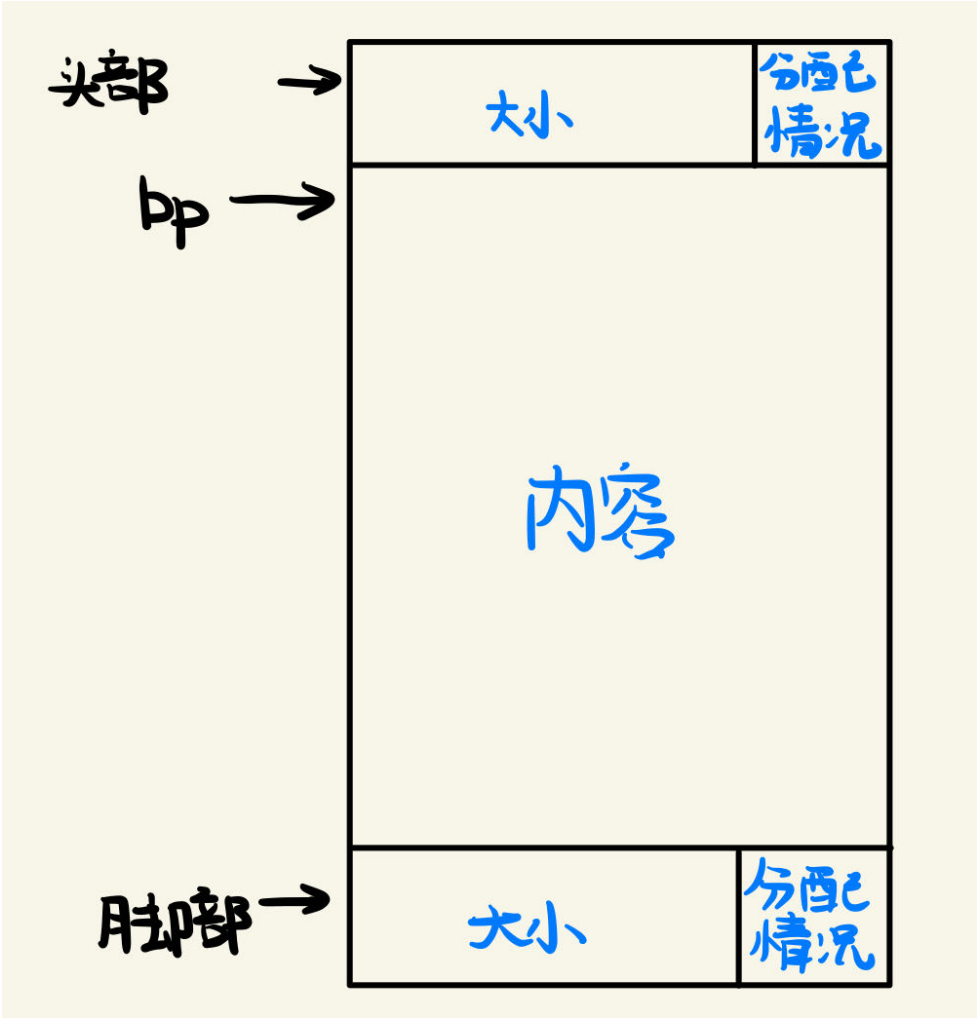
malloc 解题报告 吴佳启 2022010869

方法一：隐式链表

实现思路

1. 块的存储结构：每个块维护一个头部和尾部，这个脚部记录了自己的大小和分配情况，方便后续操作。



2. 相关操作：基于这个块的结构，我们不难构造出以下的宏定义：

- 跳转至块p的头部：由于是16对齐，同时指针bp是指向块的内容的，所以我们需要把bp的地址减去8获得块的头部。
- 块的大小：跳转至头部后去除尾部的偏移量即可获得块的大小。
- 跳转至上一个块（物理意义）：我们当前的bp减去16就可以到达上一个块的尾部，然后获得了上一个块的大小后就可以跳转至上一个块的指针。
- 跳转至下一个块（物理意义）：当前的bp获取了自己的大小后，直接加上自身的大小再减去偏移量8就可以到达下一个块的bp指针位置了。

```
#define MAX(x, y) ((x) > (y)? (x) : (y))
#define PACK(size, alloc) ((size) | (alloc))// return the value of the size and the
alloc bit, which will be stored in the footer or header
#define GET(p) (*(unsigned int *)(p))// get the value of the address p
#define PUT(p, val) (*(unsigned int *)(p) = (val))// change the value of the
address p

#define GET_SIZE(p) (GET(p) & ~0x7)// get the size of the address pz
#define GET_ALLOC(p) (GET(p) & 0x1)// get the alloc bit of the address p
#define HDRP(bp) ((char *)(bp) - WSIZE)// get the header of the address header
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)// get the address of
the footer
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE))) // get the
address of the next block
```

```
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE))) // get the address of the previous block block
```

