函钩 2025.7.24

实验要求我们编写一个适用于 c 语言程序的动态存储分配器,独立实现 malloc,free 和 realloc 函数.本实验鼓励我们多使用宏和自定义函数,尽可能少地使用全局变量.

首先简要介绍这三个函数的功能.

mm_malloc：声明 void *mm_malloc(size_t size);返回值为无类型指针.功能为根据用户请求的 size 分配对齐的内存块,并返回指向该块的指针.

mm_free：声明 void mm_free(void *ptr);无返回值.实现内存释放机制,防止内存泄漏.

mm_realloc：声明 void *mm_realloc(void *ptr, size_t size);返回值为无类型指针,实现内存重分配机制,调整已分配内存块的大小.

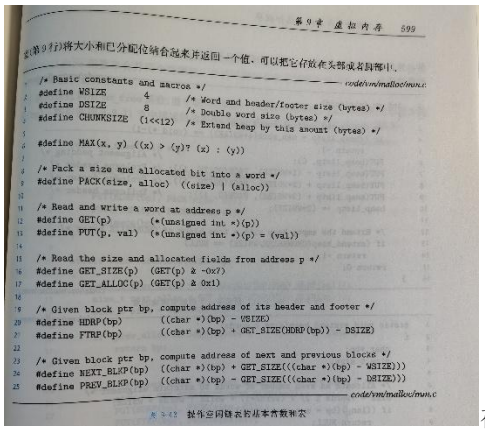
内存分配有隐式空闲链表,显示空闲链表,分离空闲链表.对于三种链表的性能,查找资料可以得知有如下情况:

维度	隐式空闲链表	显式空闲链表	分离空闲链表
查找时间(时间复杂度)	O(n)	O(n)	O(log n)
缓存未命中	极高	高	极低

所以本实验可以考虑使用分离空闲链表.分离空闲链表就是维护多个空闲链表,其中每个链表中的块有着大致相等的大小.一般的思路是将所有可能的块大小分成一些等价类,例如可以通过 2 的幂来划分块大小.

由此可见内存动态分配的核心结构是空闲链表.实验的核心思路就是操作链表组中的内存块进行动态内存分配(分离适配).分离适配的思路是,将请求块的大小进行分类,按照这个类找到适当的链表,并对适当的空闲链表做首次适配,查找一个合适的块.如果找到一个块,可以分割它,并将剩余的部分插入到适当的空闲链表中.若找不到就去寻找下一个链表.如果所有空闲链表中都没有,就想操作系统请求额外堆内存,从新的堆内存中分配出一个块,将剩余部分放在适当的大小类中.要释放一个块,就执行合并,并将结果放置到相应的空闲内存中.------部分摘抄自 CSAPP 9.9.14

实验开始前,我们需要定义一些宏,方便后续对块的头部脚部,前驱后继块等进行操作:



```
/* Basic constants and macros */
#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))
#define PUT(p, val)  (*(unsigned int *)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)       ((char *)(bp) - WSIZE)
#define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

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

在 CSAPP 课本的 page599 给出了一定的参考,我们可以进行一定补充.

补充后宏内容如下:

```
/* 单字（4 字节）或双字（8 字节）对齐,此处选择双字对齐,因此将对齐粒度定义为 8 字节.
```

```
   对齐操作可确保分配的内存地址是 ALIGNMENT 的整数倍,有助于提高内存访问效率.*/
#define ALIGNMENT 8
/* 向上舍入到最接近的 ALIGNMENT 的倍数 */
/* 此宏定义用于将给定的 size 值向上舍入到最接近的 ALIGNMENT（8 字节）的倍数.
   具体实现原理是：先将 size 加上 ALIGNMENT - 1,然后与 ~0x7 进行按位与操作.
   ~0x7 的二进制表示为 ...11111000,按位与操作会将低 3 位清零,从而实现 8 字节对齐.*/
#define ALIGN(size) (((size) + (ALIGNMENT - 1)) & ~0x7)
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define MIN(x, y) ((x) < (y) ? (x) : (y))
#define WSIZE 4 // 头部和脚部的大小
#define DSIZE 8 // 双字的大小

// 定义扩展堆时的默认大小
#define INITCHUNKSIZE (1 << 6)
#define CHUNKSIZE (1 << 12)
#define LISTMAX 16

