    return 0;
}
```

对上面的源码进行编译，然后使用 gdb 进行调式，结果如下所示：

gdb ./test1

```
[root@localhost testdir]# gdb ./test1
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-45.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /mnt/home/renyl/testdir/test1...done.
(gdb) list
4       #include <list>
5
6       #define NUM 5
7       #define SIZE 16
8
9       int main()
10      {
11
12              char *ptrs[NUM];
13              for (size_t i = 0; i < NUM; ++i) {
(gdb) list
14
15                      ptrs[i] = (char *)malloc(SIZE);
16                      memset(ptrs[i],1,SIZE);
17              }
18
19              for (size_t i = 0; i < NUM; ++i) {
20                      free(ptrs[i]);
21              }
22
23              return 0;
(gdb) b 19
Breakpoint 1 at 0x4006cf: file test1.cpp, line 19.
(gdb) start
Temporary breakpoint 2 at 0x400688: file test1.cpp, line 13.
```

```
Starting program: /mnt/home/renyl/testdir/./test1

Temporary breakpoint 2, main () at test1.cpp:13
13              for (size_t i = 0; i < NUM; ++i) {
Missing separate debuginfos, use: debuginfo-install glibc-2.17-36.el7.x86_64
libgcc-4.8.2-3.el7.x86_64 libstdc++-4.8.2-3.el7.x86_64
(gdb) c
Continuing.

Breakpoint 1, main () at test1.cpp:19
19              for (size_t i = 0; i < NUM; ++i) {
(gdb) p /x ptrs[0]
$1 = 0x602010
(gdb) x /64x 0x602010
0x602010:       0x01010101      0x01010101      0x01010101      0x01010101
0x602020:       0x00000000      0x00000000      0x00000021      0x00000000
0x602030:       0x01010101      0x01010101      0x01010101      0x01010101
0x602040:       0x00000000      0x00000000      0x00000021      0x00000000
0x602050:       0x01010101      0x01010101      0x01010101      0x01010101
0x602060:       0x00000000      0x00000000      0x00000021      0x00000000
0x602070:       0x01010101      0x01010101      0x01010101      0x01010101
0x602080:       0x00000000      0x00000000      0x00000021      0x00000000
0x602090:       0x01010101      0x01010101      0x01010101      0x01010101
0x6020a0:       0x00000000      0x00000000      0x00020f61      0x00000000
0x6020b0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020c0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020d0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020e0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020f0:       0x00000000      0x00000000      0x00000000      0x00000000
0x602100:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

gdb ./test2

```
[root@localhost testdir]# gdb ./test2
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-45.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /mnt/home/renyl/testdir/test2...done.
(gdb) list
5
6       #define NUM 5
7       #define SIZE 16
8
9       int main()
```

```
10      {
11
12              char *p[NUM];
13
14              std::list < char *>ptrs;
(gdb) list
15              for (size_t i = 0; i < NUM; ++i) {
16
17                      p[i] = (char*)malloc(SIZE);
18                      memset(p[i],1,SIZE);
19                      ptrs.push_back(p[i]);
20              }
21
22              for (size_t i = 0; i < NUM; ++i) {
23                      free(ptrs.back());
24                      ptrs.pop_back();
(gdb) b 22
Breakpoint 1 at 0x400a0a: file test2.cpp, line 22.
(gdb) start
Temporary breakpoint 2 at 0x400999: file test2.cpp, line 14.
Starting program: /mnt/home/renyl/testdir/./test2
Temporary breakpoint 2, main () at test2.cpp:14
14              std::list < char *>ptrs;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-36.el7.x86_64
libgcc-4.8.2-3.el7.x86_64 libstdc++-4.8.2-3.el7.x86_64
(gdb) c
Continuing.

Breakpoint 1, main () at test2.cpp:22
22              for (size_t i = 0; i < NUM; ++i) {
(gdb) p /x p[0]
$1 = 0x603010
(gdb) x /64x 0x603010
0x603010:       0x01010101      0x01010101      0x01010101      0x01010101
0x603020:       0x00000000      0x00000000      0x00000021      0x00000000
0x603030:       0x00603070      0x00000000      0xffffe280      0x00007fff
0x603040:       0x00603010      0x00000000      0x00000021      0x00000000
0x603050:       0x01010101      0x01010101      0x01010101      0x01010101
0x603060:       0x00000000      0x00000000      0x00000021      0x00000000
0x603070:       0x006030b0      0x00000000      0x00603030      0x00000000
0x603080:       0x00603050      0x00000000      0x00000021      0x00000000
0x603090:       0x01010101      0x01010101      0x01010101      0x01010101
0x6030a0:       0x00000000      0x00000000      0x00000021      0x00000000
0x6030b0:       0x006030f0      0x00000000      0x00603070      0x00000000
0x6030c0:       0x00603090      0x00000000      0x00000021      0x00000000
0x6030d0:       0x01010101      0x01010101      0x01010101      0x01010101
0x6030e0:       0x00000000      0x00000000      0x00000021      0x00000000
0x6030f0:       0x00603130      0x00000000      0x006030b0      0x00000000
0x603100:       0x006030d0      0x00000000      0x00000021      0x00000000
(gdb)
```