具体实现

1. init函数
   我们给整个堆分配一个首尾项作为哨兵，把尾哨兵的size设置为0，作为遍历时的终止条件，然后给整个堆适当扩容。

```
int mm_init(void) {
    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void*)-1)
          return -1;
    PUT(heap_listp, 0); // alignment padding
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); //prologue header
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); //prologue footer
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1)); //epilogue header
    heap_listp += (2 * WSIZE);
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

2. malloc函数

- 作为任务实现的核心，我们需要找到一个空闲块并且给他分配空间，这里我直接采用了first fit算法，即找到第一个满足大小的块，然后分配。
- 具体来说，first fit算法会从堆的起始位置开始遍历，直到找到一个满足大小并且未被分配的块，然后调用place函数进行分配。 而place函数则会考虑这个块的大小是否能够分为两块：即一个用于分配，另一个则作为一个缩小了的空闲块，然后设置块的头部尾部完成分配工作。
- 以上讨论的是查找成功，即找到空闲块的情况，那自然会存在遍历到了哨兵仍然没有空闲，那这个时候就需要对堆进行扩容，扩容后再调用place函数完成分配。

```
void *mm_malloc(size_t size) {
    size_t asize;      // Adjusted block size
    size_t extendsize; // Amount to extend heap if no fit
    char *bp;
    if (size == 0)
        return NULL;
    if(size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
    //if succeed in finding the fit
```

```
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }
    // no fit found, get more memory
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

3. free函数

   free函数的实现比较简单，首先需要找到要释放的块，然后将这个块标记为空闲，然后调用coalesce函数将这个块与相邻的空闲块进行合并，这里的合并则是根据首尾相邻的两个块的空闲情况来决定合并策略，基于我们之前定义的宏可以很方便地获得首尾块的情况。

```
static void* coalesce(void* bp){// handle the job of merging
    unsigned int prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    unsigned int next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));//get the size of the current block, which
will be the final amount of the total block
    //handle the job in four conditions
    if(prev_alloc && next_alloc)
        return 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));
    }
    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));
        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));
        bp = PREV_BLKP(bp);
    }
    return bp;
}

void mm_free(void *ptr) {
    size_t size = GET_SIZE(HDRP(ptr));//get the size of the block which will be
free
    PUT(HDRP(ptr), PACK(size, 0));//set the header and footer to 0
    PUT(FTRP(ptr), PACK(size, 0));
```

```
    coalesce(ptr);//coalesce the free block with the next block if it is free
}
```

4. realloc函数

   这里要求给一个块重新分配大小，大小值给定，这里我们就采用实例代码的思路，直接调用malloc函数重新分配一个内存块，然后根据新旧内存块的大小，在保证复制的字节数不超过内存块大小的前提下，将原内存块中的数据复制到新内存块中，最后释放原内存块。

```
void *mm_realloc(void *ptr, size_t size) {
    void *oldptr = ptr;
    void *newptr;
    size_t copySize;
    newptr = mm_malloc(size);
    if (newptr == NULL)
      return NULL;
    size = GET_SIZE(HDRP(oldptr));
    copySize = GET_SIZE(HDRP(newptr));
    if (size < copySize)
      copySize = size;
    memcpy(newptr, oldptr, copySize-WSIZE);
    mm_free(oldptr);
    return newptr;
}
```

