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 计算机类

学　　号 1170500913

班　　级 1703002

学　　生 熊健羽

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2018.12.16

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

CPU：Intel(R) Core(TM) i5-7200U @ 2.50GHz (64 位)

GPU：Intel(R) HD Graphics 620
　　Nvidia GeForce 940MX

物理内存：16.00GB

磁盘：1TB HDD
　　128GB SSD

1.2.2 软件环境

Windows10 64 位；
Vmware 14.11；
Ubuntu 18.04 64 位；

1.2.3 开发工具

Visual Studio 2010 64 位；

Code::Blocks；

gedit，gcc，notepad++；

1.3 实验预习

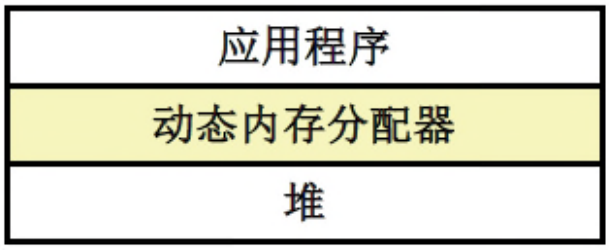
填写

第 2 章 实验预习

总分 20 分

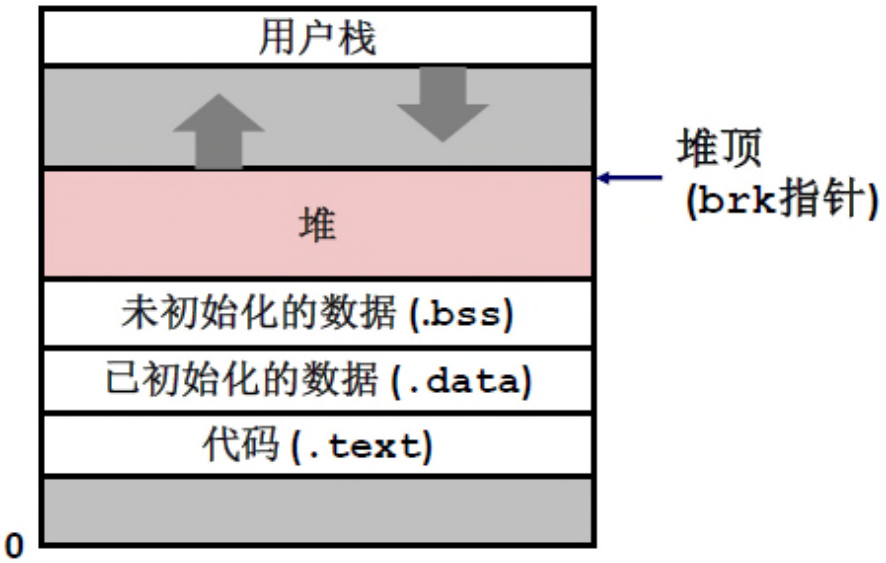
2.1 动态内存分配器的基本原理（5 分）

在程序运行时程序员使用动态内存分配器(比如 malloc) 获得虚拟内存。如图：



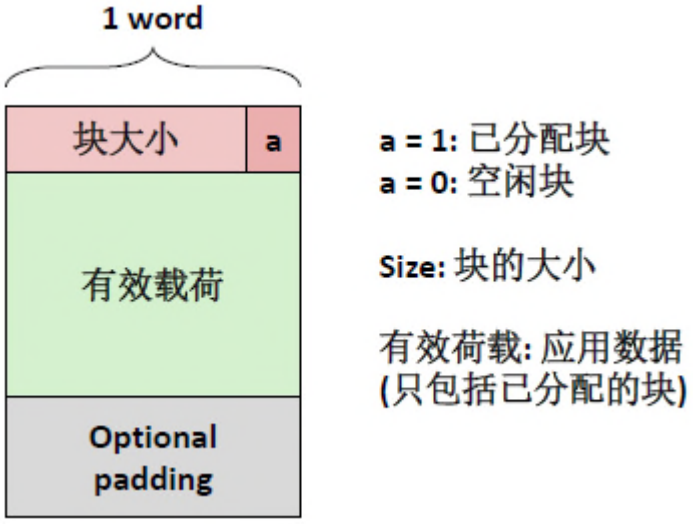
动态内存分配器维护着一个进程的虚拟内存区域，称为堆（heap）（见下图）。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在为初始化的数据区域后开始，并向上生长（向更高的地址）。对于每一个进程，内核维护着一个变量 brk，它指向堆的顶部。

