MALLOC LAB 实验报告

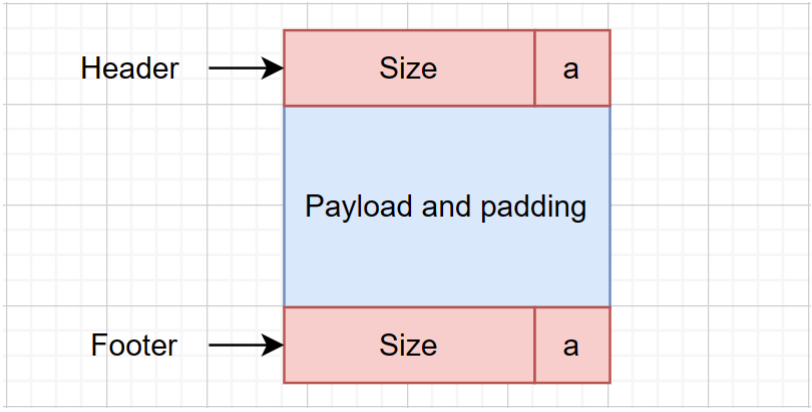
实现思路

V1 Implicit free list 隐式空闲链表

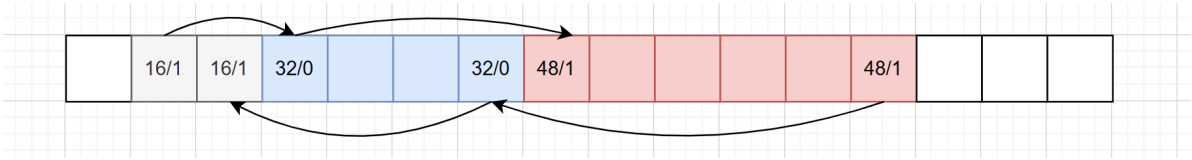
这是csapp课本上介绍最为详细的一种方法，也是最简单的一种实现方法。

数据结构

使用header+payload|padding+footer，其中header和footer各占8字节，由于Size满足16字节对齐，因此低4位均为0，可以用最低位置a表示此区域是否为空



通过每个块的起始位置和对应的Size大小，可以找到相邻的块儿，这样就构成了一条双向链表，可以在线性正比于块总数的时间内查找



起始被染成灰色的块为**序言块**，其含义可以类比于链表数据结构中的头指针，这一设计可以在之后的合并块拆分块时省去对边界情况的考虑，大大降低了讨论的难度。

重点代码解读

```
#define WSIZE 4 /* Word and header/footer size (bytes) */
#define DSIZE 8 /* Double word size (bytes) */
#define ALIGNMENT 16 /* 16 bytes alignment */
#define MINBLOCKSIZE 16 /* Min size of block(bytes) */
#define CHUNKSIZE (1 << 12) /* Extend heap by this amount (bytes) */
```

大部分函数都可以完全照搬csapp中的实现方法，其中`find_fit`和`place`两个自定义函数实现思路如下：

- `static void* find_fit(size_t size)`：采用首次适配搜索方法，从头线性扫描链表，找到首个未分配且容量为空的块。
- `static void place(void* bp, size_t asize)`：从给定的未分配块中“挖去”`asize`大小的一部分分配，剩余部分如果满足大于最小块大小（16 bytes），则将其分割成新的块。

测试结果及分析

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   98%    5694  0.021321    267
 2       cccp-bal.rep       yes   99%    5848  0.014530    402
 3    cp-decl-bal.rep       yes   99%    6648  0.032795    203
 4       expr-bal.rep       yes   99%    5380  0.025134    214
 5 coalescing-bal.rep       yes   66%   14400  0.000162  88670
 6     random-bal.rep       yes   92%    4800  0.012242    392
 7    random2-bal.rep       yes   91%    4800  0.011406    421
 8     binary-bal.rep       yes   54%   12000  0.306643     39
 9    binary2-bal.rep       yes   47%   24000  0.523258     46
10    realloc-bal.rep       yes   27%   14401  0.149848     96
11   realloc2-bal.rep       yes   34%   14401  0.003991   3608
Total                            73%  112372  1.101331    102

