实验：Malloclab

周炎亮 2018202196 信息学院

一、实验目标

完成一个 C 版本的动态内存分配管理程序，要求支持 malloc, free 和 realloc 操作且正 确、尽可能的快速、高利用率。

二、实验思路

1、最原始版代码

最原始版代码参考课件中的代码并加以修改，旨在对本次实验有初步了解并能使程序正常运行。

根据书中提示，本次实验除了需要规定的mm_malloc、mm_init、mm_realloc和mm_free函数外，还需构造coalesce、extend_heap、find_fit和place函数来实现整个功能。

该版本代码使用**隐式链表**的思路。因此每个块都有一个头部块和尾部块，两个块中都储存这该块的大小，且最后一个的值代表是否为空闲块。该方法的一个优点是可以根据块的大小计算偏移地址从而得到位于前面和后面块的信息。同时，根据8字节对齐的要求可知，一个有效块的最小大小为16（4+8+4）字节。

因此首先需要引入以下宏定义：

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

#define MAX(x, y) ((x) > (y)? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc)  ((size) | (alloc))

/* Read and write a word at address p */
#define GET(p)       (*(unsigned int *)(p))            //读取p的值
#define PUT(p, val)  (*(unsigned int *)(p) = (val))    //对p赋值

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)  (GET(p) & ~0x7)                   //该块的大小
#define GET_ALLOC(p) (GET(p) & 0x1)                    //是否已分配

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)       ((char *)(bp) - WSIZE)                      //块头部的指针
#define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE) //块尾部的指针

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))    //后
面块的指针
#define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))    //前
面块的指针
```

又因为该版本的代码是使用隐式链表的方法，则根据下图，在一开始调用**mm_init**函数初始化时，至少要分配16个字节的空间，第一个字是用以对齐而不使用的块，第二个和第三个是整个malloc堆的开始块（序言块），分别代表头部块和尾部块，而最后一个则是象征着整个堆结尾的结尾块。

根据隐式链表的思路，每个块中表示大小的最后一位用以表示是否已被分配：1表示已分配，0表示未分配。

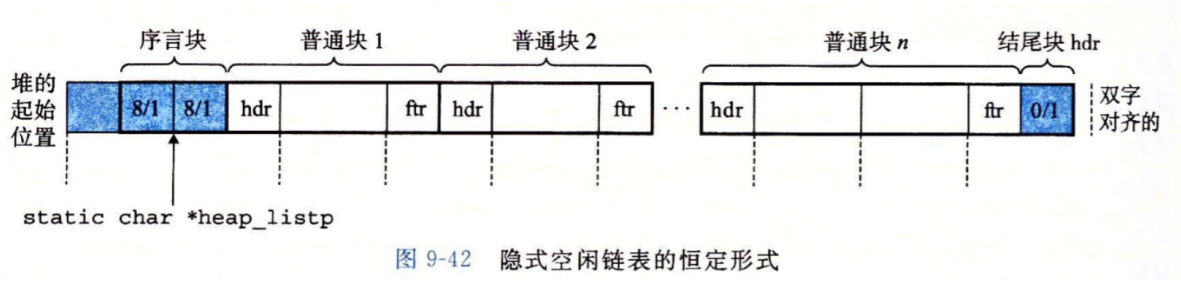


图 9-42　隐式空闲链表的恒定形式

而因为本实验中无法使用全局变量，故使用堆区的首地址（调用mem_heap_lo()）作为参照，以它的偏移值来获取需要的指针。

在对上面16个字节的块初始化之后。调用extend_heap对整个堆进行第一次内存空间的请求。为了提高峰值利用率，本次extend_heap只申请1024*4字节的空间。

```
int mm_init(void)
{
    void* heap_listp = mem_heap_lo();
    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0);                          /* 对齐块 */
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* 序言块的头部块 */
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* 序言块的尾部块 */
    PUT(heap_listp + (3*WSIZE), PACK(0, 1));     /* 结尾块 */

    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

对于请求内存空间的**mm_malloc**函数，只需先对读入的size值进行8字节对齐的操作，且若size<16，则直接将其对其为16。然后使用find_fit函数根据对齐后的size值在空闲的块中查找，如果查找到合适的块，则用place函数将其放入该块中；否则，说明空闲块中的块大小都小于size值，则需调用extend_heap函数申请新的空间，且该空间最小为1024*4字节。

```
void *mm_malloc(size_t size)
{
    size_t asize;
    size_t extendsize;
    char *bp;
    /* 错误处理 */
    if (size == 0)
        return NULL;
    /* 对齐操作 */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
    /* 寻找合适的块 */
    if ((bp = find_fit(asize)) != NULL) {
```

```
        place(bp, asize);
        return bp;
    }
    /* 未找到合适的块，重新申请空间 */
    extendsize = MAX(asize,CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

对于**mm_free**函数，只需将要释放的当前块的头部块和尾部块的最后一位置为0（即代表为空闲块），然后调用coalesce函数对前后可能存在的空闲块进行合并。