实验结果与问题思考

1. 最终得到了44+1=45分，可以发现这里吞吐率的得分极低，效率差。

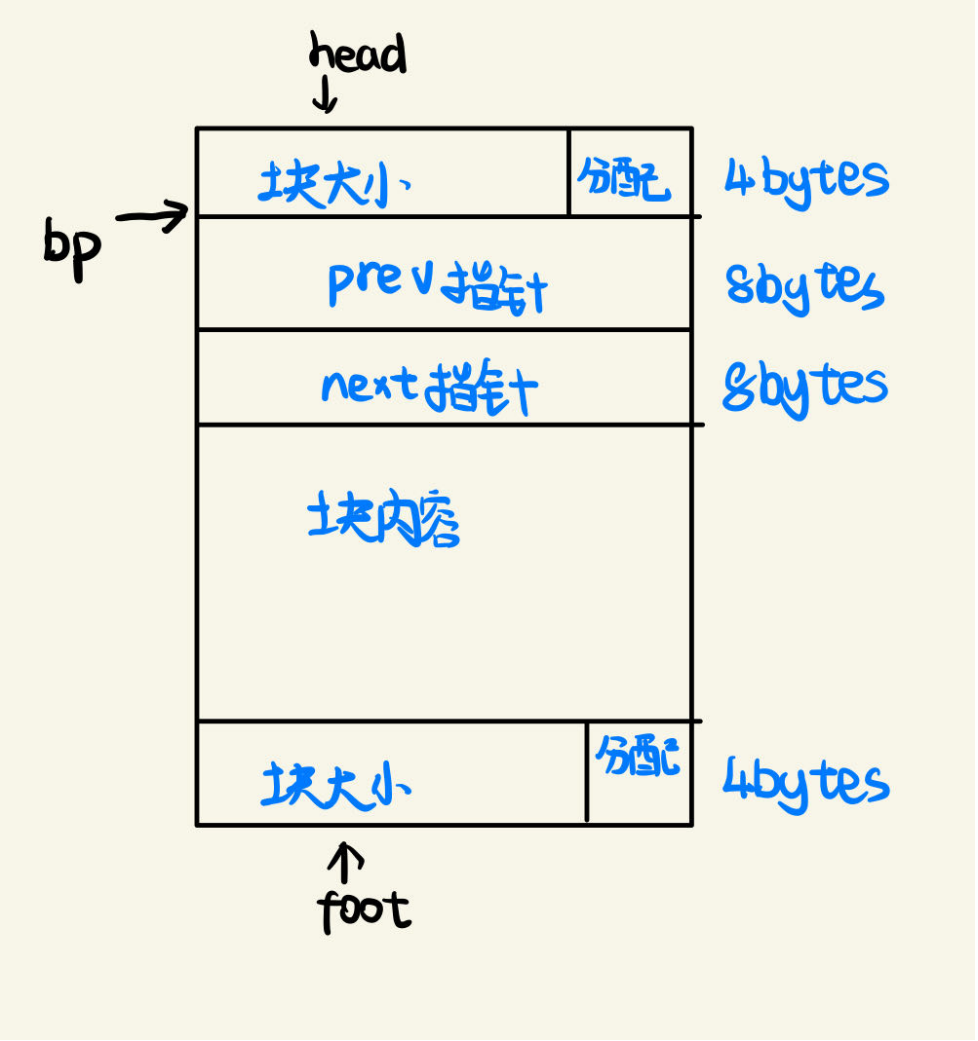
```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   98%    5694  0.009320    611
 2       cccp-bal.rep       yes   99%    5848  0.008365    699
 3    cp-decl-bal.rep       yes   99%    6648  0.014268    466
 4       expr-bal.rep       yes   99%    5380  0.012096    445
 5 coalescing-bal.rep       yes   66%   14400  0.000097 148760
 6     random-bal.rep       yes   92%    4800  0.006966    689
 7    random2-bal.rep       yes   91%    4800  0.006499    739
 8     binary-bal.rep       yes   54%   12000  0.118056    102
 9    binary2-bal.rep       yes   47%   24000  0.409111     59
10    realloc-bal.rep       yes   27%   14401  0.049742    290
11   realloc2-bal.rep       yes   34%   14401  0.001065  13528
Total                            73%  112372  0.635584    177

Score = (44 (util) + 1 (thru)) * 11/11 (testcase) = 45/100
```

2. 原因分析：不难发现，块的分配与堆块的总数呈现线性关系，也就是说执行的效率最坏可能达到$O(n)$，这个自然不是我们所希望的，于是想办法将复杂度降低是当务之急。

方法二：显式空闲链表+分离适配

实验思路

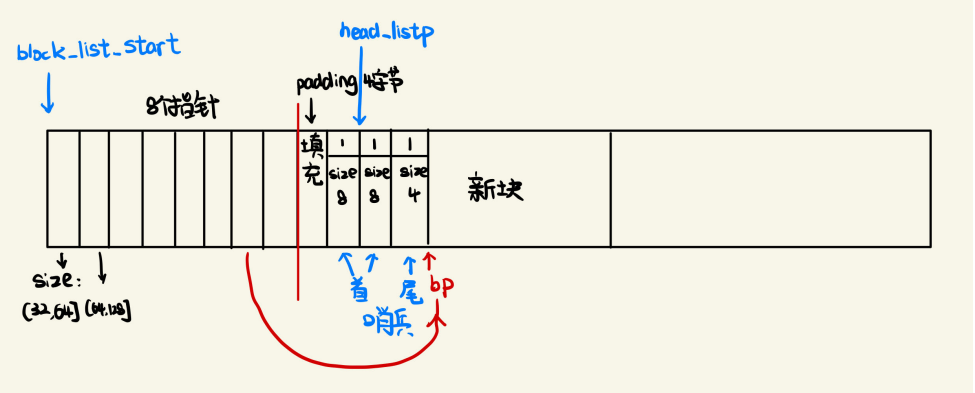
1. 块结构：这里我们用一个显式的链表结构完成，与隐式链表不同之处在于，这里我存储了两个指针，一个是他的前向块，一个是后向块，通过这种指针结构访问链表中的相邻块。



