MallocLab 实验报告

涂奕腾 2020201018

1. 隐式空闲链表

首先参(fu)照(zhi)书上的代码，实现最基本的隐式空闲链表算法：(由于和书上代码差不多，不再赘述，完整代码可见 mm1-1.c)

```
static void *coalesce(void *bp);
static void *extend_heap(size_t words);

static void *find_fit(size_t asize){
    for(char *bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp))
        if(!GET_ALLOC(HDRP(bp)) && GET_SIZE(HDRP(bp)) >= asize)
            return bp;
    return NULL;
}

static void place(void *bp, size_t asize){
    size_t csize = GET_SIZE(HDRP(bp));
    if(csize - asize >= 2 * DSIZE){
        PUT(HDRP(bp),PACK(asize, 1));
        PUT(FTRP(bp),PACK(asize, 1));
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp),PACK(csize - asize, 0));
        PUT(FTRP(bp),PACK(csize - asize, 0));
    }
    else{
        PUT(HDRP(bp),PACK(csize, 1));
        PUT(FTRP(bp),PACK(csize, 1));
    }
}
```

结果并不理想……

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes    99%    5694  0.014391   396
 1       yes    99%    5848  0.013326   439
 2       yes    99%    6648  0.022383   297
 3       yes   100%    5380  0.016452   327
 4       yes    66%   14400  0.000158 91139
 5       yes    92%    4800  0.013731   350
 6       yes    92%    4800  0.012953   371
 7       yes    55%   12000  0.166015    72
 8       yes    51%   24000  0.561332    43
 9       yes    27%   14401  0.104022   138
10       yes    34%   14401  0.003688  3905
Total           74%  112372  0.928449   121

Perf index = 44 (util) + 8 (thru) = 52/100
```

尝试改用 next fit 策略，尝试从上一次查询结束的地方开始查找，需要额外定义一个 pre_listp 指针指向上一次查询结束的地方，初始化为 heap_listp，在每次查询(find_fit)/合并(coalesce)空闲区间时进行修改(完整代码见 mm1-2.c)：

```
static void *coalesce(void *bp){
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
```

```c
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    if(prev_alloc && next_alloc) return (pre_listp = bp);
    else if(prev_alloc && !next_alloc){
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        pre_listp = bp;
    }
    else if(!prev_alloc && next_alloc){
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        pre_listp = bp = PREV_BLKP(bp);
    }
    else{
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        pre_listp = bp = PREV_BLKP(bp);
    }
    return bp;
}

static void *find_fit(size_t asize){
    for(char *bp = pre_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp))
        if(!GET_ALLOC(HDRP(bp)) && GET_SIZE(HDRP(bp)) >= asize){
            pre_listp = bp;
            return bp;
        }
    for(char *bp = heap_listp; bp != pre_listp; bp = NEXT_BLKP(bp))
        if(!GET_ALLOC(HDRP(bp)) && GET_SIZE(HDRP(bp)) >= asize){
            pre_listp = bp;
            return bp;
        }
    return NULL;
}
```

可见 next fit 减少了链表前部小碎片的产生，提高了不少的效率，但空间利用率方面还是明显存有不足。

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes    90%    5694  0.002973  1915
 1       yes    93%    5848  0.001836  3185
 2       yes    94%    6648  0.005705  1165
 3       yes    96%    5380  0.005910   910
 4       yes    66%   14400  0.000160 90282
 5       yes    89%    4800  0.007065   679
 6       yes    87%    4800  0.006625   725
 7       yes    55%   12000  0.015627   768
 8       yes    51%   24000  0.014196  1691
 9       yes    26%   14401  0.096259   150
10       yes    34%   14401  0.003674  3920
Total           71%  112372  0.160029   702

Perf index = 43 (util) + 40 (thru) = 83/100
```

