哈爾濱工業大學

实验报告

实 验（八）

题　　目 Dynamic Storage Allocator

动态内存分配器

专　　业 计算机

学　　号 1180300308

班　　级 03003

学　　生 刘义

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2019 年 12 月 11 日

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

- 理解现代计算机系统虚拟存储的基本知识
- 掌握 C 语言指针相关的基本操作
- 深入理解动态存储申请、释放的基本原理和相关系统函数
- 用 C 语言实现动态存储分配器，并进行测试分析
- 培养 Linux 下的软件系统开发与测试能力

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

1.2.3 开发工具

Visual Studio 2010 64 位以上；HITICS-Lab8 malloc 实验包

1.3 实验预习

- **上实验课前，必须认真预习实验指导书（PPT 或 PDF）**
- **了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。**
- **熟知 C 语言指针的概念、原理和使用方法**
- **了解虚拟存储的基本原理**
- **熟知动态内存申请、释放的方法和相关函数**
- **熟知动态内存申请的内部实现机制：分配算法、释放合并算法**

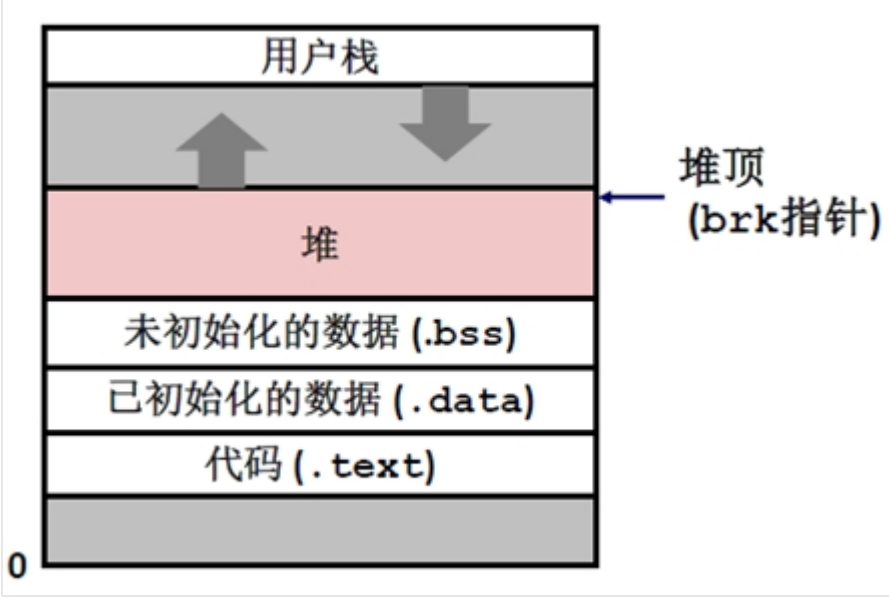
第 2 章 实验预习

总分 20 分

2.1 动态内存分配器的基本原理（5 分）

动态内存分配器：为应用程序分配虚拟内存。

动态内存分配器管理着一个虚拟内存区域，称为堆（heap）（见下图）。分配器以块为单位来维护堆，每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留以供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。



