Unknown Title

---

https://blog.csdn.net/z_ryan/article/details/79950737

https://blog.csdn.net/phenics/article/details/777053

实现细节请看：https://blog.csdn.net/songchuwang1868/article/details/90144080

目录

一、前言

二、内存布局

三、brk（sbrk）和mmap函数

1、brk() 和 sbrk()

2、mmap()

四、Allocator

五、Malloc实现原理

1、chunk 内存块的基本组织单元

2、chunk的结构

a、使用中的chunk

b、空闲的chunk

c、chunk中的空间复用

2、空闲链表bins

a、fast bins

b、unsorted bin

c、small bins

d、large bins

3、三种特殊的chunk

a、top chunk

b、mmaped chunk

c、last remainder chunk

六、sbrk与mmap

七、主分配区和非主分配区

八、内存分配malloc流程

九、内存回收流程

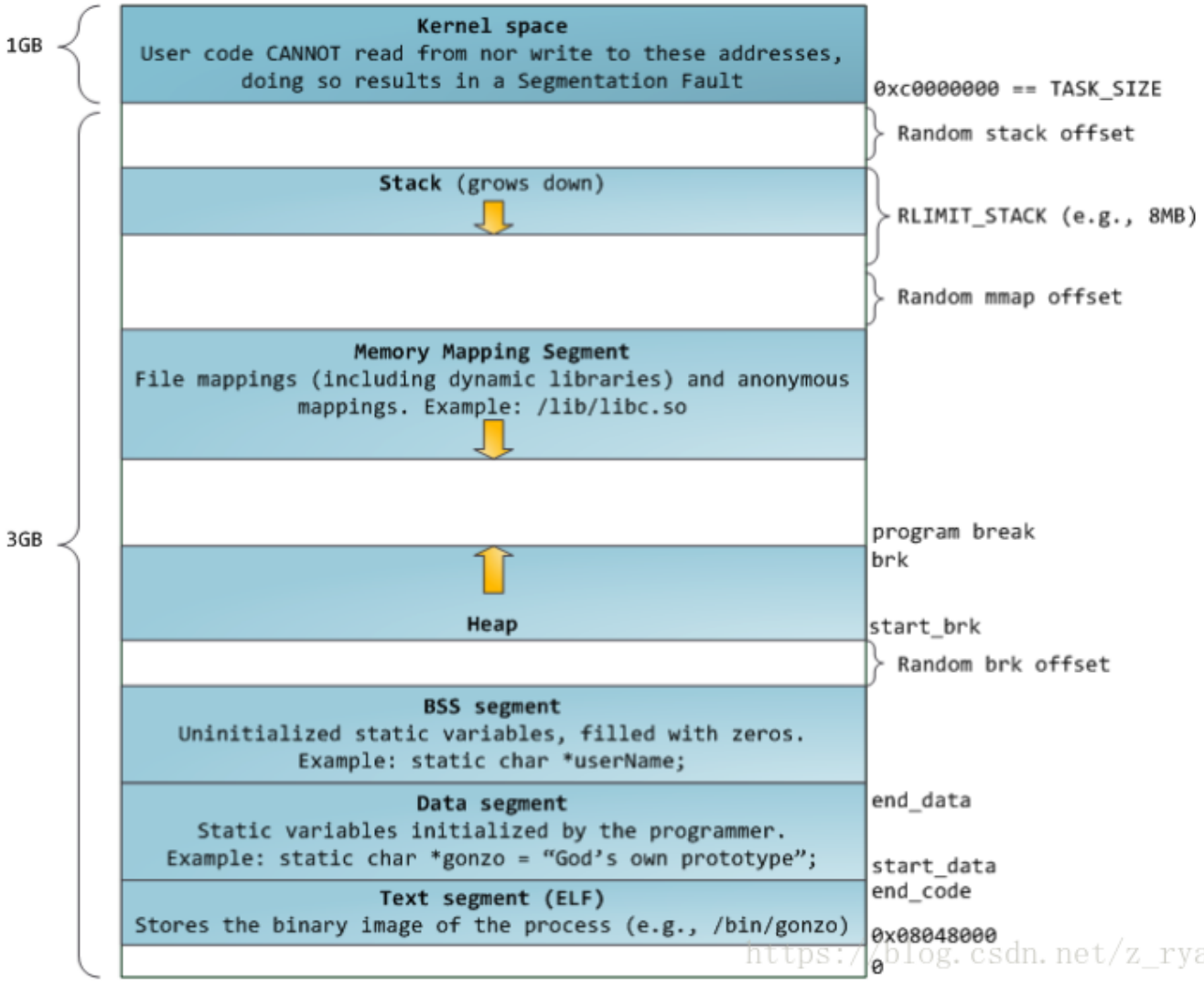
十、使用注意事项

一、前言

C语言提供了动态内存管理功能, 在C语言中, 程序员可以使用 malloc() 和 free() 函数显式的分配和释放内存. 关于 malloc() 和free() 函数, C语言标准只是规定了它们需要实现的功能, 而没有对实现方式有什么限制, 这多少让那些追根究底的人感到有些许迷茫, 比如对于 free() 函数, 它规定一旦一个内存区域被释放掉, 那么就不应该再对其进行任何引用, 任何对释放区域的引用都会导致不可预知的后果 (unperdictable effects). 那么, 到底是什么样的不可预知后果呢? 这完全取决于内存分配器(memory allocator)使用的算法. 这篇文章试图对 Linux glibc 提供的 allocator 的工作方式进行一些描述, 并希望可以解答上述类似的问题. 虽然这里的描述局限于特定的平台, 但一般的事实是, 相同功能的软件基本上都会采用相似的技术. 这里所描述的原理也许在别的环境下会仍然有效. 另外还要强调的一点是, 本文只是侧重于一般原理的描述, 而不会过分纠缠于细节, 如果需要特定的细节知识, 请参考特定 allocator 的源代码. 最后, 本文描述的硬件平台是 Intel 80x86, 其中涉及的有些原理和数据可能是平台相关的.

二、内存布局

介绍ptmalloc之前，我们先了解一下内存布局，以x86的32位系统为例：



从上图可以看到，栈至顶向下扩展，堆至底向上扩展， mmap 映射区域至顶向下扩展。 mmap 映射区域和堆相对扩展，直至耗尽虚拟地址空间中的剩余区域，这种结构便于 C 运行时库使用 mmap 映射区域和堆进行内存分配。

三、brk（sbrk）和mmap函数

首先，linux系统向用户提供申请的内存有brk(sbrk)和mmap函数。下面我们先来了解一下这几个函数。

1、brk() 和 sbrk()

```
1. #include <unistd.h>

