

MallocLab实验报告

实验目标

深入理解堆内存管理机制，实现四个函数mm_init(),mm_malloc(size_t size) mm_free(void* bp) mm_realloc(void* bp, size_t newsize)，分别用于初始化堆，按照用户的要求分配指定大小的空间，将已分配的空间释放称为未分配空间，为已分配的空间重新分配一段不同大小的内存，并将原有的内容进行拷贝操作。通过这四个函数的实现来模拟堆内存管理机制，尽可能地提高吞吐量和峰值空间利用率 (peek utility percentage)

设计思路

本质是对一系列的malloc，free请求队列的相应，同时满足一些特定的限制条件，比如说：不可以控制用户请求空间的大小；用户的请求必须立即响应，不可以使用buffer的形式，这一点是和shredlab最本质的区别；只可以使用空闲内存，不能干涉已经分配给用户的内存；为用户分配的内存必须满足一系列的alignment要求。无论使用哪一种分配手段，都需要一些通用的函数，整体的框架也不会有特别大的改变。主要使用的函数有：

- mm_init () 初始化先分配一块内存，并设置好整个heap的开始块，序言块，内存块，结尾块。
- void extend_heap(size_t size)当现有的已申请内存不够用的时候，扩展内存。
- void* coalesce (void* bp,size_t size) bp是一个空闲块的指针，这个函数的功能是检查一个空闲块的左右，并将左右可能的空闲块进行合并成为一个大的空闲块。
- void* find_fit(size_t asize)从所有的空闲块之中选取最适合当前请求的，返回指针。
- void place(void* bp,size_t asize)将asize大小的内容放置到bp所指向的空闲块，对于剩余的部分，如果超过了一定大小，会被分割成另外一个空闲块；如果比较小，就直接把整个bp所在的空闲块全部都分给请求内容，直接作为internal fragment。

由于无法使用全局变量和结构体定义，因此需要广泛地使用宏定义：

```
#define PACK(size,alloc) ((size)|(alloc))
#define GET(p) (*(unsigned int*)(p))//get 4 bytes;
#define PUT(p,val) (*(unsigned int*)(p)=(unsigned int)(val))//write 4 bytes

#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p)& 0x1)

#define HDPR(bp) ((char*)(bp)-WSIZE)
#define FTPR(bp) ((char*)(bp)+GET_SIZE(HDPR(bp))-DSIZE)
#define NEXT_BLK(p) ((char*)(p)+GET_SIZE(((char*)(p)-WSIZE)))
#define PREV_BLK(p) ((char*)(p)-GET_SIZE(((char*)(p)-DSIZE)))
```

各种不同的思路，其实只是在不断地调整以上几个函数的实现方式而已，整体大的框架没有改变。

解决方案1：implicit list + boundary tags

解决方案整体参考了教材CSAPP上的有关代码和框架。

动态内存分配器最简单的实现方案就是采取隐式链表的形式，同时对于每一个块（无论是未分配块还是已分配块），不仅含有包含header，同时在尾部块的最后4个字节还设置了一个footer，head和footer具有完全相同的内容，都是由两部分组成，size（前三个字节）和alloc（最后一个字节的最后一个bit）。

将coalesce分为四种可能的情况：

- 左右两边都已经被分配，此时无法合并，可以直接返回
- 上一块已经被分配，下一块空闲
- 上一块空闲，下一块已经被分配
- 上下两块都是空闲状态

尤其需要注意的一点是，在更改相应的header和footer时，先后次序很重要。从相关的宏定义可以看到，

```
#define HDPR(bp) ((char*)(bp)-WSIZE)
#define FTPR(bp) ((char*)(bp)+GET_SIZE(HDPR(bp))-DSIZE)
```

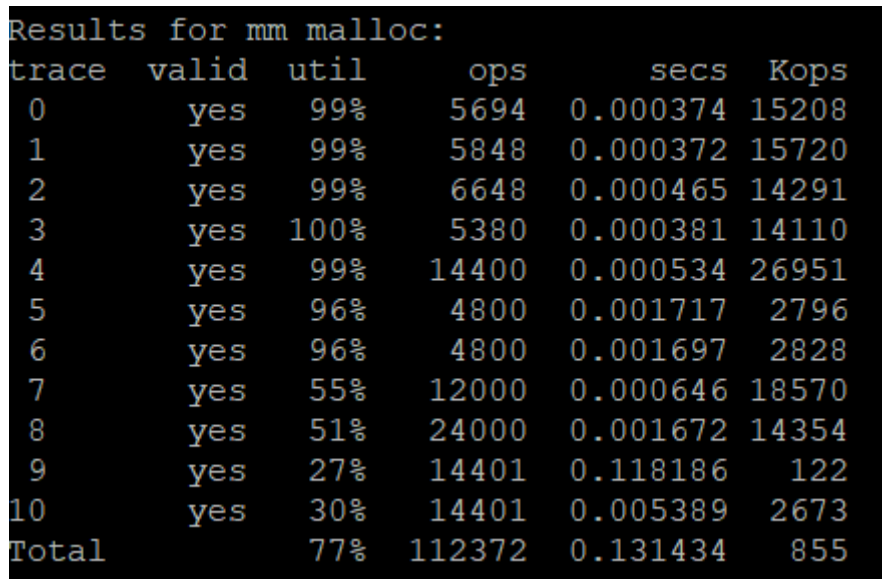
footer和header的确定并不是独立的，后者的位置确定依赖于前者。

place () 放置函数分为两种情况，如果剩余空间比较小，小于16个字节的时候，就直接将整个块分配给用户，不再做进一步的切割，形成internal fragment；如果剩余空间比较大，就再做一次切割，把剩下的部分加入到隐式空闲链表之中。