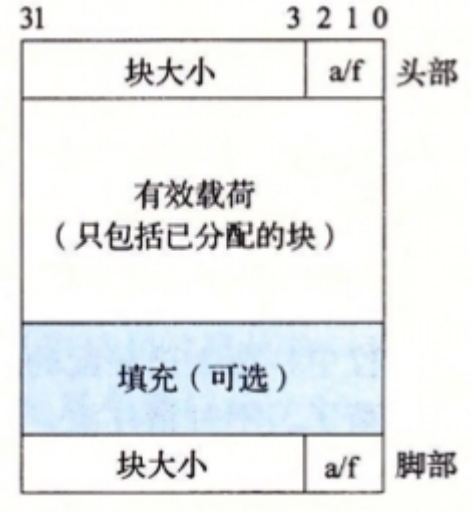
分配器分为两种基本风格：显式分配器、隐式分配器。

显式分配器：要求应用显式地释放已分配的块。

隐式分配器：自动释放不再使用的已分配的块。

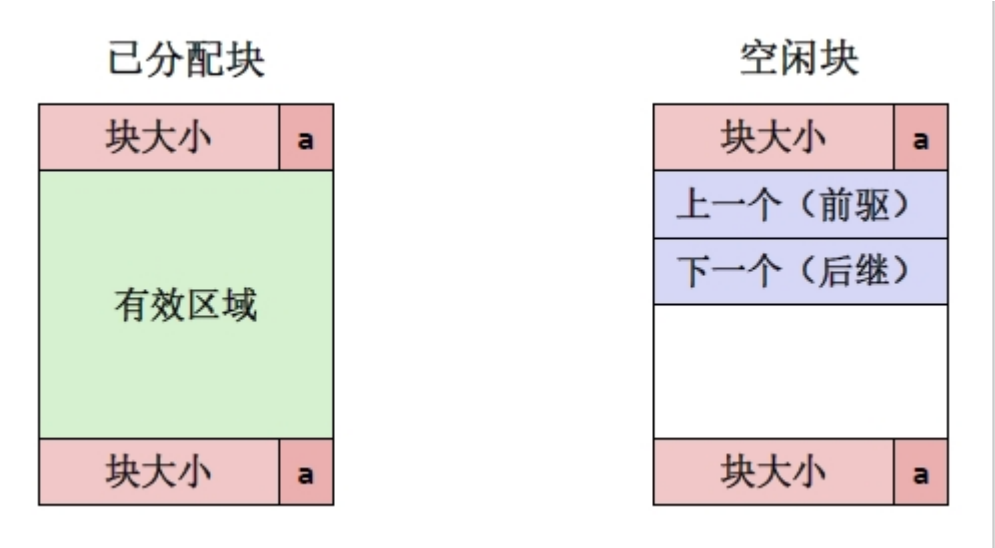
2.2 带边界标签的隐式空闲链表分配器原理（5 分）

隐式空闲链表的空闲块是通过头部中的大小字段隐含地连接着的，分配器可以通过头部遍历堆中的所有的块，从而间接地遍历整个空闲块的集合。为了方便合并空闲块，需要脚部，脚部是头部的副本，方便寻找上一个块。 如下图：



2.3 显式空闲链表的基本原理（5 分）

将堆中的空闲块组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred（前驱）和 succ（后继）指针，如图：



使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以使线性的，也可以是一个常数，这取决于我们选择的空闲链表中块的排序策略。

一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。使用 LIFO 的顺序和首次适配的放置策略，分配器会先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在线性时间内完成。

另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

一般而言，显示链表的缺点是空闲块必须足够大，以包含所有需要的指针，以及头部和可能的脚部，这就导致了更大的最小块大小，也潜在地提高了内部碎片的程度。

2.4 红黑树的结构、查找、更新算法（5 分）

结构：

红黑树是具有下列着色性质的二叉查找树：

1. 每个结点或为红色或为黑色
2. 根是黑色的
3. 如果一个结点是红色的，那么它的子结点必须为黑色。
4. 从一个结点到 NULL 指针的每一条路径必须包含相同数目的黑色结点。
5. NULL 结点是黑色的

查找算法：

```
//返回指向红黑树 x 中元素为 key 的结点的指针
Node* search(RBTree x, Type key)
{
    if (x==NULL || x->key==key)
        return x;

    if (key < x->key)
        return search(x->left, key);
    else
        return search(x->right, key);
}
```

更新算法：

插入结点之后更新：

```
void rbtree_insert_fixup(RBRoot *root, Node *node)
{
    Node *parent, *gparent;

    // 若“父节点存在，并且父节点的颜色是红色”
    while ((parent = rb_parent(node)) && rb_is_red(parent))
    {
        gparent = rb_parent(parent);

        //若“父节点”是“祖父节点的左孩子”
        if (parent == gparent->left)
        {
            // Case 1 条件：叔叔节点是红色
            {
                Node *uncle = gparent->right;
                if (uncle && rb_is_red(uncle))
                {
                    rb_set_black(uncle);
                    rb_set_black(parent);
                    rb_set_red(gparent);
                    node = gparent;
                    continue;
                }
            }
            // Case 2 条件：叔叔是黑色，且当前节点是右孩子
            if (parent->right == node)
            {
                Node *tmp;
                rbtree_left_rotate(root, parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }
            // Case 3 条件：叔叔是黑色，且当前节点是左孩子。
            rb_set_black(parent);
            rb_set_red(gparent);
```

```
            rbtree_right_rotate(root, gparent);
        }
        else//若“z 的父节点”是“z 的祖父节点的右孩子”
        {
            // Case 1 条件：叔叔节点是红色
            {
                Node *uncle = gparent->left;
                if (uncle && rb_is_red(uncle))
                {
                    rb_set_black(uncle);
                    rb_set_black(parent);
                    rb_set_red(gparent);
                    node = gparent;
                    continue;
                }
            }

            // Case 2 条件：叔叔是黑色，且当前节点是左孩子
            if (parent->left == node)
            {
                Node *tmp;
                rbtree_right_rotate(root, parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }

            // Case 3 条件：叔叔是黑色，且当前节点是右孩子。
            rb_set_black(parent);
            rb_set_red(gparent);
            rbtree_left_rotate(root, gparent);
        }
    }

    // 将根节点设为黑色
    rb_set_black(root->node);
}
```

