malloclab_report

刘慕梵 19307130248

一、结果截图

```
lalala@ubuntu:~/ICS/lab/malloclab-handout$ ./mdriver -V -t trace
Team Name:lalala
Member 1 :刘慕梵:19307130248@fudan.edu.cn
Using default tracefiles in trace/
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
```

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000311 18320
 1       yes   99%    5848  0.000313 18672
 2       yes   99%    6648  0.000362 18344
 3       yes  100%    5380  0.000304 17727
 4       yes   99%   14400  0.000346 41606
 5       yes   95%    4800  0.001664  2884
 6       yes   95%    4800  0.001414  3393
 7       yes   55%   12000  0.000606 19792
 8       yes   51%   24000  0.001549 15494
 9       yes   97%   14401  0.000488 29504
10       yes   75%   14401  0.000265 54282
Total          88%  112372  0.007624 14740

Perf index = 53 (util) + 40 (thru) = 93/100
```

二、代码解释

结构解析在注释中

```c
/*
 * mm-naive.c - The fastest, least memory-efficient malloc package.
 *
 * In this naive approach, a block is allocated by simply incrementing
 * the brk pointer.  A block is pure payload. There are no headers or
 * footers.  Blocks are never coalesced or reused. Realloc is
 * implemented directly using mm_malloc and mm_free.
 *
 * NOTE TO STUDENTS: Replace this header comment with your own header
 * comment that gives a high level description of your solution.
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>

#include "mm.h"
#include "memlib.h"

/*********************************************************
 * NOTE TO STUDENTS: Before you do anything else, please
 * provide your team information in the following struct.
 ********************************************************/
team_t team = {
    /* Team name */
    "lalala",
    /* First member's full name */
```

```
    "刘慕梵",
    /* First member's email address */
    "19307130248@fudan.edu.cn",
    /* Second member's full name (leave blank if none) */
    "",
    /* Second member's email address (leave blank if none) */
    ""
};

/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))//对齐

#define WSIZE 4
#define DSIZE 8//分别是单字双字的大小

#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define PACK(size, alloc) ((size) | (alloc))//将大小和分配位合并，返回这个值

#define GET(p) (*(unsigned int*)(p))//返回p处的字
#define PUT(p, val) (*(unsigned int*)(p) = (val))//将val存放在p处

#define GET_SIZE(p) (GET(p) & ~0x7)//p是头部或脚部，从p处返回大小
#define GET_ALLOC(p) (GET(p) & 0x1)//p是头部或脚部，从p处返回分配位

#define HDRP(bp) ((char *)(bp) - WSIZE)//返回块的头部的指针
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)//返回块的尾部的指针

#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE((char *)(bp) - WSIZE))//返回指向下一个块的指针
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE((char *)(bp) - DSIZE))//返回指向上一个块的指针

#define NEXT_FREE(bp) ((char *)(bp) + WSIZE)//返回储存下一个空闲块地址的指针
#define PREV_FREE(bp) ((char *)(bp))//返回储存上一个空闲块地址的指针

#define PAGESIZE mem_pagesize()//页大小

static char* heap_listp;//堆中储存指向空闲链表表头的首指针
static char* heap_tailp;//堆中储存指向空闲链表表头的尾指针

static void* extend_heap(size_t words);//拓展堆
static void* realloc_coalesce(void *bp);//特定用于realloc函数中合并空闲块
static void* coalesce(void *bp);//立即合并相邻空闲块
static void* find_hit(size_t asize);//从空闲链表中找到合适的空闲块
static void place(void* bp, size_t asize);//放置块到空闲链表中
static void add_free_block(void* bp);//向空闲链表插入空闲块
static void* remove_free_block(void *bp);//从空闲链表移除空闲块
static char* find_list_root(size_t size);//返回对应等级的分离链表的根,参数为块大小
static char* get_list_root(void *bp);//返回对应等级的分离链表的根，参数为块指针
static int Check(void* bp, int mod);//check函数
/*
 * mm_init - initialize the malloc package.
```