2. int brk( const void *addr )

3. void* sbrk ( intptr_t incr );
```

两者的作用是扩展heap的上界brk
Brk（）的参数设置为新的brk上界地址，成功返回1，失败返回0；

Sbrk（）的参数为申请内存的大小，返回heap新的上界brk的地址

2、mmap()

```
1. #include <sys/mman.h>

2. void *mmap(void *addr, size\_t length, int prot, int flags, int fd,
   off\_t offset);

3. int munmap(void *addr, size_t length);
```

Mmap的第一种用法是映射此盘文件到内存中；第二种用法是匿名映射，不映射磁盘文件，而向映射区申请一块内存。
Malloc使用的是mmap的第二种用法（匿名映射）。
Munmap函数用于释放内存。

四、Allocator

GNU Libc 的内存分配器(allocator) — ptmalloc 起源于 Doug Lea 的 malloc (请参看[1]). ptmalloc 实现了 malloc() , free() 以及一组其它的函数. 以提供动态内存管理的支持. 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, 为了保持高效的分配, allocator 一般都会预先分配一块大于用户请求的内存, 并通过某种算法管理这块内存. 来满足用户的内存分配要求, 用户 free 掉的内存也并不是立即就返回给操作系统, 相反, 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. 也就是说, allocator 不但要管理已分配的内存块, 还需要管理空闲的内存块, 当响应用户分配要求时, allocator 会首先在空闲空间中寻找一块合适的内存给用户, 在空闲空间中找不到的情况下才分配一块新的内存. 为实现一个高效的 allocator, 需要考虑很多的因素. 比如, allocator 本身管理内存块所占用的内存空间必须很小, 分配算法必须要足够的快. Jonathan Bartlett 给出了一个简单的 allocator 实现 [2], 事先看看或许会对理解本文有所帮助. 另外插一句, Jonathan Bartlett 的书 “Programming from Ground Up” 对想要了解 linux 汇编和工作方式的入门者是个不错的选择.

五、Malloc实现原理

因为brk、sbrk、mmap都属于系统调用，若每次申请内存，都调用这三个，那么每次都会产生系统调用，影响性能；其次，这样申请的内存容易产生碎片，因为堆是从低地址到高地址，如果高地址的内存没有被释放，低地址的内存就不能被回收。

所以malloc采用的是内存池的管理方式（ptmalloc），Ptmalloc 采用边界标记法将内存划分成很多块，从而对内存的分配与回收进行管理。为了内存分配函数malloc的高效性，ptmalloc会预先向操作系统申请一块内存供用户使用，当我们申请和释放内存的时候，ptmalloc会将这些内存管理起来，并通过一些策略来判断是否将其回收给操作系统。这样做的最大好处就是，使用户申请和释放内存的时候更加高效，避免产生过多的内存碎片。