/* 设置头部和脚部的值,将块大小和分配位组合在一起
 块大小（size）通常是一个整数（如 32 位或 64 位）.
分配状态（alloc）只需要一个二进制位（0 表示空闲,1 表示已分配）.*/
#define PACK(size, alloc) ((size) | (alloc))

/*读写指针 p 的位置*/
#define GET(p) (*(unsigned int *)(p))               // 将指针 p 转化为 unsigned int *
类型的指针,然后解引用获取该处存储的值
#define PUT(p, val) ((*(unsigned int *)(p)) = (val)) // 将指针 p 转化为 unsigned int *
类型的指针,然后将 val 赋值给该内存位置

/* 从头部或脚部获取大小或分配位 */
#define GET_SIZE(p) (GET(p) & ~0x7)
// 0x7 是...000111 反过来就是...111000,按位与操作会将低 3 位清零,从而获取头部的大小
#define GET_ALLOC(p) (GET(p) & 0x1)
// 0x1 是...000001,按位与操作只保留最后一位,从而获取 allocate 位

/* 给定有效载荷指针, 找到头部和脚部 */
#define HDRP(bp) ((char *)(bp) - WSIZE)
// 将 bp 转化为 char *类型的指针,然后减去头部的大小,从而找到头部的位置
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
// 将 bp 转化为 char *类型的指针,然后加上头部的大小,再减去脚部的大小,从而找到脚部的位置

#define SET_PTR(p, ptr) (*(unsigned int *)(p) = (unsigned int)(ptr))//设置指针 p 的值
为 ptr
/* 给定有效载荷指针, 找到前一块或下一块 */
```

```
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))//链表就像排队,下一块(NEXT)是你身后的人,前一块(PREV)是你面前的人,你比面前的人来的晚,面前的人来的比你早,你身后的人来的比你晚
// 将 bp 转化为 char *类型的指针,然后加上头部的大小,再减去脚部的大小,从而找到下一块的位置
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
// 将 bp 转化为 char *类型的指针,然后减去脚部的大小,再减去头部的大小,从而找到前一块的位置
#define PRED_PTR(ptr) ((char *)(ptr))//获取前驱指针
#define SUCC_PTR(ptr) ((char *)(ptr) + WSIZE)//获取后继指针

#define PRED(ptr) (*(char **)(ptr))//通过 ptr 获取前驱节点地址
#define SUCC(ptr) (*(char **)(SUCC_PTR(ptr)))//获取后继节点地址_successive
```

每一个宏的用途和用法在注释中有详细的说明.其中重要的新增宏有 ALIGNMENT(对齐粒度),ALIGN(将 size 向上扩展到能被 8 整除),SET_PTR(设置指针),PRED_PTR(获取前驱指针),SUCC_PTR(获取后继指针),PRED(获取前驱节点地址),SUCC(获取后继节点地址).这些宏在智能的块插入等函数中起到了重要的作用.

想要让动态内存分配高效运行,基础的函数是不可缺少的.分离空闲链表的包含了扩展,删除,合并,插入和分配五类基本操作,我们需要定义以下辅助函数:

```
static void *extend_heap(size_t size);
static void insert_node(void *ptr, size_t size);
static void *coalesce(void *ptr);
static void delete_node(void *ptr);
static void *place(void *ptr, size_t size);
```

extend_heap,返回值为无类型指针,当空闲链表中没有合适块时,用于扩展堆空间,输入参数为扩展空间的大小.

insert_node,无返回值,该函数将一个空闲块插入到分离空闲链表中,通过算法将空闲块按照大小插入到最合适的位置.

coalesce,返回值为无类型指针,用于合并空闲块.

delete_node,无返回值,用于删除节点.

place,该函数负责将空闲块分割并分配给用户,size 是用户要请求的内存大小.

extend_heap

```
static void *extend_heap(size_t size){/*该函数用于扩展堆空间*/
    void *ptr;
    /*内存对齐，将所需要的 size 向上取整为 8 的倍数*/
    size = ALIGN(size);
    /*系统调用 mem_sbrk 扩展堆  mem_sbrk - sbrk 函数的简单模型。
    通过 incr 字节扩展堆，并返回新区域的起始地址。在此模型中，堆不能收缩。*/
    ptr = mem_sbrk(size);
    if ( ptr == (void *)-1)//(void *)-1 表示错误状态，扩展失败
        return NULL
```