2. 空闲块组织结构

- 首先我们会提到一个“大小类”的概念，在这个结构中我们存储的是空闲块，根据他的大小进行分类，先考虑一下块的最小大小：$4+8+8+4$，但同时还需要对齐到16字节，因而最小块大小应该为32，阅读输入数据不难发现最大块的规模在4096左右，因此我们可以根据2的幂次来划分大小类，即32-64，64-128之类的，最后比4096大的再分为一类，我们会把不同的块归于不同的类中去。
- 通过以上的分类后我们可以得到8个类，每个类通过一个链表实现，于是在初始化的过程中我们先分配八个八字节的指针，分别指向这八个链表的头指针，后续的插入和删除操作我们就需要找到对应的链表，然后对链表进行插入和删除操作。

- 接下来的堆里我们会分配一些块，即哨兵，这里我分配了一个首尾哨兵，头部只有一个首尾部分，大小为8bytes状态为1，尾部则只有一个大小为0且分配状态为1的部分，大小为4bytes，这么分配后会出现一个问题：我们最终得到的下一个分配块的地址并未做到16字节对齐，因此我们需要在这之前先添加一个4自己的块作为padding，这样我们就可以保证下一个分配块的地址是16字节对齐的。
- 初始化以后我们来大致看看接下来的操作，接下来分配一个块以后会把他接在堆后面，然后进行左右合并，然后根据它的大小情况插入到适当的链表中去，而每一个链表我按照大小从小到大方式组织排序，通过链表的顺序模式插入到合适的位置。



3. 效率：我们通过这种方式在寻找下一个空闲块的时候仅需要遍历空闲块而不是所有块，减少了首次适配的分配时间，这样和隐式链表相比在时间复杂度上性能有了极大的优势。

具体实现

一些辅助操作

与静态操作不同，这里我们的块中存储了两个指针，为了方便对这两个指针进行获取和赋值操作，我也定义了一些宏，同时考虑到新的块最小规模为32，这个也在后续对块的补齐操作中有显著作用。

```
#define GET_PREV(p) (*(unsigned long long *)(p))
#define SET_PREV(p, prev) (*(unsigned long long *)(p) = (prev))
#define GET_NEXT(p) (*((unsigned long long *)(p)+1))
#define SET_NEXT(p, val) (*((unsigned long long *)(p)+1) = (val))
#define MINBLOCKSIZE 32
```

然后是为了对齐，即传递一个size后要为他分配多大的空间，由于一个已经分配出去的块只需要存储前后两个头即可，需要把size先加8，然后小于等于16的话补为最小的块规模，反之则直接调用align向上对齐到16的倍数即可。

```
static int align_size(int size){
    size += DSIZE;//the allocated block don't need to store the two pointers
    if(size <= 16)
        return 32;
    else
        return ALIGN(size);
}
```

删除操作

传递一个指针，我们要把这个块在它原有的链表里删除，这里和list的操作很像，即短接操作，我们只需要判断前后块是否为空，来决定条件更改，比如前方块如果不空，就把前方块存储后继的指针变成next，后方块存储前驱的指针变成prev，诸如此类与list操作一致。

```
static void remove_from_free_list(void *bp){
    if (bp == NULL || GET_ALLOC(HDRP(bp))){
        return;
    }
    //get the prev and the next, then delete the bp part
    void *root = get_free_list_head(GET_SIZE(HDRP(bp)));
    void* prev = GET_PREV(bp);
    void* next = GET_NEXT(bp);
    SET_PREV(bp, NULL);
    SET_NEXT(bp, NULL);
    if (prev == NULL){// the prev is null, which means that the removed one is the
root
        if (next != NULL) SET_PREV(next, NULL);
        PUTL(root, next);
    }
    else{
        if (next != NULL) SET_PREV(next, prev);
        SET_NEXT(prev, next);
    }
}
```

