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 计算机科学与技术

学　　号 1183200123

班　　级 1803003

学　　生 祁 天

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2019.12.11

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

1. 理解现代计算机系统虚拟存储的基本知识
2. 掌握 C 语言指针相关的基本操作
3. 深入理解动态存储申请、释放的基本原理和相关系统函数
4. 用 C 语言实现动态存储分配器，并进行测试分析
5. 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

1.3 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、释放合并算法等

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）

动态内存分配器维护着一个进程的虚拟内存区域，称为堆（heap）（见下图）。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在为初始化的数据区域后开始，并向上生长（向更高的地址）。对于每一个进程，内核维护着一个变量 brk，它指向堆的顶部。

分配器将堆视为一组不同大小的块的集合来维护。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。已分配的块显示的保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显示地被应用所分配。一个已分配的块保持已分配的状态，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。

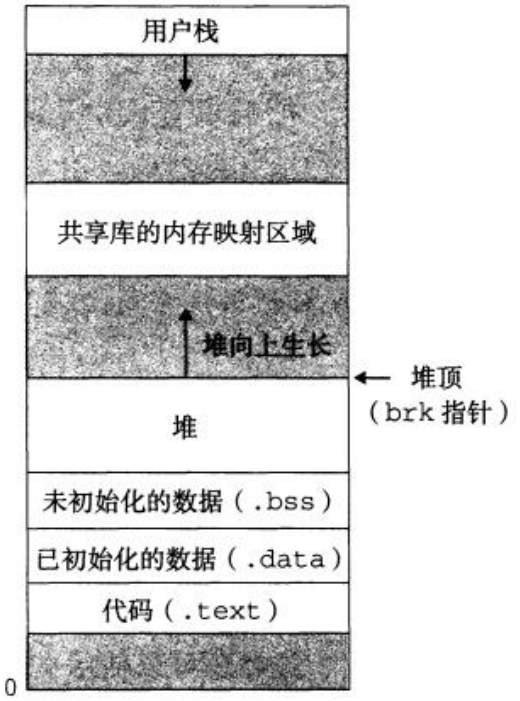


图 9-33 堆

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

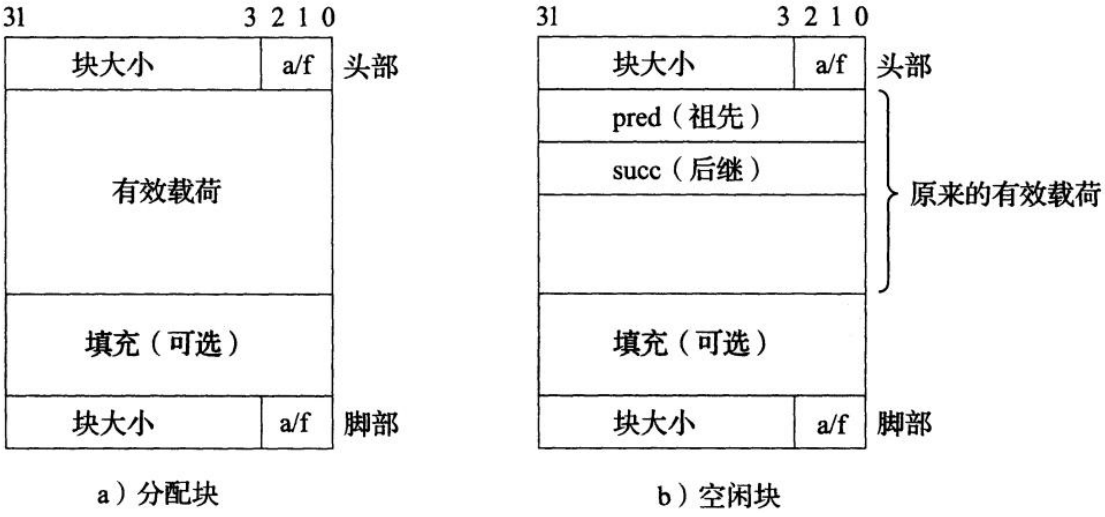
一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成，头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。头部后面就是应用调用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。