find_fit采用first fit方法，从头开始扫描，遇到尺寸超过请求的空闲块，直接放置。

realloc 采取原始的版本。

实验效果如下图所示：



trace	valid	util	ops	secs	Kops
0	yes	99%	5694	0.000374	15208
1	yes	99%	5848	0.000372	15720
2	yes	99%	6648	0.000465	14291
3	yes	100%	5380	0.000381	14110
4	yes	99%	14400	0.000534	26951
5	yes	96%	4800	0.001717	2796
6	yes	96%	4800	0.001697	2828
7	yes	55%	12000	0.000646	18570
8	yes	51%	24000	0.001672	14354
9	yes	27%	14401	0.118186	122
10	yes	30%	14401	0.005389	2673
Total		77%	112372	0.131434	855

可以看到，虽然前7个trace的测试比较理想，但是到了trace7和trace8的utility明显下降，trace10和trace9的utility下降更加严重，最终只能拿到53分

解决方案2：best fit+realloc改进

我们从解决方案一的整体框架开始进行优化，首先一种优化办法是采用best fit来替代原有的find fit策略。虽然这样做会导致place的时间复杂度成为标准的O (N) 级别，但是作为trade-off可以选择尺寸最合适的块，减少external fragment

best fit代码如下

```
void* find_fit(size_t asize)
{
    void* heap_listp=(void*)((char*)(mem_heap_lo())+2*WSIZE);
    void* rover;
    void* best_bp=(void*)-1;
```

```

unsigned residue=32768;
for(rover=heap_listp;GET_SIZE(HDPR(rover))>0;rover=NEXT_BLKPR(rover))
    if(GET_ALLOC(HDPR(rover))==0&&(asize<GET_SIZE(HDPR(rover))))
    {
        if(GET_SIZE(HDPR(rover))-asize<residue)
        {
            residue=GET_SIZE(HDPR(rover))-asize;
            best_bp=rover;
        }
    }
if(best_bp==(void*)-1)
    return NULL;
return best_bp;
}

```

同时将realloc进行改进。按照mm_realloc的要求，分为以下几种情况：

- 指针为空，这种情况下相当于直接进行malloc操作
- 新请求的空间大小为0，这种情况归结为free操作
- 将新请求的空间进行alignment对齐操作。
- 新请求的空间比原有的空间小，不做任何处理
- 新请求的空间比原有的空间更大，首先将原有的块进行一次合并操作，即coalesce，如果合并之后的尺寸超过了已请求的尺寸，就只需要复制内容就好
- 如果进行合并之后的尺寸依然比请求尺寸要小，就直接malloc，将内容搬移过来之后，再次free

优化之后得分效果如下：

```

Results for mm malloc:
trace  valid  util    ops    secs  Kops
0      yes   35%   5694  0.013174  432
1      yes   24%   5848  0.013157  444
2      yes   35%   6648  0.018272  364
3      yes   44%   5380  0.013939  386
4      yes   99%  14400  0.000290 49689
5      yes   95%   4800  0.021580  222
6      yes   94%   4800  0.020911  230
7      yes   55%  12000  0.112176  107
8      yes   51%  24000  0.389759   62
9      yes   93%  14401  0.038177  377
10     yes   22%  14401  0.008257 1744
Total                59% 112372  0.649693  173

Perf index = 35 (util) + 12 (thru) = 47/100

```

可以看到，虽然trace9和trace10的utility有了非常大的改进，但是整体上效果变得更差了，因为trace0，trace1，trace2和trace3的utility变得很差。

