


# ICS Malloc Lab WriteUp

王浩然 计23 2022010229

2023.12.13

## 0. 写在前面

malloc lab 比较逆天，据说是 csapp 里最难的一个 lab，需要模拟 libc 中的内存分配器，涉及 malloc, free, realloc, init 四个函数。主要考察充分的  力和面对 Segmentation Fault 时的心理承受能力。

```
Testing mm malloc
Reading tracefile: amptjp-bal.rep
Segmentation fault
```

各种 Segmentation Fault

开始时，我抄了 csapp 上的 demo，这是基于 implicit list 的。将 8 字节对齐调为 16 字节对齐后，我惊奇地发现居然只有二三十分（具体记不清了），我跑 handout 附赠的 naive 算法得分也比这个高呀！然后我乱调了一通，在痛苦 gdb 找 bug 后，我突然发现程序似乎跑通了。在出结果前，我甚至想，这个实验如此简单！然后拿到了可喜可贺的分数。

```
Score = (40 (util) + 4 (thru)) * 11/11 (testcase) = 44/100
```

起初的尴尬结果

嗯...我此时认为评分中的  $w=0.4$ ，也就是认为我的 util 分跑满了，看到 AVG\_LIBC\_THRUPUT 是 10000e 而非注释中的 10000。于是找学峰助教报告了这个锅，得到的答复是这里注释和取值的单位不同，前者是 ops 而后者是 Kops，后面会转化。省流：我的 thru 分数就是 4，没错！我尴尬地向学峰助教表示了谢意，然后回炉重造 CSAPP

CSAPP 提到了三种内存分配方案，分别是

### 1. Implicit List

将空闲内存块直接组织在分配的内存块中。每个内存块保存其大小以及指向下一个内存块的指针。分配内存时，从链表中查找第一个满足大小的空闲块，并将其切割成所需大小和剩余大小的两个块。释放内存时，将空闲块合并到相邻的空闲块，以合并连续的空闲内存。

### 2. Explicit List

使用一个链表来显式地维护所有的空闲内存块。每个内存块含有一个头部，其中包含了块的大小和指向下一个块的指针。空闲块通过链表链接在一起，使得查找合适大小的可用块变得更加高效。分配和释放内存是通过操作链表节点来完成的。



```

1 static inline int getIndex(size_t v) {
2     size_t r, shift;
3     r      = (v > 0xffff) << 4; v >= r;
4     shift = (v > 0xff ) << 3; v >= shift; r |= shift;
5     shift = (v > 0xf  ) << 2; v >= shift; r |= shift;
6     shift = (v > 0x3   ) << 1; v >= shift; r |= shift;
7                                     r |= (v >> 1);
8     int x = r - 5;
9     if(x < 0) x = 0; // 若当前大小类过小, 归入最小大小类
10    if(x >= SEGSUM) x = SEGSUM - 1; // 若当前大小类过大, 归入最大大小类
11    return x;
12 }

```

- extend\_heap 堆扩展

调用 mm\_sbrk, 上移 brk, 并返回匹配空闲块的头指针。在设置好头尾标记后通过 coalesce 合并前后空闲块后插入到空闲块链表中。

```

1 static void *extend_heap(size_t asize) {
2     char *bp;
3     if((long)(bp = mem_sbrk(asize)) == -1) return NULL;
4     // 初始化新空闲内存块
5     PUT(HDRP(bp), PACK(asize, FREE));
6     PUT(FTRP(bp), PACK(asize, FREE));
7     PUT(HDRP(NEXT_BLKp(bp)), PACK(0, ALLOCATED));
8     return coalesce(bp); // 不要忘记合并, 刚开始直接返回bp了...
9 }

```

- coalesce 先合并再插入

```

1 size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKp(bp))); //获取地址前块标记
2 size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKp(bp))); //获取地址后块标记
3 size_t size = CUR_SIZE(bp); //初始化新空闲内存块大小

```

分四种情况, 都是基本的链表更新。

```

1 // CASE 1: 前后块都已分配
2 INSERT(bp);
3 return bp;
4
5 // CASE 2: 前块已分配, 后块未分配
6 size += NEXT_SIZE(bp); //更新内存块大小
7 DELETE(NEXT_BLKp(bp));
8 UPDATE({HDRP(bp), FTRP(bp), PRED(bp), SUCC(bp)})
9
10 // CASE 3: 前块未分配, 后块已分配
11 size += PREV_SIZE(bp);
12 DELETE(PREV_BLKp(bp));
13 UPDATE({FTRP(bp), HDRP(PREV_BLKp(bp)), PRED(bp), SUCC(bp)})
14

```

```

15 // CASE 4: 前后块都未分配
16 size += NEXT_SIZE(bp) + PREV_SIZE(bp);
17 DELETE(PREV_BLKBP(bp));
18 DELETE(NEXT_BLKBP(bp));
19 UPDATE({HDRP(PREV_BLKBP(bp)), FTRP(NEXT_BLKBP(bp)), PRED(bp), SUCC(bp)})