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。



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：

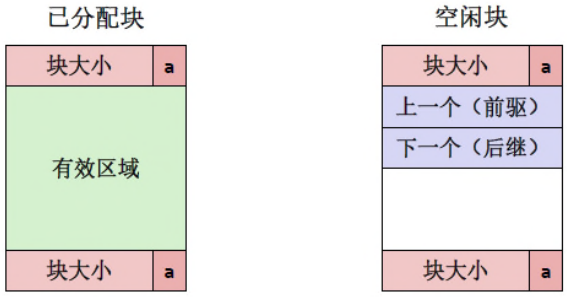


一个块是由一个字的头部、有效载荷，以及可能的一些额外的填充组成，头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。头部后面就是应用调用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。

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。

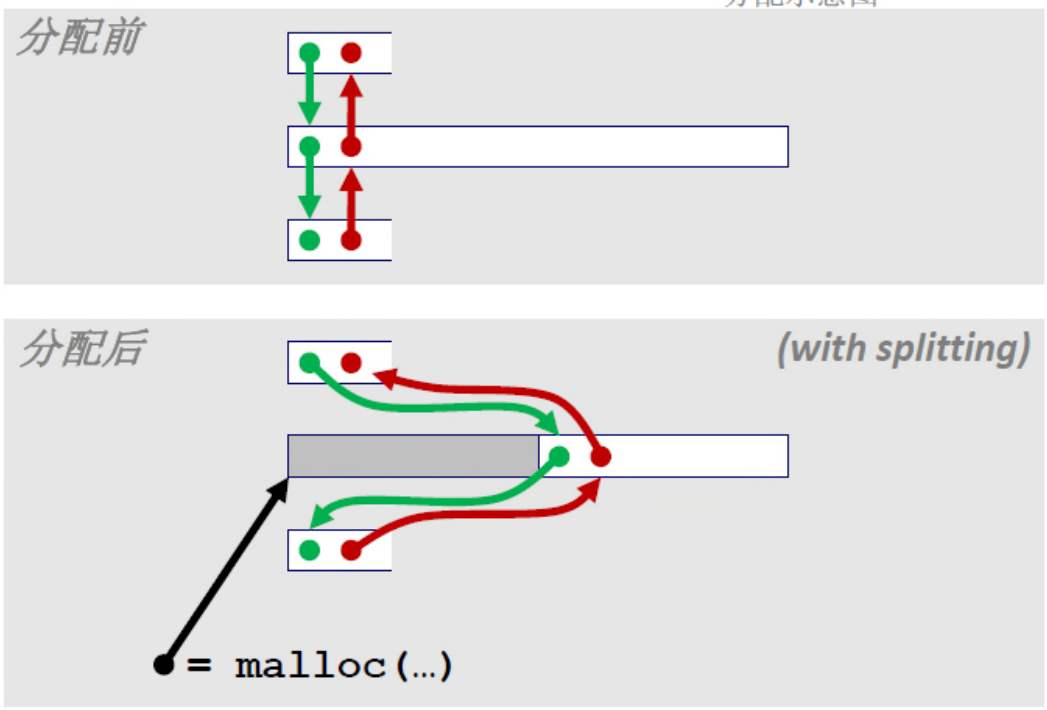
一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。使用 LIFO 的顺序和首次适配的放置策略，分配器会先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在线性时间内完成。