```
void mm_free(void *ptr)
{
    if (ptr == 0)
        return;

    size_t size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    coalesce(ptr);
}
```

对于**mm_realloc**函数，根据定义有：

1. 如果 ptr 参数为 NULL，函数等价于 mm_malloc(size)；
2. 如果 size 参数为 0，函数等价于 mm_free(ptr)；
3. 如果 ptr 参数不为空，则它必须是之前调用的 mm_malloc 或 mm_realloc 函数的返回值， 且不曾被释放过。函数需要更改分配过的内存块的大小，可以原地修改内存块（ptr 不变）， 或者重新寻找可用的地址（ptr 改变），这取决于同学们自己的实现。但注意，新内存块的内容应保持与原内存块相同。 如旧内存块 8 字节，新内存块 12 字节，则新内存块的前 8 个字 节应与旧内存块完全一致；如旧内存块 8 字节，新内存块 4 字节，则新内存块应与旧内存块 的前 4 字节完全一致。

在该版本代码中，对于第3条，直接重新寻找新的可用地址，并判断要改变的大小size是否比当前已有的大小oldsize大，若是，则将数据全部使用memcpy拷贝到新地址中，利用mm_free释放当前指针；否则，释放刚刚寻找到的可用地址的指针。

```
void *mm_realloc(void *ptr, size_t size)
{
    size_t oldsize;
    void *newptr;
    /* 情况1 */
    if(ptr == NULL) {
        return mm_malloc(size);
    }
    /* 情况2 */
    if(size == 0) {
        mm_free(ptr);
        return 0;
    }
    /* 情况3 */
    newptr = mm_malloc(size);
    if(!newptr) {
```

```
        return 0;
    }
    oldsize = GET_SIZE(HDRP(ptr));
    if(size < oldsize) oldsize = size;
    memcpy(newptr, ptr, oldsize);
    mm_free(ptr);
    return newptr;
}
```

在需要申请新的内存空间的**extend_heap**函数的操作中，首先对读入的内存大小进行8字节对齐的操作。然后调用已给定的mm_sbrk函数从堆区中申请新的空间，并将新的空闲块的头部块和尾部块初始化。又因为新块是从尾部插入，即原来的结尾块现在已是该新块的首地址，故也需重新初始化结尾块。且在该块之前可能也已有一个原来空闲块，故需调用coalesce函数判断并进行合并。

```
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;
    /* 对大小进行8字节对齐 */
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;
    /* 初始化新块 */
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    return coalesce(bp);
}
```

而对于合并空闲块操作的**coalesce**函数，在获取前后块是否已被分配的信息后，存在四种情况：1、前后两块都已被分配，可以直接返回。2、前面块被分配而后面块未被分配，则合并当前块和后面块。3、前面块未被分配而后面块已被分配，则合并前面块和当前块。4、前面块和后面块都未被分配，则合并这三块。

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(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) {             /* 1 前后两块都已被分配，可以直接返回。
*/
        return bp;
    }
    else if (prev_alloc && !next_alloc) {       /* 2 前面块被分配而后面块未被分配 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
    }
    else if (!prev_alloc && next_alloc) {       /* 3 前面块未被分配而后面块已被分配 */
        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 {                                          /* 4 前面块和后面块都未被分配，则合并这
三块。*/
        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));
        bp = PREV_BLKP(bp);
    }

    return bp;
}
```

在该版本代码中**find_fit**函数使用first fit的策略，即顺序遍历所有块，若该块为空且大小满足要求则直接返回当前地址。若最后找不到符合条件的块则返回NULL。

```
static void *find_fit(size_t asize)
{
    void *bp;
    void *heap_listp = mem_heap_lo();
    heap_listp += 2*WSIZE;
    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            return bp;
        }
    }
    return NULL; /* No fit */
}
```

对于**place**函数，只需比较该空闲块的大小和需要分配的大小。因为最小可用块的大小为16字节，因此若两者之差小于16就直接把整个空闲块都分配；否则分离后面多余的空闲块。

```
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));
    }
}
```

最终得分：56（44+12）。

缺点：使用fisrt fit的策略可能会将一个大的空闲块分割成很多小的块从而导致块的外部碎片过多。且因为顺序遍历，若满足要求的块在很后面，会使遍历需要花费很多时间。realloc操作简单粗暴，在效率低下的同时也会导致内存利用率低。

