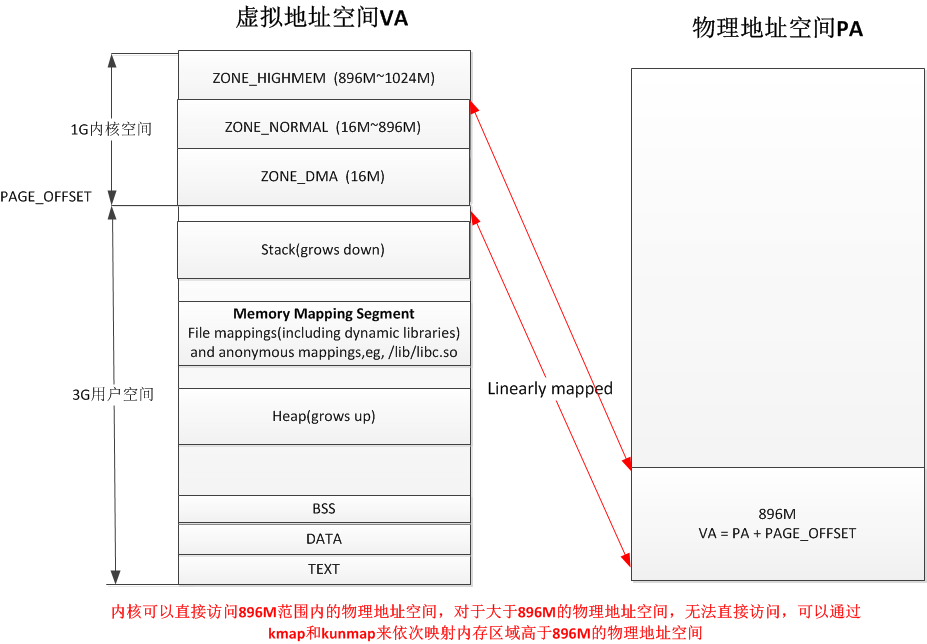
**ptmalloc**

**1 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 (unpredictable 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.

**2 x86平台Linux程序的内存分布**

Linux 程序载入内存后，loader 会把可执行文件中的各个段依次载入到从某一地址开始的空间中（载入地址取决于 link editor(ld)），如图所示。首先被载入的是 “.text” 段，然后是 “.data” 段， 最后是 “.bss” 段，这可以看作是程序的开始空间。程序所能访问的最后的地址是0xbfffffff，也就是到3G地址处，3G以上的1G空间是内核使用的，应用程序不可以直接访问。**应用程序的堆栈从最高地址处开始向下生长**，**“.bss”段与堆栈之间的空间是空闲的**。空闲空间被分成两部分，一部分为heap，一部分为mmap映射区域。在不同的Linux内核和机器上，mmap区域的开始位置一般是不同的。Heap和mmap区域都可以供用户自由使用。但是它在刚开始的时候并没有映射到内存空间内，是不可访问的。在向内核请求分配该空间之前，对这个空间的访问会导致一个 “segmentation fault”。用户程序可以直接使用系统调用来管理这块空间，但更多的时候都是程序都是使用C语言提供的 malloc() 和 free() 函数来动态的申请和释放内存。



**3 Allocator**

GNU Libc 的内存分配器(allocator) — ptmalloc 起源于 Doug Lea 的 malloc。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，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。

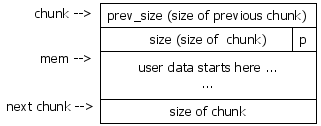
**4 chunk的组织**

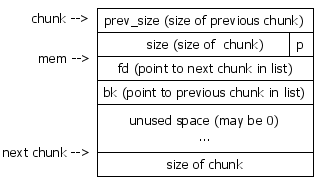
不管内存是在哪里被分配的，用什么方法分配，用户请求分配的空间在 ptmalloc 中都使用一个 **chunk** 来表示。用户调用 free() 函数释放掉的内存也并不是立即就归还给操作系统，相反， 它们也会被表示为一个 chunk，**ptmalloc 使用特定的数据结构来管理这些空闲的 chuck**。