```

最后 `INSERT(bp)` 即可

- `find_fit` 用首次适配(first-fit)策略，直接暴力在当前Segregated List里找第一个适配块

```

1 while(seg < SEG_LEN) {
2     char *root = segregated_listp + seg * WSIZE;
3     char *bp = (char*)SUCC_BLKBP(root);
4     while(bp) {
5         if((size_t)CUR_SIZE(bp) >= size) return bp;
6         bp = (char*)SUCC_BLKBP(bp);
7     }
8     seg++;
9 }

```

- `place`

```

1 // CASE 1: 若当前空闲块未分配，从链表中删除
2 if(rm_size >= 8 * DSIZE) { // 剩余空闲块足够大
3     if(asm_size > 8 * DSIZE) { // 分割空闲块，注意这里的阈值
4         PUT(HDRP(bp), PACK(rm_size, FREE));
5         PUT(FTRP(bp), PACK(rm_size, FREE));
6         PUT(HDRP(NEXT_BLKBP(bp)), PACK(asm_size, ALLOCATED));
7         PUT(FTRP(NEXT_BLKBP(bp)), PACK(asm_size, ALLOCATED));
8         insert_free_block(bp);
9         return NEXT_BLKBP(bp);
10    } else { // 不分割空闲块
11        PUT(HDRP(bp), PACK(asm_size, ALLOCATED));
12        PUT(FTRP(bp), PACK(asm_size, ALLOCATED));
13        PUT(HDRP(NEXT_BLKBP(bp)), PACK(rm_size, FREE));
14        PUT(FTRP(NEXT_BLKBP(bp)), PACK(rm_size, FREE));
15        coalesce(NEXT_BLKBP(bp));
16    }
17 }
18 // CASE 2: 已分配，直接更新
19 PUT(HDRP(bp), PACK(bsize, ALLOCATED));
20 PUT(FTRP(bp), PACK(bsize, ALLOCATED));

```

下面是重点的四个函数！

- `mm_init`

额外分配 `SEGSUM+3` 个地址，前 `SEGSUM` 个用来存 Segregated List 的各链表首地址，后三个存序言块和结尾块

```

1 // 初始化空闲块大小类头指针
2 int i;
3 for(i = 0; i < SEGSUM; ++i) PUT(heap_listp + i * WSIZE, NULL);
4 // 分配块
5 PUT(heap_listp + (i + 0) * WSIZE, PACK(DSIZE, ALLOCATED)); // 序言块header
6 PUT(heap_listp + (i + 1) * WSIZE, PACK(DSIZE, ALLOCATED)); // 序言块footer
7 PUT(heap_listp + (i + 2) * WSIZE, PACK(0, ALLOCATED)); // 结尾块header, 注意结尾块大小为
8 0, 没有footer, 只用作哨兵
9 segregated_listp = heap_listp; // 初始化分类大小类头指针
10 heap_listp += (i + 1) * WSIZE; // 初始化堆开始地址

```

- mm\_malloc

也是调各种接口函数。先对齐大小, 尝试找适配块, 若找不到则创建新块

```

1 size_t asize = ALIGN(size); // 对齐后大小
2 size_t extendsize; // 扩展堆大小
3 char *bp;
4 // 无需分配
5 if(size == 0) return NULL;
6 // 找到适配空间
7 if((bp = find_fit(asize, get_index(asize))) != NULL) return place(bp, asize);
8 // 未找到适配空间
9 extendsize = MAX(asize, CHUNKSIZE);
10 if((bp = extend_heap(extendsize)) == NULL) return NULL;
11 return place(bp, asize);

```

- mm\_free

直接置空标记, 然后合并前后块即可

```

1 size_t size = CUR_SIZE(ptr);
2 PUT(HDRP(ptr), PACK(size, FREE)); //修改头部标记
3 PUT(FTRP(ptr), PACK(size, FREE)); //修改尾部标记
4 coalesce(ptr);

```

- mm\_realloc

讲讲核心逻辑

```

1 // CASE 1: 前块已分配, 后块未分配, 且合并后空间足够
2 total_size += next_size;
3 DELETE(next_bp);
4 UPDATE(HDRP(ptr), FTRP(ptr))
5 place(ptr, total_size);
6
7 // CASE 2: 后块不存在, 且空间不足
8 size_t extend_size = asize - total_size;
9 ALLOCATE(extend_size)
10 UPDATE(HDRP(ptr), FTRP(ptr), NEXT_BLKPTR(ptr))

```

```

11 place(ptr, asize);
12
13 // CASE 3: 其他情况 (无需更改大小, 可直接分配)
14 if((newptr = mm_malloc(asize)) == NULL) return NULL;
15 memcpy(newptr, ptr, MIN(total_size, size));
16 mm_free(ptr);
17 return newptr;

```

## 2. 性能评估细节和分析

我的程序最终在所有数据点的 thru 测试上取得了满分，但在部分测试点上 util 分数较低：coalescing-bal.rep、binary-bal.rep、binary2-bal.rep，故主要分析二者即可