2、简单分离存储+分离适配+best fit+改进版realloc

为了解决之前描述的可能存在"若满足要求的块在很后面，会使遍历需要花费很多时间"的问题，可以将空闲块按一定大小分开存放，也即**简单分离存储**，这样在寻找合适的块时就没必要遍历比所需size还要小的块，可以提高搜索时间。但因为简单分离存储将一整个块全部分配，可能会导致过多的内部碎片，因此还需要**分离适配**将多余的块重新放入相应的位置中。

要实现分离存储的功能，首先根据块的大小，共将其分为10组，以2的幂为界：<16（因为观察数据集后发现最小malloc的是9，故不再单独设<8的组），17~32，……，2048~4096，>4096。然后可以使用链表将对应大小的块前后相连。因为空闲块中的内容对之后的malloc没有影响，因此可以将空闲块中除了头部块的第一个块作为存放链表中前一个块地址的块，而第二个块作为存放链表中后一个块地址的块。至于链表的第一个元素，因为本题无法使用全局变量，故可以在mm_init函数中寻求途径：可以在上述mm_init函数一开始分配16字节块的基础上，在这些块中多分配10*4字节的块，作为存放这是个链表首地址的块。

因为涉及到链表的操作，自然有增加元素和删除元素，因此还需定义两个新的函数**add_block**和**delete_block**，分别为在链表中添加新的块和在链表中删除块。在添加块时，根据**块的大小**从小大小定位块应插入的位置，这样find_fit函数使用first fit的策略也实现了**best fit**的功能。

此外，还需定义一个**find_asize_loc**函数来查找大小为size块所处的链表。

为了得到链表前一个和后一个元素的位置，需要另外使用两个宏定义：

```
#define PREV_LINKED_BLKP(bp) ((char *)(bp)) //同一链表中前一个的指针
#define NEXT_LINKED_BLKP(bp) ((char *)(bp)+WSIZE) //同一链表中后一个的指针
```

又因为新的空闲块中可能存留着无用信息，即可能会影响PREV_LINKED_BLKP(bp)和NEXT_LINKED_BLKP(bp)从而破坏链表的性质，故每次当有一个新的空闲块产生时，需要将该块前两个块中内容置为NULL。

根据思路，该版本的mm_init函数为：

```
int mm_init(void)
{
    void* heap_listp = mem_heap_lo();
    if ((heap_listp = mem_sbrk(14*WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0);
    PUT(heap_listp + (1 * WSIZE), NULL); //<=16
    PUT(heap_listp + (2 * WSIZE), NULL); //<=32
    PUT(heap_listp + (3 * WSIZE), NULL); //<=64
    PUT(heap_listp + (4 * WSIZE), NULL); //<=128
    PUT(heap_listp + (5 * WSIZE), NULL); //<=256
    PUT(heap_listp + (6 * WSIZE), NULL); //<=512
    PUT(heap_listp + (7 * WSIZE), NULL); //<=1024
    PUT(heap_listp + (8 * WSIZE), NULL); //<=2048
    PUT(heap_listp + (9 * WSIZE), NULL); //<=4096
    PUT(heap_listp + (10 * WSIZE), NULL); //>4096
    PUT(heap_listp + (11 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (12 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (13 * WSIZE), PACK(0,1));

    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

mm_malloc、mm_free和extend_heap的基本思路同上且改动不大，故不做展示。

对于coalesce函数，因为链表操作的加入，在四种情况之下还需分别进行额外操作：

1、前后两块都已被分配，则将当前空闲块使用add_block函数放入对应位置后返回。

2、前面块被分配而后面块未被分配，则用delete_block函数删除后面块，合并当前块和后面块后再使用add_block函数将新块放入对应链表。

3、前面块未被分配而后面块已被分配，则用delete_block函数删除前面块，合并当前块和前面块后再使用add_block函数将新块放入对应链表。

4、前面块和后面块都未被分配，则使用delete_block函数删除前后两块，合并这三块并使用add_block函数将新块放入对应链表。