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。如图所示：

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。

2.3 显示空间链表的基本原理（5 分）

堆可以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。

2.4 红黑树的结构、查找、更新算法（5 分）

红黑树是一种特殊的二叉查找树，红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树的结构:

（1）每个节点或者是黑色，或者是红色。

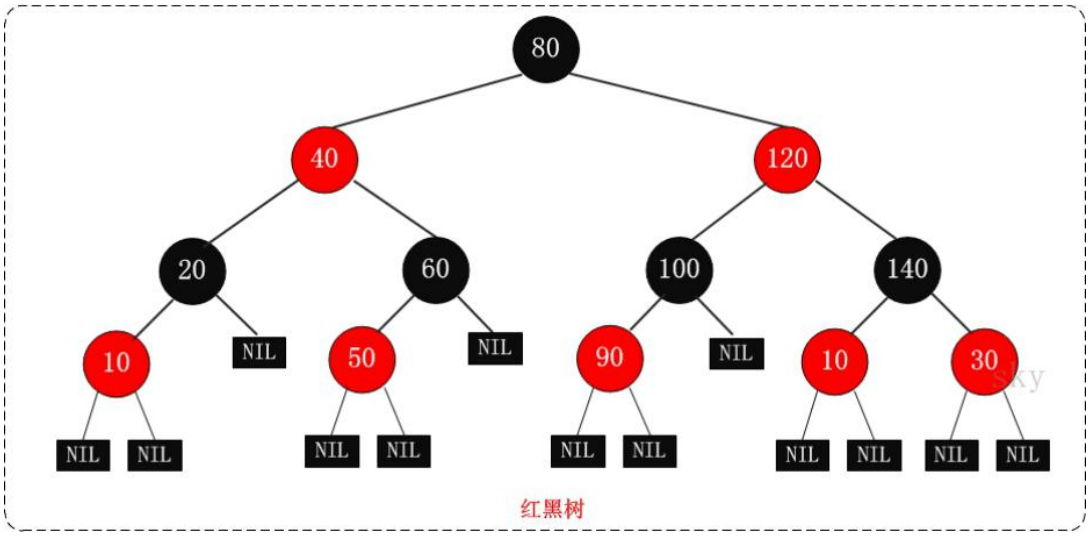
（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL 或 NULL)的叶子节点！

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

如图：



查找算法：注意红黑树是一种特殊的二叉查找树

```
PtrToNode RB_Find(PtrToNode &node, elementType x) {
    if (node == NULL)  //没找到元素
    {
        return NULL;
```

```
    } else if (x < node->data) {
        return RB_Find(node->left, x); //在左子树里面查找
    } else if (node->data < x) {
        return AVL_Find(node->right, x); //在右子树里面查找
    } else //相等
        return node;
}
```

更新算法：更新主要是为了保持插入和删除之后的平衡。

结构定义：

```
// 红黑树的节点
template <typename T>
struct RB_Node{
    char color;        // 颜色(RED 或 BLACK)
    T key;                    // 关键字(键值)
    struct RB_Node<T> *lchild;    // 左孩子
    struct RB_Node<T> *rchild;    // 右孩子
    struct RB_Node<T> *parent;    // 父结点
};
// 红黑树的根
template <typename T>
struct RBRoot{
    RB_Node<T> *node;
    };
```