**note：**用户free掉的内存并不是都会马上归还给系统，ptmalloc会统一管理heap和mmap映射区域中的空闲的chunk，当用户进行下一次分配请求时，ptmalloc会首先试图在空闲的chunk中挑选一块给用户，这样就避免了频繁的系统调用，降低了内存分配的开销

**4.1 chunk**

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

在图中，chunk 指针指向一个 chunk 的开始，一个chunk 中包含了用户请求的内存区域和相关的控制信息。**图中的 mem 指针才是真正返回给用户的内存指针**，chunk 的第二个域的最低一位为p， 它表示前一个块是否在使用中，p为0则表示前一个 chunk 为空闲，这时 chunk 的第一个域prev\_size才有效，prev\_size 表示前一个 chunk 的 size，程序可以使用这个值来找到前一个 chunk 的开始。当p为1时，表示前一个 chunk 正在使用中，prev\_size 无效，程序也就不可以得到前一个 chunk 的大小，而不能对前一个 chunk 进行任何操作。ptmalloc 分配的第一个块总是将p设为1，以防止程序引用到不存在的区域。

空闲 chunk 在内存中的结构如图所示：

当 chunk 空闲时，原本是用户数据区的地方存储了两个指针，指针 fd 指向后一个空闲的 chunk，而 bk 指向前一个空闲的 chunk，ptmalloc 通过这两个指针将大小相近的 chunk 连成一个双向链表。 而不同的 chunk 链表又是通过 bins 或者 fastbins 来组织的。

**4.2 chunk中的空间复用**

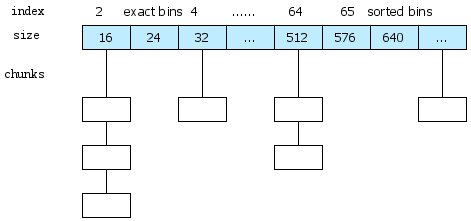
为了使得 chunk 所占用的空间最小，ptmalloc 使用了**空间复用**。一个 chunk 或者正在被使用，或者已经被 free 掉，所以 chunk 的中的一些域可以在使用状态和空闲状态表示不同的意义，来达到空间复用的效果。空闲时，一个 chunk 中至少要4个 size\_t 大小的空间，用来存储 prev\_size、size、 fd、bk，也就是16 bytes。chunk 的大小要 align 到8 bytes。当一个 chunk 处于使用状态时，它的下一个 chunk 的 prev\_size 域肯定是无效的。所以实际上，这个空间也可以被当前 chunk 使用。这听起来有点不可思议，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。

**5 空闲chunk容器**

**5.1 Bins**

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



数组中的前64个 bin 称为 “exact bins”，“exact bins”中的 chunk 具有相同的大小，两个相邻的 bin 中的 chunk 大小相差8 bytes。“exact bins”中的 chunk 按照最近使用顺序进行排列，最后释放的 chunk 被链接到链表的头部，而 allocation 是从尾部开始。这样，每一个 chunk 都有相同的机会被 ptmalloc 选中。后面的 bin 被称作 “ordered bins”。“ordered bins”中的每一个 bin 分别包含了一个给定范围内的 chunk，其中的 chunk 按大小序排列，相同大小的 chunk 同样按照最近使用顺序排列。ptmalloc 使用 “smallest-first，best-fit” 原则在空闲 “ordered bins” 中查找合适的 chunk。当空闲的 chunk 被链接到bin中的时候，ptmalloc 会把表示该 chunk 是否处于使用中的标志 p 设为0(注意，这个标志实际上处在下一个 chunk 中)，同时 ptmalloc 还会检查它前后的 chunk 是否也是空闲的， 如果是的话， ptmalloc 会首先把它们合并为一个大的 chunk， 然后将合并后的 chunk 放到 bin 中。要注意的是，**并不是所有的 chunk 被释放后就立即被放到bin中**。**ptmalloc 为了提高分配的速度，会把一些小的 chunk 先放到一个叫做 fastbin的容器内**。**5.2 fastbins**