```
    /* 设置刚刚扩展的 free 块的头和尾 */
    PUT(HDRP(ptr), PACK(size, 0));//标记块 alloc=0 表示为空闲状态
    PUT(FTRP(ptr), PACK(size, 0));//头部和脚部存储相同信息 便于向前或向后合并空闲块

    /* 注意这个块是堆的结尾，所以还要设置一下结尾 */
    PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 1));/*`NEXT_BLKP`获取下一个块的指针
    哨兵块设计 ：
    大小 0 ：表示这是一个特殊块
    alloc=1 ：标记为已分配
    作用：防止堆越界访问，简化边界条件判断*/
    /* 扩展块完成后将其插入到分离空闲链表中 */
    insert_node(ptr, size);
    /* 另外这个 free 块的前面也可能是一个 free 块，可能需要合并 */
    return coalesce(ptr);//合并当前块与前后相邻的空闲块，减少内存碎片
}
```

extend_heap 函数设计思路如下.在实验中已经提供了一个 sbrk 的模拟函数叫做 mem_sbrk(无类型指针),通过 incr 字节扩展堆,并返回新区域的起始地址.extend_heap 函数首先对 size 进行内存的对齐(向上取整为 8 的倍数,此后所有涉及到 size 的函数都要进行内存对齐),然后通过 mem_sbrk 设置扩展的指针. `PUT(HDRP(ptr), PACK(size, 0));` `PUT(FTRP(ptr), PACK(size, 0));`这样为新扩展的块设置头部块和脚部块,这样就形成了一个双向链表.在新块的下一个块,还要设置一个 size 为 0,allocated 的哨兵块.哨兵块不储存实际数据,用于消除链表操作中的边界检查.获取扩展后的块,将块插入(insert_node)合适的链表.

insert_node

```
static void insert_node(void *ptr, size_t size)
/*该函数将一个空闲块插入到分离空闲链表中，首先根据块的大小找到对应的链表，
然后在该链表中寻找合适的插入位置，保证链表中块按大小从小到大排序，最后根据不同情况完成插入操作。*/
{
    int listnumber = 0; // 链表编号
    void *search_ptr = NULL;
    void *insert_ptr = NULL;

    /* 通过块的大小找到对应的链 */
    while ((listnumber < LISTMAX - 1) && (size > 1))
    {//通过右移操作计算块大小对应的链表编号
        size = size>> 1; /* size /= 2 */
        listnumber++;
    }

    /* 找到对应的链后，在该链中继续寻找对应的插入位置，以此保持链中块由小到大的特性 */
    search_ptr = segregated_free_lists[listnumber];
    while ((search_ptr != NULL) && (size > GET_SIZE(HDRP(search_ptr))))
```

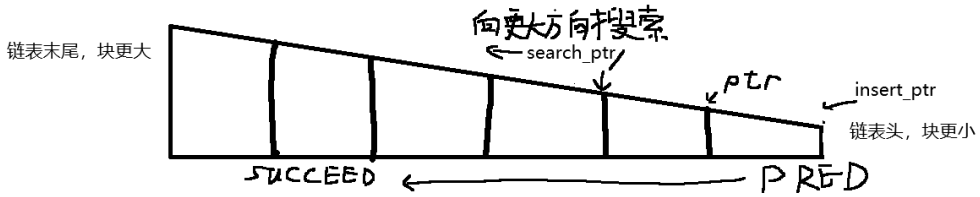
```
    { //   从链表头开始遍历            比较块大小，寻找第一个大于当前块的位置
        insert_ptr = search_ptr;
        search_ptr = PRED(search_ptr); // PRED 获取链表节点的前驱指针
    }