插入节点后更新：

```
template <typename T>
void R_BTree<T>::RbTree_Insert_ResetColorRotate(RBRoot<T>*
root, RB_Node<T>* node)
{
    RB_Node<T>*parent,*gparent,*uncle,*temp;

    while ((parent = node->parent)!=NULL && parent->color ==
RED)  // 若"父节点存在，并且父节点的颜色是红色"
    {
        gparent = parent->parent;

        if (parent == gparent->lchild)  //若"父节点"是"祖父节点
的左孩子"
        {
            uncle = gparent->rchild;
            if (uncle && uncle->color == RED)   // Case 1: 叔叔
节点是红色
            {
                uncle->color = BLACK;
```

```
            parent->color = BLACK;
            gparent->color = RED;
            node = gparent;
            continue;
        }

        if (parent->rchild == node)   // Case 2: 叔叔是黑色，且当前节点是右孩子
        {
            RbTree_LeftRotate(root,parent);
            temp = parent;
            parent = node;
            node = temp;
        }

        parent->color = BLACK;  // Case 3: 叔叔是黑色，且当前节点是左孩子。
        gparent->color = RED;
        RbTree_RightRotate(root, gparent);
    }
    else
    {
        uncle = gparent->lchild;
        if (uncle && uncle->color == RED)   // Case 1: 叔叔节点是红色
        {
            uncle->color = BLACK;
            parent->color = BLACK;
            gparent->color = RED;
            node = gparent;
            continue;
        }

        if (parent->lchild == node)  // Case 2: 叔叔是黑色，且当前节点是左孩子
        {
            RbTree_RightRotate(root, parent);
            temp = parent;
            parent = node;
            node = temp;
        }

        parent->color = BLACK;  // Case 3: 叔叔是黑色，且当前节点是右孩子。
        gparent->color = RED;
        RbTree_LeftRotate(root, gparent);
```

```
        }
    }
    root->node->color = BLACK;  // 将根节点设为黑色
}
```

删除节点之后更新：

```
template <typename T>
void R_BTree<T>::RbTree_Delete_Reset(RBRoot<T>* root,
RB_Node<T>* node, RB_Node<T>* parent)
{
    RB_Node<T>* other;

    while ((!node || node->color==BLACK) && node != root->node)
    {
        if (parent->lchild == node)
        {
            other = parent->rchild;
            if (other->color==RED)  // Case 1: node 的兄弟是红色
的
            {
                other->color = BLACK;
                parent->color = RED;
                RbTree_LeftRotate(root, parent);
                other = parent->rchild;
            }
            if ((!other->lchild || other->lchild->color ==
BLACK) &&
                (!other->rchild || other->rchild->color ==
BLACK))   // Case 2: node 的兄弟是黑色

{
                //且兄弟的两个孩子也都是黑色的
                other->color = RED;
                node = parent;
                parent = node->parent;
            }
            else
            {
                if (!other->rchild || other->rchild->color ==
BLACK)  // Case 3: node 的兄弟是黑色的

{
             //并且兄弟的左孩子是红色，右孩子为黑色
                    other->lchild->color = BLACK;
                    other->color = RED;
                    RbTree_RightRotate(root, other);
                    other = parent->rchild;
```

```
                }
                other->color = parent->color;   // Case 4: node
的兄弟是黑色的
                parent->color = BLACK;          //并且 node 的右孩子
是红色的，左孩子任意颜色
                other->rchild->color = BLACK;
                RbTree_LeftRotate(root, parent);
                node = root->node;
                break;
            }
        }
        else
        {
            other = parent->lchild;
            if (other->color == RED)  // Case 1: node 的兄弟是红
色的
            {
                other->color = BLACK;
                parent->color = RED;
                RbTree_RightRotate(root, parent);
                other = parent->lchild;
            }
            if ((!other->lchild || other->lchild->color ==
BLACK) &&   // Case 2: node 的兄弟是黑色
                (!other->rchild || other->rchild->color ==
BLACK))         //且 node 的两个孩子也都是黑色的
            {
                other->color = RED;
                node = parent;
                parent = node->parent;
            }
            else
            {
                if (!other->lchild || other->lchild->color ==
BLACK)    // Case 3: node 的兄弟是黑色的

{
              //并且 node 的左孩子是红色，右孩子为黑色
                    other->rchild->color = BLACK;
                    other->color = RED;
                    RbTree_LeftRotate(root, other);
                    other = parent->lchild;
                }
                other->color = parent->color;    // Case 4: node
的兄弟是黑色的
                parent->color = BLACK;          //并且 node 的右孩
```

```
子是红色的，左孩子任意颜色
                other->lchild->color = BLACK;
                RbTree_RightRotate(root, parent);
                node = root->node;
                break;
            }
        }
    }
    if (node)   /*********/
        node->color = BLACK;
}
```

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。

1.堆：动态内存分配器维护着一个进程的虚拟内存区域，称为堆。它紧接在为初始化的数据区域后开始，并向上生长（向更高的地址）。对于每一个进程，内核维护着一个变量 brk，它指向堆的顶部。