```
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) {             /* 1、前后两块都已被分配 */
        add_block(bp, size);
        return bp;
    }
    else if (prev_alloc && !next_alloc) {       /* 2、前面块被分配而后面块未被分配 */
        delete_block(NEXT_BLKP(bp), GET_SIZE(HDRP(NEXT_BLKP(bp))));
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
        PUT(PREV_LINKED_BLKP(bp), NULL);
        PUT(NEXT_LINKED_BLKP(bp), NULL);
        add_block(bp, size);
    }
    else if (!prev_alloc && next_alloc) {       /* 3、前面块未被分配而后面块已被分配
*/
        delete_block(PREV_BLKP(bp), GET_SIZE(HDRP(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);
        PUT(PREV_LINKED_BLKP(bp), NULL);
        PUT(NEXT_LINKED_BLKP(bp), NULL);
        add_block(bp, size);
    }
    else {                                      /* 4、前面块和后面块都未被分配 */
        delete_block(NEXT_BLKP(bp), GET_SIZE(HDRP(NEXT_BLKP(bp))));
        delete_block(PREV_BLKP(bp), GET_SIZE(HDRP(PREV_BLKP(bp))));
        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));
        bp = PREV_BLKP(bp);
        PUT(PREV_LINKED_BLKP(bp), NULL);
        PUT(NEXT_LINKED_BLKP(bp), NULL);
        add_block(bp, size);
    }
    return bp;
}
```

对于find_fit函数，只需先使用find_asize_loc函数找出该大小的块在链表中的位置，然后顺序遍历该链表即可。但可能会出现该链表中没有空闲块而更大块的链表中有空闲块的情况，因此当在当前链表中找不到合适块的情况下，在更大的链表中寻找，直至遍历完之后所有的链表。

```
static void *find_fit(size_t asize)
{
    void* heap_listp = find_asize_loc(asize);
    for(;heap_listp<=mem_heap_lo() + 10*WSIZE;heap_listp+=WSIZE){
        void *st = GET(heap_listp);
        while(st!=NULL){
            if(GET_SIZE(HDRP(st)) >= asize) return st;
            st = GET(NEXT_LINKED_BLKP(st));
        }
    }
    return NULL;
}
```

对于place函数，只需在原来的基础上用delete_block函数删除原本空闲块，并使用add_block函数插入可能多余的块即可。

```
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));
    delete_block(bp, csize);
    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));
        PUT(PREV_LINKED_BLKP(bp), NULL); //对新块初始化
        PUT(NEXT_LINKED_BLKP(bp), NULL);
        add_block(bp,csize-asize);
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

对于新定义的find_asize_loc函数，只需根据输入的size大小返回原本在mm_init函数中定义的对应链表的首地址即可。

```
static void* find_asize_loc(size_t asize)
{
    void* heap_listp = mem_heap_lo();
    if(asize <= 16) return(heap_listp + 1 * WSIZE);
    if(asize <= 32) return(heap_listp + 2 * WSIZE);
    if(asize <= 64) return(heap_listp + 3 * WSIZE);
    if(asize <= 128) return(heap_listp + 4 * WSIZE);
    if(asize <= 256) return(heap_listp + 5 * WSIZE);
    if(asize <= 512) return(heap_listp + 6 * WSIZE);
    if(asize <= 1024) return(heap_listp + 7 * WSIZE);
    if(asize <= 2048) return(heap_listp + 8 * WSIZE);
    if(asize <= 4096) return(heap_listp + 9 * WSIZE);
```

```
    return(heap_listp + 10 * WSIZE);
}
```

对于add_block函数，先使用find_asize_loc函数定位该块应在的链表。在根据块大小进行插入操作时可能会存在三种情况：

1、该链表中尚无元素，则直接将该块放在链表首位。

2、该链表中已有元素，但该块小于所有块（即应被放在第一位），则将其放在链表首位，然后更新他的NEXT_LINKED_BLKP指针和下一个块的PREV_LINKED_BLKP指针。

3、该链表中已有元素，且该块位置处于这些块的中间，则将该块插入对应位置，并更新前后块的对应指针。

```
void add_block(void* bp, size_t asize)
{
    void* heap_listp = find_asize_loc(asize);
    if(GET(heap_listp) == NULL){    /* 该链表中尚无元素 */
        PUT(heap_listp, bp);
        PUT(PREV_LINKED_BLKP(bp), heap_listp);
    }
    else{
        void* next = GET(heap_listp);
        void* prev = heap_listp;
        while(next != NULL && GET_SIZE(next)<asize){ //根据块大小寻找相应位置
            prev = next;
            next = NEXT_LINKED_BLKP(next);
        }
        PUT(PREV_LINKED_BLKP(bp), prev);
        PUT(NEXT_LINKED_BLKP(bp), next);
        if(prev == heap_listp){     /* 该链表中已有元素，但该块小于所有块（即应被放在第一位） */
            PUT(prev, bp);
        }
        else{                       /* 该链表中已有元素，且该块位置处于这些块的中间 */
            PUT(NEXT_LINKED_BLKP(prev), bp);
        }
        if(next != NULL) PUT(PREV_LINKED_BLKP(next), bp);
    }

}
```