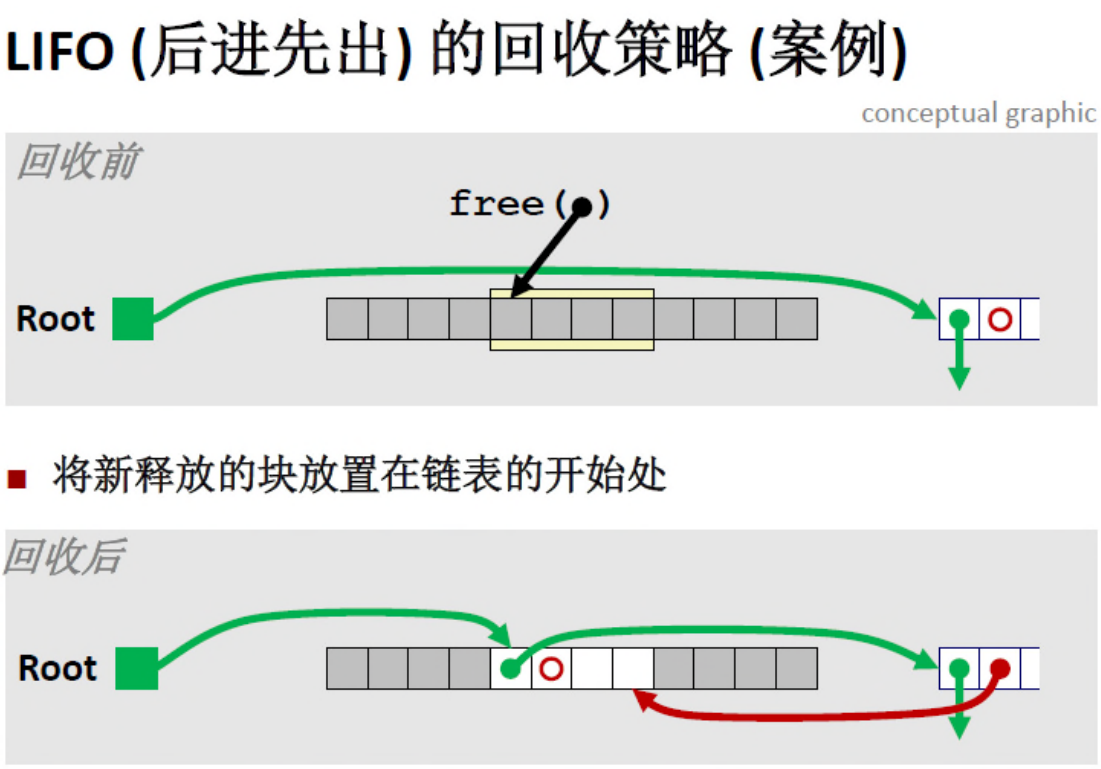
另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