删除结点之后更新：

```
void rbtree_delete_fixup(RBRoot *root, Node *node, Node *parent)
{
    Node *other;

    while ((!node || rb_is_black(node)) && node != root->node)
    {
        if (parent->left == node)
        {
            other = parent->right;
            if (rb_is_red(other))
            {
                // Case 1: x 的兄弟 w 是红色的
                rb_set_black(other);
                rb_set_red(parent);
                rbtree_left_rotate(root, parent);
                other = parent->right;
            }
            if ((!other->left || rb_is_black(other->left)) &&
                (!other->right || rb_is_black(other->right)))
            {
                // Case 2: x 的兄弟 w 是黑色，且 w 的俩个孩子也都是黑色
的
                rb_set_red(other);
                node = parent;
                parent = rb_parent(node);
            }
            else
            {
                if (!other->right || rb_is_black(other->right))
                {
                    // Case 3: x 的兄弟 w 是黑色的，并且 w 的左孩子是红
色，右孩子为黑色。
                    rb_set_black(other->left);
                    rb_set_red(other);
                    rbtree_right_rotate(root, other);
                    other = parent->right;
```

```
                }
                // Case 4: x 的兄弟 w 是黑色的；并且 w 的右孩子是红色的，
左孩子任意颜色。
                rb_set_color(other, rb_color(parent));
                rb_set_black(parent);
                rb_set_black(other->right);
                rbtree_left_rotate(root, parent);
                node = root->node;
                break;
            }
        }
        else
        {
            other = parent->left;
            if (rb_is_red(other))
            {
                // Case 1: x 的兄弟 w 是红色的
                rb_set_black(other);
                rb_set_red(parent);
                rbtree_right_rotate(root, parent);
                other = parent->left;
            }
            if ((!other->left || rb_is_black(other->left)) &&
                (!other->right || rb_is_black(other->right)))
            {
                // Case 2: x 的兄弟 w 是黑色，且 w 的俩个孩子也都是黑色
的
                rb_set_red(other);
                node = parent;
                parent = rb_parent(node);
            }
            else
            {
                if (!other->left || rb_is_black(other->left))
                {
                    // Case 3: x 的兄弟 w 是黑色的，并且 w 的左孩子是红
色，右孩子为黑色。
```

```
                        rb_set_black(other->right);
                        rb_set_red(other);
                        rbtree_left_rotate(root, other);
                        other = parent->left;
                    }
                    // Case 4: x 的兄弟 w 是黑色的；并且 w 的右孩子是红色的，
左孩子任意颜色。
                    rb_set_color(other, rb_color(parent));
                    rb_set_black(parent);
                    rb_set_black(other->left);
                    rbtree_right_rotate(root, parent);
                    node = root->node;
                    break;
                }
            }
        }
        if (node)
            rb_set_black(node);
    }
```