一般的情况是，程序在运行时会经常需要分配和释放一些较小的内存空间。当 allocator 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 allocator 又需要从大的空闲内存中分出一块出来，这样无疑是比较低效的。故而，ptmalloc 中在分配过程中引入了 fastbins， 不大于 max\_fast (72 bytes) 的 chunk 被 free 后，首先会被放到 fastbins 中，fastbins 中的 chunk 并不改变它的使用标志p。这样也就无法将它们合并，当需要给用户分配的 chunk 小于或等于 max\_fast 时，ptmalloc 首先会在 fastbins 中查找相应的空闲块，然后才会去查找 bins 中的空间 chunk。在某个特定的时候，ptmalloc 会遍历 fastbins 中的 chunk，将相邻的空闲 chunk 进行合并，并将合并后的 chunk 放到 bins 中去。

**5.3 unsorted bins**

如果被用户释放的 chunk 大于 max\_fast, 则按上面的叙述它应该会被放到 bins中. 但实际上, ptmalloc 还引入了一个称为 “unsorted bins”的队列. 这些大于 max\_fast 的chunk 首先会被放到 “unsorted bins” 队列中, 在进行 malloc 操作的时候, 如果在 fastbins 中没有找到合适的 chunk, 则 ptmalloc 会先在 “unsorted bins”中查找合适的空闲 chunk, 然后才查找 bins. 如果 “unsorted bins” 不能满足分配要求. malloc 便会将 “unsorted bins” 中的 chunk 放到 bins 中, 然后再在 bins 中继续进行查找和分配过程. 从这个过程可以看出来, “unsorted bins”可以看做是 bins 的一个缓冲区, 增加它只是为了加快分配的速度, 忽略它对我们理解 ptmalloc 没有太大的影响, 在本文中, 这个过程就不被考虑了.

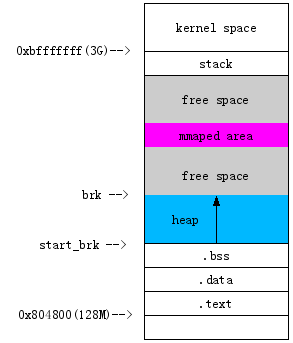
**5.4  例外的 chunk**并不是所有的 chunk 都按照上面的方式来组织, 实际上, 有两种例外情况：

**top chunk：**在前面一直提到, ptmalloc 会预先分配一块较大的空闲内存（也就是所谓的heap）, 而通过管理这块内存来响应用户的需求, 因为内存是按地址从低向高进行分配的, 在空闲内存的最高处, 必然存在着一块空闲 chunk, 叫做 “top chunk”. **当 bins 和 fastbins 都不能满足分配需要的时候, ptmalloc 会设法在 “top chunk” 中分出一块内存给用户**, 如果 “top chunk” 本身不够大, 则 ptmalloc 会适当的增加它的大小(也就增加了 heap 的大小). 以满足分配的需要, 实际上, “top chunk” 在分配时总是在 fastbins 和 bins 之后被考虑, 所以, 不论 “top chunk” 有多大, 它都不会被放到 fastbins 或者是 bins 中.

**mmaped chunk：**当需要分配的 chunk 足够大, 而且 fastbins 和 bins 都不能满足要求, 甚至 “top chunk” 本身也不能满足分配需求时, ptmalloc 会使用 mmap 来直接使用内存映射来将页映射到进程空间(具体的情况, 请参考第6节). 这样分配的 chunk 在被 free 时将直接解除映射, 于是就将内存归还给了系统, 再次对这样的内存区的引用将导致一个“segmentation fault”错误. 这样的 chunk 也不会包含在任何 bin 中.

