# 深入理解Glibc的内存管理

------------------------------------------------------------------

renyl 2016/3/24

目录

[深入理解Glibc的内存管理 1](#_Toc477392443)

[目录 2](#_Toc477392444)

[1 介绍 4](#_Toc477392445)

[2 内存基础 5](#_Toc477392446)

[2.1 内存地址 5](#_Toc477392447)

[2.2 内存分布 5](#_Toc477392448)

[2.3 延迟分配 7](#_Toc477392449)

[3 Glibc内存管理 8](#_Toc477392450)

[3.1 背景知识 8](#_Toc477392451)

[3.2 内存分配器（Allocator） 8](#_Toc477392452)

[3.3 Chunk结构 8](#_Toc477392453)

[3.4 空闲chunk容器 12](#_Toc477392454)

[3.4.1 bins 12](#_Toc477392455)

[3.4.2 fast bins 13](#_Toc477392456)

[3.4.3 unsorted bins 13](#_Toc477392457)

[3.5 分配区 14](#_Toc477392458)

[3.6 malloc过程 15](#_Toc477392459)

[3.6.1 源代码 15](#_Toc477392460)

[3.6.2 流程图 15](#_Toc477392461)

[3.6.3 说明 17](#_Toc477392462)

[3.7 free过程 19](#_Toc477392463)

[3.7.1 源代码 19](#_Toc477392464)

[3.7.2 流程图 19](#_Toc477392465)

[3.7.3 说明 21](#_Toc477392466)

[4 内存分配的相关函数 22](#_Toc477392467)

[4.1 库函数 22](#_Toc477392468)

[4.2 系统调用 22](#_Toc477392469)

[4.3 调优选项 23](#_Toc477392470)

[4.3.1 M\_MMAP\_MAX 23](#_Toc477392471)

[4.3.2 M\_MMAP\_THRESHOLD 24](#_Toc477392472)

[4.3.3 M\_MXFAST 26](#_Toc477392473)

[4.3.4 M\_TOP\_PAD 28](#_Toc477392474)

[4.3.5 M\_TRIM\_THRESHOLD 30](#_Toc477392475)

[5 实例分析 32](#_Toc477392476)

[5.1 指针数组 32](#_Toc477392477)

[5.1.1 源代码 32](#_Toc477392478)

[5.1.2 代码分析 32](#_Toc477392479)

[5.2 内存泄露1 33](#_Toc477392480)

[5.2.1 源代码 33](#_Toc477392481)

[5.2.2 测定结果 34](#_Toc477392482)

[5.2.3 原因分析 34](#_Toc477392483)

[5.2.4 解决方法 35](#_Toc477392484)

[5.3 内存泄露2 35](#_Toc477392485)

[5.3.1 源代码 35](#_Toc477392486)

[5.3.2 测定结果 36](#_Toc477392487)

[5.3.3 原因分析 37](#_Toc477392488)

[5.3.4 解决方法 42](#_Toc477392489)

[6 检测方法与工具 43](#_Toc477392490)

[6.1 mtrace 43](#_Toc477392491)

[6.2 mallopt 44](#_Toc477392492)

[6.3 valgrind 46](#_Toc477392493)

[7 参考文献 48](#_Toc477392494)

# 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：

1. 代码段：该部分空间只能读不能写，存储程序的代码。
2. 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。
3. 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。
4. 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。
5. 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。
6. 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）。
7. 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。

1. 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](http://ltech.fnst.cn.fujitsu.com/source/line/RHEL7.0RC/HTML/files.html)/[Documentation](http://ltech.fnst.cn.fujitsu.com/source/line/RHEL7.0RC/HTML/files/43343.html)/[x86](http://ltech.fnst.cn.fujitsu.com/source/line/RHEL7.0RC/HTML/files/43572.html)/[x86\_64](http://ltech.fnst.cn.fujitsu.com/source/line/RHEL7.0RC/HTML/files/43574.html)/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。

1. 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;  };  … |

1. 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：

1. chunk 指针指向一个 chunk 的开始
2. mem 指针才是真正返回给用户的内存指针
3. 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。
4. 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。
5. 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。
6. 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。
7. 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。
8. 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)。
9. 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：

1. 当chunk空闲时，其M状态不存在，只有AP状态。
2. 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：

