Lab4 动态存储器分配优化实验

李仁杰 13307130279

一、实验目的

1.了解分配器的工作原理并编程实现；

2.学习程序优化，提高代码能力。

二、实验内容

这个 Lab 的要求是自己实现类似 GNU Libc 的 malloc 和 free 函数，也就是实现一个动态内存分配器，让你亲手管理一个程序的堆内存分配。最后会从吞吐量（单位时间可执行次数）和空间利用率两个方面进行评估。

三、实验原理

1.教材中介绍了空闲链表的形式、分配策略，以及每个堆块的布局。

组织策略：

(1)隐式空闲链表：只记录每个块的大小，分配时遍历整个堆寻找大小适合的块；

(2)显式空闲链表：在空闲块中额外记录前一个、后一个空闲块的位置，可以节省遍历时间；

(3)分离空闲链表：维护多个链表，将不同大小类的块分到同一个链表中。

选择策略：

(1)首次适配：选择第一个合适的块；

(2)下次适配：每次搜索从上次结束的地方开始；

(3)最佳适配：选择大小合适的最小块。

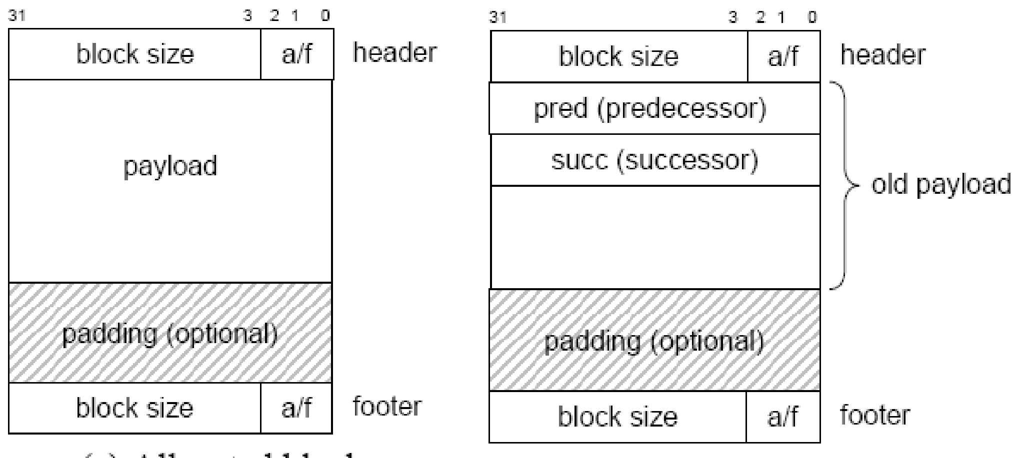
分割策略：确定分配块的大小（找到块后，是否分割令剩余块并入空闲块）；合并策略：立即合并，推迟合并。