对于delete_block函数，也存在两种情况：

1、该块为链表中的第一个元素，则直接将链表的第一个元素指针更新为他的下一个元素

2、该块处于链表中间，则更新他前面元素的NEXT_LINKED_BLKP指针为他的下一个元素

并且判断该块下一个元素是否为空在更新下一个块的PREV_LINKED_BLKP指针。

```
void delete_block(void* bp, size_t asize)
{
    void* heap_listp = find_asize_loc(asize);
    void* prev = GET(PREV_LINKED_BLKP(bp));
    void* next = GET(NEXT_LINKED_BLKP(bp));
    if(prev == heap_listp){                 /* 该块为链表中的第一个元素 */
        PUT(prev, next);
    }
```

```
    else{                                       /* 该块处于链表中间 */
        PUT(NEXT_LINKED_BLKP(prev), next);
    }
    if(next != NULL){
        PUT(PREV_LINKED_BLKP(next), prev);
    }
}
```

对于之前简单粗暴的mm_realloc函数，做出了较大调整。

1、若原本的块大小oldsize大于新的需要重新分配的大小asize，则对原来的块取前面asize大小的元素，并利用place函数的思路判断多余的块大小是否大于16字节，若是，则将其放入对应链表中重新利用，这样可以提高空间利用率。

2、但若oldsize小于asize，则判断该块后面的块是否为空闲块，若是，则将两块合并，并且同上，将可能多余的部分重新放入链表中，这不同于直接使用mm_malloc函数，可以大大节省时间并提高空间利用率。如果都不满足上述条件，只能调用mm_malloc函数重新寻找空间。

（在与同学交流时，也听到过将前面块一起放入并判断的思路。但如果当前块和后面块的总大小都无法满足要求，那么即使加入前面块，也意味着原本数据需要进行移动，我认为这和调用mm_malloc再将数据移动到相应位置所需的成本是差不多的。不过因为时间原因，没有单独再写一个版本加以验证。）

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

    size_t asize;
    if(size <= DSIZE) asize  = 2 * DSIZE;
    else asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1))/DSIZE);

    size_t oldsize = GET_SIZE(HDRP(ptr));
    if(oldsize == asize) return ptr;        /* 需要重分配的空间和原始大小一致，可以直接返回 */
    else if(oldsize > asize){               /* 1、oldsize大于asize */
        if(oldsize - asize>=2*DSIZE){       /* 思路同place */
            PUT(HDRP(ptr), PACK(asize, 1));
            PUT(FTRP(ptr), PACK(asize, 1));
            void *bp = ptr;
            bp = NEXT_BLKP(bp);
            PUT(HDRP(bp), PACK(oldsize - asize, 0));
            PUT(FTRP(bp), PACK(oldsize - asize, 0));
            PUT(PREV_LINKED_BLKP(bp), NULL);
            PUT(NEXT_LINKED_BLKP(bp), NULL);

            coalesce(bp);
        }
        else{
            PUT(HDRP(ptr), PACK(oldsize, 1));
            PUT(FTRP(ptr), PACK(oldsize, 1));
        }
        //immediate or delay
```

```
        return ptr;
    }
    else{                                   /* 2、oldsize小于asize */
        size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
        //判断后面块是否为空闲块并且两块大小之和是否不小于asize
        if(!next_alloc && GET_SIZE(HDRP(NEXT_BLKP(ptr))) + oldsize >= asize) {
            delete_block(NEXT_BLKP(ptr), GET_SIZE(HDRP(NEXT_BLKP(ptr))));
            size_t ssize = GET_SIZE(HDRP(NEXT_BLKP(ptr))) + oldsize;
            size_t last = ssize - asize;
            if(last >= 2*DSIZE){            /* 思路同place */
                PUT(HDRP(ptr), PACK(asize, 1));
                PUT(FTRP(ptr), PACK(asize, 1));
                char *bp = NEXT_BLKP(ptr);
                PUT(HDRP(bp), PACK(last, 0));
                PUT(FTRP(bp), PACK(last, 0));
                PUT(NEXT_LINKED_BLKP(bp), NULL);
                PUT(PREV_LINKED_BLKP(bp), NULL);
                add_block(bp, last);
            }
            else{
                PUT(HDRP(ptr), PACK(ssize, 1));
                PUT(FTRP(ptr), PACK(ssize, 1));
            }
            return ptr;
        }
        //使用mm_malloc重新寻找合适的空间
        else{
            char *newptr = mm_malloc(asize);
            if(newptr == NULL) return NULL;
            memcpy(newptr, ptr, oldsize - DSIZE);
            mm_free(ptr);
            return newptr;
        }
    }
}
```

最终得分：83（43+40）

优缺点：可以看出速度与上一版本相比有了很大提升，但空间利用率提升不大。

3、在第二版基础上使用改进版的first fit策略

在查阅资料时发现，如果链表中的块是按地址而不是大小排序，可以更好地利用内存空间，因此对add_block函数进行了修改，在寻找合适的插入位置时比较的是块的地址大小而不是块的大小。