    /*插入操作： 循环后有四种情况 */
    if (search_ptr != NULL)
    {
        /* 1. ->xx->insert->xx 在中间插入*/
        if (insert_ptr != NULL)//search 和 insert 都不为空
        {
            SET_PTR(PRED_PTR(ptr), search_ptr);//ptr 的前一个节点是 search_ptr
            SET_PTR(SUCC_PTR(search_ptr), ptr);//search_ptr 的后一个节点是 ptr
            SET_PTR(SUCC_PTR(ptr), insert_ptr);//ptr 的后一个节点是 insert_ptr
            SET_PTR(PRED_PTR(insert_ptr), ptr);//insert_ptr 的前一个节点是 ptr
        }
        /* 2. [listnumber]->insert->xx 在开头插入，而且后面有之前的 free 块*/
        else//insert 为空，search 不为空
        {
            SET_PTR(PRED_PTR(ptr), search_ptr);//ptr 的前一个节点是 search_ptr
            SET_PTR(SUCC_PTR(search_ptr), ptr);//search_ptr 的后一个节点是 ptr
            SET_PTR(SUCC_PTR(ptr), NULL);//ptr 的后一个节点为空
            segregated_free_lists[listnumber] = ptr;//链表头指向 ptr
        }
    }
    else
    {
        if (insert_ptr != NULL)//search 为空，insert 不为空
        { /* 3. ->xxxx->insert 在结尾插入*/
            SET_PTR(PRED_PTR(ptr), NULL);//search_ptr 为空
            SET_PTR(SUCC_PTR(ptr), insert_ptr);//ptr 的后一个节点是 insert_ptr
            SET_PTR(PRED_PTR(insert_ptr), ptr);//insert_ptr 的前一个节点是 ptr
        }
        else//search 为空，insert 为空
        { /* 4. [listnumber]->insert 该链为空，这是第一次插入 */
            SET_PTR(PRED_PTR(ptr), NULL);
            SET_PTR(SUCC_PTR(ptr), NULL);
            segregated_free_lists[listnumber] = ptr;//insert 和 search 都为空，ptr 直接
成为链表头
        }
    }
}
```

要设计 insert_node 函数,我们首先要回顾分离适配方法的定义,将内存块插入到合适的链表中.所以我们需要定义并初始化 listnumber=0(链表序号).我们定义的链表是以 2 的幂次方进行分级的,所以要确定

listnumber,我们每当 size/=2,就给 listnumber++,直到 size=1 停止.这样就能确定$\log_2(size)$向下取整为大小的链表序号。

为了让整个程序保持一种一致性，我们需要将链表中的插入块也按照大小进行排序，这样能够提高分配效率。



我们通过定义两个指针 search_ptr 和 insert_ptr 来进行顺序插入.search_ptr 始终指向一个更大的节点,当下一个节点的 GET_SIZE>=size 时,就停止,insert_ptr 指向这个节点,然后进行插入操作.插入节点有四种情况,节点在表头,表中间,表末尾和空链表.

在表中间则直接设置头部脚部指针即可;在表头时则 ptr 设置为链表头;在表末尾时,search_ptr 为空,即 ptr 的前一个块为空,后一个块为 insert_ptr;在空链表中时,ptr 的前后块都为空,所以 ptr 要设置为链表头.指针设置的一个要点就是,要构造双向链表,保持链表结构的一致性.当前后都不为空时,要将"前指针设置为后指针的前指针,后指针设置为前指针的后指针",便于搜索.

coalesce

```
static void *coalesce(void *ptr)
{
    _Bool is_prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(ptr)));//查看前一个、后一个块是否已分配
    _Bool is_next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
    size_t size = GET_SIZE(HDRP(ptr));//从头部获取当前块的大小
    /* 根据 ptr 所指向块前后相邻块的情况，可以分为四种可能性 */
    /* 另外注意到由于我们的合并和申请策略，不可能出现两个相邻的 free 块 */
    /* 1.前后均为 allocated 块，不做合并，直接返回 */
    if (is_prev_alloc && is_next_alloc)
    {
        return ptr;
    }
    /* 2.前面的块是 allocated，但是后面的块是 free 的，这时将两个 free 块合并 */
    else if (is_prev_alloc && !is_next_alloc)
    {
        delete_node(ptr);
        delete_node(NEXT_BLKP(ptr));
        size += GET_SIZE(HDRP(NEXT_BLKP(ptr)));
        PUT(HDRP(ptr), PACK(size, 0));//将合并后的块信息更新到头部和脚部
        PUT(FTRP(ptr), PACK(size, 0));
    }
```