解决方案3 segment list

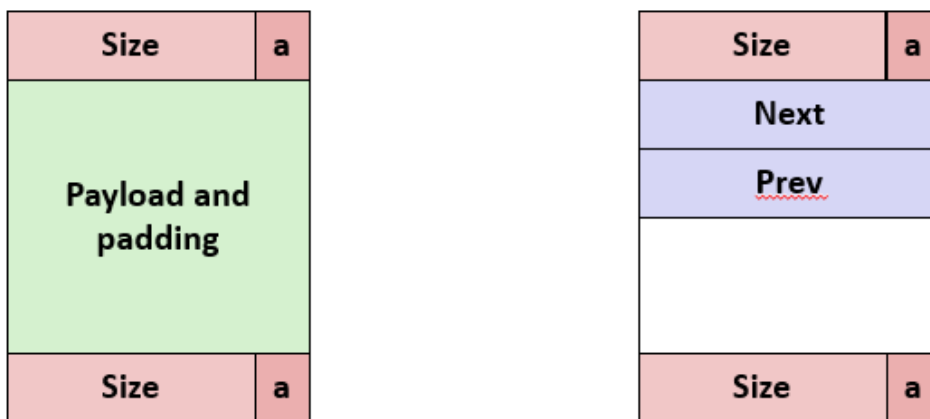
为了进一步提高效率，我们把所有的空闲块用显式的链表串联起来，并根据空闲块的大小，分别分配到不同的链表之中。为此，所有的块之间，有两种关系，一种关系是物理地址上的相邻，另一种关系是在同一条链表上的相邻。为此，需要添加两个宏定义

```

#define NEXT_PTR(bp) ((void*)((char*)(bp)+WSIZE))
#define PREV_PTR(bp) ((void*)bp)

```

由此我们可以看到，对于分配块，只有header和footer，其余的部分全部都是payload。而对于未分配块，则分为5个部分。header， footer， next pointer previous pointer， payload五个部分。



要做到segment list，就需要对大小不同的块进行分类，放到不同的链表之中。由于不允许使用全局变量，所以我把所有的链表头都放在了heap的头部。并按照64byte， 128byte， 256byte， 512byte， 1024byte， 2048byte， 4096byte进行分级。

每次由于用户的free导致出现了新的空闲块之后，需要将空闲块加入到列表之中。首先，我们需要根据这个块的大小来确定链表头的位置，这一部分使用一个函数来实现，这个函数叫做seekroot ()

```
void* seekroot(size_t size)
{
    if(size<=32)
        return (char*)mem_heap_lo()+1*WSIZE;
    else if (size<=64)
        return (char*)mem_heap_lo()+2*WSIZE;
    else if(size<=128)
        return (char*)mem_heap_lo()+3*WSIZE;
    else if(size<=512)
        return (char*)mem_heap_lo()+4*WSIZE;
    else if(size<=1024)
        return (char*)mem_heap_lo()+5*WSIZE;
    else if(size<=2048)
        return (char*)mem_heap_lo()+6*WSIZE;
    else if(size<=4096)
        return (char*)mem_heap_lo()+7*WSIZE;
    else
    {
        return (char*)mem_heap_lo()+8*WSIZE;
    }
}
```

在每一次coalesce的过程中，需要做两件事，将前后的空闲块从链表之中删除；将新合并之后的空闲块插入应有的位置上去。我按照尺寸的有序性插入，也就是在同一个segment的范围之内，大块在后，小块在前。一个新的空闲块分为四种可能的情况：

- 空插，链表头是空的，新的空闲块成为唯一的元素
- 头插，新块插入到紧接在链表头之后
- 中插，插入在中间位置
- 尾插，新的空闲块最大，被插入在最后

与之相对的，删除也是同样分为这几种情况。

尤其需要注意的一点是，正常的mm_free所使用的coalesce函数和mm_realloc使用的coalesce不能混为一谈。我在实现mm_realloc的过程中，曾经想要直接借助实现mm_free阶段已经写好的coalesce函数，但是发现其实是行不通的。因为mm_free进行合并的时候，是把当前的块当成了空闲块，合并完之后是会直接加入到新的对应大小的list之中。但是mm_realloc需要在合并完之后依然当作分配块，不可以加入到链表中去。否则的话，由于空闲块具有NEXT和PREV两个指针，所以如果对已经分配的块调用这两个并不存在的指针，会导致segment fault。

下图是解决方案3的最终评价得分。

```
Results for mm malloc:
```

trace	valid	util	ops	secs	Kops
0	yes	99%	5694	0.000373	15274
1	yes	99%	5848	0.000364	16048
2	yes	99%	6648	0.000459	14484
3	yes	100%	5380	0.000377	14274
4	yes	99%	14400	0.000534	26966
5	yes	96%	4800	0.001741	2757
6	yes	96%	4800	0.001598	3005
7	yes	55%	12000	0.000743	16151
8	yes	51%	24000	0.001685	14247
9	yes	99%	14401	0.001849	7790
10	yes	98%	14401	0.000965	14919
Total		90%	112372	0.010687	10515

Perf index = 54 (util) + 40 (thru) = 94/100

反思与总结

1. 整个实验可以分步进行，realloc的部分，在一开始可以直接使用malloc和free来完成，后期把malloc和free写的差不多了的时候可以从新开始写realloc
2. realloc之中使用的合并一定不可以直接利用在free的过程之中使用的合并，两者虽然本质是相同的，但是一个处理的是空闲块，一个处理的是已分配块，不可同日而语。
3. 基本上能用宏定义的尽量都用宏定义，哪怕仅仅是一个地址解析，因为中间涉及到了两层指针，会有点混乱
4. 基本上bug都是segment fault，基本上所有的段错误都是由于使用宏定义的时候对分配块使用了只用空闲块才具有的结构。