1、chunk 内存块的基本组织单元

在 ptmalloc 的实现源码中定义结构体 malloc_chunk 来描述这些块。malloc_chunk 定义如下：

```
 1. struct malloc_chunk {
 2.   INTERNAL_SIZE_T      prev_size;    /* Size of previous chunk (if
    free).  */
 3.   INTERNAL_SIZE_T      size;         /* Size in bytes, including
    overhead. */
 4.
 5.   struct malloc_chunk* fd;           /* double links -- used only if
    free. */
 6.   struct malloc_chunk* bk;
 7.
 8.   /* Only used for large blocks: pointer to next larger size.  */
 9.   struct malloc_chunk* fd_nextsize;      /* double links -- used only
    if free. */
10.   struct malloc_chunk* bk_nextsize;
11. };
```

chunk 的定义相当简单明了，对各个域做一下简单介绍：

prev_size: 如果前一个 chunk 是空闲的，该域表示前一个 chunk 的大小，如果前一个 chunk 不空闲，该域无意义。（这里的描述有点模糊，一段连续的内存被分成多个chunk，prev_size记录的就是相邻的前一个chunk的size，知道当前chunk的地址，减去prev_size便是前一个chunk的地址。prev_size主要用于相邻空闲chunk的合并）

size ：当前 chunk 的大小，并且记录了当前 chunk 和前一个 chunk 的一些属性，包括前一个 chunk 是否在使用中，当前 chunk 是否是通过 mmap 获得的内存，当前 chunk 是否属于非主分配区。

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

fd_nextsize 和 bk_nextsize: 当前的 chunk 存在于 large bins 中时， large bins 中的空闲 chunk 是按照大小排序的，但同一个大小的 chunk 可能有多个，增加了这两个字段可以加快遍历空闲 chunk ，并查找满足需要的空闲 chunk ， fd_nextsize 指向下一个比当前 chunk 大小大的第一个空闲 chunk ， bk_nextszie 指向前一个比当前 chunk 大小小的第一个空闲 chunk 。（同一大小的chunk可能有多块，在总体大小有序的情况下，要想找到下一个比自己大或小的chunk，需要遍历所有相同的chunk，所以才有

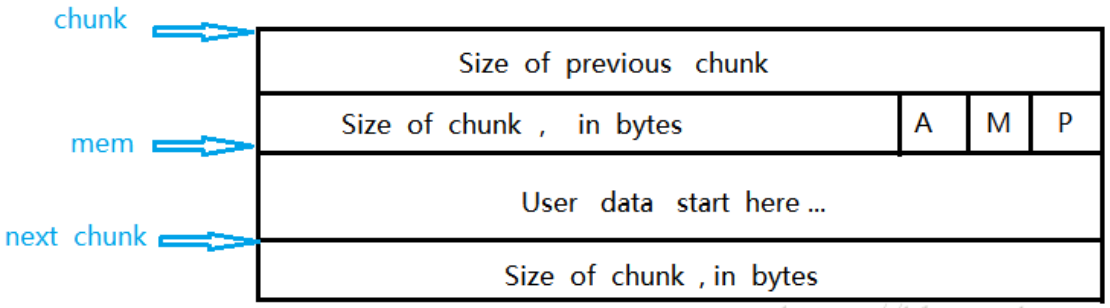
fd_nextsize和bk_nextsize这种设计） 如果该 chunk 块被分配给应用程序使用，那么这两个指针也就没有用（该chunk 块已经从 size 链中拆出）了，所以也当作应用程序的使用空间，而不至于浪费。

（下面马上可以看到，当chunk为空时才有fd、bk、fd_nextsize、bd_nextsize四个指针，当chunk不为空，这四个指针的空间是直接交给用户使用的）

2、chunk的结构

chunk的结构可以分为使用中的chunk和空闲的chunk。使用中的chunk和空闲的chunk数据结构基本相同，但是会有一些设计上的小技巧，巧妙的节省了内存。