插入操作

我们同样传递一个指针，然后把该指针插入到合适的位置，这里分为两步，首先我们先确定它属于那八个基础指针的哪一个序列，于是我们先根据插入的空闲块的大小找到序列，然后在这个序列中通过顺序遍历的方式找到合适的地方，这里的合适就是从小到大的插入，基于链表的结果就通过线性查找合适的位置，然后和list一样完成插入。

```
static void insert_to_free_list(void* bp)
{
    // the whole sequence is sorted by the size
    if (bp == NULL)
        return;
    //get the head, the prev and the next
    void* root = get_free_list_head(GET_SIZE(HDRP(bp)));
    void* prev = root;
    void* next = GETL(root);
    //by linear search, find the right position to insert
    while (next != NULL){
        if (GET_SIZE(HDRP(next)) >= GET_SIZE(HDRP(bp))) break;
        prev = next;
        next = GET_NEXT(next);
    }
    //insert the bp to the right position
    if (prev == root){// insert at the root, then the root will be the bp
        PUTL(root, bp);
        SET_PREV(bp, NULL);
        SET_NEXT(bp, next);
        if (next != NULL) SET_PREV(next, bp);
    }
    else{// change the prev, next conditions, which is the same as that in the list
        SET_PREV(bp, prev);
        SET_NEXT(bp, next);
        SET_NEXT(prev, bp);
        if (next != NULL) SET_PREV(next, bp);
    }
}
```

块的合并-coalesce

这里的基本思路和隐式一样，还是根据物理上连续的块进行合并，与先前操作不同的是，这里我们使用的是显式，所以对于之前已经分配好了的空闲块，如果又被合并了的话，我们要先把这个块从原有的链表里移除，然后把合并后的新块再调用insert_to_free_list函数插入到合适的空闲块链表中。
这里需要注意的一个细节在于，如果是和前面的那个块合并的话，合并后新的块的指针不应该为传入的bp，而是那个指向前面块的指针了，新块的地址也变成了前面块的地址。

```
static void* coalesce(void* bp){// handle the job of merging
    void* prev_bp = PREV_BLKP(bp);
    void* next_bp = NEXT_BLKP(bp);
    //get the conditions
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    //get the size of the current block, which will be the final amount of the
total block
    size_t size = GET_SIZE(HDRP(bp));
```

```
    //handle the job in four conditions
    if(prev_alloc && next_alloc){
        SET_PREV(bp, NULL);
        SET_NEXT(bp, NULL);
    }
    //the total logic is remove the part which can be merged, and then merge bp
from forward and backward, then insert the final part
    else if(prev_alloc && !next_alloc){
        remove_from_free_list(next_bp);
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        SET_PREV(bp, NULL);
        SET_NEXT(bp, NULL);
    }
    else if(!prev_alloc && next_alloc){
        remove_from_free_list(prev_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);
        SET_PREV(bp, NULL);
        SET_NEXT(bp, NULL);
    }
    else{
        remove_from_free_list(prev_bp);
        remove_from_free_list(next_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);
        SET_PREV(bp, NULL);
        SET_NEXT(bp, NULL);
    }
    insert_to_free_list(bp);
    return bp;
}
```

extend_heap操作

对这里传入的参数bytes我要求他已经通过了一些手段完成实现了16的倍数要求，因此这里只需要调用mem_sbrk函数申请一块新的空间即可，申请结束后对这个新的块的四个属性：首尾记录大小为bytes，状态为未分配，然后给他两个指针置为null，同时在他后面的位置标记上一个状态1，表示后面的块已占用——即后面的块不是空闲的不可访问。注意这里申请了内存后并没有立刻把他插入序列中，而是调用coalesce函数，因为这里可能在前面还存在一个比较大的空闲块，我们先通过合并以后再把它插入到序列中去，避免空间的碎片化。

```
static void* extend_heap(size_t bytes){//assure that bytes are 16k
    char* bp;
```

```
    // allocate bytes
    if ((long)(bp = mem_sbrk(bytes)) == -1)
        return NULL;
    // init the footer and the header
    PUT(HDRP(bp), PACK(bytes, 0));
    PUT(FTRP(bp), PACK(bytes, 0));
    SET_PREV(bp, 0);
    SET_NEXT(bp, 0);
    // set the next part's alloc to be 1
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));
    // coalesce if the previous block is free.
    return coalesce(bp);
}
```