2.显式空闲链表分配器

(1)采用首次适配策略；

(2)边界标记立即合并方式；

(3)双向单链表结构。



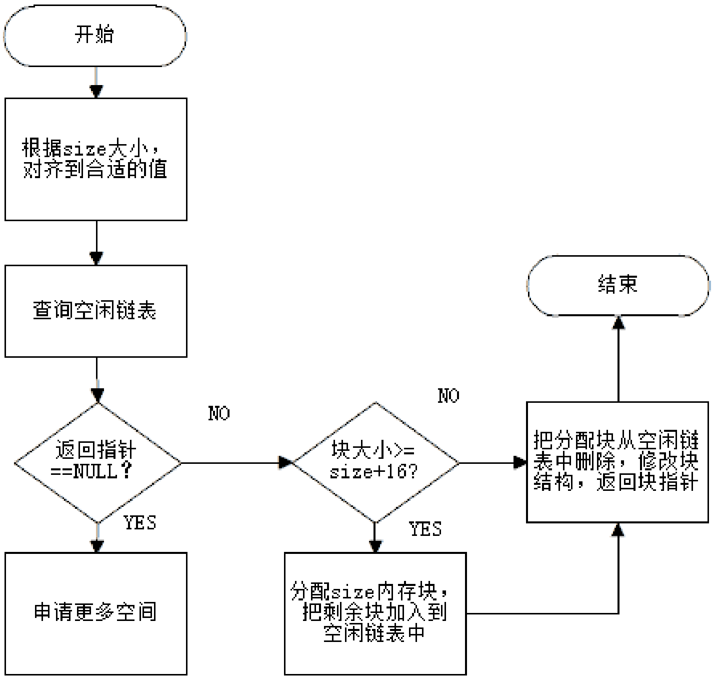
(a) Allocated block (b) Free block

四、实验过程

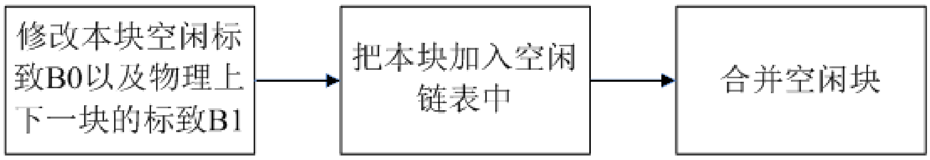
首先直接将教材上的隐式空闲链表的例子写入 mm.c 程序，经过运行，发现得分为 45(util)+0(thru)。经过分析，认为因为没有进行任何的优化，导致程

序运行过慢，数据的吞吐量收到极大的限制。

mm_malloc 程序的基本框图如下：



mm_free 程序的基本框图如下：



接着我改用显式空闲链表分配器的方法。我们正常的是把前面一小块分配出来放数据（allocated），而留下后面一半作为新的 free block。这样做在隐式的情况下是不会有明显问题的，但是因为在显式空闲链表中，我们的双向链表都存放在 free block 的靠近头的位置，如果把前面的小块分配出去了，我们必须把原先指向这整块 free block 的链表指针重新指向剩下的后面一小块。我们的时间基本上都耗费在这样没有意义的移动指针上了，因为如果我们是选择把后面的小块分配出去，是根本就不用改变链表的结构的。

后来我发现，遍历速度慢也是一个很重要的影响程序性能的地方。因为链表长度很长，所以我们也可以通过想办法把链表长度降下来来提高速度。怎么降？这个时候就要用上分离的空闲列表了，把不同长度区间的 block 分开在不同的链表中，这样链表的长度自然就低了，关键是要分得均匀，让每条表长度均衡，我的分组方法是

{1} {2～3} {4～7} {8～15} {16～31}…… {2048～4095} {4096～8191} {8192～12287} {12288～16383}…… {28672～32767} {32768～65535} {65536～131071} {131072～262143} {262144～524287} {524287～无穷大}