2. 显式空闲链表

只需在之前的代码中做一些简单改动，这里只给出链表插入、删除操作部分的代码，(完整代码详见 mm2.c):

```
#define PREV_PTR(bp) (*(char **)(bp))
#define NEXT_PTR(bp) (*(char **)(bp + DSIZE))
#define SET_PREV(bp, val) (PREV_PTR(bp) = (val))
#define SET_NEXT(bp, val) (NEXT_PTR(bp) = (val))

static void erase(void *bp){
    if(bp == NULL || GET_ALLOC(HDRP(bp))) return;
    void *prev = PREV_PTR(bp);
    void *next = NEXT_PTR(bp);
    SET_PREV(bp, 0);
    SET_NEXT(bp, 0);
    if(prev == NULL && next ==NULL) list_head = NULL;
    else if(prev ==NULL) SET_PREV(next, 0), list_head = next;
    else if(next == NULL) SET_NEXT(prev, 0);
    else SET_NEXT(prev, next), SET_PREV(next, prev);
}

static void insert(void *bp){
    if(bp == NULL) return;
    if(list_head == NULL){
        list_head = bp;
        return;
    }
    SET_NEXT(bp, list_head);
    SET_PREV(list_head, bp);
    list_head = bp;
}
```

```
Results for mm malloc:
trace  valid  util     ops      secs   Kops
 0       yes   88%    5694  0.000357 15950
 1       yes   91%    5848  0.000233 25120
 2       yes   94%    6648  0.000482 13787
 3       yes   96%    5380  0.000353 15249
 4       yes   66%   14400  0.000203 70796
 5       yes   88%    4800  0.000723  6640
 6       yes   85%    4800  0.000807  5950
 7       yes   52%   12000  0.004287  2799
 8       yes   44%   24000  0.004491  5344
 9       yes   26%   14401  0.096798   149
10       yes   34%   14401  0.003740  3850
Total          70%  112372  0.112474   999

Perf index = 42 (util) + 40 (thru) = 82/100
```

由于使用了双向链表，只需查询空闲块而非所有块，时间上有所进步，但在空间上没有太大优化，空间利用率依然不够理想，继续改进。

3. 分离适配

(1)分离适配策略

对于分离适配，我是按照块的大小范围[1, 1], [2, 2], [3, 4], [5, 8], [9, 16],...,[2049, 4096], [4097, ∞]划分其所属的不同的链表，每个链表中的块按照 size 的递增顺序放置，表头指向每个链表的尾部。注意在初始化时，应当为每个表头分配空间。

```
static char  **list_head;
#define GET_LIST(i) (*(list_head + i))
#define SET_LIST(i, bp)(GET_LIST(i) = bp)

static void erase(void *bp){
    if(bp == NULL || GET_ALLOC(HDRP(bp))) return;
    int i = 0;
    size_t size = GET_SIZE(HDRP(bp));
    for(; i < NUM_OF_LIST - 1 && size > 1; size >>= 1, ++i);
    void *pre = PREV_PTR(bp);
    void *nxt = NEXT_PTR(bp);
    if(pre == NULL && nxt == NULL) SET_LIST(i, NULL);
    else if(pre == NULL && nxt != NULL) SET_PREV(nxt, NULL);
    else if(pre != NULL && nxt == NULL) SET_NEXT(pre, NULL), SET_LIST(i, pre);
    else SET_NEXT(pre, nxt), SET_PREV(nxt, pre);
}

static void insert(void *bp, size_t size){
    if(bp == NULL) return;
    int i = 0;
    for(;i < NUM_OF_LIST - 1 && size > 1; size >>= 1, ++i);
    void *pre = GET_LIST(i) , *nxt = NULL;
    for(; pre != NULL && size > GET_SIZE(HDRP(pre)); nxt = pre, pre =  PREV_PTR(pre));
    if(pre == NULL && nxt == NULL){
        SET_PREV(bp, NULL);
        SET_NEXT(bp, NULL);
        SET_LIST(i, bp);
    }
    else if(pre == NULL && nxt != NULL){
        SET_PREV(bp, NULL);
        SET_NEXT(bp, nxt);
        SET_PREV(nxt, bp);
    }
    else if(pre != NULL && nxt == NULL){
        SET_NEXT(pre, bp);
        SET_PREV(bp, pre);
        SET_NEXT(bp, NULL);
        SET_LIST(i, bp);
    }
    else{
        SET_NEXT(pre, bp);
        SET_PREV(bp, pre);
        SET_PREV(nxt, bp);
        SET_NEXT(bp, nxt);
```

```
    }
}

int mm_init(void){
    if((list_head = mem_sbrk(NUM_OF_LIST * sizeof(char *))) == (void *)-1) return -1;
    for(int i = 0; i < NUM_OF_LIST; ++i) SET_LIST(i, NULL);
    if((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1) return -1;
    PUT(heap_listp, 0);
    PUT(heap_listp + 1 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 2 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 3 * WSIZE, PACK(0, 1));
    heap_listp += DSIZE;
    if(extend_heap(CHUNKSIZE / WSIZE) == NULL) return -1;
    return 0;
}
```