Score = (44 (util) + 0 (thru)) * 11/11 (testcase) = 44/100
```

整体表现平平，在不涉及重分配的情况下空间利用率尚可，但吞吐率偏低。对此结果的分析如下：

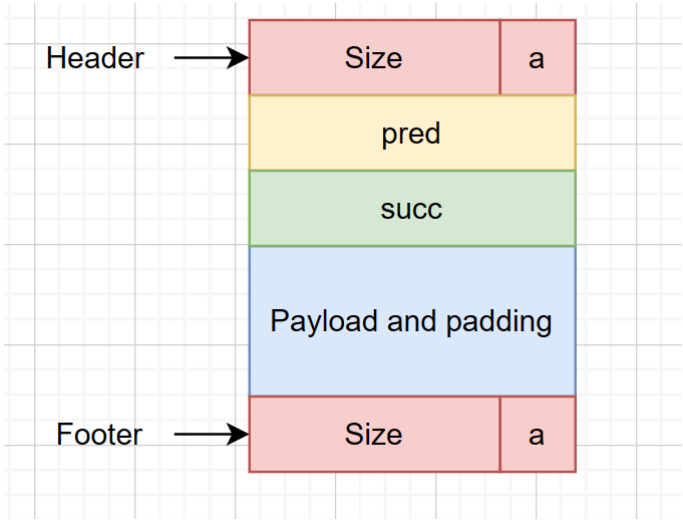
- 吞吐率偏低：隐式空闲链表上存有所有已分配和未分配的块，这样做的好处是数据结构简单内部碎片化少，但也会导致每次搜索时时间复杂度正比于块的个数，在块总数较多的情况下会显著增加时间开销，因而导致吞吐率降低。
- 涉及重分配操作的部分表现不佳：V1版本实现时采取的是最简单的realloc逻辑，即不管如何都删去原有的块并为之分配新的块，这样做也会增加不必要的时间开销和外部碎皮化的风险，导致吞吐率和空间复杂度双双下降。
- 某些特殊测例导致：一些特殊测例制造的极端情况，下文提到时会进一步具体说明。

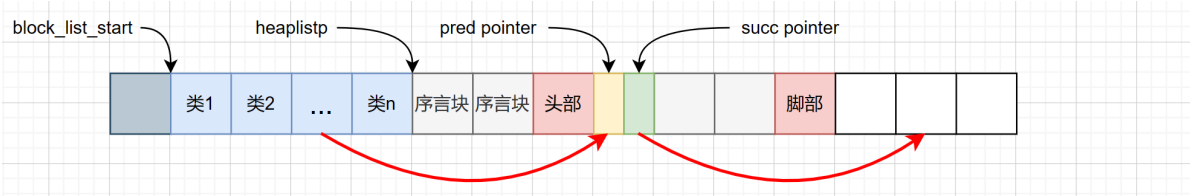
基于以上分析，接下来在隐式空闲链表的基础上进一步改造优化

V2 Segregated Free Lists 分离适配

数据结构

这种数据结构的思想是在原有隐式空闲链表的基础上，在块之间再添加多路链表，将大小接近空闲块链接起来，从而实现更快查找。具体实现方法就是在每个块中，添加pred和succ指针，指向其对应类的前驱和后继。同时在起始地址处维护一个指针数组，用于指向各个类的第一个块。





按理说n应越大越好，但考虑到n过大可能会占用过多内存导致空间利用率低，经过实验，将n选取在21效果最好，即将所有空闲块分成$\{32\},\{33\sim64\}\cdots\{2049\sim4096\}\cdots\{2^{24}\sim\infty\}$共21类（由于加入了两指针，MINBLOCKSIZE为32，无需考虑小于32的空闲块）

重点代码解读

新增加`NEXT_BLKP`和`PREV_BLKP`宏，实现前驱后继的查找