分析：（根据 gdb 的调试结果）

a) list 在执行 push_back 操作时会动态申请小块内存，用来存储链表中的内容。
b) list 在执行 pop_back 操作时会释放小块内存，但由于内存块只有 32B，小于 M_MXFAST（默认值为 128B），小块内存会被放入 fast bins 中。
c) fast bins 中的小块不会修改使用标志 P，致使不能与相邻的 chunk 就进行合并。
d) fast bins 中的小块贯穿于整个 heap 空间，导致 Top chunk 过小，因此 heap 空间无法收缩，内存也就无法归还给操作系统。

找到了原因，如何解决呢？

5.3.4 解决方法

1) 调整 mmap 分配阀值。这种方法虽然可以解决问题，但效率极差，不可采取。

2) 调用 mallopt 函数来禁用 fast bins。对于该程序所出现的问题可以使用这种方法来解决，但是在某些程序中不易采用禁用 fast bins 这种方法，因为小块内存放入 fast bins 中不需要修改使用标志 P，也就不会进行合并操作，当用户再次申请小块内存时，可以直接从 fast bins 中获取，效率高。

3) 调用 malloc_trim（0）函数来主动释放空闲内存。

注：

man 手册中说该函数只能释放 Top Chunk 中的内存，但经过测试，也可释放非 Top chunk 的内存。不过，free list bins/fast bins 中依然维护着这些内存地址，当再次需要申请小内存块时，总是从这些 bins 中存储的地址分配内存给用户。

采用方法 2 和方法 3 对 test2.c 进行测试，输出结果如下所示：

mallopt（M_MXFAST,0）

```
----------------before malloc-----------------
size   Resident  Shared  Trs  Lrs  Drs   dt
3123    207       172     2    0    70    0
----------------after malloc------------------
size   Resident  Shared  Trs  Lrs  Drs   dt
16224   13329     217     2    0  13171   0
----------------after free--------------------
size   Resident  Shared  Trs  Lrs  Drs   dt
3156    293       219     2    0   103    0
```

malloc_trim(0)

```
----------------before malloc-----------------
size   Resident  Shared  Trs  Lrs  Drs   dt
3123    207       172     2    0    70    0
----------------after malloc------------------
size   Resident  Shared  Trs  Lrs  Drs   dt
16224   13328     216     2    0  13171   0
----------------after free--------------------
size   Resident  Shared  Trs  Lrs  Drs   dt
16224   13345     218     2    0  13171   0
-------------after malloc_trim(0)--------------
size   Resident  Shared  Trs  Lrs  Drs   dt
3124    261       219     2    0    71    0
```

6 检测方法与工具

Linux 下有多种方法与工具检测内存相关的问题，接下来将分别介绍。

6.1 mtrace

1) 设置环境变量 MALLOC_TRACE 指向一个文件 memory_trace.log，如下所示：

```
[root@localhost testdir]# export MALLOC_TRACE=memory_trace.log
```

2) 准备测试代码 malloc.c，如下所示：

```
[root@localhost testdir]# cat malloc.c
#include <stdio.h>
#include <stdlib.h>
#include <mcheck.h>
int main(void)
{
   #ifdef DEBUGGING
   mtrace( ); //设置 malloc 钩子函数
   #endif

   int *p, *q ;
   p = malloc( sizeof( int ) ) ;
   q = malloc( sizeof( int ) ) ;
   printf("p=%p, q=%p \n",p,q);
   *p = 1 ;
   *q = 2 ;

   free( p ) ;  // q 没有被 free
   return 0;
}
```