**6 sbrk & mmap**

**6.1 sbrk**



.bss 段之上的这块分配给用户程序的空间被称为 heap (堆). start\_brk 指向 heap 的开始, 而 brk 指向 heap 的顶部. 可以使用系统调用 brk 和 sbrk 来增加标识 heap 顶部的 brk 值, 从而线性的增加分配给用户的 heap 空间. 在使用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DEFAULT\_MMAP\_THRESHOLD (128K bytes)时, ptmalloc 会调用sbrk增加一块大小为 ( 128 KB + chunk\_size ) align 4K 的空间作为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**(具体的分配过程见第7节). 若需要分配的 chunk 大小小于 DEFAULT\_MMAP\_THRESHOLD, 而 heap 空间又不够, 则此时 ptmalloc 会通过 sbrk 调用来增加 heap 值, 也就是增加 “top chunk”的大小, 每次 heap 增加的值都会 align 到4k bytes.

**6.2  mmap**当用户的请求超过 DEFAULT\_MMAP\_THRESHOLD , 并且使用 sbrk 分配失败的时候, ptmalloc 会尝试使用 mmap 直接映射一块内存到进程内存空间(我机器上是在0x40159000地址处). 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).**7 malloc()**

ptmalloc 的响应用户内存分配要求的具体步骤为：

1、 获取分配区的锁, ptmalloc 对 Doug Lea malloc 的主要扩展便是增加了线程支持. 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, 如果主分配区域的锁不能得到, 那么会 ptmalloc 会建立一个新的分配区域供当前线程使用.

2、 将用户的请求大小转换为实际需要分配的空间大小(见第4.2节的相关介绍).

3、 判断所需分配 chunk 的大小是否满足 chunk\_size <= max\_fast (**max\_fast 默认为 72 bytes**) , 如果是的话, 则转下一步, 否则跳到第5步.

4、 首先尝试在 fastbins 中摘取一个所需大小的 chunk 分配给用户. 如果可以找到, 则分配结束. 否则转到下一步.

5、 判断所需大小是否处在 “exact bins” 中, 即判断 chunk\_size < 512 bytes 是否成立(见图4). 如果 chunk 大小处在 “exact bins”中, 则转下一步, 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 “exact bins”, 并从该 bin 的尾部摘取一块恰好满足大小的 chunk. 若成功, 则分配结束, 否则, 转到下一步.

7、 到了这一步, 说明需要分配的是一块大的内存, 或者, “exact bins” 中找不到合适的 chunk. 于是, ptmalloc 首先会遍历 fastbins 中的 chunk , 将相邻的 chunk 进行合并, 并链接到 bins 中, 然后从 “sorted bins” 中按照 “smallest-first, best-fit” 原则, 找一块合适的 chunk, 从中划分一块所需大小的chunk, 并将剩下的部分链接回到 bins 中. 若操作成功, 则分配结束, 否则转到下一步.

8、 如果搜索 fastbins 和 bins 都没有找到合适的 chunk, 那么就需要操作 top chunk 来进行分配了. 判断 top chunk 大小是否满足所需 chunk 的大小, 如果是, 则从 top chunk 中分出一块来. 否则转到下一步.

9、 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: 调用 sbrk, 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 DEFAULT\_MMAP\_THRESHOLD(128KB), 如果是的话, 则转下一步, 调用 mmap 分配, 否则跳到第11步, 使用 sbrk 来增加 top chunk 的大小.

10、 使用 mmap 系统调用在大约 0x40159000 (大约为1G) 地址处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.

11、 判断是否为第一次调用 malloc, 若是, 则需要进行一次初始化工作, 分配一块大小为 (chunk\_size + 128K) align 4KB 大小的空间作为初始的 heap. 若已经初始化过了, 则调用 sbrk 增加 heap 空间, 使之满足分配需求, 并将内存指针返回给用户.

