Malloclab 实验报告

一、宏定义：

```
#define WSIZE       4
#define DSIZE       8
#define INITCHUNKSIZE (1<<6)
#define CHUNKSIZE (1<<12)
#define MAX(x, y) ((x) > (y) ? (x) : (y))  #最大值
#define PACK(size, alloc) ((size) | (alloc))
#define GET(p)                  (*(unsigned int *)(p))  #读
#define PUT(p, val)             (*(unsigned int *)(p) = (val))  #写
#define GET_SIZE(p)  (GET(p) & ~0x7)    #读块的大小
#define GET_ALLOC(p) (GET(p) & 0x1)    #读块的使用情况
#define HDRP(bp) ((char *)(bp) - WSIZE)  #首部
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)  #脚步
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE((char *)(bp) - WSIZE))
#下一块
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE((char *)(bp) - DSIZE))
#上一块
#define PRED(bp) (*(char **)(bp))  #前驱结点
#define SUCC(bp) (*(char **)((char *)(bp) + WSIZE))  #后继节点
#define HSIZE 16
static void* heads[HSIZE];  #链表头结点数组
```

二、基本思路：

采用分离式链表，按 size 的二进制的位数进行分组，一共 16 组，size 二进制位数为 i 的块分在第 i-1 组。每一组里面是一个双线链表。同一组内按 size 从小到大的顺序排列。链表中插入新元素时也是在不破坏这个顺序的前提下进行插入。在从链表中找合适的空闲块时按组号从小到大、size 从小到大的顺序查找，找到的第一个作为结果。

三、函数及其思路：

工具函数

一、static void *extend_heap(size_t words)

```
static void *extend_heap(size_t words)
{
    void *bp;
    words = ALIGN(words);
    if ((bp = mem_sbrk(words)) == (void *)-1) return NULL;
    PUT(HDRP(bp), PACK(words, 0));
    PUT(FTRP(bp), PACK(words, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));
    return coalesce(bp);
}
```

这是用一个新的空闲块扩展堆的函数，和书上的没什么变化。在创建完新的空闲块后考虑到这个空闲块的前一块有可能也是空闲块，所以要用 coalesce 函数合并空闲块。

二、static void *coalesce(void *bp)

```
static void *coalesce(void *bp)
{
    size_t prev_alloc=GET_ALLOC(HDRP(PREV_BLKP(bp)));
    size_t next_alloc=GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size=GET_SIZE(HDRP(bp));
    if(prev_alloc&&next_alloc){
        insert(bp);
        return bp;
    }else if(!prev_alloc&&next_alloc){
        delete(PREV_BLKP(bp));
        size+=GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp),PACK(size,0));
        PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
        bp=PREV_BLKP(bp);
    }else if(prev_alloc&&!next_alloc){
        delete(NEXT_BLKP(bp));
        size+=GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp),PACK(size,0));
        PUT(FTRP(bp),PACK(size,0));
    }else{
        delete(PREV_BLKP(bp));
        delete(NEXT_BLKP(bp));
        size+=GET_SIZE(HDRP(PREV_BLKP(bp)))+GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));
        PUT(FTRP(NEXT_BLKP(bp)),PACK(size,0));
        bp=PREV_BLKP(bp);
    }
    insert(bp);
    return bp;
}
```

这是合并空闲块的函数，考虑四种情况：

1、前后都是已分配的块

直接将块插入到链表里，然后直接返回原块。

2、前面一块是空闲块，后面一块已分配

将前一块从链表里删除，将两块合并，然后将新的块插入到链表中，返回新生成的块。

3、后面一块是空闲块，前面一块已分配

将后一块从链表里删除，将两块合并，然后将新的块插入到链表中，返回新生成的块。

4、前后都是空闲块

将前后两块从链表里删除，将三块合并，然后将新的块插入到链表中，返回新生成的块。

三、static int findhead(size_t size)

```
static int findhead(size_t size){
    int i=0;
    for(;(i<HSIZE-1)&&(size>1);i++){
        size>>=1;
    }
    return i;
}
```

按 size 的大小找寻对应组的函数，简单的循环加移位操作。

四、static void delete(void *bp)

```
static void delete(void *bp)
{
    if(PRED(bp)==NULL){
        int head=findhead(GET_SIZE(HDRP(bp)));
        if(SUCC(bp)==NULL)
            heads[head]=NULL;
        else{
            PRED(SUCC(bp))= NULL;
            heads[head]=SUCC(bp);
        }
    }
    else if(SUCC(bp)==NULL)
            SUCC(PRED(bp))= NULL;
    else{
        SUCC(PRED(bp))=SUCC(bp);
        PRED(SUCC(bp))=PRED(bp);
    }
}
```