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。

Allocating From Explicit Free Lists显式空闲链表的分配

分配示意图





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。

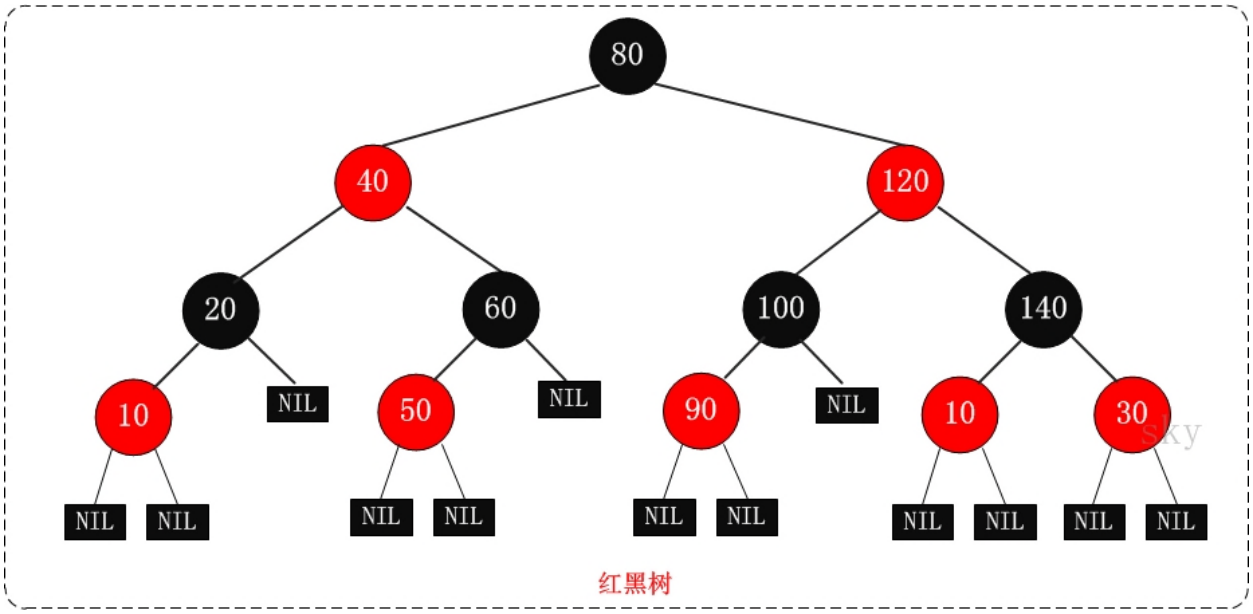
（2）根节点是黑色。

（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;
}
```

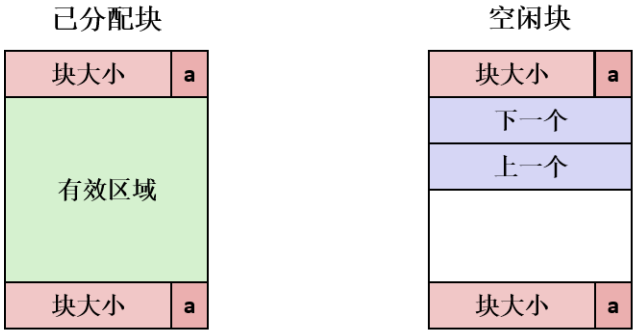
第 3 章 分配器的设计与实现

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。

采用：显式空闲链表+首次适配+立即合并



堆的格式如图：

```
* begin                                                                                                 end
* heap                                                                                                  heap
*  ---------------------------------------------------------------------------------------------------
*  |hdr(16:a)|pred|succ|ftr(16:a)|  pad   | hdr(8:a) | ftr(8:a) | zero or more usr blks | hdr(8:a) |
*  ---------------------------------------------------------------------------------------------------
*  |          root node          |        |       prologue      |                       | epilogue |
*  |            block            |        |         block       |                       |   block  |
*
```

解释：为了插入、删除操作的方便，在堆顶放置一个表头。

每个块增加前趋(pred)、后趋(succ)节点指针。

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。

除了程序中定义的宏，我自行增加了两个宏：
#define PRED(bp) (GET(bp))
#define SUCC(bp) (GET((char *)(bp) + WSIZE))
用于访问某块的前趋指针和后趋指针。
同时，定义了一个静态全局变量：
static char *root;
用于保存根节点的地址

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：应用程序（例如轨迹驱动的测试程序 mdriver）在使用 mm_malloc、mm_realloc 或 mm_free 之前，首先要调用该函数进行初始化。例如申请初始堆区域。返回值：0 表示正常，-1 表示有错误；

处理流程：mm_init 函数从内存系统得到 8 个字，并将它们初始化，创建一个空的空闲链表、显式空闲链表的表头。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：

调用 mem_sbrk 函数，从内存系统得到字

初始化空的空闲链表、显式空闲链表的表头

扩展堆、创建初始空闲块

代码实现：

```
int mm_init(void)
{
    if ((heap_listp = mem_sbrk(8 * WSIZE)) == NULL) //4 WSIZE -> 7 WSIZE
        return -1;