(2)修改 realloc

主要根据以下两个策略重写 realloc 函数：

空闲块融合：在重分配时候，如果后方有空闲块可以进行融合，再看空间是否充足，如果足够就不用释放再分配

尾部堆扩展：如果重分配的块是尾部块执行 extend_heap 即可，不需要释放再分配

```
void *mm_realloc(void *ptr, size_t size){
    if(ptr == NULL)  return mm_malloc(size);
    else if(size == 0){
        mm_free(ptr);
        return NULL;
    }
    size_t asize, cur_size = GET_SIZE(HDRP(ptr));
    asize= DSIZE * ((size + DSIZE - 1) / DSIZE + 3 );
    char *oldptr = ptr, *newptr;
    if(cur_size == asize) return ptr;

    char *next = NEXT_BLKP(ptr);
    size_t next_alloc =  GET_ALLOC(HDRP(next));
    size_t next_size = GET_SIZE(HDRP(next));
    size_t total_size = cur_size;

    if(!next_alloc && (cur_size + next_size >= asize)){
        total_size += next_size;
        erase(next);
        PUT(HDRP(ptr), PACK(total_size, 1));
        PUT(FTRP(ptr), PACK(total_size, 1));
        place(ptr, total_size);
    }
    else if(!next_size && asize >= cur_size){
        size_t extend_size = asize - cur_size;
        if((long)(mem_sbrk(extend_size)) == -1) return NULL;
        PUT(HDRP(ptr), PACK(total_size + extend_size, 1));
        PUT(FTRP(ptr), PACK(total_size + extend_size, 1));
        PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 1));
        place(ptr, asize);
```

```
    }
    else{
        newptr = mm_malloc(asize);
        if(newptr == NULL) return NULL;
        memcpy(newptr, ptr, MIN(cur_size, size));
        mm_free(ptr);
        return newptr;
    }
    return ptr;
}
```

(3)放置策略：根据数据的实际情况优化放置(place)时的策略：如果需求的空间小则考前放置，反之则靠后放置，这里的阈值在 100 左右都差不多

```
static void* place(void *bp, size_t asize){
    size_t csize = GET_SIZE(HDRP(bp)), tmp = csize - asize;
    erase(bp);
    if(tmp <= MINBLOCKSIZE){
        PUT(HDRP(bp),PACK(csize, 1));
        PUT(FTRP(bp),PACK(csize, 1));
    }
    else if(asize >= 112){
        PUT(HDRP(bp),PACK(tmp, 0));
        PUT(FTRP(bp),PACK(tmp, 0));
        insert(bp, tmp);
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp),PACK(asize, 1));
        PUT(FTRP(bp),PACK(asize, 1));
    }
    else{
        PUT(HDRP(bp),PACK(asize, 1));
        PUT(FTRP(bp),PACK(asize, 1));
        void *nxt = NEXT_BLKP(bp);
        PUT(HDRP(nxt),PACK(tmp, 0));
        PUT(FTRP(nxt),PACK(tmp, 0));
        insert(nxt, tmp);
    }
    return bp;
}
```

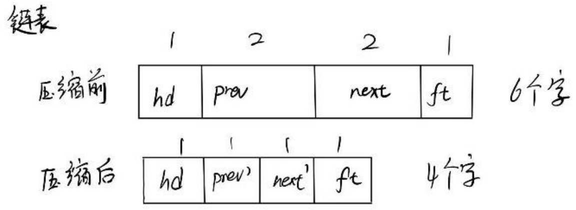
基于以上三个策略，可以进一步优化(完整代码见 mm3-1.c)：

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000488 11668
 1       yes   99%    5848  0.000503 11622
 2       yes   99%    6648  0.000566 11756
 3       yes   99%    5380  0.000453 11887
 4       yes   66%   14400  0.000881 16339
 5       yes   93%    4800  0.000669  7179
 6       yes   91%    4800  0.000672  7148
 7       yes   90%   12000  0.000758 15835
 8       yes   73%   24000  0.001926 12460
 9       yes   99%   14401  0.000463 31104
10       yes   87%   14401  0.000476 30235
Total          90%  112372  0.007854 14308

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

(4)压缩存储地址

对于平衡树来说，一般至少要维护父亲节点、两个孩子节点一共三个指针，所以如过继续直接将指针存到两个字中，势必会造成很大的浪费，于是考虑对指针进行压缩存储，先在分离适配的方法上进行实验。