a、使用中的chunk



说明：

1、 chunk指针指向chunk开始的地址；mem指针指向用户内存块开始的地址。

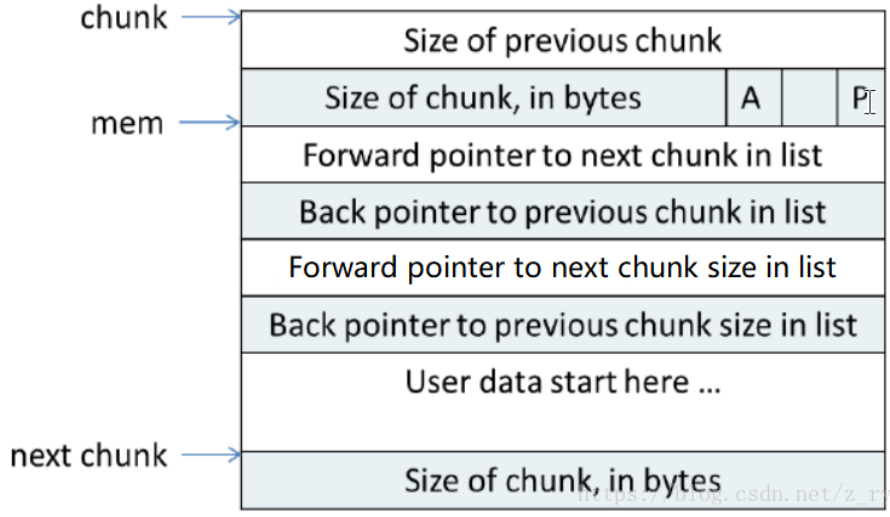
2、 p=0时，表示前一个chunk为空闲，prev_size才有效

3、p=1时，表示前一个chunk正在使用，prev_size无效 p主要用于内存块的合并操作；ptmalloc 分配的第一个块总是将p设为1, 以防止程序引用到不存在的区域

4、M=1 为mmap映射区域分配；M=0为heap区域分配

5、 A=0 为主分配区分配；A=1 为非主分配区分配。

b、空闲的chunk



说明：

1、当chunk空闲时，其M状态是不存在的，只有AP状态（因为M表示是由brk还是mmap分配的内存，

而mmap分配的内存free时直接ummap，不会放到空闲链表中。换言之空闲链表中的都死brk分配的，所以不用额外记录）

２、原本是用户数据区的地方存储了四个指针，

指针fd指向后一个空闲的chunk,而bk指向前一个空闲的chunk，malloc通过这两个指针将大小相近的chunk连成一个双向链表。

在large bin中的空闲chunk，还有两个指针，fd_nextsize和bk_nextsize，用于加快在large bin中查找最近匹配的空闲chunk。不同的chunk链表又是通过bins或者fastbins来组织的。

c、chunk中的空间复用

为了使得 chunk 所占用的空间最小, ptmalloc 使用了空间复用, 一个 chunk 或者正在被使用, 或者已经被 free 掉, 所以 chunk 的中的一些域可以在使用状态和空闲状态表示不同的意义, 来达到空间复用的效果. 空闲时, 一个 chunk 中至少要4个 size_t 大小的空间, 用来存储 prev_size, size , fd 和 bk ， 也就是16 bytes （？？为什么不是6个size_t呢？不是还有fd_nextsize和bk_nextsize吗？——并不是所有bin中都需要这两个指针，比如在fast_bin中，每隔8个Byte就有一个链表，每个链表中的所有chunk的size都是一样的，显然不用这两个指针） chuck 的大小要 align 到8 bytes. 当一个 chunk 处于使用状态时, 它的下一个 chunk 的 prev_size 域肯定是无效的. 所以实际上, 这个空间也可以被当前 chunk 使用. 这听起来有点不可思议, 但确实是合理空间复用的例子. 故而实际上, 一个使用中的 chunk 的大小的计算公式应该是:

in_use_size = (用户请求大小 + 8 - 4) align to 8 bytes 这里加8是因为需要存储 prev_size 和 size, 但又因为向下一个 chunk “借”了4个bytes, 所以要减去4. 最后, 因为空闲的 chunk 和使用中的 chunk 使用的是同一块空间. 所以肯定要取其中最大者作为实际的分配空间. 即最终的分配空间 chunk_size = max(in_use_size, 16). 这就是当用户请求内存分配时, ptmalloc 实际需要分配的内存大小, 在后面的介绍中. 如果不是特别指明的地方, 指的都是这个经过转换的实际需要分配的内存大小, 而不是用户请求的内存分配大小.

2、空闲链表bins

当用户使用free函数释放掉的内存，ptmalloc并不会马上交还给操作系统，而是被ptmalloc本身的空闲链表bins管理起来了，这样当下次进程需要malloc一块内存的时候，ptmalloc就会从空闲的bins上寻找一块合适大小的内存块分配给用户使用。这样的好处可以避免频繁的系统调用，降低内存分配的开销。

malloc将相似大小的chunk用双向链表链接起来，这样一个链表被称为一个bin。ptmalloc一共维护了128bin。每个bins都维护了大小相近的双向链表的chunk。基于chunk的大小，有下列几种可用bins：