```
#define NEXT_BLKP(bp) ((char*)(bp) + GET_SIZE(((char*)(bp)-WSIZE)))
#define PREV_BLKP(bp) ((char*)(bp)-GET_SIZE(((char*)(bp)-DSIZE)))
```

`insert`和`delete`函数，实现对空块的插入和删除，其操作和书中`coalesce`函数相似，本质上都是在链表中插入或者删除结点，对于删除操作，需要对于其前后的空块是否存在，分四种情况具体讨论。

```
/*
 * insert - insert block to the head
 */
static void insert(void* bp)
{

    /* block size */
    size_t size = GET_SIZE(HDRP(bp));
    /* search the id */
    int num = search(size);
    /* if list is blank, put the block directly */
    if (GET_HEAD(num) == NULL) {
        PUT(block_list_start + WSIZE * num, bp); /* head */
        PUT(bp, NULL); /* pred */
        PUT((size_t*)bp + 1, NULL); /* succ */
    } else { /* if list is not blank, put it after the head */
        PUT((size_t*)bp + 1, GET_HEAD(num));
        PUT(GET_HEAD(num), bp);
        PUT(bp, NULL);
        PUT(block_list_start + WSIZE * num, bp);
    }
}
/*
 * delete - delete the block
 */
static void delete(void* bp)
{

    size_t size = GET_SIZE(HDRP(bp));
    int num = search(size);
    if (GET_PRE(bp) == NULL && GET_SUC(bp) == NULL) { /* Case 1 only one block: succ and pred both null */
        PUT(block_list_start + WSIZE * num, NULL);
    } else if (GET_PRE(bp) != NULL && GET_SUC(bp) == NULL) { /* Case 2 tail block*/
```

```
        PUT(GET_PRE(bp) + 1, NULL);
    } else if (GET_SUC(bp) != NULL && GET_PRE(bp) == NULL) { /* Case 3 head block*/
        PUT(block_list_start + WSIZE * num, GET_SUC(bp));
        PUT(GET_SUC(bp), NULL);
    } else if (GET_SUC(bp) != NULL && GET_PRE(bp) != NULL) { /*Case 4 common block: both succ and pred exist*/
        PUT(GET_PRE(bp) + 1, GET_SUC(bp));
        PUT(GET_SUC(bp), GET_PRE(bp));
    }
}
```

当调用`coalesce`和`place`时，也对块进行了合并以及切分的操作，因此，也需要在相应位置调用`delete`和`insert`来维护分离链表，其时间开销也都是$O(1)$的。

基于分离适配方法，重新设计`find_fit`函数，每次现根据size大小找到其落在的区间，对该链表进行线性扫描，如果并没有找到合适的空快，则转向更大的区间。

```
/*
 * find_fit - begin from the smallest class acording to size, find the fit block
 */
static void* find_fit(size_t size)
{
    for (int num = search(size); num < CLASS_SIZE; num++) {
        /* scan one list, if no suitable block, jump to bigger list*/
        for (size_t* bp = GET_HEAD(num); bp; bp = GET_SUC(bp)) {
            if (GET_SIZE(HDRP(bp)) >= size) {
                return (void*)bp;
            }
        }
    }
    return NULL;
}
```

最后，在debug过程中课本基于32位机器所写的代码所制造的隐患渐渐显现，我采取的办法较为简单暴力，将所有原书中`unsigned int`改为`size_t`（定义为`unsigned long`），并将DSIZE和WSIZE扩大为两倍，这样做虽然可能制造额外的空间开销，但是可以较好复用原书代码，并且经过和其它同学对比，这种操作对结果的影响并不大。

测试结果及分析

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   97%    5694  0.000640   8900
 2       cccp-bal.rep       yes   96%    5848  0.000406  14407
 3    cp-decl-bal.rep       yes   97%    6648  0.000496  13390
 4       expr-bal.rep       yes   98%    5380  0.000390  13809
 5 coalescing-bal.rep       yes   96%   14400  0.000537  26836
 6     random-bal.rep       yes   93%    4800  0.000536   8954
 7    random2-bal.rep       yes   90%    4800  0.000534   8996
 8     binary-bal.rep       yes   54%   12000  0.000567  21153
 9    binary2-bal.rep       yes   47%   24000  0.001022  23490
10    realloc-bal.rep       yes   88%   14401  0.000465  30997
11   realloc2-bal.rep       yes   17%   14401  0.000425  33909
Total                             79%  112372  0.006016  18678

Score = (48 (util) + 40 (thru)) * 11/11 (testcase) = 88/100
```