3) 编译并运行程序，如下所示：

```
[root@localhost testdir]# gcc malloc.c  -o malloc -DDEBUGGING
[root@localhost testdir]# ./malloc
p=0x1e6c460, q=0x1e6c480
```

4) 查看文件 memory_trace.log 内容，如下所示：

```
[root@localhost testdir]# cat memory_trace.log
= Start
@ ./malloc:[0x400627] + 0x1e6c460 0x4
@ ./malloc:[0x400635] + 0x1e6c480 0x4
@ ./malloc:[0x400673] - 0x1e6c460      //p 被释放，q 没被释放
```

6.2 mallopt

1) mallopt 中的参数选项 M_CHECK_ACTION 可以用来检测内存相关问题，该参数设置不同的值将会表示不同含义，如下所示：（其它值将被忽略）

M_CHECK_ACTION	说明（如果遇到错误情况）
0	忽略错误情况，然后继续执行
1	打印一个详细的错误信息，然后继续执行
2	终止程序
3	打印详细的错误信息、栈调用、内存映射，然后终止程序
5	打印一个简单的错误信息，然后继续执行
7	打印简单的错误信息、栈调用、内存映射，然后终止程序

注：

a) 参数选项 M_CHECK_ACTION 默认值为 3。

b) 参数选项 M_CHECK_ACTION 对应的环境变量为 MALLOC_CHECK_。

c) 使用环境变量 MALLOC_CHECK_=num 的方式不需要修改源代码。

2) 准备测试代码 malloc.c，如下所示：

```
[root@localhost testdir]# cat malloc.c
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *p;
    p = malloc(1000);

    free(p);
    printf("main(): returned from first free() call\n");

    free(p);
    printf("main(): returned from second free() call\n");

    return 0;
}
```

3) 编译并运行程序，如下所示：

```
[root@localhost renyl]# gcc malloc.c  -o malloc
[root@localhost renyl]# MALLOC_CHECK_=0 ./a.out
main(): returned from first free() call
main(): returned from second free() call
[root@localhost renyl]# MALLOC_CHECK_=1 ./a.out
main(): returned from first free() call
*** Error in `./a.out': free(): invalid pointer: 0x0000000000642010 ***
main(): returned from second free() call
[root@localhost renyl]# MALLOC_CHECK_=2 ./a.out
main(): returned from first free() call
Aborted (core dumped)
[root@localhost renyl]# MALLOC_CHECK_=3 ./a.out
main(): returned from first free() call
*** Error in `./a.out': free(): invalid pointer: 0x0000000002258010 ***
======= Backtrace: =========
/lib64/libc.so.6[0x3de907dec6]
./a.out[0x4005ff]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x3de9021af5]
./a.out[0x4004f9]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:02 135805085                  /home/renyl/a.out
00600000-00601000 r--p 00000000 08:02 135805085                  /home/renyl/a.out
00601000-00602000 rw-p 00001000 08:02 135805085                  /home/renyl/a.out
02258000-02279000 rw-p 00000000 00:00 0                          [heap]
3de8800000-3de8821000 r-xp 00000000 08:02 276081912              /usr/lib64/ld-2.17.so
3de8a20000-3de8a21000 r--p 00020000 08:02 276081912              /usr/lib64/ld-2.17.so
3de8a21000-3de8a22000 rw-p 00021000 08:02 276081912              /usr/lib64/ld-2.17.so
3de8a22000-3de8a23000 rw-p 00000000 00:00 0
3de9000000-3de91b6000 r-xp 00000000 08:02 276081913              /usr/lib64/libc-2.17.so
3de91b6000-3de93b6000 ---p 001b6000 08:02 276081913              /usr/lib64/libc-2.17.so
3de93b6000-3de93ba000 r--p 001b6000 08:02 276081913              /usr/lib64/libc-2.17.so
3de93ba000-3de93bc000 rw-p 001ba000 08:02 276081913              /usr/lib64/libc-2.17.so
3de93bc000-3de93c1000 rw-p 00000000 00:00 0
…
7f27e733f000-7f27e7342000 rw-p 00000000 00:00 0
7f27e7368000-7f27e736b000 rw-p 00000000 00:00 0
7fffa026d000-7fffa028e000 rw-p 00000000 00:00 0                  [stack]
7fffa02de000-7fffa02e0000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
Aborted (core dumped)
```