```
void add_block(void *bp, size_t asize){
    void* heap_listp = find_asize_loc(asize);
    void* list_table = mem_heap_lo();
    if(GET(heap_listp) == NULL){
        PUT(heap_listp, bp);
        PUT(PREV_LINKED_BLKP(bp), heap_listp);
    }
    else{
        void* prev = heap_listp;
        void* next = GET(prev);
        while(next != NULL && next < bp){       //此处修改为按地址大小排序
            prev = next;
```

```
            next = GET(NEXT_LINKED_BLKP(next));
        }
        if(prev != heap_listp){
            PUT(PREV_LINKED_BLKP(bp),prev);
            PUT(NEXT_LINKED_BLKP(bp),next);
            PUT(NEXT_LINKED_BLKP(prev),bp);
        }
        else{
            PUT(PREV_LINKED_BLKP(bp),prev);
            PUT(NEXT_LINKED_BLKP(bp),next);
            PUT(prev,bp);
        }
        if(next != NULL) PUT(PREV_LINKED_BLKP(next), bp);

    }

}
```

最终得分：86（46+40）

优缺点：相比第二版代码在空间利用率上的确有了一定提升，但尚有提升空间。

4、在第2、3版基础上的place函数改进版

在放置过程中，可以抽象出一个问题：如果有一系列申请空间的请求，且申请请求的空间大小为大、小、大、小、大、小……这样的组合，那么当较大的块被释放后，由于中间较小块的存在，这些块不能被合并，那么当下一次更大块的请求到来时，就需要去其他地方寻找空间从而使得这些块被浪费，导致空间利用率的降低。

如果将这些请求块按一定大小划分，较小的都与原有空闲块的头对齐，而较大的都与空闲块的尾部对齐，这样较大块被释放时也能通过coalesce函数合并，就会减少上述情况的发生，提高空间利用率。

因此，在原来place函数的基础上做如下改进：如果请求大小size小于m（m为自定义的大小）或者free size-size小于16字节，则同之前一样；否则，将前面的free size-size大小的作为新的空闲块，将后面size大小的块作为存放内容的块。

因为输入的bp地址发生的变化，因此在更新过的place函数中，要返回对应新的分配块的首地址给mm_malloc函数，以防出错。

```
static void* place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));
    delete_block(bp, csize);
    if ((csize - asize) < (2*DSIZE)) {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
        return bp;
    }
    else if (asize <= 96){
        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));
        PUT(PREV_LINKED_BLKP(bp), NULL); //对新块初始化
        PUT(NEXT_LINKED_BLKP(bp), NULL);
        add_block(bp,csize-asize);
        return PREV_BLKP(bp);
```