第 3 章 分配器的设计与实现

总分 50 分

3.1 总体设计（10 分）

介绍堆、堆中内存块的组织结构，采用的空闲块、分配块链表/树结构和相应算法等内容。

空闲块的组织方法：隐式空闲链表。

放置策略：首次适配（First Fit）：每次从头进行搜素，找到第一个合适的块。

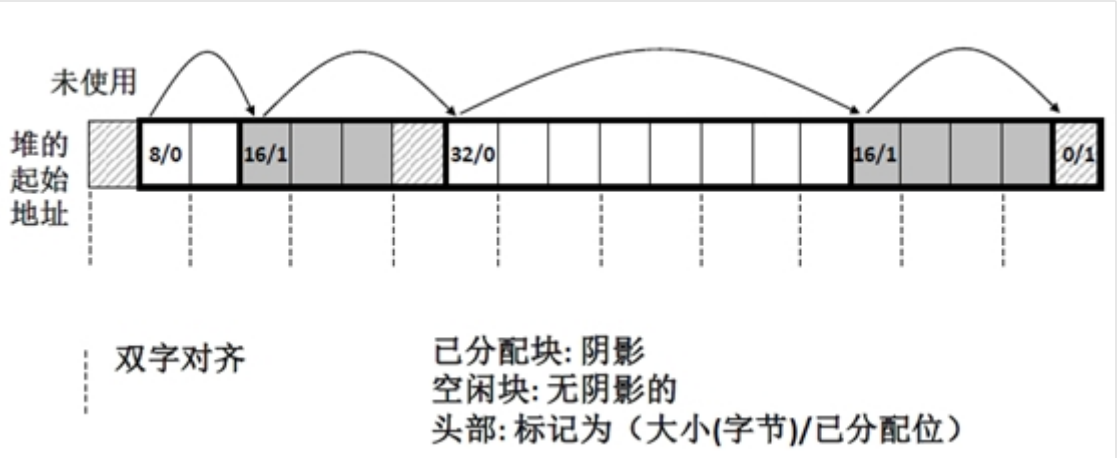
合并策略：立即合并。

1. 隐式空闲链表的空闲块是通过头部中的大小字段隐含地连接着的，分配器可以通过遍历堆中的所有的块，从而间接地遍历整个空闲块的集合，这里我们需要某种特殊标记的结束块。

一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成，头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。头部后面就是应用调用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。

2. 空闲块、分配块链表

用隐式空闲链表来组织堆，空闲块通过头部中的大小字段隐含着连接着，分配器可以通过遍历堆中的所有的块，从而间接地遍历整个空闲块的集合。



隐式链表的优点是简单。显著的缺点是任何操作的开销要求对空闲链表进行搜素，该搜素所需时间与堆中已分配块和空闲块的总数呈线性关系。

3. 立即合并

在每次一个块被释放时就合并所有的相邻块，这可以在常数时间内完成，但是对于某种请求模式，这种方式会产生一种形式的抖动，块会反复地合并，然后马上分割，这会产生大量不必要的分割和合并，从而降低吞吐率，不过在本实验中吞吐率尚可。

3.2 关键函数设计（40 分）

3.2.1 int mm_init(void)函数（5 分）

函数功能：执行初始化操作，包括分配初始的堆区域。成功返回 0，否则返回-1。
处理流程：mm_init 函数从内存系统得到 4 个字，并将它们初始化，创建一个空的空闲链表。然后它调用 extend_heap 函数，这个函数将堆拓展 CHUNKSIZE 字节，并且创建初始的空闲块，此时，分配器已经初始化了，并且准备好接受来自应用的分配和释放请求。
要点分析：在空闲链表中操作头部和脚部需要大量使用强制类型转换和指针运算，比较麻烦，因此定义了一小组宏来访问和遍历空闲链表。例如：

1. PACK(DSIZE, 0x1 | 0x2))将大小和已分配位、前一个块的已分配位进行“按位或”得到头部编码、尾部编码
2. PUT 宏将 PACK(DSIZE, 0x1 | 0x2))存放在参数 p 指向的字节中。

代码如下：

```
int mm_init(void)
{
    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1) //line:vm:mm:begininit
        return -1;
    PUT(heap_listp, 0);                                   /* Alignment padding */
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 0x1 | 0x2)); /* Prologue header */
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 0x1 | 0x2)); /* Prologue footer */
    PUT(heap_listp + (3 * WSIZE), PACK(0, 0x1 | 0x2));     /* Epilogue header */
    heap_listp += (2 * WSIZE);                     //line:vm:mm:endinit

    rover = heap_listp;

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
        return -1;
    return 0;
}
```

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放参数 ptr 指向的已分配内存块。
参　　数：ptr
处理流程：释放所请求的块（bp），然后合并与之邻接的空闲块。
要点分析：