由于栈的大小有限，所以指针地址中真正有用的信息完全可以在一个字 32 位中存下来(4GB)，所以我们将当前的指针减去 mem_heap_lo()可以得到一个 unsigned int 类型的整数，作为指针的压缩存储使用，特别地，NULL 用 0 表示：

```
#define int2ptr(x) (void*)((char *)mem_heap_lo() + (x))
#define ptr2int(ptr) (unsigned int)((char *)(ptr) - (char *)mem_heap_lo())

inline void* PREV_PTR(void *bp){
    return *(unsigned int*)(bp) == 0 ? NULL : (void*)((char *)mem_heap_lo() + *(unsigned int*)(bp));
}

inline void* NEXT_PTR(void *bp){
    return *((unsigned int*)(bp) + 1) == 0 ? NULL : (void*)((char *)mem_heap_lo() + *((unsigned int*)(bp) + 1));
}

inline void SET_PREV(void *bp, void *ptr){
    *(unsigned int *)(bp) = ptr == NULL ? (unsigned int) 0 :  ptr2int(ptr);
}

inline void SET_NEXT(void *bp, void* ptr){
    *((unsigned int*)(bp) + 1) = ptr == NULL ?  (unsigned int) 0 : ptr2int(ptr);
}

inline void* GET_LIST(int i){
    return *(unsigned int *)(list_head + i) == 0 ?  NULL : (int2ptr(*(list_head + i)));
}

inline void SET_LIST(int i, void *bp){
    *(unsigned int *)(list_head + i) = bp == NULL ?  (unsigned int) 0 : ptr2int(bp);
}
```

另外，还修改了之前代码中一些不必要的 insert 和 erase 操作，所以运行时间相比 3.中还有所下降。(完整代码见 mm3-2.c)

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000516 11039
 1       yes   97%    5848  0.000543 10768
 2       yes   98%    6648  0.000634 10489
 3       yes   99%    5380  0.000469 11466
 4       yes   79%   14400  0.000650 22154
 5       yes   93%    4800  0.000696  6894
 6       yes   92%    4800  0.000683  7030
 7       yes   81%   12000  0.000718 16706
 8       yes   88%   24000  0.001323 18135
 9       yes  100%   14401  0.000421 34199
10       yes   93%   14401  0.000426 33829
Total          93%  112372  0.007079 15873

Perf index = 56 (util) + 40 (thru) = 96/100
```

4. Splay 平衡树

在平衡树的部分需要实现插入(insert)，删除(erase)，查找(find fit)三个操作，其中查找操作应该是寻找 lower_bound。

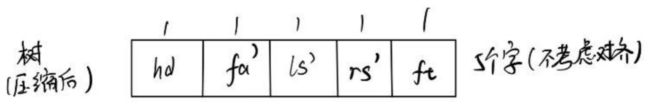
在实现前，我考虑了以下几种平衡树作为选择:

(1)无旋 treap：fhq-treap 的删除是按照关键字大小 split，由于我水平有限，如果有多个相同的值时不知道改如何进行 split，于是放弃

(2)有旋 treap：常数小，实现简单，本来是最优选择，但是在已经基本实现后发现了 bug：有旋 treap 并不维护父亲节点信息，常规的按照关键字进行删除是从根递归到节点，但由于在该实验中删除时直接传入节点，不知道其父节点的指针(无法修改父节点的孩子信息)，如果从根按照关键字大小递归下来的话，有多个相同关键字的节点时难以判断目标节点的具体位置，所以有旋 treap 也不行。

(3)红黑树：太难写，留做备选。

(4)Splay：虽然常数略大(实际上在测试中和红黑树的耗时差不多)，但比较好写，特别是在删除操作中，传入目标节点时可以直接将其 Splay 到根再进行删除，非常方便。



下面给出了平衡树部分的代码，除了上述的三个操作外还包括了 rotate 和 Splay 函数：(完整代码见 mm4-1.c)

```
Static int getpos(void *bp){//判断左儿子还是右儿子
    return (char *)(bp) == (char *)GET_RS(GET_FA(bp));
}

static void rotate(unsigned int *x){
    unsigned int *y = GET_FA(x);
    unsigned int *z = GET_FA(y);
    int chk = getpos(x);
    if(chk){
        unsigned int *tmp = GET_LS(x);
        SET_RS(y, tmp);
        if(tmp != NULL) SET_FA(tmp, y);
        SET_LS(x, y);
```