1. 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：

1. 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。
2. 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

1. Top chunk：
2. 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。
3. 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”。
4. 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。

1. chunk中的空间复用：
2. 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。
3. 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。
4. 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：

1. 数组中编号从0到1的bin称为unsorted bin（后面将会详细介绍）。
2. 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。
3. 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 |

1. 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。

1. 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。
2. 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：

1. 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。
2. 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。
3. 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。
4. 分配区的创建过程：
5. 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。
6. 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。
7. 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。
8. 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 [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=__libc_malloc&type=reference)[size\_t](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=size_t&type=definitions) [bytes](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=bytes&type=symbol))  {  [mstate](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/371.html#L12) [ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol); //分配区的结构体描述  **void** \*victim[; //](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=victim&type=symbol)返回给user的实际虚拟内存地址  **void** \*(\*hook[) (](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=hook&type=symbol)[size\_t](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=size_t&type=definitions), **const** **void** \*)  = atomic\_forced\_read [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/324.html#L541)[\_\_malloc\_hook](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=__malloc_hook&type=symbol)); //malloc的hook函数  **if** (\_\_builtin\_expect [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=__builtin_expect&type=definitions)[hook](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=hook&type=symbol) != [NULL](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=NULL&type=definitions), 0))  **return** (\*hook[)(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=hook&type=symbol)[bytes](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=bytes&type=symbol), [RETURN\_ADDRESS](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/234.html#L189) (0));    arena\_lookup [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/380.html#L96)[ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol)); //查看当前线程存储中是否有分配区的指针  arena\_lock [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/380.html#L101)[ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol), [bytes](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=bytes&type=symbol)); //获取分配区的锁  **if** (!ar\_ptr[)](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol)  **return** 0;  victim [=](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=victim&type=symbol) [\_int\_malloc](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L3287) ([ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol), [bytes](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=bytes&type=symbol)); //核心函数，用来从分配区中获取虚拟内存  **if** (!victim[) //](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=victim&type=symbol)分配失败，尝试到其它分配区获取虚拟内存  {  LIBC\_PROBE [(memory\_malloc\_retry, 1,](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=LIBC_PROBE&type=definitions) [bytes](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=bytes&type=symbol));  ar\_ptr [=](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol) [arena\_get\_retry](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/380.html#L885) ([ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol), [bytes](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=bytes&type=symbol));  **if** (\_\_builtin\_expect [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=__builtin_expect&type=definitions)[ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol) != [NULL](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=NULL&type=definitions), 1))  {  victim [=](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=victim&type=symbol) [\_int\_malloc](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L3287) ([ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol), [bytes](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=bytes&type=symbol));  (**void**) mutex\_unlock [(&](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mutex_unlock&type=definitions)[ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol)->[mutex](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mutex&type=symbol));  }  }  **else**  (**void**) mutex\_unlock [(&](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mutex_unlock&type=definitions)[ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol)->[mutex](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mutex&type=symbol)); //释放分配区锁  assert [(!](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=assert&type=definitions)[victim](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=victim&type=symbol) || [chunk\_is\_mmapped](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L1277) ([mem2chunk](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L1216) ([victim](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=victim&type=symbol))) ||  ar\_ptr [==](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol) [arena\_for\_chunk](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/380.html#L112) ([mem2chunk](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L1216) ([victim](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=victim&type=symbol))));  **return** victim[;](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=victim&type=symbol)  } |

### 3.6.2 流程图

ptmalloc响应malloc函数申请内存的过程如下所示：



### 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：

1. 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）
2. 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：