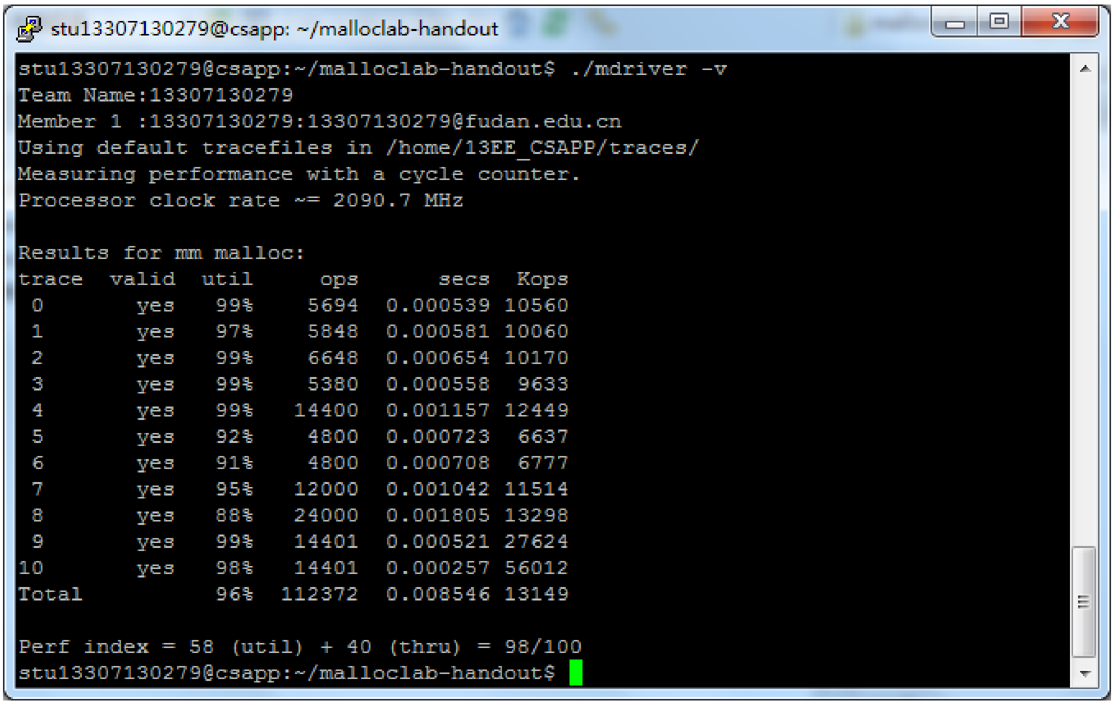
这样根据书上内容改进之后，程序性能大大提升。

五、实验结果

最近我的 Ubuntu Linux 系统在更新的时候崩溃了，在做这个 Lab 的时候还

没来得及重装，所以这次采用的是 PUTTY 调试。先通过 WinSCP 软件将 mm.c 文件上传到 10.92.13.14 的目录下，然后通过 PUTTY 登录到自己的目录下，执行如下指令：

```
unix> make clean
unix> make mdriver
unix> ./mdriver -v
```



之后可以得到如上图的运行结果。可以看出来，程序性能良好。

六、心得和体会

这次的实验明显比前几次要难得多，代码长度也比以往要长，调试过程极其繁琐，花在这上面的时间也较长。不过这个实验也让我能够在课下用自己的方法了解书中的原理，收获还是蛮大的。

这次实验的文件也会在截止日期 6 月 7 日 12 点之后在我的 github 上开源。地址 https://github.com/lirenjie95/CSAPP