```
    /* 3.后面的块是 allocated，但是前面的块是 free 的，这时将两个 free 块合并 */
    else if (!is_prev_alloc && is_next_alloc)//    |prev 是 free|在此处合并|ptr 是
free|next 是 allocated|
    {
        delete_node(ptr);
        delete_node(PREV_BLKP(ptr));
        size += GET_SIZE(HDRP(PREV_BLKP(ptr)));
        PUT(FTRP(ptr), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(ptr)), PACK(size, 0));
        ptr = PREV_BLKP(ptr);
    }
    /* 4.前后两个块都是 free 块，这时将三个块同时合并 */
    else
    {
        delete_node(ptr);
        delete_node(PREV_BLKP(ptr));
        delete_node(NEXT_BLKP(ptr));
        size += GET_SIZE(HDRP(PREV_BLKP(ptr))) + GET_SIZE(HDRP(NEXT_BLKP(ptr)));
        PUT(HDRP(PREV_BLKP(ptr)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(ptr)), PACK(size, 0));
        ptr = PREV_BLKP(ptr);
    }

    /* 将合并好的 free 块加入到空闲链接表中 */
    insert_node(ptr, size);

    return ptr;
}
```

动态分配内存性能差的一个重要原因就是内存碎片多,内存块大小未排序,导致搜索效率下降.通过合并空闲块并重新插入合适位置,可以很好的防止内存碎片化.coalesce 函数也判断了四种合并块的可能性,前块和后块各自为 allocated 或 free.合并块需要删除当前块和相邻的空闲块(如果有),然后将 size 设置为合并后的大小,然后设置新的头部和脚部.因为块的大小已经改变,所以最后还要使用 insert_node 重新按顺序插入合并后的块.

delete_node

```
static void delete_node(void *ptr){/*删除节点*/
    int listnumber = 0;
    size_t size = GET_SIZE(HDRP(ptr));

    /* 通过块的大小找到对应的链 操作方式和 insert_node 相同*/
    while ((listnumber < LISTMAX - 1) && (size > 1)){
```

```
        size >>= 1;
        listnumber++;
    }

    /* 根据这个块的情况分四种可能性 前导指针和后继指针空或非空*/
    if (PRED(ptr) != NULL){
        /* 1. xxx-> ptr -> xxx 两者都非空，表示该块处于链表中间位置*/
        if (SUCC(ptr) != NULL){
            SET_PTR(SUCC_PTR(PRED(ptr)), SUCC(ptr));
            SET_PTR(PRED_PTR(SUCC(ptr)), PRED(ptr));
        }
        /* 2. [listnumber] -> ptr -> xxx */
        /*前导指针为非空，后继指针空，表示该块是链表的尾节点，有前驱节点没有后继节点 */
        else{
            SET_PTR(SUCC_PTR(PRED(ptr)), NULL);
            segregated_free_lists[listnumber] = PRED(ptr);
        }
    }
    else if(PRED(ptr) == NULL){
        /* 3. [listnumber] -> xxx -> ptr */
        if (SUCC(ptr) != NULL){
            SET_PTR(PRED_PTR(SUCC(ptr)), NULL);
        }
        /* 4. [listnumber] -> ptr */
        else{
            segregated_free_lists[listnumber] = NULL;//链表只有一个节点的话就直接删除该
链表
        }
    }
}
```

该函数负责删除节点.该函数搜索对应链表的方式和 insert_node 是相同的,都是循环/=2 得到链表序号.delete_node 的删除节点并不是真的把这个内存块中所包含的内容全部格式化,而是将想要删除的节点的前一个节点和后一个节点连起来:**ptr 的前节点的后继指针设置为 ptr 的后继指针,ptr 后节点的前驱指针设置为 ptr 的前驱指针**,链表只有一个块的情况就直接删除链表.整体思路与前面几个函数是相同的,都有表头,中间,末尾和独占链表几种情况.

删除的内存块的内容并没有被删除,去哪里了?节点中的内容仍保留在内存堆中,在之后调用 malloc 时这个块可能会被重新分配,其中的内容会被覆盖为新内容,不影响使用.

place

```
static void *place(void *ptr, size_t size){/*该函数负责将空闲块分割并分配给用户，size
是要请求的内存大小*/
```