**总结**:

根据用户请求分配的内存的大小, ptmalloc 有可能会在两个地方为用户分配内存空间. 在第一次分配内存时, brk 值等于 start\_brk, 所以实际上 heap 大小为0, top chunk 大小也是0. 这时, 如果不增加 heap 大小, 就不能满足任何分配要求. 所以, 若用户的请求小于 DEFAULT\_MMAP\_THRESHOLD, 则 ptmalloc 会初始化heap. 然后在 heap 中分配空间给用户, 以后的分配就基于这个 heap 进行. 若第一次用户的请求就大于DEFAULT\_MMAP\_THRESHOLD, 则 ptmalloc 直接使用 mmap 分配一块给用户, 而 heap 也就没有被初始化, 直到用户第一次请求小于DEFAULT\_MMAP\_THRESHOLD 的内存分配. 第一次以后的分配就比较复杂了, 简单说来, ptmalloc 首先会查找 fastbins, 如果不能找到匹配的 chunk, 则查找 “exact bins”. 若还是不行, 则查找 “sorted bins”. 在 fastbins 和 “exact bins” 中的查找都需要精确匹配, 而在sorted bins 中查找时, 则遵循 “smallest-first, best-fit” 的原则, 不需要精确匹配. 若以上方法都失败了, 则 ptmalloc 会考虑使用 top chunk. 若top**8  free()**free() 函数接受一个指向分配区域的指针作为参数, 释放该指针所指向的 chunk. 而具体的释放方法则看该 chunk 所处的位置和该 chunk 的大小. free()函数的工作步骤如下:1、free() 函数同样首先需要获取分配区的锁, 来保证线程安全.2、判断传入的指针是否为0, 如果为0, 则什么都不做, 直接return. 否则转下一步:3、判断所需释放的 chunk 是否为 mmaped chunk, 如果是, 则直接释放 mmaped chunk, 解除内存空间映射. 该空间不再有效. 释放完成. 否则跳到下一步.4、判断 chunk 的大小和所处的位置, 若 chunk\_size <= max\_fast , 并且 chunk 并不位于 heap 的顶部, 也就是说并不与 top chunk 相邻, 则转到下一步, 否则跳到第6步. (因为与 top chunk 相邻的小 chunk 也和 top chunk 进行合并, 所以这里不仅需要判断大小, 还需要判断相邻情况.)5、将 chunk 放到 fastbins 中, chunk 放入到 fastbins 中时, 并不设置该 chunk 使用位. 也不与相邻的 chunk 进行合并. 只是放进去, 如此而已. 做实验的结果还发现**ptmalloc 放入 fastbins 中的 chunk 中的用户数据去全置为 0**. 但是在源代码中找不到相关的代码. 这一步做完之后释放便结束了, 程序从 free() 函数中返回.

6、判断前一个 chunk 是否处在使用中, 如果前一个块也是空闲块, 则合并. 并转下一步.7、判断当前释放 chunk 的下一个块是否为 top chunk, 如果是, 则转第9步, 否则转下一步.8、判断下一个 chunk 是否处在使用中, 如果下一个 chunk 也是空闲的. 则合并, 并将合并后的 chunk 放到 bins 中. 注意, 这里在合并的过程中, 要更新 chunk 的大小, 以反映合并后的 chunk 的大小. 并转到第10步.9、 如果执行到这一步, 说明释放了一个与 top chunk 相邻的chunk. 则无论它有多大, 都将它与 top chunk 合并, 并更新 top chunk 的大小等信息. 转下一步.10、判断合并后的 chunk 的大小是否大于 FASTBIN\_CONSOLIDATION\_THRESHOLD, 如果是的话, 则会触发进行 fastbins 的合并操作, fastbins 中的 chunk 将被遍历, 并于相邻的空闲 chunk 进行合并, 合并后的 chunk 会被放到 bins 中. fastbins 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 DEFAULT\_TRIM\_THERESHOLD. 如果是的话, 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的空间是不会归还. ptmalloc 会一直控制这部分内存. 用于响应用户的分配请求. 做完这一步之后, 释放结束, 从 free 函数退出.