1. 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)
2. 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](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=__libc_free&type=symbol) (**void** \*[mem](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mem&type=symbol))  {  mstate[ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol); //分配区的结构体描述  mchunkptr[p](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29); /\* chunk corresponding to mem \*/  **void** (\*hook[) (](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=hook&type=symbol)**void** \*, **const** **void** \*)  = atomic\_forced\_read [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/324.html#L541)[\_\_free\_hook](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=__free_hook&type=symbol)); //free的hook函数  **if** (\_\_builtin\_expect [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=__builtin_expect&type=definitions)[hook](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=hook&type=symbol) != [NULL](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=NULL&type=definitions), 0))  {  (\*hook[)(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=hook&type=symbol)[mem](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mem&type=symbol), [RETURN\_ADDRESS](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/234.html#L189) (0));  **return**;  }  **if** (mem [== 0)](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mem&type=symbol) /\* free(0) has no effect \*/  **return**;  p [=](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29) [mem2chunk](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L1216) ([mem](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mem&type=symbol)); //获取mem所指向的chunk指针  **if** (chunk\_is\_mmapped [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L1277)[p](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29))) /\* 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[\_](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mp_&type=symbol).[no\_dyn\_threshold](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=no_dyn_threshold&type=symbol)  && p[->](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29)[size](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=size&type=definitions) > [mp\_](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mp_&type=symbol).[mmap\_threshold](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mmap_threshold&type=symbol)  && p[->](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29)[size](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=size&type=definitions) <= [DEFAULT\_MMAP\_THRESHOLD\_MAX](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=DEFAULT_MMAP_THRESHOLD_MAX&type=symbol)){  mp\_[.](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mp_&type=symbol)[mmap\_threshold](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mmap_threshold&type=symbol) = [chunksize](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L1300) ([p](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29));  mp\_[.](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mp_&type=symbol)[trim\_threshold](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=trim_threshold&type=symbol) = 2 \* [mp\_](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mp_&type=symbol).[mmap\_threshold](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mmap_threshold&type=symbol);  LIBC\_PROBE [(memory\_mallopt\_free\_dyn\_thresholds, 2,](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=LIBC_PROBE&type=definitions)  mp\_[.](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mp_&type=symbol)[mmap\_threshold](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mmap_threshold&type=symbol), [mp\_](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=mp_&type=symbol).[trim\_threshold](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=trim_threshold&type=symbol));  }  munmap\_chunk [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L2797)[p](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29)); //释放内存给系统  **return**;  }  ar\_ptr [=](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol) [arena\_for\_chunk](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/380.html#L112) ([p](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29));  \_int\_free [(](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/399.html#L3799)[ar\_ptr](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/cgi-bin/global.cgi?pattern=ar_ptr&type=symbol), [p](http://10.167.225.115/sourceweb/opensource/sourceware.org:git:glibc.git/HTML/S/12646.html#L29), 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：



### 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 | test2.c |
| #include <stdlib.h>  #include <unistd.h>  #include <string.h>  int main(void)  {  char \*p = malloc(1024\*1024); //1MB  memset(p,1,1024\*1024);  free(p);  return 0;  } | #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #include <malloc.h>  int main(void)  {  mallopt(M\_MMAP\_MAX,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 | strace ./test2 |
| …  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) = ? | …  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：
2. 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。
3. free的时候会调用munmap系统调用直接把内存归还给操作系统。
4. test2：
5. 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。
6. 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。

1. 使用mmap分配内存的优点：
2. mmap分配内存的大小不受数据段大小（RLIMIT\_DATA）的限制。
3. 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。
4. 使用mmap分配内存的缺点：
5. 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。
6. 由于mmap分配内存需要按页对齐，会导致内存浪费。
7. 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 | test2.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;  } | #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：
2. 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。
3. 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。
4. test2：
5. 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。
6. 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。
7. 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 ==========  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): 135168  # 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): 960  Total free space (fordblks): 134208  Topmost releasable block (keepcost): 134208  ========== After freeing blocks ===========  Total non-mmapped bytes (arena): 135168  # of free chunks (ordblks): 1  # of free fastbin blocks (smblks): 20  # of mapped regions (hblks): 0  Bytes in mapped regions (hblkhd): 0  Max. total allocated space (usmblks): 0  Free bytes held in fastbins (fsmblks): 960  Total allocated space (uordblks): 0  Total free space (fordblks): 135168  Topmost releasable block (keepcost): 134208 | ========= 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): 135168  # 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): 960  Total free space (fordblks): 134208  Topmost releasable block (keepcost): 134208  ========== After freeing blocks ============  Total non-mmapped bytes (arena): 135168  # 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): 135168  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。

1. 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。
2. 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：

|  |  |
| --- | --- |
| test1.c | test2.c |
| #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;  } | #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;  } |