    /*build root node*/
    char *root_start = heap_listp;
    PUT(root_start, PACK(16, 1)); //hdr
    root = root_start + WSIZE;    //set root
    SUCC(root) = (size_t)NULL;    //succ = NULL
    PRED(root) = (size_t)NULL;    //pred = NULL
    PUT(root + DSIZE, PACK(16, 1)); //set foot

    heap_listp += DSIZE + DSIZE;

    /* create the initial empty heap */
    PUT(heap_listp, 0);                          /* alignment padding */
    PUT(heap_listp + WSIZE, PACK(OVERHEAD, 1));  /* prologue header */
    PUT(heap_listp + DSIZE, PACK(OVERHEAD, 1));  /* prologue footer */
    PUT(heap_listp + WSIZE + DSIZE, PACK(0, 1)); /* epilogue header */
    heap_listp += DSIZE;

#ifdef NEXT_FIT
    rover = heap_listp;
#endif

    /* 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，没有返回值。
参　　数：指针值 ptr 应该是之前调用 mm_malloc 或 mm_realloc 返回的值，并且没有释放过。
处理流程：释放所请求的块（bp），然后使用边界标记合并技术将之与邻接的空闲块合并起来。
要点分析：该函数比较简单
1. 宏 HDRP(bp)返回指向这个块的头部的指针
2. 宏 GET_SIZE(HDRP(bp))从地址(HDRP(bp)处的头部分别返回大小和已分配位
3. PACK(OVERHEAD, 1)将大小和已分配位结合起来
4. PUT 宏将 PACK(OVERHEAD, 1)存放在参数 heap_listp+WSIZE 指向的字节中。

代码实现：

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

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

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 指向内存块，size 内存块大小
处理流程：
1. 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。

代码实现：

```
void *mm_realloc(void *ptr, size_t size)
{
```

```
    size_t oldsize;
    void *newptr;

    /* 如果参数 size 为 0，等价于 mm_free(ptr) */
    if (size == 0)
    {
        mm_free(ptr);
        return 0;
    }

    /* 如 ptr 是空指针 NULL,等价于 mm_malloc(size) */
    if (ptr == NULL)
    {
        return mm_malloc(size);
    }

    newptr = mm_malloc(size);

    /*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 */
    /* 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_realloc(void *ptr, size_t size)
{
    size_t oldsize;
    void *newptr;

    /* 如果参数 size 为 0，等价于 mm_free(ptr) */
    if (size == 0)
    {
        mm_free(ptr);
        return 0;
    }

    /* 如 ptr 是空指针 NULL,等价于 mm_malloc(size) */
    if (ptr == NULL)
    {
        return mm_malloc(size);
    }

    newptr = mm_malloc(size);

    /*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 */
    /* 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;
}
```

3.2.5 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）。申请的整个块应该在对的区间内，并且不能与其他已经分配的块重叠。返回的地址应该是 8 字节对齐的（地址%8==0）

参　　数：size：申请内存块的大小

处理流程：

1. 在检查完请求的真假之后，分配器调整请求块的大小
2. 搜索空闲链表，寻找一个合适的空闲块
3. 如果有合适的，放置这个请求块
4. 如果没有合适的，用一个新的空闲块来拓展堆

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。

代码实现：

```
void *mm_malloc(size_t size)
{
    size_t asize;      /* adjusted block size */
    size_t extendsize; /* amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size <= 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
        asize = DSIZE + OVERHEAD;
    else
        asize = DSIZE * ((size + (OVERHEAD) + (DSIZE - 1)) / 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;
```

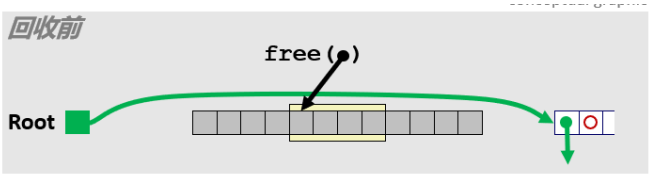
```
    place(bp, asize);

    return bp;
}
```