```c
 * 堆的组织方式为分离式空闲链表，一共设置了9个大小的空闲块链表，分别为32B以下，64B以下，128B以下，256B以下，512B以下，
 * 1024B以下，2048B以下，4096B以下，以及4096B以上，除了4096B以上（第8级）的和32B以下（第0级）的，第i级链表中空闲块大小b为
 * 2^(i+4) < b <= 2^(i+5)。初始化链表有12个块，第0到8块储存第0到8级链表的表头指针，第9、10块为序言块，11块为堆尾块，
 * 用来标志堆开头和结束。heap_listp指向第0级链表，heap_tailp指向第一个序言块。分配块与空闲块有4字节的块头与块尾，
 * 空闲块在块头后面还有两个4字节的指针，分别指向上一个空闲块与下一个空闲块，空闲块按块大小排序。
 */
int mm_init(void)
{
    char *ptr;
    if((heap_listp = mem_sbrk(12 * WSIZE)) == (void *)-1)//获得12个块的大小存放初始信息
    {
        return -1;
    }
    PUT(heap_listp, 0);//小于32B块的链表
    PUT(heap_listp + WSIZE, 0);//小于64B块的链表
    PUT(heap_listp + (2 * WSIZE), 0);//小于128B块的链表
    PUT(heap_listp + (3 * WSIZE), 0);//小于256B块的链表
    PUT(heap_listp + (4 * WSIZE), 0);//小于512B块的链表
    PUT(heap_listp + (5 * WSIZE), 0);//小于1024B块的链表
    PUT(heap_listp + (6 * WSIZE), 0);//小于2048B块的链表
    PUT(heap_listp + (7 * WSIZE), 0);//小于4096B块的链表
    PUT(heap_listp + (8 * WSIZE), 0);//大于4096B块的链表
    PUT(heap_listp + (9 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (10 * WSIZE), PACK(DSIZE, 1));//两个序言块
    PUT(heap_listp + (11 * WSIZE), PACK(0, 1));//堆末尾标识块
    heap_tailp = heap_listp + 9 * WSIZE;//指向链表尾

    if(extend_heap(PAGESIZE / DSIZE) == NULL)
        return -1;
    return 0;
}

/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 * 实现方式与隐式空闲链表类似，主要通过find_hit与place函数实现
 */
void *mm_malloc(size_t size)
{
    if(size == 0)
        return NULL;
    size_t newsize = ALIGN(size + SIZE_T_SIZE);//对齐
    char *bp;

    if((bp = find_hit(newsize)) != NULL)//找到合适的块
    {
        place(bp, newsize);//放置块
        return bp;
    }

    //如果找不到，则拓展堆
    size_t extendsize = MAX(newsize, PAGESIZE);//拓展大小为两者较大值
```

```c
    if((bp = extend_heap(extendsize / DSIZE)) == NULL)
        return NULL;
    place(bp, newsize);//放置块
    return bp;
}

/*
 * mm_free - Freeing a block does nothing.
 * 释放块，先将两个指针设置为0，再合并。
 */
void mm_free(void *ptr)
{
    if(ptr == NULL)
        return;

    size_t size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));//设置分配位为0
    PUT(NEXT_FREE(ptr), 0);
    PUT(PREV_FREE(ptr), 0);//将指向前后两个空闲块的两个指针设置为0

    coalesce(ptr);//立即合并相邻空闲块
}

/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 * 如果ptr为NULL，调用mm_malloc，如果size为0，调用mm_free。
 * 如果重新分配的size小于等于原来的size，则直接返回原来的块指针
 * 如果重新分配的size大于原来的size，首先调用realloc_coalesce尝试将ptr与相邻空闲块连接成一个更大的块，暂时不加入空闲链表，
 * 比较新块的大小和需要重新分配的大小，如果匹配，则直接将整个新块作为新分配的块，并将数据拷贝。
 * 如果不匹配，则调用mm_malloc得到需要的大小的块，将数据拷贝进去，并将之前合并得到的块加入空闲链表。
 */
void *mm_realloc(void *ptr, size_t size)
{
    if(ptr == NULL)//对特殊输入的处理
        return mm_malloc(size);
    else if(size == 0)
    {
        mm_free(ptr);
        return NULL;
    }

    size_t oldsize = GET_SIZE(HDRP(ptr)), asize = ALIGN(size + SIZE_T_SIZE), newsize;//分别是原本块大小，需要分配块的大小和新合并块的大小
    char *newfree, *newptr;//分别是指向新合并块的指针以及最后返回的指针
    if(asize <= oldsize)//需要的大小小于原来块的大小
        return ptr;
    else//需要的大小大于原来块的大小
    {
        PUT(HDRP(ptr), PACK(oldsize, 0));
        PUT(FTRP(ptr), PACK(oldsize, 0));//将原本的块设置为空闲块
        newfree = realloc_coalesce(ptr);//调用realloc的合并函数
        newsize = GET_SIZE(HDRP(newfree));
        if(newsize >= asize)//新块的大小大于等于需要重新分配的大小，直接分配整个新块
```

```c
        {
            memmove(newfree, ptr, oldsize - DSIZE);//拷贝数据
            PUT(HDRP(newfree), PACK(newsize, 1));
            PUT(FTRP(newfree), PACK(newsize, 1));//设置为已分配块
            newptr = newfree;
        }
        else//新块的大小小于需要重新分配的大小
        {
            newptr = mm_malloc(asize);//调用mm_malloc函数
            memmove(newptr, ptr, oldsize - DSIZE);//拷贝数据
            add_free_block(newfree);//将原本合并得到的新块加入空闲链表
        }
    }
    return newptr;
}

static void* extend_heap(size_t words)//拓展堆，与书上类似
{
    char *bp;
    size_t size;

    size = (words % 2)? (words + 1) * DSIZE : words * DSIZE;//需要拓展的字节
    if((bp = mem_sbrk(size)) == (void *)-1 )
        return NULL;