从链表中删除特定元素的函数，有以下四种情况：

1、前驱后继都为空

则该组中只有这一个元素，将该组链表设为空链表即可。

2、前驱为空后继不为空

说明该元素是该组链表的头结点，需要将该元素的后继的前驱设为空，然后将该组链表的头结点设为该 0 元 0 素的后继。

3、前驱不为空，后继为空

说明该元素是该组链表的最后一个元素，只需要将该元素的前驱的后继设为空就可以了。

4、前驱后继均不为空

此时需要将该元素的前驱的后继设为该元素的后继，将该元素后继的前驱设为该元素的前驱。

五、static void insert(void *bp)

```
static void insert(void *bp)
{
    size_t size=GET_SIZE(HDRP(bp));
    int head = findhead(size);
    void *temp = NULL;
    void* t = heads[head];
    if(t==NULL){
        SUCC(bp)=NULL;
        PRED(bp)=NULL;
        heads[head] = bp;
        return;
    }
    if(size<=GET_SIZE(HDRP(t))){
        SUCC(bp)=t;
        PRED(t)=bp;
        PRED(bp)=NULL;
        heads[head] = bp;
        return;
    }
    while (t != NULL){
        if(GET_SIZE(HDRP(t))>=size) break;
        temp = t;
        t = SUCC(t);
    }
    if (t != NULL){
        SUCC(bp)=t;
        PRED(t)=bp;
        PRED(bp)=temp;
        SUCC(temp)=bp;
    }else{
        SUCC(bp)=NULL;
        PRED(bp)=temp;
        SUCC(temp)=bp;
    }
}
```

在链表中插入新元素的函数，首先计算该元素应该被插入到哪个组里，然后判断是否出现以下情况：

1、链表为空

那么直接将该元素设为头结点，将其前驱后继均设为空即可。

2、链表的首结点的对应的块的大小大于等于新插入的块的大小。

将该元素的后继设为原头结点，将原头结点的前驱设为该元素，然后将该元素设为新的头结点。

如果没有出现这两种情况，就从头结点开始往后找第一个 size 不小于新插入的块的大小的元素，有两种情况：

1、没有满足条件的元素

将链表的尾元素的后继设为该元素，将该元素的前驱设为链表的尾元素。

2、找到了满足条件的元素 t，t 的前驱为 temp

将 temp 的后继设为该元素，将该元素的前驱设为 temp；将 t 的前驱设为该元素，将该元素的后继设为 t。

六、static void *place(void *bp, size_t size)

```
static void *place(void *bp, size_t size)
{
    size_t total=GET_SIZE(HDRP(bp));
    size_t left=total-size;
    delete(bp);
    if(left<2*DSIZE){
        PUT(HDRP(bp),PACK(total,1));
        PUT(FTRP(bp),PACK(total,1));
        return bp;
    }
    if(size>=96){
        PUT(HDRP(bp),PACK(left,0));
        PUT(FTRP(bp),PACK(left,0));
        PUT(HDRP(NEXT_BLKP(bp)),PACK(size,1));
        PUT(FTRP(NEXT_BLKP(bp)),PACK(size,1));
        insert(bp);
        return NEXT_BLKP(bp);
    }
    PUT(HDRP(bp),PACK(size,1));
    PUT(FTRP(bp),PACK(size,1));
    PUT(HDRP(NEXT_BLKP(bp)),PACK(left,0));
    PUT(FTRP(NEXT_BLKP(bp)),PACK(left,0));
    insert(NEXT_BLKP(bp));
    return bp;
}
```

将空闲块分割，提取出需要使用的部分，以此来提高空间利用率。

首先先判断空闲块的大小减去需要的部分后还剩多少，如果剩余的部分小于16bytes，就不进行分割，直接使用整个空闲块。

如果剩余的部分大于或等于 16bytes，这时候采用在网上学到的一种可以减少外部碎片的分割方法：如果需要的 size 较大，就将空闲块的最后大小为 size 的部分用来分配，否则就将空闲块前面的大小为 size 的部分用来分配，这样可以一定程度上避免两个大的已分配的块中间有一个较小的空闲块。