```
    }
    else{
        PUT(HDRP(bp), PACK(csize-asize, 0));
        PUT(FTRP(bp), PACK(csize-asize, 0));
        PUT(PREV_LINKED_BLKP(bp), NULL); //对新块初始化
        PUT(NEXT_LINKED_BLKP(bp), NULL);
        add_block(bp,csize-asize);
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        return bp;
    }
}
```

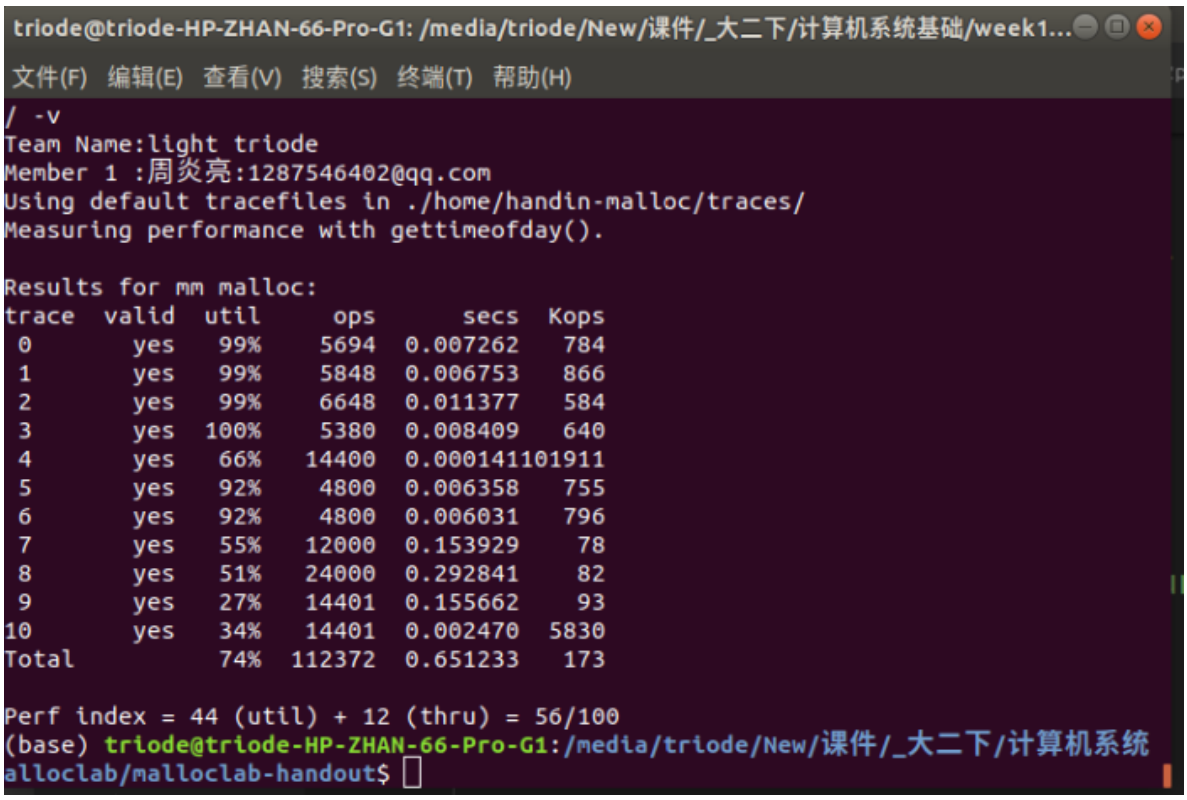
最终得分:

m的值	64	96	128	256	512
