ICS Malloc Lab Report

杜一阳　计11　2021011778

2023.1

1　设计思路

采用分离适配的方式，维护空闲链表以供分配时快速查找最优块。

1.1　堆上存储布局

堆上首先填充 1word 用于16字节对齐，紧接着是 prologue 块，它是一个永不释放的已分配块，内部存储 16words 用于存放空闲列表的头指针。后面是若干个常规负载块，最后以一个标记长度为0的 epilogue 块结尾。对于任意的块，用其第二个 word 所在低字节作为其地址，即除了 header 以外的最底地址。（图1）

当常规块被分配时，它仅由头尾标签和内部负载组成；当未被分配时，内部负载的前两个字被用于存储 PREV 和 SUCC ：它所在的空闲链表的下一个元素和前一个元素的地址。正因如此，在对齐时还要求分配请求至少为两个字长。头尾标签是该块的大小，并用后三位为标签记录块状态，从低到高的三位分别表示：是否已分配、是否为 prologue、是否为 epilogue。

1.2　分离的空闲链表

每个空闲链表只存储特定大小范围内的空闲块，称为一个桶(bin)。为了提高查找效率，设计两种桶：fast bin 和 regular bin 。前者只存储某个特定长度的块，而后者以升序存储长度在某个范围内的块。二者的数量可以调整，本实现中用了 10 个 fast bin 和 6 个 regular bin 。根据块的大小，很容易算出对应的桶。

初始化时，将各链表头指针置为 NULL，并记录在堆中的地址。

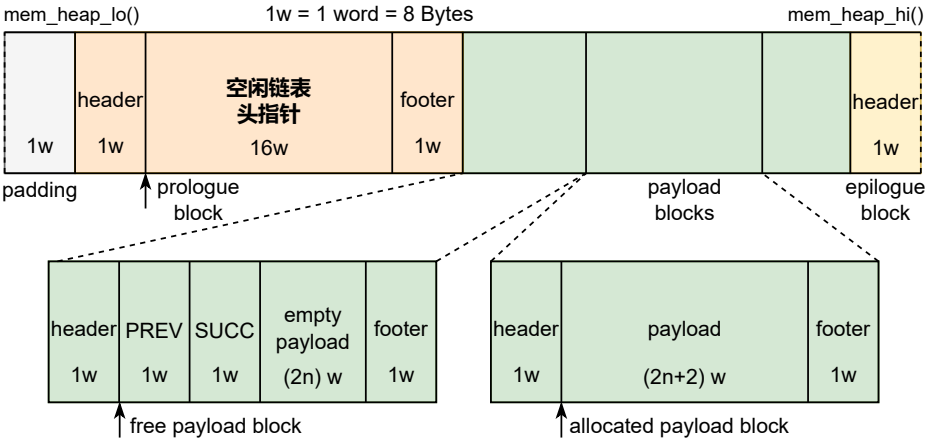

图 1: 堆上存储布局

为了减少内存碎片，要求任意两个空闲块不相邻，这意味着总是要在插入空闲块后检查是否能合并，并将合并后的块重新放到合适的桶中。

链表的插入和删除操作与常规的双向链表类似，只是在插入时，如果当前桶为空，那么 PREV 指针在指向 prologue 块中的链表头指针地址时需要增加与 SUCC 引用相反的偏移量，这是因为后续在对头指针进行（可能的）修改时，是将它视作一个块进行 SUCC 调用的，而 SUCC 调用会添加一个偏移量。

1.3 malloc 算法

对齐请求后在对应的桶中寻找是否有合适的块。对于 fast bin，由于桶中块大小一致，只需关注第一个块是否存在；对于 regular bin，由于长度升序，只需遍历找到满足要求的最小块。如果当前桶中没有，就在下一个桶中查找。如果所有桶都为空，则扩展堆空间。

在找到合适的块后，考察该块的长度与请求的长度，如果分配请求后剩余空间足够形成一个块（$\geq 32Bytes$），则将原块分配给请求后在空闲链表删除并将剩余块插入空闲链表，否则直接将整个块分配给请求并将其在空闲链表中删除。

当存在 realloc 操作时，为了避免 malloc 的空间对 realloc 的影响，从低地址开始采用隐式空闲链表的方式查找合适的块，返回首次匹配。

1.4 free 算法

将对应的块置为空闲，合并相邻空闲块后插入对应的桶即可。

1.5 realloc 算法

当 ptr 或 size 为 0 时，根据定义调用 malloc 或 free。

若 realloc 后块大小没有增加，则考察大小的变化量。若这部分多出来的空间足够形成一个块，则将其插入对应的桶中，并将原来的块更改大小后返回，否则不进行任何改变。这里一定返回的指针一定没有变化，并且无需 memcpy 操作。