init操作

根据我们之前的想法，我们分配八个八字节的指针，然后是四个四字节的块，接下来我们把block_list_start和heap_list分别指向最开始和第一个开始块（序言块），然后调用extend_heap函数先分配一些内存空间。

```
int mm_init(void){
    if ((heap_listp = mem_sbrk(8*DSIZE+4*WSIZE)) == (void*)-1)
        return -1;
    block_list_start = heap_listp;
    //allocate the free block
    PUTL(heap_listp, 0);            //block size <= 64
    PUTL(heap_listp+(1*DSIZE), 0); //block size <= 128
    PUTL(heap_listp+(2*DSIZE), 0); //block size <= 256
    PUTL(heap_listp+(3*DSIZE), 0); //block size <= 512
    PUTL(heap_listp+(4*DSIZE), 0); //block size <= 1024
    PUTL(heap_listp+(5*DSIZE), 0); //block size <= 2028
    PUTL(heap_listp+(6*DSIZE), 0); //block size <= 4096
    PUTL(heap_listp+(7*DSIZE), 0); //block size > 4096
    heap_listp+=(8*DSIZE);
    //allocate the prologue/epilogue and the padding to assure the bp to be 16k
    PUT(heap_listp, 0); //padding
    PUT(heap_listp+(1*WSIZE), PACK(DSIZE, 1)); //prologue header
    PUT(heap_listp+(2*WSIZE), PACK(DSIZE, 1)); //prologue footer
    PUT(heap_listp+(3*WSIZE), PACK(0, 1)); //epilogue header
    heap_listp+=2*WSIZE;
    extend_heap(128);
    return 0;
}
```

mm_malloc

然后就是mm_malloc函数，核心逻辑和静态是一样的，但是由于各个函数不同的实现最终的效果有很大差距。

```
void *mm_malloc(size_t size) {
    size_t asize;      // Adjusted block size
    size_t extendsize; // Amount to extend heap if no fit
    char *bp;
    if (size == 0)
        return NULL;
    asize = align_size(size);
    //if succeed in finding the fit
    if ((bp = find_fit(asize)) != NULL) {
        bp = place(bp, asize);
        return bp;
    }
    // no fit found, get more memory
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize)) == NULL)
        return NULL;
    bp = place(bp, asize);
    return bp;
}
```

1. 首先我们把传入的参数size通过align_size函数对齐到16的倍数。
2. 然后通过find_fit函数来寻找一个合适的空闲块，这里find_fit的方法是，我们先根据要求的大小在8个链表头中找到最适合的链表头，接下来进行顺序遍历看能否找到一个合适的，这里有可能在这个链表里仍然没有满足要求的，举个例子，如果只有一块1024的空闲块，查询32的块的时候就会查不到，因此如果某一个链表查到最后了仍然没找到，我们要立刻进入下一个链表——即下一个存储块更大的链表里去寻找有无合适的，如果扫到最大的那块了仍然不行，那就结束返回null，然后申请更大的内存空间，bp指向这个更大的空间。

```
static void* find_fit(size_t asize){
    for (void* root = get_free_list_head(asize); root != (heap_listp-2*WSIZE);
root+=DSIZE){
        void* bp = GETL(root);
        while (bp){// enum the free blocks
            if (GET_SIZE(HDRP(bp)) >= asize){
              return bp;
            }
            bp = GET_NEXT(bp);
        }
        // if this root doesn't have enough space, then go to a bigger one
    }
    return NULL;
}
```

3. 接下来无论是找到了还是没找到，我们都需要把当前指针指向的这个块进行分配，即调用place函数，这里place函数大概思路为：考察目前这个空闲块的大小，根据

我们find的搜索条件可以断定这个块的size一定大于我们需要的，但是他能否再分出一个新的空闲块呢？这个就要考虑这个块的大小减去需要的大小能否到达最小块的大小了，如果可以我们就把这个块分割成两部分，反之整个块来存储这部分内存。

如何让这个块变为已分配的也十分简单，只需要更改他的头部和尾部信息即可。

这里有一个小技巧，当可以分出来新的块时，我们要考虑到底是把前面的作为新的空闲块还是把后面的作为新的空闲块，如果我们不做这个考虑的话最终得分只有89左右，在后续调整参数与技巧中会说明详细原因。

最终的修改方案为，如果插入的块大于96，就把它作为原来块的尾部，反之则为头部，这样可以得到比较高的性能。最后和之前分析一样，把新的空闲块通过coalesce函数插入到合适的位置。

```
static void* place(void* bp, size_t asize){
    size_t size = GET_SIZE(HDRP(bp));
    remove_from_free_list(bp);
    if ((size - asize) >= MINBLOCKSIZE){//if the block is large enough to split
        //the trick here is to insert the larger one at the back
        if(asize > 96){
            PUT(HDRP(bp), PACK(size-asize, 0));
            PUT(FTRP(bp), PACK(size-asize, 0));
            void* new_bp = NEXT_BLKP(bp);
            PUT(HDRP(new_bp), PACK(asize, 1));
            PUT(FTRP(new_bp), PACK(asize, 1));
            coalesce(bp);
            return new_bp;
        }
        else{
            PUT(HDRP(bp), PACK(asize, 1));
            PUT(FTRP(bp), PACK(asize, 1));
            void* new_bp = NEXT_BLKP(bp);
            PUT(HDRP(new_bp), PACK(size-asize, 0));
            PUT(FTRP(new_bp), PACK(size-asize, 0));
            coalesce(new_bp);
        }
    }
    else{
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    }//if the block is not large enough to split
    return bp;
}
```