该分配模式是交替分配一个小型内存块和一个大的块。小块

(16或64)被故意设置为2的幂，而较大的块(112或448)不是2的幂。由于我使用了segregated list，其在块大小为2的幂次时复杂度能做到  $O(\log n)$ ，其余情况需要考虑块大小的二进制分解，复杂度可以到  $O(n)$ ，我们在这种情况下效率很低。我尝试改变块大小，分数突然上升了，于是进入到如下部分：

### 痛苦调参

篇幅有限，只讲两处

首先放出我的最后成绩（在服务器环境测试）

Results for mm malloc:						
trace	name	valid	util	ops	secs	Kops
1	amptjp-bal.rep	yes	98%	5694	0.000368	15460
2	cccp-bal.rep	yes	98%	5848	0.000330	17700
3	cp-decl-bal.rep	yes	99%	6648	0.000352	18876
4	expr-bal.rep	yes	99%	5380	0.000265	20325
5	coalescing-bal.rep	yes	50%	14400	0.000410	35096
6	random-bal.rep	yes	94%	4800	0.000420	11420
7	random2-bal.rep	yes	93%	4800	0.000421	11399
8	binary-bal.rep	yes	88%	12000	0.002272	5281
9	binary2-bal.rep	yes	75%	24000	0.001789	13414
10	realloc-bal.rep	yes	99%	14401	0.000285	50565
11	realloc2-bal.rep	yes	87%	14401	0.000217	66487
Total			89%	112372	0.007130	15760

Score = (53 (util) + 40 (thru)) \* 11/11 (testcase) = 93/100

最终测试结果图

#### 1. CHUNKSIZE（单次扩展堆大小）

我开始时将 CHUNKSIZE 的值取为书上 demo 给出的  $1 \ll 10$ ，但在 binary-bal.rep 上效率很差。随手将其改为  $1 \ll 12$ ，一下子涨了二十分。分析 binary-bal.rep，其连续插入 4000 次，这意味着我们需要很大的空间。如果单次扩容小，那么扩容次数就会增多，效率拉低。提高单次扩容空间极大程度地减少了扩容次数，因此提高效率

```
Results for mm malloc:
trace      name      valid  util    ops      secs    Kops
1      amptjp-bal.rep    yes   98%    5694  0.000336 16951
2      cccp-bal.rep     yes   99%    5848  0.000319 18344
3      cp-decl-bal.rep   yes   99%    6648  0.000330 20139
4      expr-bal.rep     yes   99%    5380  0.000247 21826
5      coalescing-bal.rep yes   87%   14400  0.000408 35329
6      random-bal.rep    yes   94%    4800  0.000413 11625
7      random2-bal.rep   yes   93%    4800  0.000407 11797
8      binary-bal.rep    yes   70%   12000  0.008915 1346
9      binary2-bal.rep   yes   66%   24000  0.015756 1523
10     realloc-bal.rep   yes  100%   14401  0.000305 47201
11     realloc2-bal.rep  yes   96%   14401  0.000214 67232
Total                                91%  112372  0.027649 4064

Score = (55 (util) + 16 (thru)) * 11/11 (testcase) = 71/100
CHUNKSIZE=(1<<10)
```

## 2. 块分割阈值

在 `p1ace` 函数中，我们需要对是否分割当前块给出一个阈值。

```
1  if(rm_size >= ?1 * DSIZE) { // 剩余空闲块足够大
2      if(asm_size > ?2 * DSIZE) { // 分割空闲块，注意这里
3          ...
4      }
5  }
```

我开始将 `?1` 和 `?2` 都填为 `2*DSIZE`，结果如图

```
Results for mm malloc:
trace      name      valid  util    ops      secs    Kops
1      amptjp-bal.rep    yes   98%    5694  0.000361 15755
2      cccp-bal.rep     yes   98%    5848  0.000323 18133
3      cp-decl-bal.rep   yes   99%    6648  0.000353 18817
4      expr-bal.rep     yes   99%    5380  0.000264 20402
5      coalescing-bal.rep yes   50%   14400  0.000430 33488
6      random-bal.rep    yes   94%    4800  0.000426 11278
7      random2-bal.rep   yes   93%    4800  0.000424 11323
8      binary-bal.rep    yes   55%   12000  0.010045 1195
9      binary2-bal.rep   yes   74%   24000  0.001530 15684
10     realloc-bal.rep   yes   99%   14401  0.000287 50248
11     realloc2-bal.rep  yes   87%   14401  0.000217 66364
Total                                86%  112372  0.014659 7666

Score = (51 (util) + 31 (thru)) * 11/11 (testcase) = 82/100
块大小>=8*DSIZE时分割
```

改为 `8*DSIZE` 后，又变成了上面的最终结果。其实 `DSIZE` 前的系数是 `7,8,9,etc.` 都无所谓，但不能太小，否则导致块数多，①小块对应的 `Segregated List` 元素过多；②需要插入的次数增多。但也不能太大，否则基本不会发生块分割，这步操作也没有意义了。

写这个lab好爽