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 | test2.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));  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;  } | #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);  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--------  current brk =d46000  ---------after malloc--------  current brk =d80000  current brk =d80000  current brk =db2000  current brk =db2000  current brk =de4000  ---------after free----------  current brk =d67000 | --------before malloc--------  current brk =1da1000  ---------after malloc--------  current brk =1ddb000  current brk =1ddb000  current brk =1e0d000  current brk =1e0d000  current brk =1e3f000  ---------after free----------  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 | 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;  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;  } | #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）
2. 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。
3. 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。
4. test2.c：（一次分配内存(保证内存的连续性)）
   1. 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。
   2. 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 | 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\tTrs\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\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\tdt\n");  system(command);  return 0;  } | #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\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\tTrs\tLrs\tDrs\tdt\n");  system(command);  for (loop =0 ; loop < ARRAY\_SIZE - 1; loop++)  free(array[loop]);  printf("--------after free----------\n");  printf("size\tResident\tShared\tTrs\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 | ./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  1074 164 102 1 0 80 0 | ----------------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

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，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：

1. 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。
2. 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。
3. 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  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\tdt\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\tdt\n");  system(command);  //display\_mallinfo();  return 0;  } | #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();  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----------------  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 | ----------------before malloc----------------  size Resident Shared Trs Lrs Drs dt  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 | ./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): 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  # 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 | ============ 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): 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：

1. 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)。
2. 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。
3. Top chunk（keepcost）的大小仅仅为61696B。

根据上面几个发现，可以推断出：

1. 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。
2. 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。

1. 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 | test2.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]);    return 0;  } | #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]);  }  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) {  保存着第2块内存的malloc\_chunk结构体  (gdb) p /x ptrs[0]  $1 = 0x602010  (gdb) x /64x 0x602010  0x602010: 0x01010101 0x01010101 0x01010101 0x01010101  第1块内存的返回地址  保存着第3块内存的malloc\_chunk结构体  0x602020: 0x00000000 0x00000000 0x00000021 0x00000000  0x602030: 0x01010101 0x01010101 0x01010101 0x01010101  0x602040: 0x00000000 0x00000000 0x00000021 0x00000000  0x602050: 0x01010101 0x01010101 0x01010101 0x01010101  第2块内存的返回地址  0x602060: 0x00000000 0x00000000 0x00000021 0x00000000  heap的空闲大小，即Top chunk大小  0x602070: 0x01010101 0x01010101 0x01010101 0x01010101  0x602080: 0x00000000 0x00000000 0x00000021 0x00000000  0x602090: 0x01010101 0x01010101 0x01010101 0x01010101  0x6020a0: 0x00000000 0x00000000 0x00020f61 0x00000000  第5块内存的返回地址  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.  这就是放入fast bins中的chunk，这段内存是被list分配的，用来存储list链表中的内容。  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  第1块内存的返回地址  0x603020: 0x00000000 0x00000000 0x00000021 0x00000000  0x603030: 0x00603070 0x00000000 0xffffe280 0x00007fff  0x603040: 0x00603010 0x00000000 0x00000021 0x00000000  0x603050: 0x01010101 0x01010101 0x01010101 0x01010101  第2块内存的返回地址  保存着第2块内存的malloc\_chunk结构体  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  第3块内存的返回地址  保存着第3块内存的malloc\_chunk结构体  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）

1. 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。
2. 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。
3. 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。
4. 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） | 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 13329 217 2 0 13171 0  ----------------after free--------------------  size Resident Shared Trs Lrs Drs dt  3156 293 219 2 0 103 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 |

1. 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;  } |

1. 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 |

1. 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：

1. 参数选项M\_CHECK\_ACTION默认值为3。
2. 参数选项M\_CHECK\_ACTION对应的环境变量为MALLOC\_CHECK\_。
3. 使用环境变量MALLOC\_CHECK\_=num的方式不需要修改源代码。
4. 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;  } |

1. 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。 |

1. valgrind的使用：
2. 语法：valgrind [valgrind-options] [your-program] [your-program-options]
3. [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 |
| … |  |

1. 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 |

1. 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  进程号  ==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  第一处错误，指出错误类型  ==20946== Command: ./malloc  ==20946==  ==20946== Invalid write of size 4  指出错误发生位置  ==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==  第二处错误，指出内存泄露的大小  ==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  错误小结  ==20946==  ==20946== For counts of detected and suppressed errors, rerun with: -v  ==20946== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 3 from 3) |

# 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>