若 realloc 后块大小增加，则考察相邻的下一个块，如果它未被分配且大小足够，那么可以把它的一部分或全部（取决于剩余空间是否足够形成一个块）分配给当前块，更新桶中的信息和 header 与 footer 中的数。

若没有下一个 payload 块，即下一个块是 epilogue，可以直接要求系统增加堆空间，得到一个新的未分配块，转化为前一种情形解决。

以上情形均无需 memcpy。如果无法满足上述任意一种情形，则只能依次调用 malloc-memcpy-free ，将数据分配到新的内存中。

2 测试结果及性能分析

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   99%    5694  0.000249  22831
 2       cccp-bal.rep       yes   99%    5848  0.000239  24448
 3    cp-decl-bal.rep       yes   99%    6648  0.000288  23091
 4       expr-bal.rep       yes   99%    5380  0.000250  21537
 5 coalescing-bal.rep       yes   95%   14400  0.000292  49248
 6     random-bal.rep       yes   96%    4800  0.001037   4629
 7    random2-bal.rep       yes   94%    4800  0.001052   4563
 8     binary-bal.rep       yes   95%   12000  0.000357  33651
 9    binary2-bal.rep       yes   82%   24000  0.000679  35351
10    realloc-bal.rep       yes  100%   14401  0.000250  57535
11   realloc2-bal.rep       yes   87%   14401  0.000212  68058
Total                            95%  112372  0.004905  22910

Score = (57 (util) + 40 (thru)) * 11/11 (testcase) = 97/100
u2021011778@hp:~/malloclab-handout$
```

图 2: 测试结果[0]

从空间利用率上看，大部分测试中均能达到 95% 以上，说明和简单分离存储相比，这种总是合并相邻未分配块的时间换空间的方法时有效的，相当于每次寻找的都是 best fit 。而且堆上额外的分离空闲链表头只占 $16 \times 8Bytes$，对空间影响较小。

从吞吐量上看，平均 Kops 在 20000 以上也说明分离空闲链表能够很快速的查找到对应的块，这一方面是因为分离链表本身就在入口处排除了那些大小肯定不合适的块，另一方面是因为 fast bin 同一大小块无需遍历，regular bin 升序排列只需线性查找，因此最坏情形下查找耗时为 O(桶数量+最大桶长) ，而最好情形下（fast bin 或短的 regular bin）只需 $O(1)$ 。

当前的桶数量是 10+6 ，然而最初的设计是类似 gcc malloc 的 80+128，然后发现时间和空间都不好，这是因为：当查找失败时，耗时是和桶数量正相关的——考虑一个完全空的空闲链表，有几个桶就得查看几个桶，这十分耗时。更不用说这消耗 $1664Bytes$ 的堆上空间，对 coalescing-bal.rep 等本身使用空间较少的测例的空间利用率产生了很大的负面影响。仅有 random-bal.rep 例外：在增大桶的数量时耗时减少，这可能是由于随机数据下块的长度分布较为均匀，桶增多分布地均匀更便于查找。

binary2-bal.rep 的空间利用率仅有 82%，这是因为在它使用内存最多的时刻，有一半数量的内存申请是 16 字节的，这是最短的块负载长度，也就意味着 header 和 footer 的额外开销占比最大。realloc2-bal.rep 的空间利用率也只有 87%，这是因为第一次 realloc 时长度为 4092 的块后面分配了一个长度为 16 的块，它不得不通过 malloc-memset-free 调用换到后面长为 4097 的块，而前面 4092 的未分配区域此后至多被两个 16 长度的块使用过，其余的空间浪费了。不过在这个测例的情形下，无论是末端大块的 realloc，还是开头小块的 malloc，均可以 O(1) 时间完成，因此吞吐量很大。

3 参考资料

CSAPP 10.9 动态存储器分配

课件 8 内存分配基础

glibc wiki MallocInternals [1]

[0]测试环境：Intel(R) Xeon(R) CPU E5-4610 v2 @ 2.30GHz, 32GB RAM, Ubuntu 20.04, x86_64, where libc malloc 10000 Kops/sec .

[1]https://sourceware.org/glibc/wiki/MallocInternals