6.3 valgrind

1) valgrind 是一套基于仿真技术的内存调试和分析工具。

2) valgrind 由多个工具组成，每个工具完成不同的功能，如下所示：

工具	说明
Memcheck	用来检测程序中内存相关的各种问题。
Callgrind	类似 gprof，能够建立函数调用关系，计算出各函数的开销。
Cachegrind	Cache 分析器，能够指出程序中 cache 命中率。
Helgrind	检查多线程程序中出现的竞争问题。
Massif	堆栈分析器，能够指出堆块和栈的大小。

3) valgrind 的使用：

a) 语法：valgrind [valgrind-options] [your-program] [your-program-options]

b) [valgrind-options]分为基本选项、malloc 选项、callgrind 选项等，详细信息如下所示：

基本选项（所有工具都可使用）	
参数	说明
--tool=<name>	使用工具的类型（默认为 Memcheck）
--trace-children=<yes|no>	跟踪子进程
--track-fds=<yes|no>	跟踪文件描述符
--db-attach=no|yes	发现错误时调试
…	

Memcheck 选项	
参数	说明
--leak-check=<no|summary|yes|full>	在程序 exit 时查找内存泄漏
--leak-resolution=<low|med|high>	设置如何报告内存泄露
--alignment=<number>	设置分配的最小对齐大小
…	

4) 准备测试代码 malloc.c，如下所示：

```
[root@localhost testdir]# cat malloc.c
#include <stdlib.h>

int main(int argc, char *argv[])
{
        int *p = malloc(10 * sizeof(int));
        p[10] = 1;          //p 越界

        return 0;
}                           //内存泄露，p 没被释放
```

5) 编译并运行程序，如下所示：

```
[root@localhost renyl]# gcc malloc.c  -o malloc
 [root@localhost renyl]# valgrind --tool=memcheck --leak-check=full ./malloc
==20946== Memcheck, a memory error detector
==20946== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==20946== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==20946== Command: ./malloc
==20946==
==20946== Invalid write of size 4
==20946==    at 0x400555: main (in /home/renyl/malloc)
==20946==  Address 0x4c3b068 is 0 bytes after a block of size 40 alloc d
==20946==    at 0x4A0645D: malloc(in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==20946==    by 0x400548: main (in /home/renyl/malloc)
==20946==
==20946==
==20946== HEAP SUMMARY:
==20946==     in use at exit: 40 bytes in 1 blocks
==20946==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==20946==
==20946== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==20946==    at 0x4A0645D: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==20946==    by 0x400548: main (in /home/renyl/malloc)
==20946==
==20946== LEAK SUMMARY:
==20946==    definitely lost: 40 bytes in 1 blocks
==20946==    indirectly lost: 0 bytes in 0 blocks
==20946==      possibly lost: 0 bytes in 0 blocks
==20946==    still reachable: 0 bytes in 0 blocks
==20946==         suppressed: 0 bytes in 0 blocks
==20946==
==20946== For counts of detected and suppressed errors, rerun with: -v
==20946== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 3 from 3)
```

进程号

第一处错误，指出错误类型

指出错误发生位置

第二处错误，指出内存泄露的大小

错误小结

7 参考文献

1) http://gee.cs.oswego.edu/dl/html/malloc.html
2) https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=1120d4df8487b78a9f1ceb5394968d6ab651986e;hb=439bda3209b768c349b98b8ceecf0fa8d94600e9
3) http://blog.csdn.net/phenics/article/details/777053
4) http://stackoverflow.com/questions/10943907/linux-allocator-does-not-release-small-chunks-of-memory
5) http://www.cnblogs.com/sunyubo/archive/2010/05/05/2282170.html
6) http://www.ibm.com/developerworks/cn/linux/l-memory/
7) http://www.cnblogs.com/sunyubo/archive/2010/05/05/2282170.html