```c
    }
    else{
        unsigned int *tmp = GET_RS(x);
        SET_LS(y, tmp);
        if(tmp != NULL) SET_FA(tmp, y);
        SET_RS(x, y);
    }
    SET_FA(y, x);
    SET_FA(x, z);
    if(z != NULL){
        if((unsigned int *)y == (unsigned int *)GET_RS(z)) SET_RS(z, x);
        else SET_LS(z, x);
    }
}

static void Splay(unsigned int *x){
    for(unsigned int *f; (f = GET_FA(x)) != NULL; rotate(x)){
        if(GET_FA(f) != NULL)
            rotate(getpos(f) == getpos(x) ? f : x);
    }
    rt = x;
}

static void erase(void *bp){
    if(bp == NULL || GET_ALLOC(HDRP(bp))) return;
    Splay(bp);
    if(GET_LS(rt) == NULL && GET_RS(rt) == NULL){
        rt = NULL;
    }
    else if(GET_LS(rt) == NULL){
        rt = GET_RS(rt);
        SET_FA(rt, NULL);
    }
    else if(GET_RS(rt) == NULL){
        rt = GET_LS(rt);
        SET_FA(rt, NULL);
    }
    else{
        unsigned int *x = rt, *y = GET_LS(x);
        while(GET_RS(y) != NULL) y = GET_RS(y);
        Splay(y);
        SET_FA(GET_RS(x), y);
        SET_RS(y, GET_RS(x));
    }
}

static void insert(void *bp, size_t asize){
    if(bp == NULL) return;
    SET_FA(bp, NULL);
    SET_LS(bp, NULL);
    SET_RS(bp ,NULL);
```

```c
    if(rt == NULL){
        rt = (unsigned int *)bp;
        return;
    }
    unsigned int *x = rt, *f = NULL;
    while(1){
        f = x;
        x = GET_SIZE(HDRP(x)) <= asize ? GET_RS(x) : GET_LS(x);
        if(x == NULL){
            x = (unsigned int *)bp;
            SET_FA(x, f);
            if(GET_SIZE(HDRP(f)) <= asize) SET_RS(f, x);
            else SET_LS(f, x);
            break;
        }
    }
    Splay(x);
}

static char* find_fit(size_t val){
    unsigned int* x = rt, *res =NULL;
    while(x != NULL){
        if(GET_SIZE(HDRP(x)) >= val) res = x, x = GET_LS(x);
        else x = GET_RS(x);
    }
    if(res != NULL) Splay(res);
    return (char *)res;
}
```

由结果可见，使用平衡树对空间利用率有一定的提升(并不明显),，但可能由于数据量较小，且平衡树相较于链表常数更大，所以在时间上比链表还是慢了不少。

```
Results for mm malloc:
trace  valid  util     ops      secs   Kops
 0       yes    99%    5694  0.001453  3918
 1       yes   100%    5848  0.001373  4261
 2       yes   100%    6648  0.001693  3927
 3       yes   100%    5380  0.001327  4054
 4       yes    80%   14400  0.000484 29734
 5       yes    95%    4800  0.005831   823
 6       yes    95%    4800  0.005869   818
 7       yes    81%   12000  0.004216  2846
 8       yes    88%   24000  0.004114  5834
 9       yes   100%   14401  0.000359 40103
10       yes    93%   14401  0.000370 38964
Total           94%  112372  0.027089  4148

Perf index = 56 (util) + 40 (thru) = 96/100
```

在之前内容的基础上进行最后的优化：

(1)去掉已分配块的尾标，用头部的第二位维护某个块的前一个块是否已分配：

```c
#define GET_PREV_ALLOC(p) (GET(HDRP(p)) & 0x2)
#define SET_PREV_ALLOC(p) (GET(HDRP(p)) |= 0x2)
#define RESET_PREV_ALLOC(p) (GET(HDRP(p)) &= ~0x2)
```

(2)单独维护大小为 8 字节和 16 字节的迷你块组成链表，分别用 list_head，list 表示，其中对于 8 字节的块需要单独维护：用头部的第三位维护某个块的前一个块是否为 8 字节的迷你块：

```c
#define GET_PREV_FREE(p) (GET(HDRP(p)) & 0x4)
#define SET_PREV_FREE(p) (GET(HDRP(p)) |= 0x4)
#define RESET_PREV_FREE(p) (GET(HDRP(p)) &= ~0x4)
```

基于以上优化，需要对插入、删除、合并等函数进行修改，完整代码见 mm4-2.c，即最终的 mm.c，结果如下：

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.001399  4070
 1       yes  100%    5848  0.001462  3999
 2       yes  100%    6648  0.001683  3949
 3       yes  100%    5380  0.001360  3956
 4       yes   94%   14400  0.000563 25568
 5       yes   95%    4800  0.006031   796
 6       yes   95%    4800  0.006005   799
 7       yes   81%   12000  0.005281  2272
 8       yes   88%   24000  0.005115  4692
 9       yes  100%   14401  0.000395 36421
10       yes   98%   14401  0.000406 35488
Total          95%  112372  0.029700  3784

Perf index = 57 (util) + 40 (thru) = 97/100
```