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

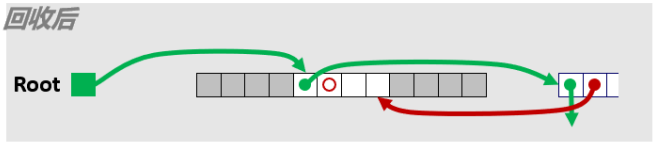
函数功能：将要回收的空闲块和临近的空闲块（如果有的话）合并成一个大的空闲块。

处理流程：

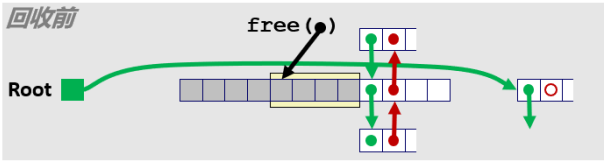
1. 前面的块和后面的块都是已分配的：不进行合并，当前块的状态只是简单地从分配变为空闲。同时，将其插入空闲链表的开头。



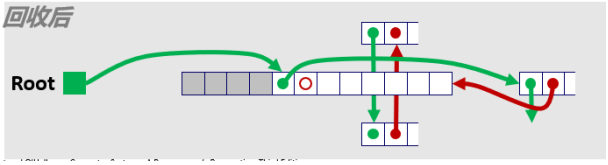
- 将新释放的块放置在链表的开始处



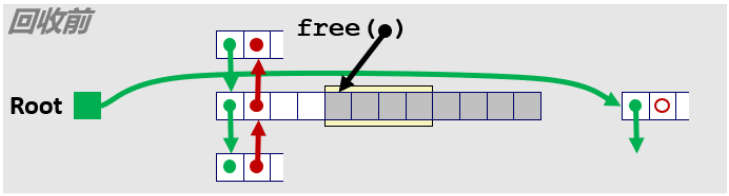
2. 前面的块是已分配的，后面的块是空闲的：当前块与后面的块合并，用当前块和后面的块的大小的和来更新当前块的头部和后面块的脚部。同时，将新合成的块插入空闲链表的开头，将后面的块从原有空闲链表中删去。



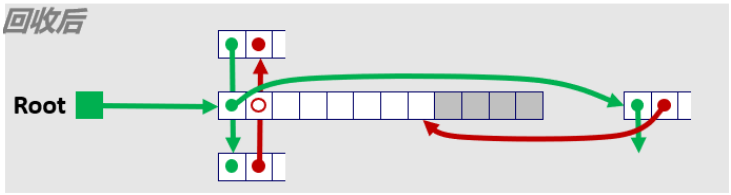
- 将后继块拼接出来，合并两个内存块，并在列表的开始处插入新块



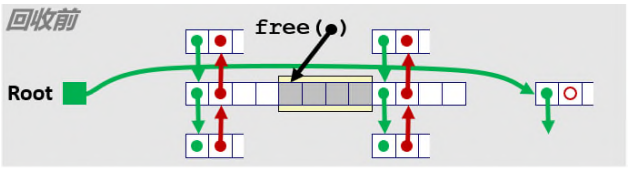
3. 前面的块是空闲的，后面的块是已分配的：前面的块和当前块合并，用两个块的大小的和来更新前面块的头部和当前块的脚部。同时，将新合成的块插入空闲链表的开头，将前面的块从原有空闲链表中删去。



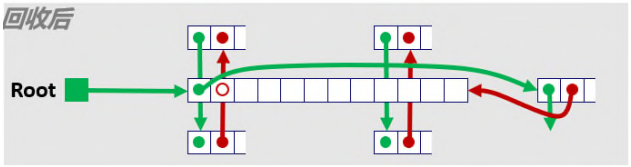
- 拼接出前块，合并两个内存块，并在列表的开始处插入新的块



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。同时，将新合成的块插入空闲链表的开头，将前面、后面的块从原有空闲链表中删去。