分离适配方法将多个测例的Kops提升了好几个数量级，吞吐率终于可以看了。熬夜DEBUG至此，不禁令人精神为之一振！下面就要对具体的测例做进一步优化。

V3 针对测例优化调整

提高初块利用率

课本提供的`mm_inti`函数要求先为其分配一段固定大小为CHUNKSIZE(4KB)的空块，而细心观察会发现测例`coalescing-bal.rep`的特点是反复分配和释放一段内存，而内存经过16字节补齐后又刚好大于初块的大小，这就导致每次程序都会认为现有的空块无法满足要求，在尾部拓展要求大小的空间，这就导致初块始终无法被用上。基于此，我们对mm_inti做如下修改，减小初始分配块的大小，从而提高空间利用率。

```
if (extend_heap(2 * DSIZE / WSIZE) == NULL) /* adjust for coalescing-bal.rep
*/
    return -1;
```

realloc 优化

正如对V1版本测试结果所分析的那样，无脑free和malloc的方法会带来大量无必要的时间开销。事实上，当新的空间大于旧的空间且待转移的那部分块前后相邻部分又恰好空闲时，其实存在利用现有块儿就地分配更大的内存的可能，具体表现为**空闲块融合**和**尾部堆扩展**两种方法。

- 空闲块融合：重分配时，如果后方有空闲块可以进行融合，再看空间够不够，如果够了就不用释放再分配了，直接就地分配新内存块
- 尾部堆扩展：如果要重新分配的块是尾部块，则只需要扩展新旧块相差的大小，就可以直接将尾部块转化成新块。

```
void* mm_realloc(void* ptr, size_t size)
{
    /* if ptr = NULL ,just mm_malloc(size) */
    if (!ptr)
        return mm_malloc(size);
    /* if size = 0 ,just mm_free(ptr) */
    if (!size) {
        mm_free(ptr);
        return NULL;
```

```
    }
    size_t asize = ALIGN(size + DSIZE);
    size_t copysize = GET_SIZE(HDRP(ptr));
    char* newptr;
    /* if copysize = asize, don't need to move */
    if (copysize == asize)
        return ptr;
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(ptr)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
    char* next_bp = NEXT_BLKP(ptr);
    size_t next_size = GET_SIZE(HDRP(next_bp));

    if (prev_alloc && !next_alloc && (copysize + next_size >= asize)) { /*
next block is blank and total size big enough */
        delete (next_bp);
        allocate(ptr, copysize + next_size, 1);
        place(ptr, copysize + next_size);
    } else if (!next_size && asize >= copysize) { /* the block is at the
end, just extend it to asize */
        if ((long)(mem_sbrk(asize - copysize)) == -1)
            return NULL;
        allocate(ptr, asize, 1);
        allocate(NEXT_BLKP(ptr), 0, 1);
        place(ptr, asize);
    } else { /* final way, mm_malloc */
        newptr = mm_malloc(asize);
        if (newptr == NULL)
            return NULL;
        memmove(newptr, ptr, MIN(copysize, size));
        mm_free(ptr);
        return newptr;
    }
    return ptr;
}
```

place 优化

在realloc优化的基础上，我们不由得进一步思考，能否在分配时尽量多的让较大的块放在后面，从而使其重分配时能够更有可能触发realloc优化机制，从而提高吞吐率。此外，在分配块时，能否通过规划好不同大小块的位置，从而减小内部碎片。这两个问题都指向了对place函数的优化。