```
    size_t ptr_size = GET_SIZE(HDRP(ptr));//获取 ptr 当前块大小
    /* allocate size 大小的空间后剩余的大小 */
    size_t remainder = ptr_size - size;
    delete_node(ptr);/*因为当一个空闲块被选中分配给用户时，必须立即从链表中删除*/

    /* 如果剩余的大小小于最小块，则不分离原块 */
    if (remainder < DSIZE * 2){
        PUT(HDRP(ptr), PACK(ptr_size, 1));//就是直接把 delete_note 删掉的东西原样放回去
        PUT(FTRP(ptr), PACK(ptr_size, 1));
    }
    else if (size >= 96){//特殊分割
        // 将原块前半部分设为空闲块
        PUT(HDRP(ptr), PACK(remainder, 0));
        PUT(FTRP(ptr), PACK(remainder, 0));
        // 将后半部分设为分配块
        PUT(HDRP(NEXT_BLKP(ptr)), PACK(size, 1));
        PUT(FTRP(NEXT_BLKP(ptr)), PACK(size, 1));
        // 将空闲部分重新插入链表
        insert_node(ptr, remainder);
        return NEXT_BLKP(ptr);
    }

    else {
        // 将当前块标记为已分配
        PUT(HDRP(ptr), PACK(size, 1));
        PUT(FTRP(ptr), PACK(size, 1));
        PUT(HDRP(NEXT_BLKP(ptr)), PACK(remainder, 0));
        PUT(FTRP(NEXT_BLKP(ptr)), PACK(remainder, 0));
        insert_node(NEXT_BLKP(ptr), remainder);
    }
    return ptr;
}
```

place 函数要将用户请求的块大小分配给用户.mm_malloc 函数会寻找大小合适的链表中的 free 块,找到合适的 free 块后会调用 place 函数来进行分割.为了避免浪费,需要将过大的块进行分割后再返回指针.首先要获取当前块的大小,然后计算去掉程序请求内存大小之后的剩余大小.我们通过一种先删除再还原剩余部分的方式,对块进行分割.

为了避免内存碎片化,若分割之后剩余块的大小过小,就不需要分割块.
当**请求**的块较大,要进行特殊分割,将块的前半部分(低地址)划为剩余,保持 free,后半部分划为已分配.
对于常规分割策略,要适用于大多情况,将块的前半部分(高地址)分配给用户,后半部分划为剩余,保持 free.

这样是为了尽量把相同大小的申请尽量放在一起,内存利用率高,碎片更少.最后要插入剩余节点,重新排序.

mm_malloc

```
void *mm_malloc(size_t size){
    if (size == 0)
        return NULL;

    size=ALIGN(size + DSIZE); /* 内存对齐 */
    int listnumber = 0;
    size_t searchsize = size;
    void *ptr = NULL;
    while (listnumber < LISTMAX){
        /* 寻找对应链 */
        if (((searchsize <= 1) && (segregated_free_lists[listnumber] != NULL))){
            ptr = segregated_free_lists[listnumber];
            /* 在该链寻找大小合适的 free 块 */
            while ((ptr != NULL) && ((size > GET_SIZE(HDRP(ptr))))){
                ptr = PRED(ptr);//对每个链表，通过 PRED(ptr) 遍历前向指针查找合适块,找到第一个大小≥请求 size 的块时跳出循环
            }
            /* 找到对应的 free 块 */
            if (ptr != NULL)
                break;
        }
        searchsize >>= 1;//size 每除以 2，就给 listnumber 加 1，用于确定链表块大小，searchsize 只有 1 和 0 两种情况，根据循环需求确定。
        listnumber++;//listnumber 就是 searchsize 除以 2 的次数
    }

    /* 没有找到合适的 free 块，扩展堆 */
    if (ptr == NULL){
        ptr = extend_heap(MAX(size, CHUNKSIZE));
        if ( ptr == NULL )
            return NULL;
    }