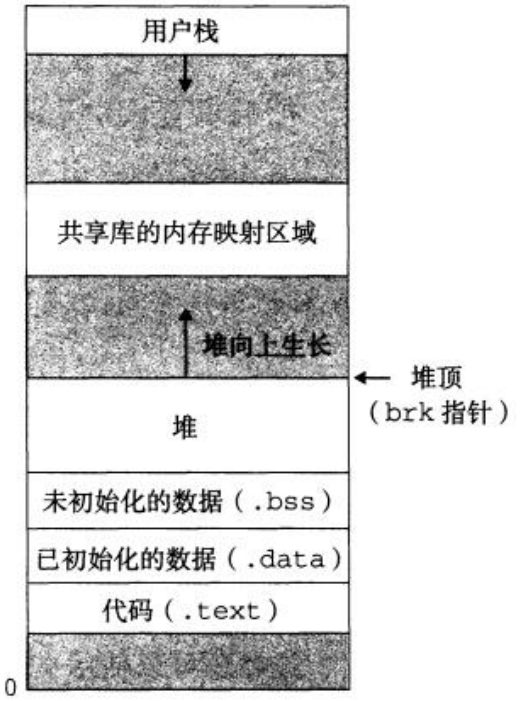


图 9-33 堆

2.堆中内存块的组织结构：用隐式空闲链表来组织堆，具体组织的算法在 mm_init 函数中。对于带边界标签的隐式空闲链表分配器，一个块是由一个字的头部、有效载荷、可能的一些额外的填充，以及在块的结尾处的一个字的脚部组成的。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。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

3.对于空闲块和分配块链表：采用分离的空闲链表。全局变量： void *Lists[MAX_LEN]; 因为一个使用单向空闲块链表的分配器需要与空闲块数量呈线性关系的时间来分配块，而此堆的设计采用分离存储的来减少分配时间，就是维护多个空闲链

表，每个链表中的块有大致相等的大小。将所有可能的块大小根据 2 的幂划分。

4.放置策略（适配方式）：首次适配（其实有着最佳适配的效果）。malloc 搜索块的时间从所有空的空闲块降低到局部链表的空闲块中，当分到对应的大小类链表的时候，它的空间也会在大小类链表的范围里面，这样使得即使是首次适配也可以是空间利用率接近最佳适配。进一步解释：当空闲链表按照块大小递增的顺序排序时，首次适配是选择第一个合适的空闲块，最佳适配是选择所需请求大小最小的空闲块，也是会选择第一个合适的空闲块，后面的块大小递增，不再选择。因此两种适配算法效率近似。

5.关于链表操作主要函数和算法：

static void InsertNode(void *bp, size_t size)

static void DeleteNode(void *bp)

程序中大部分函数在后面都会介绍到，因此在这里只简单分析 InsertNode 和 DeleteNode 函数，分别用来插入分离的空闲链表，和从分离的空闲链表中删除。

在介绍这两个函数之前，先阐明新的宏定义：

```
#define SET_PTR(p, bp) (*(unsigned int *)(p) = (unsigned int)(bp))
/*将 bp 写入参数 p 指的字中*/
#define PRED_PTR(bp) ((char *)(bp))   /*祖先节点*/
#define SUCC_PTR(bp) ((char *)(bp) + WSIZE)   /*后继节点*/
#define PRED(bp) (*(char **)(bp))
#define SUCC(bp) (*(char **)(SUCC_PTR(bp)))
```

根据分配器的设计，后面两个宏定义分别表示 size 更大块的指针和 size 更小块的指针。InsertNode(void *bp, size_t size)函数：

1.将 free 块插入分离空闲链表，首先要在链表数组中，找到块的大小类，从而找到对应的分离空闲链表；其次，找到链表后，需要根据 size 的比较一直循环，直到链中的下一个块比 bp 所指的块大为止，以保持链表中的块由小到大排列，方便之后的适配。