    PUT(HDRP(bp), PACK(size, 0));//设置新分配的堆块的头部
    PUT(FTRP(bp), PACK(size, 0));//设置尾部
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));//设置堆末尾标识
    PUT(NEXT_FREE(bp), 0);
    PUT(PREV_FREE(bp), 0);//两个指针设置为0

    return coalesce(bp);//与前面的空闲块合并
}

static void* realloc_coalesce(void *bp)//用于mm_realloc函数中使用的合并函数，其合并后不插入到空闲链表中
{
    size_t prev_alloc, next_alloc, size;

    prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));//前一个块的分配位
    next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));//后一个块的分配位
    size = GET_SIZE(HDRP(bp));//最后空闲块的大小

    if(prev_alloc && next_alloc);//独立的空闲块
    else if(!prev_alloc && next_alloc)//前面有空闲块而后面没有
    {
        remove_free_block(PREV_BLKP(bp));//从空闲链表中移除前面空闲块
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        bp = PREV_BLKP(bp);//设置新的空闲块
    }
    else if(prev_alloc && !next_alloc)//前面没有空闲块而后面有
    {
        remove_free_block(NEXT_BLKP(bp));//从空闲链表中移除后面空闲块
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));//设置新的空闲块
```

```
    }
    else//前后都有空闲块
    {
        remove_free_block(PREV_BLKP(bp));//从空闲链表中移除前面空闲块
        remove_free_block(NEXT_BLKP(bp));//从空闲链表中移除后面空闲块
        size += GET_SIZE(HDRP(NEXT_BLKP(bp))) + GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);//设置新的空闲块
    }
    return bp;
}

static void* coalesce(void *bp)//合并空闲块，其就比realloc_coalesce函数多了加入空闲链表的部分
{
    char *newbp = realloc_coalesce(bp);
    add_free_block(newbp);//新空闲块插入空闲链表
    return newbp;
}

static void* find_hit(size_t asize)//找到合适的块
{
    char *bp;
    char *headptr = find_list_root(asize);//指向储存对应大小链表表头的指针，即等级为i的块
    //首次适配法， 也是最佳适配法， 因为空闲块按照块大小排序
    for(; headptr != heap_tailp; headptr += WSIZE)//从合适的最小等级开始搜索，按各个分离链表依次搜索
    {
        for(bp = GET(headptr); bp != NULL; bp = GET(NEXT_FREE(bp)))//从空闲链表从头开始搜索
        {
            if(GET_SIZE(HDRP(bp)) >= asize)
                return bp;
        }
    }
    return NULL;
}

static void place(void* bp, size_t asize)//放置块
{
    size_t size = GET_SIZE(HDRP(bp));
    size_t leftsize = size - asize;

    remove_free_block(bp);//移除该空闲块
    if(leftsize >= 2 * DSIZE)//如果分割后剩下的块大小大于等于最小块大小
    {
        PUT(HDRP(bp), PACK(asize, 1));//前面部分设置为已分配
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(leftsize, 0));//后面块设置为空闲
        PUT(FTRP(bp), PACK(leftsize, 0));
        PUT(NEXT_FREE(bp), 0);
        PUT(PREV_FREE(bp), 0);//后面块指针设置为0
        add_free_block(bp);//将分割后的空闲块加入空闲链表
    }
```

```
    else//否则整个块设置为已分配
    {
        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    }
}

static void add_free_block(void *bp)//插入空闲链表
{
    char *headptr = get_list_root(bp);//得到对应大小链表的根
    char *head = GET(headptr);//对应大小链表的表头

    if(head == NULL)//空的空闲链表，将bp加入该链表
    {
        PUT(headptr, bp);
        PUT(NEXT_FREE(bp), 0);
        PUT(PREV_FREE(bp), 0);
        return;
    }

    size_t insert_size = GET_SIZE(HDRP(bp));//bp的快大小
    char *nextbp = head, *prevbp = headptr;//插入时的遍历搜索变量
    for(; nextbp != NULL; nextbp = GET(NEXT_FREE(nextbp)))//循环寻找合适的插入位置
    {
        if(GET_SIZE(HDRP(nextbp)) >= insert_size)
            break;
        prevbp = nextbp;//prevbp指向nextbp前面的块
    }
    if(nextbp == head)//插入空闲块最小，插入在表头
    {
        PUT(headptr, bp);//将表头设置为bp
        PUT(NEXT_FREE(bp), nextbp);
        PUT(PREV_FREE(bp), 0);
        PUT(PREV_FREE(nextbp), bp);
    }
    else//插入在中间或在表尾
    {
        PUT(NEXT_FREE(prevbp), bp);
        PUT(PREV_FREE(bp), prevbp);
        PUT(NEXT_FREE(bp), nextbp);
        if(nextbp != NULL)//在中间
            PUT(PREV_FREE(nextbp), bp);
    }
    return;
}