我们规定，如果当前适配的块大小比需要分配的大很多（大于MINBLOCKSIZE）时，我们将空间较大的块分配在空闲块的后部，较小的块分配在前部，并为之设定一个阈值SMALLSIZE（最后选为128 bytes）这样的组织方式有两方面好处，一是对块的大小进行了粗分类，有利于之后合并，另外针对`binary2-bal.rep`，这样做大大提高了尾部堆扩展的触发次数，从而进一步提高吞吐率。

```
static void* place(void* bp, size_t asize)
{
    size_t size = GET_SIZE(HDRP(bp));
    size_t rsize = size - asize;
    if (!GET_ALLOC(HDRP(bp))) /* for realloc, if bp is not blank, don't
delete it */
        delete (bp);
    if (rsize >= MINBLOCKSIZE) { // split block
```

```
        if (asize >= SMALLSIZE) { /* if asize >= SMALLSIZE, then allocat it at the back of the block*/
            allocate(bp, rsize, 0);
            allocate(NEXT_BLKP(bp), asize, 1);
            return NEXT_BLKP(bp);
        } else { /* if asize < SMALLSIZE, then allocat it at the front of the block */
            allocate(bp, asize, 1);
            allocate(NEXT_BLKP(bp), rsize, 0);
            return bp;
        }
    } else // do not split
    {
        allocate(bp, size, 1);
        return bp;
    }
}
```

经过上述优化后，最终得分

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   97%    5694  0.000396  14386
 2       cccp-bal.rep       yes   97%    5848  0.000424  13786
 3    cp-decl-bal.rep       yes   97%    6648  0.000509  13056
 4       expr-bal.rep       yes   98%    5380  0.000385  13974
 5 coalescing-bal.rep       yes   97%   14400  0.000549  26239
 6     random-bal.rep       yes   93%    4800  0.000513   9355
 7    random2-bal.rep       yes   90%    4800  0.000585   8207
 8     binary-bal.rep       yes   91%   12000  0.000557  21559
 9    binary2-bal.rep       yes   81%   24000  0.000993  24181
10    realloc-bal.rep       yes   99%   14401  0.000356  40418
11   realloc2-bal.rep       yes   69%   14401  0.000360  39958
Total                            92%  112372  0.005627  19971

Score = (55 (util) + 40 (thru)) * 11/11 (testcase) = 95/100
```

结论

从几次优化的测试结果来看，分离空闲链表对性能的提升最为明显，以极小的空间利用率代价取得了吞吐率的巨大提升，几乎让得分翻了一倍！不过这也并非意味着隐式空闲链表再无用武之地，课本中指出，当总体块个数比较确定且并不是很多的情况下，隐式空闲链表可以减小内部碎片化程度并提高空间利用效率。

此外，适当调整常数对于提升程序性能也可能有一定的帮助，如在实验过程中，发现CHUNKSIZE设为2^{13}性能会更好，但因为自身能力原因，还无法给出合理的解释。

总结

本次实验对我个人的编程能力也是一次极大的挑战，引用一篇博客的话，可以作为我完成作业时的内心写照。

> 这个实验写得人头皮发麻，欲罢不能。定义一堆宏，指针满天飞，写时一时爽，debug 火葬场。我整个周末几乎都在与 segmentation fault 打交道，随意一个小笔误就要找几个小时，唉。

经过这次实验的洗礼，我对指针和地址的理解都更深了一层，对于malloc的内部原理也有了进一步的理解，收获颇丰，感谢助教和老师们的辛苦付出，春节快乐！

参考资料

CSAPP-malloclab 解题思路记录 - 找一个吃麦旋风的理由 (zero4drift.github.io)

malloc lab的一些奇技淫巧 - 知乎 (zhihu.com)

与 Malloc Lab(in csapp) 大战三天三夜纪实 - 知乎 (zhihu.com)

CSAPP-Lab08 Malloc Lab 深入解析 - 知乎 (zhihu.com)

CSAPP:Lab5-Malloc Lab - 知乎 (zhihu.com)

高性能 Malloc Lab —— 不上树 97/100 - 知乎 (zhihu.com)