- 将前块和后继块拼接起来，将所有3个内存块合并并在列表的开始处插入新块



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。

同样的，增加的表头节点，省去了删除节点时，还需要判断是否为 root 节点的麻烦。

第 4 章测试

总分 10 分

4.1 测试方法

1. 对单个轨迹文件进行测试：

```
./mdriver -v -V -a -l -f traces/ amptjp-bal.rep
./mdriver -v -V -a -l -f traces/ cccp-bal.rep
./mdriver -v -V -a -l -f traces/ cp-decl-bal.rep
./mdriver -v -V -a -l -f traces/ expr-bal.rep
./mdriver -v -V -a -l -f traces/ coalescing-bal.rep
./mdriver -v -V -a -l -f traces/ random-bal.rep
./mdriver -v -V -a -l -f traces/ random2-bal.rep
./mdriver -v -V -a -l -f traces/ binary-bal.rep
./mdriver -v -V -a -l -f traces/ binary2-bal.rep
./mdriver -v -V -a -l -f traces/ realloc-bal.rep
./mdriver -v -V -a -l -f traces/ realloc2-bal.rep
```

2. 整体测试：

```
./mdriver -v
```

4.2 测试结果评价

1. 如果参照 csapp 书中对动态内存分配器的设计，采用隐式空闲链表+首次适配+立即合并，测试结果为：perf index = 44(util)+7(thru)=51/100, 跑出来的峰值利用率还挺高，但是吞吐量很小，因为隐式空闲链表每次 malloc 的时候需要从头遍历所有的块，以及立即合并可能会产生抖动。

3. 我采用的，显式空闲链表+首次适配+立即合并的方法测试结果为：Perf index = 43 (util) + 40 (thru) = 83/100。显式空闲链表的结构，使得寻找空闲块时，遍历的是整个空闲块，而不是所有块，因此时间复杂度下降，吞吐量大大提高。

4.3 自测试结果

1. 隐式空闲链表+首次适配+立即合并

```
xjy1170500913@ubuntu:~/桌面/hitics/lab8/malloclab-handout$ ./mdriver -v
Team Name:implicit first fit
Member 1 :Xiong Jianyu:1170500913
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.009113   625
 1       yes   99%    5848  0.007698   760
 2       yes   99%    6648  0.013121   507
 3       yes  100%    5380  0.009652   557
 4       yes   66%   14400  0.000107134958
 5       yes   92%    4800  0.009246   519
 6       yes   92%    4800  0.008214   584
 7       yes   55%   12000  0.175876    68
 8       yes   51%   24000  0.343469    70
 9       yes   27%   14401  0.075185   192
10       yes   34%   14401  0.002600  5539
Total          74%  112372  0.654281   172

Perf index = 44 (util) + 11 (thru) = 56/100
```

2.（优化后）显式空闲链表+首次适配+立即合并

```
xjy1170500913@ubuntu:~/桌面/hitics/lab8/malloclab-handout$ ./mdriver -v
Team Name:explicit first fit
Member 1 :Xiong Jianyu:1170500913
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   92%    5694  0.000253 22479
 1       yes   94%    5848  0.000159 36849
 2       yes   96%    6648  0.000258 25777
 3       yes   97%    5380  0.000213 25294
 4       yes   66%   14400  0.000152 94799
 5       yes   89%    4800  0.000510  9414
 6       yes   85%    4800  0.000554  8672
 7       yes   55%   12000  0.004273  2808
 8       yes   51%   24000  0.003565  6731
 9       yes   26%   14401  0.075794   190
10       yes   34%   14401  0.002599  5541
Total          71%  112372  0.088329  1272

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

深入理解了计算机系统虚拟存储的基本知识，掌握了C语言指针相关的基本操作，深入理解了动态存储申请、释放的基本原理和相关系统函数，用 C 语言实现了动态存储分配器，并进行了测试分析

5.2 请给出对本次实验内容的建议

暂无

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.