1、Fast bin
2、Unsorted bin
3、Small bin
4、Large bin

保存这些bin的数据结构为：

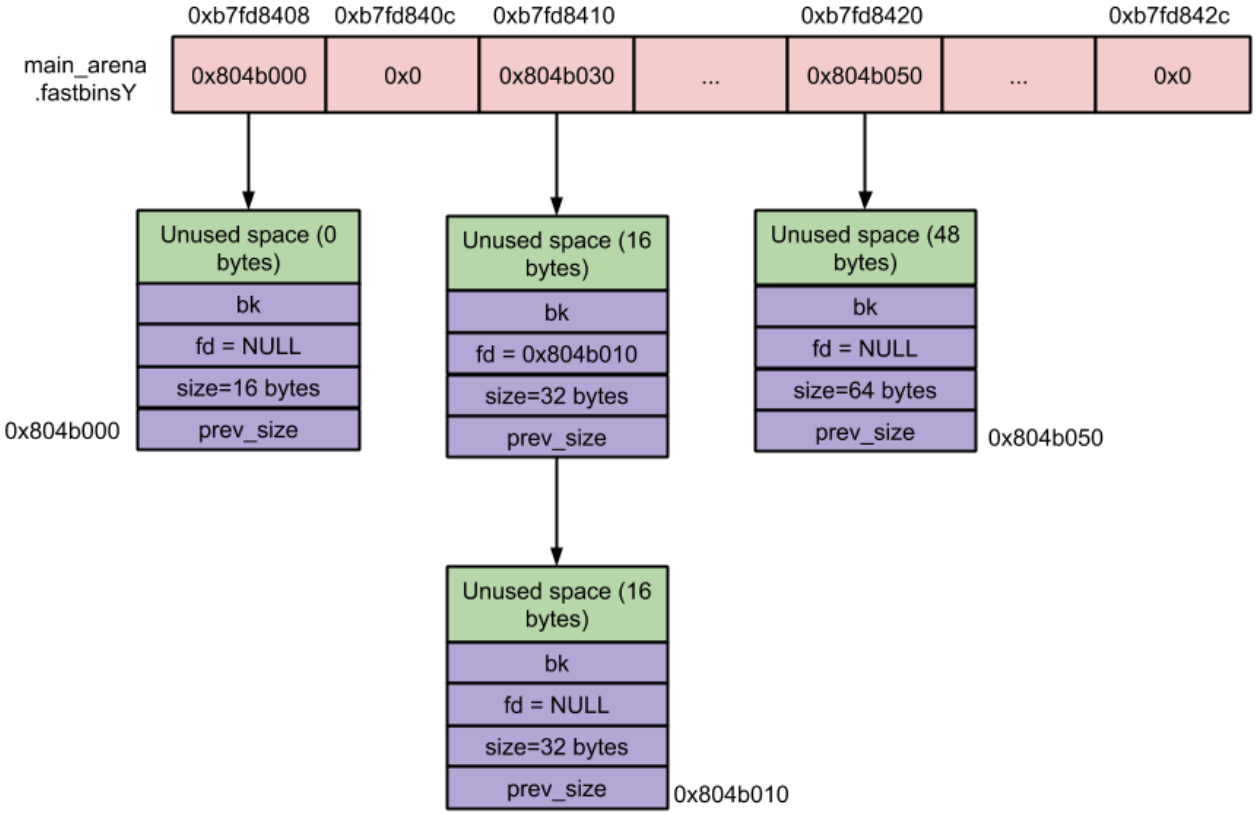
fastbinsY：这个数组用以保存fast bins。

bins：这个数组用以保存unsorted、small以及large bins，共计可容纳126个：

当用户调用malloc的时候，能很快找到用户需要分配的内存大小是否在维护的bin上，如果在某一个bin上，就可以通过双向链表去查找合适的chunk内存块给用户使用。

a、fast bins

程序在运行时会经常需要申请和释放一些较小的内存空间。当分配器合并了相邻的几个小的 chunk 之后,也许马上就会有另一个小块内存的请求,这样分配器又需要从大的空闲内存中切分出一块,这样无疑是比较低效的,故而,malloc 中在分配过程中引入了 fast bins



Fast Bin Snapshot

fast bins是bins的高速缓冲区，大约有10个定长队列。每个fast bin都记录着一条free chunk的单链表（称为binlist ，采用单链表是出于fast bin中链表中的chunk不会被摘除的特点），增删chunk都发生在链表的前端。 fast bins 记录着大小以8字节递增的bin链表（？？从上面的图看好像是4字节递增啊——4字节递增是因为指针占4字节，下面挂的块的确是8字节递增）。

当用户释放一块不大于max_fast（默认值64B）的chunk的时候，会默认会被放到fast bins上。当需要给用户分配的 chunk 小于或等于 max_fast 时,malloc 首先会到fast bins上寻找是否有合适的chunk。（一定大小内的chunk无论是分配还是释放，都会先在fast bin中过一遍）