第二版	92	94	92	91	88
第三版	91	93	/	/	/

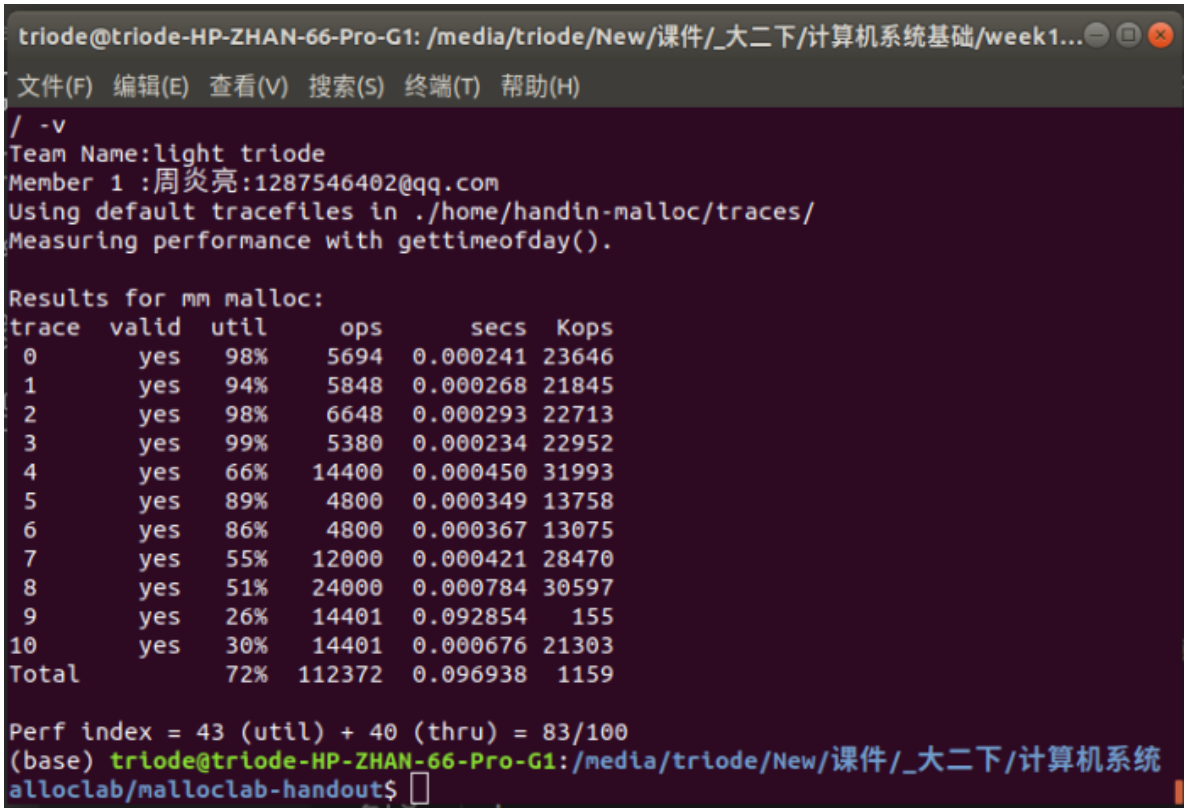
优缺点：因为先用第二版的代码根据不同的m大小进行测试，故第三版的测试数量较少。这些得分的两个组成部分只有第一个部分不同，故可以发现m的值和内存利用率呈一个凸函数的关系，且在m为96时内存利用率最高。

三、部分实验截图

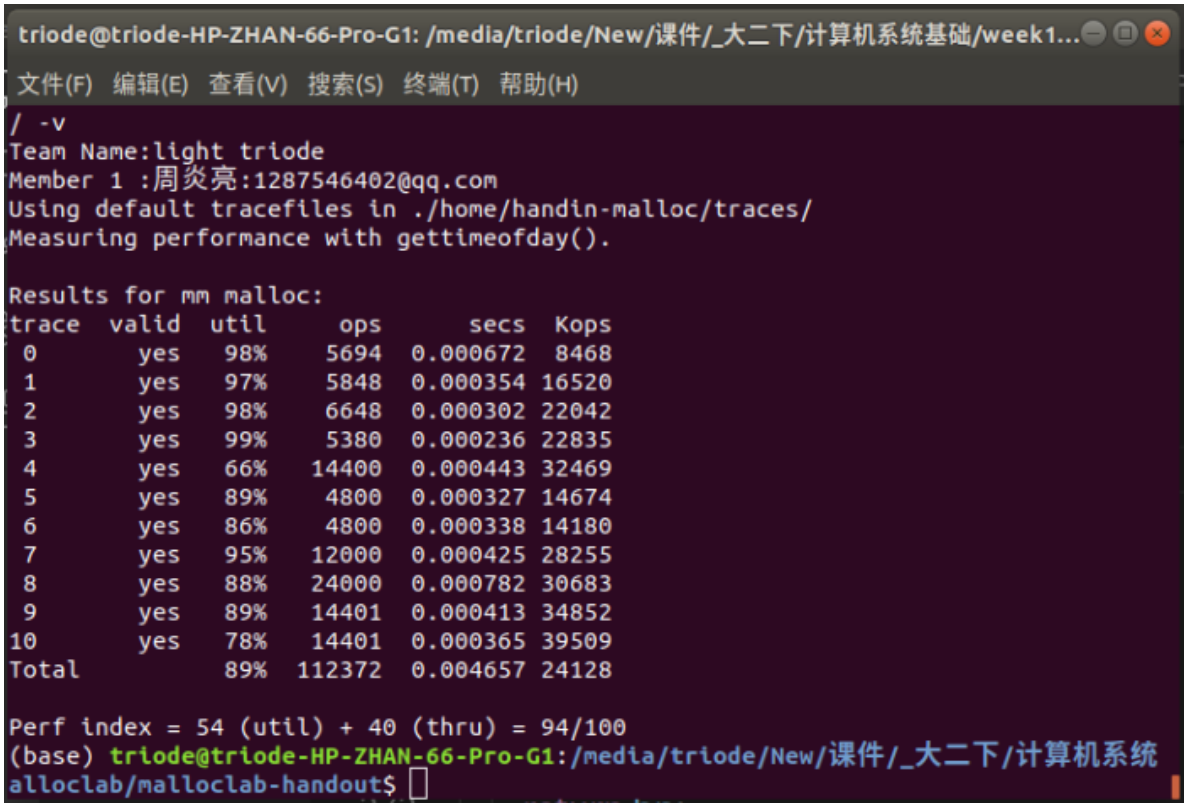
第一版代码:



第二版代码:



第四版代码最高分：



四、实验感想

本次实验提高了我对malloc部分知识点的掌握程度，并且在编写代码时对指针操作也有了更多的认识。

此外还提高了我在编写代码时的严谨性（毕竟一点小小的瑕疵就会导致程序出现段错误，调bug时有时要调好久，最后有时就会发现其实只是一些不起眼的问题）。