static void* remove_free_block(void *bp)//从空闲链表中移除空闲块
{
    char *headptr = get_list_root(bp);
    char *prev_free = GET(PREV_FREE(bp));//指向上一个空闲块的指针
    char *next_free = GET(NEXT_FREE(bp));//指向下一个空闲块的指针

    if(!prev_free && !next_free)//唯一的空闲块
        PUT(headptr, 0);
    else if(prev_free && !next_free)//末尾空闲块
        PUT(NEXT_FREE(prev_free), 0);
    else if(!prev_free && next_free)//队首空闲块
```

```
    {
        PUT(PREV_FREE(next_free), 0);
        PUT(headptr, next_free);
    }
    else//中间空闲块
    {
        PUT(NEXT_FREE(prev_free), next_free);
        PUT(PREV_FREE(next_free), prev_free);
    }
    PUT(PREV_FREE(bp), 0);
    PUT(NEXT_FREE(bp), 0);
    return bp;
}

static char* find_list_root(size_t size)//从对应大小得到对应等级链表的根
{
    int level = 0;
    if(size <= 32) level = 0;
    else if(size <= 64) level = 1;
    else if(size <= 128) level = 2;
    else if(size <= 256) level = 3;
    else if(size <= 512) level = 4;
    else if(size <= 1024) level = 5;
    else if(size <= 2048) level = 6;
    else if(size <= 4096) level = 7;
    else level = 8;
    return heap_listp + WSIZE * level;//对应等级链表的根
}

static char* get_list_root(void *bp)//find_list_root的包装
{
    size_t size = GET_SIZE(HDRP(bp));
    return find_list_root(size);
}

static int Check(void* bp, int mod)
{
    switch(mod)
    {
        case 0://返回的地址检查，如果不为8对齐，则出错
        {
            if((unsigned int)bp != (unsigned int)bp & ~0x7)
            {
                fprintf(stderr, "not 8 Bytes align at address: %p\n", bp);
                return 1;
            }
            break;
        }
        case 1://检查bp是否标志设置为0
        {
            if(GET_ALLOC(HDRP(bp)))
            {
                fprintf(stderr, "free block(%p) did not set alloc free\n", bp);
                return 1;
            }
            break;
        }
```

```c
        case 2://检查是否空闲链表中每一块标志都设置为0
        {
            for(char *headptr = heap_listp; headptr != heap_tailp; headptr += WSIZE)
            {
                for(char* fbp = GET(headptr); fbp != NULL; fbp = GET(NEXT_FREE(fbp)))
                {
                    if(GET_ALLOC(HDRP(fbp)) || GET_ALLOC(FTRP(fbp)))
                    {
                        fprintf(stderr, "free block(%p) in list did not set alloc free\n", fbp);
                        return 1;
                    }
                }
            }
            break;
        }
        case 3://检查bp是否加入到空闲块链表
        {
            if(GET_ALLOC(HDRP(bp)))
                return 0;
            char *headptr = get_list_root(bp);
            for(char *cmpbp = GET(headptr); cmpbp != NULL; cmpbp = GET(NEXT_FREE(cmpbp)))
            {
                if(cmpbp == bp)
                    return 0;
            }
            fprintf(stderr, "free block(%p) is not in list\n", bp);
            return 1;
        }
        case 4://检查是否每个空闲块都加入到了空闲链表
        {
            for(char *nowbp = heap_tailp + 3 * WSIZE; GET_SIZE(HDRP(nowbp)) != 0; nowbp = NEXT_BLKP(nowbp))
            {
                if(GET_ALLOC(HDRP(nowbp)))
                    continue;
                char *headptr = get_list_root(nowbp);
                for(char *cmpbp = GET(headptr); cmpbp != NULL; cmpbp = GET(NEXT_FREE(cmpbp)))
                {
                    if(cmpbp == nowbp)
                        return 0;
                }
                fprintf(stderr, "free block(%p) is not in list\n", nowbp);
                return 1;
            }
            break;
        }
        case 5://用于realloc中，打印该地址下的内容
        {
            static int realloc_checktimes = 0;
            printf("%d:\n",realloc_checktimes);
            realloc_checktimes++;
            size_t size = GET_SIZE(HDRP(bp)) - DSIZE;
            for(size_t i = 0; i < size; i++)
```

```c
                    printf("%d ",*(char *)(bp + i));
                printf("\n\n");
                fflush(stdout);
                return 0;
                break;
            }
    }
    return 0;
}
```