1. 处理特殊情况，ptr 若为空函数返回，若未初始化调用 mm_init 函数。
2. 设置当前块的头部、尾部以及下一个块的次低位（指示上一个块是否已分配）
3. 调用 coalesce 函数合并空闲块

代码实现：

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

    size_t size = GET_SIZE(HDRP(bp));

    if (heap_listp == 0) {
        mm_init();
    }

    if (GET_PREVIOUS_ALLOC(HDRP(bp)))
    {
        PUT(HDRP(bp), PACK(size, 0x2));
    }
    else
    {
        PUT(HDRP(bp), PACK(size, 0));
    }
    PUT(FTRP(bp), PACK(size, 0));
    /* pack next block */
    *HDRP(NEXT_BLKP(bp)) &= ~0x2;
    coalesce(bp);
}
 /* $end mmfree */
```

3.2.3 void *mm_realloc(void *ptr, size_t size)函数（5 分）

函数功能：将 ptr 所指向内存块的大小变为 size，返回新内存块的地址
参　　数：旧内存块地址 ptr，新内存块大小 size
处理流程：
1. 如果 ptr 是空指针 NULL,等价于 mm_malloc(size)
2. 如果参数 size 为 0，等价于 mm_free(ptr)
3. 否则申请一个大小为 size 的新块，将旧块复制到新块，再将旧块释放

要点分析：
新内存块中，前 min(旧块 oldsize，新块 size 的较小值)个字节的内容与旧块相同，其他字节未做初始化。
代码实现：

```
void *mm_realloc(void *ptr, size_t size)
{
    size_t oldsize;
    void *newptr;
    /* If size == 0 then this is just free, and we return NULL. */
    if (size == 0) {
        mm_free(ptr);
        return 0;
    }
    /* If oldptr is NULL, then this is just malloc. */
    if (ptr == NULL) {
        return mm_malloc(size);
    }
    newptr = mm_malloc(size);
    /* If realloc() fails the original block is left untouched  */
    if (!newptr) {
        return 0;
    }
    /* Copy the old data. */
    oldsize = GET_SIZE(HDRP(ptr));
    if (size < oldsize) oldsize = size;
    memcpy(newptr, ptr, oldsize);

    /* Free the old block. */
    mm_free(ptr);

    return newptr;
}
```

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查重要的不变量和一致性条件。当且仅当堆是一致的，才能返回非 0 值。

处理流程：

1. 首先从堆的起始位置处检查序言块的头部和脚部是否都是双字
2. 从第一个块开始循环，直到结尾，依次检查当前块
3. 检查结尾块是否大小为 0 且已分配

要点分析：在检查堆的函数中调用检查每个块的函数，主要检查已分配块是不是双字对齐，头部和脚部是否匹配。

代码实现：

```
void mm_checkheap(void)
{
    char *bp = heap_listp;

    if (verbose)
        printf("Heap (%p):\n", heap_listp);

    if ((GET_SIZE(HDRP(heap_listp)) != DSIZE) || !GET_ALLOC(HDRP(heap_listp)))
        printf("Bad prologue header\n");
    checkblock(heap_listp);

    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (verbose)
            printblock(bp);
        checkblock(bp);
    }

    if (verbose)
        printblock(bp);
    if ((GET_SIZE(HDRP(bp)) != 0) || !(GET_ALLOC(HDRP(bp))))
        printf("Bad epilogue header\n");
}

static void checkblock(void *bp)
{
    if ((size_t)bp % 8)
        printf("Error: %p is not doubleword aligned\n", bp);
    if (GET(HDRP(bp)) != GET(FTRP(bp)))
        printf("Error: header does not match footer\n");
}
```

3.2.5 void *mm_malloc(size_t size)函数（10 分）

函数功能：申请有效载荷至少是参数 size 指定大小的内存块，返回该内存块地址首地址（可以使用的区域首地址）。返回的地址应该是 8 字节对齐的（地址%8==0）

参　　数：申请空闲块的大小 size

处理流程：

1. 处理特殊情况，若 size 为 0 函数返回，若未初始化调用 mm_init 函数。
2. 若堆中存在合适的空闲块，放置这个请求块
3. 否则，用一个新的空闲块来拓展堆

要点分析：

1. 获得所需空闲块的大小，请求大小+头部，8 字节对齐。
2. 分割策略：尽可能地分割多余的部分

代码实现：