主函数

一、int mm_init(void)

```
int mm_init(void)
{
    for (int i = 0; i < HSIZE; i++)
        heads[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));
    if(extend_heap(INITCHUNKSIZE)==NULL) return -1;
    return 0;
}
```

初始化堆的函数，与书上的比加了一个链表的初始化。

二、void *mm_malloc(size_t size)

```
void *mm_malloc(size_t size)
{
    if(size==0) return NULL;
    size=ALIGN(size+DSIZE);
    int head=findhead(size);
    void* t=NULL;
    for(;head<HSIZE;head++){
        t=heads[head];
        while((t!=NULL)&&size>GET_SIZE(HDRP(t)))
            t=SUCC(t);
        if(t!=NULL) break;
    }
    if(t!=NULL){
        t=place(t,size);
        return t;
    }
    if((t=extend_heap(MAX(size,CHUNKSIZE)))==NULL)
       return NULL;
    t=place(t,size);
    return t;
}
```

分配块的函数，思路是从链表里面找满足条件的最小的空闲块，如果没有就利用 extend_heap 函数获取一个新的空闲块，然后将该空闲块进行分割；如果有直接将找到的空闲块进行分割就行。

三、void mm_free(void *bp)

```
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}
```

释放已分配的块的函数，与书上相同。

四、void *mm_realloc(void *bp, size_t size)

```
void *mm_realloc(void *bp, size_t size)
{
    void *new_bp=bp;
    int left;
    if (size == 0) return NULL;
    if (size <= DSIZE) size = 2 * DSIZE;
    else size = ALIGN(size + DSIZE);
    if ((left = GET_SIZE(HDRP(bp)) - size) == 0) return bp;
    else if ((left = GET_SIZE(HDRP(bp)) - size) == 0){
        bp=place(bp,size);
        return bp;
    }else if (!GET_ALLOC(HDRP(NEXT_BLKP(bp))) || !GET_SIZE(HDRP(NEXT_BLKP(bp)))){
        if ((left = GET_SIZE(HDRP(bp)) + GET_SIZE(HDRP(NEXT_BLKP(bp))) - size) < 0){
            if (extend_heap(MAX(-left, CHUNKSIZE)) == NULL)
                return NULL;
            left += MAX(-left, CHUNKSIZE);
        }
        delete(NEXT_BLKP(bp));
        PUT(HDRP(bp), PACK(size + left, 1));
        PUT(FTRP(bp), PACK(size + left, 1));
    }
    else{
        new_bp = mm_malloc(size);
        memcpy(new_bp, bp, GET_SIZE(HDRP(bp)));
        mm_free(bp);
    }
    return new_bp;
}
```

重新分配已分配好的块的函数。

主要思路是如果快的大小刚好够，就直接分配即可，如果块的大小比需要的大小大，就利用 place 函数分割一下。

而对于块大小不够的情况，一开始我的思路是看块前面和后面的块是否有空闲块，如果有的而且加起来大小够的话，就将空闲块合并之后进行分配。但我发现当块的前面是空闲块时，这样分配会出现“段错误”，我没有找到解决方法，于是只保留了只有后面是空闲块的情况，这种情况得分只有八十几分，最后两个测试分数很低。然后我在网上看到了这个考虑块在堆末尾的情况的方法，当块在堆末尾时，我们可以使用 extend_heap 函数新创建一个我们需要大小的块直接接在后面，这样就大大提高了空间利用率。

如果块大小不够，块不在堆末尾，后面又没有空闲块时就只能利用 memcpy 函数进行重分配了。

四、运行结果：

```
zjy@ubuntu:~/桌面/malloclab-handout$ make
gcc -Wall -O2 -m32   -c -o mm.o mm.c
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
zjy@ubuntu:~/桌面/malloclab-handout$ ./mdriver -v -t traces/
Team Name:Zhang Junyao
Member 1 :19307130226:19307130226@fudan.edu.cn
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000319 17855
 1       yes   99%    5848  0.000266 21985
 2       yes   99%    6648  0.000292 22806
 3       yes   99%    5380  0.000574  9379
 4       yes   99%   14400  0.000322 44762
 5       yes   95%    4800  0.000576  8330
 6       yes   95%    4800  0.000498  9639
 7       yes   95%   12000  0.000465 25795
 8       yes   88%   24000  0.002688  8929
 9       yes   99%   14401  0.000212 68090
10       yes   98%   14401  0.000163 88622
Total          97%  112372  0.006373 17632

Perf index = 58 (util) + 40 (thru) = 98/100
```