2.insert_bp 表示的是待插入的位置，search_bp 表示的是比 bp 所指块更大的块的指针，找到对应位置，有四种情况：如果 search_bp != NULL，那么可能是在中间插入，或者在 List[i]首地址之后插入（并且此时 List[i]后面不是空）否则 search_bp = NULL，那么可能是在结尾插入，或者该 List[i]链表原本就为空，在其首地址插入即可。

DeleteNode(void *bp)函数：

将块从分离空闲链表中删除，其实和插入的操作类似。

1. 首先要在链表数组中，找到块的大小类，从而找到对应的分离空闲链表

2.从链表中删除块的时候，也分四种情况：在链表的中间删除；在表头删除，并且删除后 List[i]不是空表；在链表的结尾删除；在 List[i]表头删除，并且原本 bp 所指的块就是表中最后一个块。

3.2.1 int mm_init(void)函数（5 分）

函数功能：初始化堆和分离空闲链表

处理流程：

1、 初始化分离空闲链表。根据申请的链表数组，将分离空闲链表全部初始化

为 NULL。

```
/* 初始化分离空闲链表 */
for (i = 0; i < MAX_LEN; i++)
{
    Lists[i] = NULL;
}
```

2、mm_init 函数从内存中得到 4 个字，并且将堆初始化，创建一个空的空闲链表。

```
/*建立一个空堆*/
if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
    return -1;
PUT(heap_listp, 0);    //对齐填充
PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));  //序言块头部
PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));  //序言块脚部
PUT(heap_listp + (3 * WSIZE), PACK(0, 1));      //结尾块
```

3、调用 extend_heap 函数，这个函数将堆扩展 INITCHUNKSIZE 字节，并且创建初始的空闲块。

```
/* 将堆拓展INITCHUNKSIZE bytes字节 */
if (extend_heap(INITCHUNKSIZE) == NULL)
return -1;
```

要点分析：

1. 分配器使用最小块的大小是 16 字节，空闲链表组织成一个隐式空闲链表；
2. 空闲链表创建之后需要使用 extend_heap 函数来扩展堆

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放一个以前分配的块
参　　数：指针 ptr

处理流程：

1、 获得请求块的大小；
2、 将请求块的头部和脚部的已分配位置为 0，表示为 free；
3、 将 free 块插入到分离空闲链表中；
4、 使用边界标记合并技术将释放的块 bp 与相邻的空闲块合并起来

```
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    InsertNode(bp, size);
    coalesce(bp);
}
```

要点分析：

1. PACK(OVERHEAD, 1)将大小和已分配位结合起来
2. 将请求块 bp 标记为 free 后，需要将它插入到分离的空闲链表中；
3. free 块需要和与之相邻的空闲块使用边界标记合并技术进行合并

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（旧块)的大小变为 size，并返回新内存块的地址
参　　数：指针 ptr 、内存块大小 size
处理流程：

1. 要点分析：如 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. 如 ptr 非空, 它应该是之前调用 mm_malloc 或 mm_realloc 返回的数值，指向一个已分配的内存块。

要点分析：

返回的地址与原地址可能相同，也可能不同，这依赖于算法的实现、旧块内部碎片大小、参数 size 的数值。新内存块中，前 min(旧块 size，新块 size)个字节的内容与旧块相同，其他字节未做初始化。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆的一致性
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：

1、检查已分配块是不是双字对齐，头部和脚部是否匹配。

2、checkheap 函数只是对堆一致性的简单检查，如空闲块是否都在空闲链表等方面并没有展开检查

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

函数功能：从空闲链表分配一个块
参　　数：内存块大小 size
处理流程：

1、判断请求的真假，调整请求块的大小

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

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

2、 寻找一个合适的空闲块。如果有合适的，那么分配器就用 place 函数放置这个请求块

```
while (i < MAX_LEN)
{
    /* 先找合适的空闲链表 */
    if (((asize <= 1) && (Lists[i] != NULL)))
    {
        bp = Lists[i];
        /* 找到链表，在该链寻找大小合适的未分配块 */
        while ((bp != NULL) && ((size > GET_SIZE(HDRP(bp)))))
            bp = PRED(bp);

        /* 找到对应的未分配的块 */
        if (bp != NULL)
            break;
    }
    asize >>= 1;
    i++;
}
```