```
void *mm_malloc(size_t size)
{
    size_t Wsize;       /* round up to an WSIZE */
    size_t Dsize;       /* DSIZE after align WSIZE */

    size_t asize;       /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;

    if (heap_listp == 0) {
        mm_init();
    }
    if (size == 0)
        return NULL;

    Wsize = (size + (WSIZE - 1)) / WSIZE;
    Dsize = Wsize / 2 + 1;
    asize = DSIZE * Dsize;

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize / WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
/* $end mmmalloc */
```

3.2.6 static void *coalesce(void *bp)函数（10 分）

函数功能：将要回收的空闲块和临近的空闲块（如果有的话）合并成一个大的空闲块。

处理流程：

1. 前面的块和后面的块都是已分配的：不进行合并，当前块的状态只是简单地从分配变为空闲。
2. 前面的块是已分配的，后面的块是空闲的：当前块与后面的块合并，用当前块和后面的块的大小的和来更新当前块的头部和后面块的脚部。
3. 前面的块是空闲的，后面的块是已分配的：前面的块和当前块合并，用两个块的大小的和来更新前面块的头部和当前块的脚部。
4. 前面的和后面的块都是空闲的：合并所有的三个块形成一个单独的空闲块，用三个块大小的和来更新前面块的头部和后面块的脚部。

要点分析：

合并空闲块为一个大的空闲块的时候只需要改变空闲块最前方的 Header 和最后面的 Footer。

代码实现：

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_PREVIOUS_ALLOC(HDRP(bp));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) {            /* Case 1 */
        return bp;
    }

    else if (prev_alloc && !next_alloc) {      /* Case 2 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0x2));
        PUT(FTRP(bp), PACK(size, 0));
    }

    else if (!prev_alloc && next_alloc) {      /* Case 3 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));

        if (GET_PREVIOUS_ALLOC(HDRP(PREV_BLKP(bp))))
        {
            PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0x2));
        }
        else
        {
            PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        }
        bp = PREV_BLKP(bp);
    }

    else {                                     /* Case 4 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
            GET_SIZE(FTRP(NEXT_BLKP(bp)));

        if (GET_PREVIOUS_ALLOC(HDRP(PREV_BLKP(bp))))
        {
            PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0x2));
        }
        else
        {
            PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        }
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    /* $end mmfree */
    /* Make sure the rover isn't pointing into the free block */
    /* that we just coalesced */
    if ((rover > (char *)bp) && (rover < NEXT_BLKP(bp)))
        rover = bp;
    /* $begin mmfree */
    return bp;
}
```

第 4 章测试

总分 10 分

4.1 测试方法

使用实验包中给定的测试函数。

1） make clean。清除已经有的 make 信息（在 Makefile 中有定义）。

2） make。链接、编译成可执行程序 mdriver。其中 mdriver.c 是 mm.c 的调用程序，整个测试程序的执行逻辑存放其中。

3）./mdriver -v -t traces/ 测试 traces 文件夹下的所有的轨迹文件并输出结果。

4.2 测试结果评价

采用隐式空闲链表+首次适配+立即合并，已分配的块没有脚部，用头部的次高位表示前一个块的分配状态，测试结果为：perf index = 43(util)+40(thru)=83/100。

4.3 自测试结果

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab8/malloclab-handout$ ./mdriver -v
  -t  traces/
Team Name:1180300308
Member 1 :LiuYi:2681514899@qq.com
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   90%    5694  0.002292  2485
 1       yes   92%    5848  0.001148  5093
 2       yes   95%    6648  0.003225  2062
 3       yes   96%    5380  0.003329  1616
 4       yes   66%   14400  0.000138104272
 5       yes   91%    4800  0.003865  1242
 6       yes   89%    4800  0.003608  1330
 7       yes   55%   12000  0.015425   778
 8       yes   51%   24000  0.007716  3110
 9       yes   27%   14401  0.060288   239
10       yes   34%   14401  0.002400  6001
Total          71%  112372  0.103432  1086

Perf index = 43 (util) + 40 (thru) = 83/100
```

第 5 章 总结

5.1 请总结本次实验的收获

1. 动态内存分配器的原理
2. 隐式空闲链表的基本实现方法

5.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.