除非特定情况，两个毗连的空闲chunk并不会被合并成一个空闲chunk。不合并可能会导致碎片化问题，但是却可以大大加速释放的过程！

分配时，binlist中被检索的第一个个chunk将被摘除并返回给用户。free掉的chunk将被添加在索引到的binlist的前端。

b、unsorted bin

unsorted bin 的队列使用 bins 数组的第一个，是bins的一个缓冲区，加快分配的速度。当用户释放的内存大于max_fast或者fast bins合并后的chunk都会首先进入unsorted bin上。chunk大小 – 无尺寸限制，任何大小chunk都可以添加进这里。这种途径给予 ‘glibc malloc’ 第二次机会以重新使用最近free掉的chunk，这样寻找合适bin的时间开销就被抹掉了，因此内存的分配和释放会更快一些。

用户malloc时，如果在 fast bins 中没有找到合适的 chunk,则malloc 会先在 unsorted bin 中查找合适的空闲 chunk，如果没有合适的bin，ptmalloc会将unsorted bin上的chunk放入bins上，然后到bins上查找合适的空闲chunk。

c、small bins

大小小于512字节的chunk被称为small chunk，而保存small chunks的bin被称为small bin。数组从2开始编号，前64个bin为small bins，small bin每个bin之间相差8个字节，同一个small bin中的chunk具有相同大小。

每个small bin都包括一个空闲区块的双向循环链表（也称binlist）。free掉的chunk添加在链表的前端，而所需chunk则从链表后端摘除。

两个毗连的空闲chunk会被合并成一个空闲chunk。合并消除了碎片化的影响但是减慢了free的速度。

分配时，当samll bin非空后，相应的bin会摘除binlist中最后一个chunk并返回给用户。在free一个chunk的时候，检查其前或其后的chunk是否空闲，若是则合并，也即把它们从所属的链表中摘除并合并成一个新的chunk，新chunk会添加在unsorted bin链表的前端。

d、large bins

大小大于等于512字节的chunk被称为large chunk，而保存large chunks的bin被称为large bin，位于small bins后面。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按大小递减排序，大小相同则按照最近使用时间排列。

两个毗连的空闲chunk会被合并成一个空闲chunk。

分配时，遵循原则“smallest-first , best-fit”,从顶部遍历到底部以找到一个大小最接近用户需求的chunk。一旦找到，相应chunk就会分成两块User chunk（用户请求大小）返回给用户。
Remainder chunk 剩余部分添加到unsorted bin。free时和small bin 类似。

3、三种特殊的chunk

并不是所有chunk都按照上面的方式来组织，有三种列外情况。top chunk，mmaped chunk 和last remainder chunk

a、top chunk

top chunk相当于分配区的顶部空闲内存（可能就是由brk调用控制的brk指针），当bins上都不能满足内存分配要求的时候，就会来top chunk上分配。

当top chunk大小比用户所请求大小还大的时候，top chunk会分为两个部分：User chunk（用户请求大小）和Remainder chunk（剩余大小）。其中Remainder chunk成为新的top chunk。

当top chunk大小小于用户所请求的大小时，top chunk就通过sbrk（main arena）或mmap（thread arena）系统调用来扩容。

b、mmaped chunk

当分配的内存非常大（大于分配阀值，默认128K）的时候，需要被mmap映射，则会放到mmaped chunk上，当释放mmaped chunk上的内存的时候会直接交还给操作系统。 （chunk中的M标志位置1）

c、last remainder chunk

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

六、sbrk与mmap

在堆区中， start_brk 指向 heap 的开始，而 brk 指向 heap 的顶部。可以使用系统调用 brk()和 sbrk()来增 加标识 heap 顶部的 brk 值，从而线性的增加分配给用户的 heap 空间。在使 malloc 之前，brk 的值等于 start_brk，也就是说 heap 大小为 0。

ptmalloc 在开始时，若请求的空间小于 mmap 分配阈值（mmap threshold，默认值为 128KB）时，主分配区会调用 sbrk()增加一块大小为 (128 KB + chunk_size) align 4KB（页面大小对齐） 的空间作为 heap。非主分配区会调用 mmap 映射一块大小为 HEAP_MAX_SIZE（32 位系统上默认为 1MB，64 位系统上默认为 64MB）的空间作为 sub-heap。这就是前面所说的 ptmalloc 所维护的分配空间;

当用户请求内存分配时，首先会在这个区域内找一块合适的 chunk 给用户。当用户释放了 heap 中的 chunk 时，ptmalloc 又会使用 fastbins 和 bins 来组织空闲 chunk。以备用户的下一次分配。