mm_free函数

free相对简单，我们只需要把这个块的首尾记录改为0即未分配，然后把两个指针都设置为NULL，最后调用coalesce函数把相邻的空闲块合并后插入空闲块即可。

```
void mm_free(void *bp) {
    size_t size = GET_SIZE(HDRP(bp));
    // change the conditions of the bp
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    SET_PREV(bp, 0);
    SET_NEXT(bp, 0);
    return coalesce(bp);
}
```

realloc函数

首先我们会自然想到一个最简单的方法，就是先free掉原来的指针，然后重新malloc一个指针，可想而知，这样在时间上会有着比较高的复杂度，尤其是在realloc函数调用次数多的情况下，因而自然要进行优化。

```
void *mm_realloc(void *ptr, size_t size) {
    size_t oldsize = GET_SIZE(HDRP(ptr));
    size_t requiredSize = align_size(size);
    void* oldptr = ptr;
    size_t copySize;
    void* newptr = mm_malloc(requiredSize);
    if (newptr == NULL)
        return NULL;
    if(oldsize < size){
      copySize = oldsize;
    }
    else{
    copySize = size;
    }
    memcpy(newptr, oldptr, copySize);
    mm_free(oldptr);
    return newptr;
}
```

最终的得分也和我们预想的一致，在最后两个realloc的情况下，耗时太长最终得分很低。

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   99%    5694  0.000295  19315
 2       cccp-bal.rep       yes   99%    5848  0.000237  24644
 3    cp-decl-bal.rep       yes   99%    6648  0.000206  32288
 4       expr-bal.rep       yes   99%    5380  0.000124  43387
 5 coalescing-bal.rep       yes   97%   14400  0.000150  96000
 6     random-bal.rep       yes   95%    4800  0.000875   5486
 7    random2-bal.rep       yes   95%    4800  0.000745   6441
 8     binary-bal.rep       yes   91%   12000  0.000672  17868
 9    binary2-bal.rep       yes   81%   24000  0.000786  30519
10    realloc-bal.rep       yes   26%   14401  0.051371    280
11   realloc2-bal.rep       yes   30%   14401  0.001401  10278
Total                            83%  112372  0.056863   1976

Score = (50 (util) + 8 (thru)) * 11/11 (testcase) = 58/100
```

那如何优化呢？这种最简单的方法虽然适用但是有着很高的复杂度，我们尽量让这种方法少出现即可。

- 首先我们会很自然的想到能否与后面的块合并：如果后面的块是空闲的且size足够，那我们直接把后面这个块从原来的链表里删除，然后把这两个块合并成一个块，然后把合并后的块插入到空闲块链表中。

```
if(!next_alloc && oldsize + next_size >= requiredSize){//extend the next part
    size_t new_size = oldsize + next_size;
    remove_from_free_list(next_bp);
    PUT(HDRP(ptr), PACK(new_size, 1));
    PUT(FTRP(ptr), PACK(new_size, 1));
    ptr = place(ptr, new_size);
}
```

- 其次我们可考虑后面的块如果为空，也就是说这个块的next如果指向的是空，那我们直接调用mem_sbrk函数申请一块内存空间贴在后面即可。

```
else if(!next_size && requiredSize >= oldsize){// the next part is the tail
    size_t extend_size = requiredSize - oldsize;
    if((long)(mem_sbrk(extend_size)) == -1)
        return NULL;
    PUT(HDRP(ptr), PACK(requiredSize, 1));
    PUT(FTRP(ptr), PACK(requiredSize, 1));
    PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 1));
    ptr = place(ptr, requiredSize);
}
```

- 最后的情况我们再使用之前的方法即可，这样则可以保证realloc函数的高效。

性能评估与优化

1. 优化

- 完成上述操作后得分为91。

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   99%    5694  0.000223  25511
 2       cccp-bal.rep       yes   99%    5848  0.000257  22781
 3    cp-decl-bal.rep       yes   99%    6648  0.000171  38877
 4       expr-bal.rep       yes   99%    5380  0.000121  44316
 5 coalescing-bal.rep       yes   99%   14400  0.000186  77336
 6     random-bal.rep       yes   95%    4800  0.000790   6080
 7    random2-bal.rep       yes   95%    4800  0.001019   4710
 8     binary-bal.rep       yes   60%   12000  0.000944  12713
 9    binary2-bal.rep       yes   81%   24000  0.000961  24964
10    realloc-bal.rep       yes   99%   14401  0.000123 116702
11   realloc2-bal.rep       yes   18%   14401  0.000145  99454
Total                            86%  112372  0.004941  22745

Score = (51 (util) + 40 (thru)) * 11/11 (testcase) = 91/100
```