3、 如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来扩展堆，同样把请求块放置在这个新的空闲块里，可选地分割这个块，然后返回一个指向这个新分配块的指针。

```
/* 没有找到合适的未分配块，则扩展堆 */
if (bp == NULL){
    if ((bp = extend_heap(MAX(size, CHUNKSIZE))) == NULL)
        return NULL;
}
/* 在未分配块中allocate size大小的块 */
bp = place(bp, size);

return bp;
```

要点分析：

1. 在检查完请求的真假之后，分配器必须调整请求块的大小，从而为头部和脚部留有空间，并满足双字对齐的要求。强制最小块大小是 16 字节，8 字节用来满足对齐要求，另外 8 个用来放头部和脚部。对于超过 8 字节的请求，一般的规则是加上开销字节，然后向上舍入到最接近的 8 的整数倍。

2. 一旦分配器调整了请求的大小，它就会搜索空闲链表，寻找一个合适的空闲块，如果有合适的，那么分配器就放置这个请求块，并可选地分割出多余的部分，然后返回新分配块的地址。

如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来拓展堆，把请求块放置在这个新的空闲块里，可选地分割这个块，然后返回一个指针，指向这个新分配的块。

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

函数功能：边界标记合并。将指针返回到合并块

处理流程：

1、 获得前一块和后一块的已分配位，并且获得 bp 所指块的大小

```
    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));
```

2、 根据 bp 所指块相邻块的情况，可以得到以下四种可能性：

```
/*四种情况*/
if (prev_alloc && next_alloc)   /*case1*/
{
    return bp;
}

else if (prev_alloc && !next_alloc)   /*case2*/
{
    DeleteNode(bp);
    DeleteNode(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)   /*case3*/
{
    DeleteNode(bp);
    DeleteNode(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   /*case4*/
{
    DeleteNode(bp);
    DeleteNode(PREV_BLKP(bp));
    DeleteNode(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);
}
```

3、 将上述四种操作后更新的 bp 所指的块插入分离空闲链表

```
/* 合并后的free块插入到空闲链接表中 */
InsertNode(bp, size);
```

要点分析：

这里有一个很微妙的方面，我们选择的空闲链表格式（它的序言块和结尾块总是标记为已分配）允许我们忽略潜在的麻烦边界情况，也就是，请求块 bp 在堆的起始处或堆的结尾处。如果没有这些标记块代码将混乱得多，更加容易出错并且更慢，因为不得不在每次释放请求是，都去检查这些并不常见的边界情况。

第 4 章测试

总分 10 分

4.1 测试方法

生成可执行评测程序文件的方法：
linux>make
评测方法:
mdriver [-hvVa] [-f <file>]
选项：
-a 不检查分组信息
-f <file> 使用 <file>作为单个的测试轨迹文件
-h 显示帮助信息
-l 也运行 C 库的 malloc
-v 输出每个轨迹文件性能
-V 输出额外的调试信息
轨迹文件：指示测试驱动程序 mdriver 以一定顺序调用
性能分 pindex 是空间利用率和吞吐率的线性组合
获得测试总分 linux>./mdriver -av -t traces/

4.2 测试结果评价

总体效果良好，尽最大可能实现内存利用率和吞吐率最大化。其中采用了一些手段来减少外部碎片，在大多数的 trace 中都取得了良好的效果。

4.3 自测试结果

```
qt1183200123@ubuntu:~/Desktop/malloclab-handout$ ./mdriver -av -t traces
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.000529 10770
 1       yes   99%    5848  0.000641  9129
 2       yes   99%    6648  0.000643 10339
 3       yes   99%    5380  0.000561  9583
 4       yes   99%   14400  0.000790 18232
 5       yes   94%    4800  0.000899  5341
 6       yes   91%    4800  0.000797  6023
 7       yes   95%   12000  0.000753 15940
 8       yes   88%   24000  0.007211  3328
 9       yes   99%   14401  0.000577 24941
10       yes   98%   14401  0.000409 35227
Total          96%  112372  0.013810  8137

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

第 5 章 总结

5.1 请总结本次实验的收获

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.