    /* 在 free 块中分配 size 大小的块 */
    ptr = place(ptr, size);
```

```
    return ptr;
}
```

mm_malloc 函数是动态内存分配的核心函数.从题目一开始给出的可以看出,传入的参数为 size,按照一贯的方法,我们要对 size 进行内存对齐.一个内存块包含头部(WSIZE),有效载荷(size),脚部(WSIZE)三部分,使用定义的宏 ALIGN(size+DSIZE)对齐内存块.定义一个 searchsize=size,然后按照和之前一样的方法,通过 searchsize 循环/=2 来计算 listnumber.

在循环的最后两次,也就是当 searchsize=1 或 0 的时候,分别对当前的序号的分离空闲链表进行检查,确定是否有一个合适的块.在链表和 ptr 都不为空的情况下,由于我们在安排链表时都是按照从小到大的顺序摆放块的,我们只需要让 ptr 从链表头依次向前找,直到找到一个大小>=size 的**空闲块**,此时若 ptr 是非空的,则停止循环.

若直到循环结束都没有找到合适的块,就要使用 extend_heap 扩展一个合适大小的块.若仍不能请求到内存块,则直接返回 NULL,表示内存不足无法请求.由于我们定义了 CHUNKSIZE,这是我们程序中应该被请求的最小的块大小,所以在扩展堆的过程中,要使用 MAX(size,CHUNKSIZE)获取大小.这样同样也是为了避免内存的碎片化.最后的最后,要使用 place 函数分配所需要的块,返回指针.

mm_free

```
void mm_free(void *ptr){//free 不是删除内存块，而是将内存块标记为未分配，再次使用时可以直接覆盖原内容
    size_t size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));//将头脚部设为未分配
    PUT(FTRP(ptr), PACK(size, 0));

    /* 将已释放的块插入分离空闲链表 */
    insert_node(ptr, size);
    /* 注意合并 */
    coalesce(ptr);
}
```

mm_free 为释放内存的函数,该函数将传入的指针对应的块标记为未分配.思路很简单,获取块大小,设置块为未分配,插入链表最后合并块.

mm_realloc

```
void *mm_realloc(void *ptr, size_t size){
    void *new_block = ptr;
    int remainder;

    if (size == 0)
        return NULL;
```

```
    /* 内存对齐 */
    if (size <= DSIZE){
        size = 2 * DSIZE;
    }
    else{
        size = ALIGN(size + DSIZE);
    }

    /* 如果 size 小于原来块的大小，直接返回原来的块 */
    if ((remainder = GET_SIZE(HDRP(ptr)) - size) >= 0){
        return ptr;
    }
    /* 否则先检查地址连续下一个块是否为 free 块或者该块是堆的**结束块**，因为我们要尽可能利用相邻的 free 块，以此减小“external fragmentation” */
    else if (!GET_ALLOC(HDRP(NEXT_BLKP(ptr))) || !GET_SIZE(HDRP(NEXT_BLKP(ptr))))/*获取下一个块的大小，堆结束块（哨兵块）：被设计为 大小为 0 的特殊块，作为堆的边界标记*//*空闲块或者是堆的结束块*/
    {
        /* 即使加上后面连续地址上的 free 块空间也不够，需要扩展块 */
        remainder = GET_SIZE(HDRP(ptr)) + GET_SIZE(HDRP(NEXT_BLKP(ptr))) - size;//剩余空间大小是当前块大小加上下一个块大小减去请求大小
        if (remainder < 0){/*如果大小不够请求*/
            if (extend_heap(MAX(-remainder, CHUNKSIZE)) == NULL)/*extend_heap 函数的类型是 void 指针*/
                return NULL;
            remainder += MAX(-remainder, CHUNKSIZE);
        }

        /* 删除刚刚利用的 free 块并设置新块的头尾 */
        delete_node(NEXT_BLKP(ptr));
        PUT(HDRP(ptr), PACK(size + remainder, 1));
        PUT(FTRP(ptr), PACK(size + remainder, 1));
    }
    /* 没有可以利用的连续 free 块，而且 size 大于原来的块，这时只能申请新的不连续的 free 块、复制原块内容、释放原块 */
    else{
        new_block = mm_malloc(size);//利用 memcpy 复制原内存块中内容

        memcpy(new_block, ptr, GET_SIZE(HDRP(ptr)));
        mm_free(ptr);
    }
    return new_block;
```