附录 mm.c

```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>
```

```c
#include <assert.h>
#include <unistd.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 */
    "13307130279",
    /* First member's full name */
    "13307130279",
    /* First member's email address */
    "13307130279@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 WSIZE     4         // word and header/footer size (bytes)
#define DSIZE     8         // double word size (bytes)
#define INITCHUNKSIZE (1<<6)
#define CHUNKSIZE (1<<12)
#define LEN     20
#define REALLOC_BUFFER  (1<<7)
#define MAX(x, y) ((x)>(y)?(x):(y))
#define MIN(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) | GET_TAG(p))
#define PUT_NOTAG(p, val) (*(unsigned int *)(p) = (val))
// Store predecessor or successor pointer for free blocks
#define SET_BP(p, bp) (*(unsigned int *)(p) = (unsigned int)(bp))
// Read the size and allocation bit from address p
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)
#define GET_TAG(p)   (GET(p) & 0x2)
```

```c
#define SET_RATAG(p)   (GET(p) |= 0x2)
#define REMOVE_RATAG(p) (GET(p) &= ~0x2)
// Address of block's header and footer
#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
// Address of (physically) 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))
// Address of free block's predecessor and successor entries
#define PRED_BP(bp) ((char *)(bp))
#define SUCC_BP(bp) ((char *)(bp) + WSIZE)
// Address of free block's predecessor and successor on the segregated list
#define PRED(bp) (*(char **)(bp))
#define SUCC(bp) (*(char **)(SUCC_BP(bp)))
/* Below is the declaration of functions*/
void *heap_list[LEN];
static void *extend_heap(size_t size);
static void *coalesce(void *bp);
static void *place(void *bp, size_t asize);
static void insert_node(void *bp, size_t size);
static void delete_node(void *bp);
/* Functions*/
static void *extend_heap(size_t size)
{
    void *bp;
    size_t asize;
    asize = ALIGN(size);
    if ((bp = mem_sbrk(asize)) == (void *)-1)
        return NULL;
    PUT_NOTAG(HDRP(bp), PACK(asize, 0));
    PUT_NOTAG(FTRP(bp), PACK(asize, 0));
    PUT_NOTAG(HDRP(NEXT_BLKP(bp)), PACK(0, 1));
    insert_node(bp, asize);
    return coalesce(bp);
}
static void insert_node(void *bp, size_t size) {
    int list = 0;
    void *search_bp = bp;
    void *insert_bp = NULL;
    while ((list < LEN - 1) && (size > 1)) {
        size >>= 1;
        list++;
    }
    search_bp = heap_list[list];
```

```
    while ((search_bp != NULL) && (size > GET_SIZE(HDRP(search_bp)))) {
        insert_bp = search_bp;
        search_bp = PRED(search_bp);
    }
    if (search_bp != NULL) {
        if (insert_bp != NULL) {
            SET_BP(PRED_BP(bp), search_bp);
            SET_BP(SUCC_BP(search_bp), bp);
            SET_BP(SUCC_BP(bp), insert_bp);
            SET_BP(PRED_BP(insert_bp), bp);
        } else {
            SET_BP(PRED_BP(bp), search_bp);
            SET_BP(SUCC_BP(search_bp), bp);
            SET_BP(SUCC_BP(bp), NULL);
            heap_list[list] = bp;
        }
    } else {
        if (insert_bp != NULL) {
            SET_BP(PRED_BP(bp), NULL);
            SET_BP(SUCC_BP(bp), insert_bp);
            SET_BP(PRED_BP(insert_bp), bp);
        } else {
            SET_BP(PRED_BP(bp), NULL);
            SET_BP(SUCC_BP(bp), NULL);
            heap_list[list] = bp;
        }
    }
    return;
}
static void delete_node(void *bp) {
    int list = 0;
    size_t size = GET_SIZE(HDRP(bp));
    while ((list < LEN - 1) && (size > 1)) {
        size >>= 1;
        list++;
    }
    if (PRED(bp) != NULL) {
        if (SUCC(bp) != NULL) {
            SET_BP(SUCC_BP(PRED(bp)), SUCC(bp));
            SET_BP(PRED_BP(SUCC(bp)), PRED(bp));
        } else {
            SET_BP(SUCC_BP(PRED(bp)), NULL);
            heap_list[list] = PRED(bp);
        }
```

```c
    } else {
        if (SUCC(bp) != NULL) {
            SET_BP(PRED_BP(SUCC(bp)), NULL);
        } else {
            heap_list[list] = NULL;
        }
    }
    return;
}
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));
    // Do not coalesce with previous block if the previous block is tagged with
Reallocation tag
    if (GET_TAG(HDRP(PREV_BLKP(bp))))
        prev_alloc = 1;
    if (prev_alloc && next_alloc) {                         // Case 1
        return bp;
    }
    else if (prev_alloc && !next_alloc) {                   // Case 2
        delete_node(bp);
        delete_node(NEXT_BLKP(bp));
        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) {                 // Case 3
        delete_node(bp);
        delete_node(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 {                                                // Case 4
        delete_node(bp);
        delete_node(PREV_BLKP(bp));
        delete_node(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_node(bp, size);
```

```
    return bp;
}
static void *place(void *bp, size_t asize)
{
    size_t bp_size = GET_SIZE(HDRP(bp));
    size_t remainder = bp_size - asize;
    delete_node(bp);
    if (remainder <= DSIZE * 2) {
        // Do not split block
        PUT(HDRP(bp), PACK(bp_size, 1));