- 不难发现，最后一个点得分的空间利用率很低，观察数据不难发现这里是一个大块一个小块，大块的大小接近于4096，小块则很小，然后删除小块，因此这就解释了一开始我们为什么要先分配出一块128大小的空间了，通过一个偏移量保证了每次的删除都不会有太大影响。同时我们在place中提到的一个技巧，如果插入的块太大了就把他放在尾巴上，这样在realloc可以直接通过申请内存来提高效率了，这样可以得到96分。

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   99%    5694  0.000417  13668
 2       cccp-bal.rep       yes   99%    5848  0.000197  29670
 3    cp-decl-bal.rep       yes   99%    6648  0.000172  38741
 4       expr-bal.rep       yes   99%    5380  0.000130  41448
 5 coalescing-bal.rep       yes   97%   14400  0.000203  70936
 6     random-bal.rep       yes   95%    4800  0.000858   5594
 7    random2-bal.rep       yes   95%    4800  0.000904   5312
 8     binary-bal.rep       yes   60%   12000  0.000948  12664
 9    binary2-bal.rep       yes   81%   24000  0.000756  31733
10    realloc-bal.rep       yes   99%   14401  0.000121 119017
11   realloc2-bal.rep       yes   99%   14401  0.000105 137152
Total                            93%  112372  0.004810  23364

Score = (56 (util) + 40 (thru)) * 11/11 (testcase) = 96/100
```

- 接下来就是针对于binary-bal的优化了，查看数据不难发现，binary-bal里是规模为64和448的交替插入，然后删除大块，因此我们可以继续调整place里空块的参数，我们把它设置为96，可以把64和448区分开，保证大小为448的分配块可以放在尾巴上，这样在后续的删除操作中就可以获取不错的性能，最终得到98分。

```
Results for mm malloc:
trace            name     valid  util     ops      secs   Kops
 1     amptjp-bal.rep       yes   99%    5694  0.000164  34783
 2       cccp-bal.rep       yes   99%    5848  0.000133  44103
 3    cp-decl-bal.rep       yes   99%    6648  0.000172  38584
 4       expr-bal.rep       yes   99%    5380  0.000124  43422
 5 coalescing-bal.rep       yes   97%   14400  0.000296  48616
 6     random-bal.rep       yes   95%    4800  0.000985   4873
 7    random2-bal.rep       yes   95%    4800  0.001100   4365
 8     binary-bal.rep       yes   91%   12000  0.000764  15713
 9    binary2-bal.rep       yes   81%   24000  0.000748  32077
10    realloc-bal.rep       yes   99%   14401  0.000113 126993
11   realloc2-bal.rep       yes   99%   14401  0.000141 102207
Total                            96%  112372  0.004740  23709

Score = (58 (util) + 40 (thru)) * 11/11 (testcase) = 98/100
```

2. 性能评估：
   在完成以上全部操作后得分为98分，这也是可以预料的。

- 首先，和隐式不同，这里我们只需要在八个链表中查询空闲块的大小即可，不用像之前那样进行$O(n)$的扫描，在于之前的隐式空间的结果图中通过对比可以发现吞吐率的极大提升。
- 同时我们在一个链表里维护了大小关系，即一个链表里从小到大存储，最终可以降低分配和释放的复杂度，因为这里所用的时间都是正比于查找到正确插入位置的时间的，而这里查找的插入时间与空闲块的数量有关，即$O(m)$，m为空闲块数量，不会涉及到已经分配的，搜索被限制在了堆的某一个部分，复杂度大大降低。
- 其次，我们采用了优化后的realloc函数，尽量向后合并与扩展而不是用最原始的方法进行，提高了重分配的效率，由58分提升至90。
- 最后从空间利用率的角度就是通过以上优化操作处理一些特殊情况，比如小块和大块相间隔分布等提高空间利用率到百分之九十六。

心得与感悟

1. 本次编程任务最为痛苦的一点就是在于16进制的对齐要求，以至于多次报错，WSIZE和DSIZE的使用不当也导致了这一点，以后可以一个一个函数逐一完成，否则整体调试的工作量太大，但这也提高了我对指针的运用与理解能力。
2. 其次是最后完成任务后并未得到高分，接下来参数的调整也是很重要的一步，通过阅读输入文件确定输入特点给出相应解决方案，提高了我的代码能力。
3. 当然本次实验也让我掌握了显式空闲链表和分离适配的方法，由于书上的简介比较简单一直没有深入了解，通过代码的完善提高了我的理解。