```
}
```

realloc 的核心功能是重新分配.实验开始预留的两个参数是*ptr 和 size,意为将 ptr 指向内存块设置为 size 大小.

在 realloc 函数中,我们可以猜测到有以下几种情况.

1:size 大小比原来还小

2:size 大小比原来大,且可以从当前块的 NEXT 的 free 块找到可用的空间

3:size 大小比原来大,且有 NEXT 的 free 块，但是当前块加上 NEXT 的 free 块,大小也不够用

4:size 大小比原来大,但是相邻没有 free 块

先定义一个无类型指针 new_block,最后用来作为新分配内存块的指针.然后是通常的对 size 的对齐和大小检查.确定剩余空间,若要再分配的大小小于当前大小,直接返回 ptr.否则检查地址连续的下一个块是否为 free 块或者哨兵块.

下一个块未分配(allocate 位=0)　　下一个块大小为 0(哨兵块)

```
!GET_ALLOC(HDRP(NEXT_BLKP(ptr))) || !GET_SIZE(HDRP(NEXT_BLKP(ptr)))
```

为了高效起见,我们需要尽可能不使用 mm_malloc,而是尽量通过操作当前链表中的内存块来调节再分配的大小.设置 `remainder = GET_SIZE(HDRP(ptr)) + GET_SIZE(HDRP(NEXT_BLKP(ptr))) - size;//剩余空间大小是当前块大小加上下一个块大小减去请求大小.`当然有可能大小不够用,若 remainder<0,就用 extend_heap(MAX(-remainder, CHUNKSIZE)) == NULL 来进行一个扩展,以及条件判断,若 extend_heap==NULL 则直接返回 NULL,若不满足条件,堆扩展也会产生一个合适大小的块. remainder += MAX(-remainder, CHUNKSIZE)计算得到扩展后块的大小.然后用通用的方法,先删掉原块,再设置新块.

若实在没有连续可用的 free 块,且 size 大于原来的块,就只能申请新的不连续的 free 块,复制原块内容,释放(mm_free(ptr))原块.此处可以使用 C 标准库 string.h 中的 memcpy 函数. memcpy(new_block, ptr, GET_SIZE(HDRP(ptr)))复制内存块到 new_block 中.

```
/*extern void *memcpy (void *__restrict __dest, const void *__restrict __src,
size_t __n) __THROW __nonnull ((1, 2));*/
        // 将 源内存块 （ __src ）的 前 __n 字节 复制到 目标内存块 （ __dest ），返回目标内存块指针
```

使用 make 对整个项目进行编译，./mdriver -v -t traces/指令进行案例测试，运行结果如下：

```
root@localhost:/home/malloclab# ./mdriver -v -t traces/
Team Name:TJU
Member 1 :FunctionHook:tjdx1225230685@tju.edu.cn
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.000174 32649
 1       yes   99%    5848  0.000173 33784
 2       yes   99%    6648  0.000247 26882
 3       yes   99%    5380  0.000195 27533
 4       yes   99%   14400  0.000341 42229
 5       yes   94%    4800  0.000244 19704
 6       yes   91%    4800  0.000234 20487
 7       yes   95%   12000  0.000268 44726
 8       yes   88%   24000  0.001818 13201
 9       yes   99%   14401  0.000162 88676
10       yes   98%   14401  0.000127113304
Total          96%  112372  0.003985 28199

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

测试案例总得分为 98 分。

实验完成，总共包含了 3 个核心函数，5 个基本操作函数，24 个自定义宏(包含原实验的 3 个宏)．

实验感想

本次实验深刻了解到了动态内存分配的实现方式，尤其是分离空闲链表这种途径，有着最高效的方式。

隐式空闲链表在运行时,每个块都要检查头部和脚部,每次循环都要访问两个元数据位置,且为强制遍历,缓存行利用率极低.显式空闲链表要给每个块分配多达 32 个字节的指针,潜在提高了内部碎片的程度．

动态分配内存的高效运行离不开代码中对诸多内存块的统一性管理，例如链表中从小到大安排内存块，合并较小的内存块并重新插入，对申请后内存块的分情况分割，确定可申请内存块的最小长度（CHUNKSIZE）以及及时释放不需要的内存。我们知道内存过度碎片化导致的缓存未命中惩罚是命中的十万倍，远远高于高速缓存的未命中惩罚，是完全不能容忍的现象。代码中所做的一切努力，一切办法都是为了减少内存碎片化,减少未命中惩罚。