        PUT(FTRP(bp), PACK(bp_size, 1));
    }
    else if (asize >= 100) {
        // Split block
        PUT(HDRP(bp), PACK(remainder, 0));
        PUT(FTRP(bp), PACK(remainder, 0));
        PUT_NOTAG(HDRP(NEXT_BLKP(bp)), PACK(asize, 1));
        PUT_NOTAG(FTRP(NEXT_BLKP(bp)), PACK(asize, 1));
        insert_node(bp, remainder);
        return NEXT_BLKP(bp);

    }
    else {
        // Split block
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        PUT_NOTAG(HDRP(NEXT_BLKP(bp)), PACK(remainder, 0));
        PUT_NOTAG(FTRP(NEXT_BLKP(bp)), PACK(remainder, 0));
        insert_node(NEXT_BLKP(bp), remainder);
    }
    return bp;
}
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    int list;
    char *heap_start; // Pointer to beginning of heap
    // Initialize segregated free lists
    for (list = 0; list < LEN; list++) {
        heap_list[list] = NULL;
    }
```

```
    // Allocate memory for the initial empty heap
    if ((long)(heap_start = mem_sbrk(4 * WSIZE)) == -1)
        return -1;
    PUT_NOTAG(heap_start, 0);                          /* Alignment padding */
    PUT_NOTAG(heap_start + (1 * WSIZE), PACK(DSIZE, 1)); /* Prologue header */
    PUT_NOTAG(heap_start + (2 * WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
    PUT_NOTAG(heap_start + (3 * WSIZE), PACK(0, 1));     /* Epilogue header */
    if (extend_heap(INITCHUNKSIZE) == 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.
 */
void *mm_malloc(size_t size)
{
    size_t asize;      /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    void *bp = NULL;  /* Pointer */
    // Ignore size 0 cases
    if (size == 0)
        return NULL;
    // Align block size
    if (size <= DSIZE) {
        asize = 2 * DSIZE;
    } else {
        asize = ALIGN(size+DSIZE);
    }
    int list = 0;
    size_t searchsize = asize;
    // Search for free block in segregated list
    while (list < LEN) {
        if ((list == LEN - 1) || ((searchsize <= 1) && (heap_list[list] !=
NULL))) {
            bp = heap_list[list];
            // Ignore blocks that are too small or marked with the reallocation
bit
            while ((bp != NULL) && ((asize > GET_SIZE(HDRP(bp))) ||
(GET_TAG(HDRP(bp)))))
            {
                bp = PRED(bp);
            }
            if (bp != NULL)
```

```
                break;
            }
            searchsize >>= 1;
            list++;
        }
        // if free block is not found, extend the heap
        if (bp == NULL) {
            extendsize = MAX(asize, CHUNKSIZE);

            if ((bp = extend_heap(extendsize)) == NULL)
                return NULL;
        }
        // Place and divide block
        bp = place(bp, asize);
        // Return pointer to newly allocated block
        return bp;
    }
    /*
     * mm_free - Freeing a block does nothing.
     */
    void mm_free(void *bp)
    {
        size_t size = GET_SIZE(HDRP(bp));
        REMOVE_RATAG(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        insert_node(bp, size);
        coalesce(bp);
        return;
    }
    /*
     * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
     */
    void *mm_realloc(void *bp, size_t size)
    {
        void *new_bp = bp;    /* Pointer to be returned */
        size_t new_size = size; /* Size of new block */
        int remainder;          /* Adequacy of block sizes */
        int extendsize;         /* Size of heap extension */
        int block_buffer;       /* Size of block buffer */
        // Ignore size 0 cases
        if (size == 0)
            return NULL;
        // Align block size
```

```
    if (new_size <= DSIZE) {
        new_size = 2 * DSIZE;
    } else {
        new_size = ALIGN(size+DSIZE);
    }
    /* Add overhead requirements to block size */
    new_size += REALLOC_BUFFER;
    /* Calculate block buffer */
    block_buffer = GET_SIZE(HDRP(bp)) - new_size;
    /* Allocate more space if overhead falls below the minimum */
    if (block_buffer < 0) {
        /* Check if next block is a free block or the epilogue block */
        if (!GET_ALLOC(HDRP(NEXT_BLKP(bp))) || !GET_SIZE(HDRP(NEXT_BLKP(bp)))) {
            remainder = GET_SIZE(HDRP(bp)) + GET_SIZE(HDRP(NEXT_BLKP(bp))) -
new_size;
            if (remainder < 0) {
                extendsize = MAX(-remainder, CHUNKSIZE);
                if (extend_heap(extendsize) == NULL)
                    return NULL;
                remainder += extendsize;
            }
            delete_node(NEXT_BLKP(bp));
            // Do not split block
            PUT_NOTAG(HDRP(bp), PACK(new_size + remainder, 1));
            PUT_NOTAG(FTRP(bp), PACK(new_size + remainder, 1));
        } else {
            new_bp = mm_malloc(new_size - DSIZE);
            memcpy(new_bp, bp, MIN(size, new_size));
            mm_free(bp);
        }
        block_buffer = GET_SIZE(HDRP(new_bp)) - new_size;
    }
    // Tag the next block if block overhead drops below twice the overhead
    if (block_buffer < 2 * REALLOC_BUFFER)
        SET_RATAG(HDRP(NEXT_BLKP(new_bp)));
    // Return the reallocated block
    return new_bp;
}
```