若需要分配的 chunk 大小小于 mmap分配阈值，而 heap 空间又不够，则此时主分配区会通过 sbrk()调用来增加 heap 大小，非主分配区会调用 mmap 映射一块新的 sub-heap，也就是增加 top chunk 的大小，每次 heap 增加的值都会对齐到 4KB。当用户的请求超过 mmap 分配阈值，并且主分配区使用 sbrk()分配失败的时候，或是非主分配区在 top chunk 中不能分配到需要的内存时，ptmalloc 会尝试使用 mmap()直接映射一块内存到进程内存空间。使用 mmap()直接映射的 chunk 在释放时直接解除映射，而不再属于进程的内存空间。任何对该内存的访问都会产生段错误。而在 heap 中或是 sub-heap 中分配的空间则可能会留在进程内存空间内，还可以再次引用（当然是很危险的）。

七、主分配区和非主分配区

内存分配器中，为了解决多线程锁争夺问题，分为主分配区main_area（分配区的本质就是内存池，管理着chunk，一般用英文area表示）和非主分配区no_main_area。 （主分配区和非主分配区的区别）

1. 主分配区和非主分配区形成一个环形链表进行管理。
2. 每一个分配区利用互斥锁使线程对于该分配区的访问互斥。
3. 每个进程只有一个主分配区，也可以允许有多个非主分配区。
4. ptmalloc根据系统对分配区的争用动态增加分配区的大小，分配区的数量一旦增加，则不会减少。
5. 主分配区可以使用brk和mmap来分配，而非主分配区只能使用mmap来映射内存块
6. 申请小内存时会产生很多内存碎片，ptmalloc在整理时也需要对分配区做加锁操作。

当一个线程需要使用malloc分配内存的时候，会先查看该线程的私有变量中是否已经存在一个分配区。若是存在。会尝试对其进行加锁操作。若是加锁成功，就在使用该分配区分配内存，若是失败，就会遍历循环链表中获取一个未加锁的分配区。若是整个链表中都没有未加锁的分配区，则malloc会开辟一个新的分配区，将其加入全局的循环链表并加锁，然后使用该分配区进行内存分配。当释放这块内存时，同样会先获取待释放内存块所在的分配区的锁。若是有其他线程正在使用该分配区，则必须等待其他线程释放该分配区互斥锁之后才能进行释放内存的操作。

需要注意几个点：

- 主分配区通过brk进行分配，非主分配区通过mmap进行分配
- 从分配区虽然是mmap分配，但是和大于128K直接使用mmap分配没有任何联系。大于128K的内存使用mmap分配，使用完之后直接用ummap还给系统
- 每个线程在malloc会先获取一个area，使用area内存池分配自己的内存，这里存在竞争问题
- 为了避免竞争，我们可以使用线程局部存储，thread cache（tcmalloc中的tc正是此意），线程局部存储对area的改进原理如下：

1. 如果需要在一个线程内部的各个函数调用都能访问、但其它线程不能访问的变量（被称为static memory local to a thread 线程局部静态变量），就需要新的机制来实现。这就是TLS。
2. thread cache本质上是在static区为每一个thread开辟一个独有的空间，因为独有，不再有竞争
3. 每次malloc时，先去线程局部存储空间中找area，用thread cache中的area分配存在thread area中的chunk。当不够时才去找堆区的area
4. C++11中提供thread_local方便于线程局部存储

- 可以看出主分配区其实是鸡肋，实际上tcmalloc和jemalloc都不再使用主分配区，直接使用非主分配区

八、内存分配malloc流程

1、获取分配区的锁，防止多线程冲突。（一个进程有一个malloc管理器，而一个进程中的多个线程共享这一个管理器，有竞争，加锁）

2、计算出实际需要分配的内存的chunk实际大小。

3、判断chunk的大小，如果小于max_fast（64 B），则尝试去fast bins上取适合的chunk，如果有则分配结束。否则，下一步；

4、判断chunk大小是否小于512B，如果是，则从small bins上去查找chunk，如果有合适的，则分配结束。否则下一步；

5、ptmalloc首先会遍历fast bins（注：这里是第二次遍历fast bins了，虽然fast bins一般不会合并，但此时会）中的chunk，将相邻的chunk进行合并，并链接到unsorted bin中然后遍历 unsorted bins。（总体而言，第五部遍历unsorted bin，只是在遍历前先合并fast bin，遍历unsorted bin时一边遍历，一边放到small bin和large bin中）

- 如果unsorted bins上只有一个chunk并且大于待分配的chunk，则进行切割，并且剩余的chunk继续扔回unsorted bins；
- 如果unsorted bins上有大小和待分配chunk相等的，则返回，并从unsorted bins删除；
- 如果unsorted bins中的某一chunk大小 属于small bins的范围，则放入small bins的头部；
- 如果unsorted bins中的某一chunk大小 属于large bins的范围，则找到合适的位置放入。若未分配成功，转入下一步；

6、从large bins中查找找到合适的chunk之后，然后进行切割，一部分分配给用户，剩下的放入unsorted bin中。

7、如果搜索fast bins和bins都没有找到合适的chunk，那么就需要操作top chunk来进行分配了 。当top chunk大小比用户所请求大小还大的时候，top chunk会分为两个部分：User chunk（用户请求大小）和

Remainder chunk（剩余大小）。其中Remainder chunk成为新的top chunk。 当top chunk大小小于用户所请求的大小时，top chunk就通过sbrk（main arena）或mmap（thread arena）系统调用来扩容。

8、到了这一步，说明 top chunk 也不能满足分配要求，所以，于是就有了两个选择: 如 果是主分配区，调用 sbrk()，增加 top chunk 大小；如果是非主分配区，调用 mmap 来分配一个新的 sub-heap，增加 top chunk 大小；或者使用 mmap()来直接分配。在 这里，需要依靠 chunk 的大小来决定到底使用哪种方法。判断所需分配的 chunk 大小是否大于等于 mmap 分配阈值，如果是的话，则转下一步，调用 mmap 分配， 否则跳到第 10 步，增加 top chunk 的大小。

9、使用 mmap 系统调用为程序的内存空间映射一块 chunk_size align 4kB 大小的空间。 然后将内存指针返回给用户。

10、判断是否为第一次调用 malloc，若是主分配区，则需要进行一次初始化工作，分配 一块大小为 (chunk_size + 128KB) align 4KB 大小的空间作为初始的 heap。若已经初 始化过了，主分配区则调用 sbrk()增加 heap 空间，分主分配区则在 top chunk 中切 割出一个 chunk，使之满足分配需求，并将内存指针返回给用户。

九、内存回收流程

1. 获取分配区的锁，保证线程安全。
2. 如果free的是空指针，则返回，什么都不做。
3. 判断当前chunk是否是mmap映射区域映射的内存，如果是，则直接munmap()释放这块内存。前面的已使用chunk的数据结构中，我们可以看到有M来标识是否是mmap映射的内存。
4. 判断chunk是否与top chunk相邻，如果相邻，则直接和top chunk合并（和top chunk相邻相当于和分配区中的空闲内存块相邻）。转到步骤8
5. 如果chunk的大小大于max_fast（64b），则放入unsorted bin，并且检查是否有合并，有合并情况并且和top chunk相邻，则转到步骤8；没有合并情况则free。
6. 如果chunk的大小小于 max_fast（64b），则直接放入fast bin，fast bin并没有改变chunk的状态。没有合并情况，则free；有合并情况，转到步骤7
7. 在fast bin，如果当前chunk的下一个chunk也是空闲的，则将这两个chunk合并，放入unsorted bin上面。合并后的大小如果大于64B，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 bin会变为空。合并后的chunk和topchunk相邻，则会合并到topchunk中。转到步骤8
8. 判断top chunk的大小是否大于mmap收缩阈值（默认为128KB），如果是的话，对于主分配区，则会试图归还top chunk中的一部分给操作系统。free结束。

十、使用注意事项

为了避免Glibc内存暴增，需要注意：

1. 后分配的内存先释放，因为ptmalloc收缩内存是从top chunk开始，如果与top chunk相邻的chunk不能释放，top chunk以下的chunk都无法释放。

2. Ptmalloc不适合用于管理长生命周期的内存，特别是持续不定期分配和释放长生命周期的内存，这将导致ptmalloc内存暴增。

3. 不要关闭 ptmalloc 的 mmap 分配阈值动态调整机制，因为这种机制保证了短生命周期的 内存分配

尽量从 ptmalloc 缓存的内存 chunk 中分配，更高效，浪费更少的内存。

4. 多线程分阶段执行的程序不适合用ptmalloc，这种程序的内存更适合用内存池管理 （因为同一个进程下的多线程要加锁后才能使用malloc分配器）

5. 尽量减少程序的线程数量和避免频繁分配/释放内存。频繁分配，会导致锁的竞争，最终导致非主分配区增加，内存碎片增高，并且性能降低。

6. 防止内存泄露，ptmalloc对内存泄露是相当敏感的，根据它的内存收缩机制，如果与top chunk相邻的那个chunk没有回收，将导致top chunk一下很多的空闲内存都无法返回给操作系统。

7. 防止程序分配过多的内存，或是由于glibc内存暴增，导致系统内存耗尽，程序因为OOM被系统杀掉。预估程序可以使用的最大物理内存的大小，配置系统的/proc/sys/vm/overcommit_memory ,/proc/sys/vm/overcommit_ratio,以及使用ulimit -v限制程序能使用的虚拟内存的大小，防止程序因OOM被杀死掉。