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 1190200910

班　　级 1903012

学 生 姓 名 严幸

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 2021-06-09

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 Code

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”。假设堆是一个请求二进制零的区域，它在 Linux 的内存映像中紧接在未初始化的数据区域后开始，并向上生长(向更高的地址)。对于每个进程，内核维护着一个变量 brk，它是一个始终指向堆的顶部的指针。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(chunk)，要么是已分配的，要么是空闲的。已分配的块显式地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。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

动态内存分配器维护的堆中的每一个块，都包含一个头部(header)和脚部(footer)，分别位于这个块的开头和结尾。头部和脚部的内容完全相同，其中都包含了这个块的大小信息、是否已经分配情况。此外，每个块还包含有效载荷、额外填充(padding)的几个字节。由于块的大小都是 8 的倍数，因此块大小二进制表示的低三位一定全都是 0，据此，我们可以利用头部和脚部的最低位来表示这个块是否已分配。

这样一来，只需要从头开始遍历堆，用每个块的起始地址加上这个块的大小，就能得到下一个块的起始地址。由于块的头部包含了这个块的分配情况，因此通过这样的流程即可完成对整个堆中所有空闲块的遍历。并且遍历所有空闲块的时间与块的总数(包括已分配块和空闲块)呈线性关系。

为了搜索到一个足够大的未分配块，在遍历空闲块的过程中可以使用首次适配/下一次适配/最佳适配几种策略。找到空闲块之后，如果空闲块比已分配块大很多，可能还需要对空闲块进行分割，分割出的新的空闲块也要对头部和脚部进行标记，相当于将新的空闲块加到空闲链表中。如果遍历整个空闲链表也找不到一个足够大的空闲块用于分配，则分配器调用 sbrk 函数来申请额外的堆内存。在释放已分配块时，如果已分配块的前面或后面也存在空闲块，则要进行合并。由于

每个块的开始和结束分别有一个头部和脚部，因此可以在常数时间内得到前一块和后一块的大小及分配情况，从而进行合并。

2.3 显式空闲链表的基本原理（5 分）

采用显式空闲链表的方案，对于已分配块的表示，与隐式空闲链表相同。对于空闲块，头部、脚部、padding 的策略都与隐式空闲链表相同，二者的主要区别在于，空闲块的有效载荷被充分利用起来，在空闲块有效载荷的开头存放两个指针，分别指向它的前一个空闲块和后一个空闲块。这样一来，从第一个空闲块开始，可以直接找到下一个空闲块，而无需经过已分配块。在这个方案下，遍历所有空闲块的时间与空闲块总数呈线性关系。在已分配块很多时，显式空闲链表会比隐式空闲链表的效率高得多。

显式空闲链表的空闲块搜索方式、空闲块分割策略、堆内存不足处理策略、释放空闲块合并策略，都与隐式空闲链表相同。显式空闲链表的主要优势在于遍历空闲块时省去了很多无用的已分配块的遍历，从而大大提高了效率。

2.4 红黑树的结构、查找、更新算法（5 分）

1) 红黑树的结构

R-B Tree，全称是 Red-Black Tree，又称为“红黑树”，它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)。

红黑树的特性:

- 每个节点或者是黑色，或者是红色。
- 根节点是黑色。
- 每个叶子节点是黑色。
- 如果一个节点是红色的，则它的子节点必须是黑色的。
- 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。

红黑树的应用比较广泛，主要是用它来存储有序的数据，它查找的时间复杂度是 O(lgn)，效率非常之高。

2) 红黑树的查找

红黑树是一棵二叉查找树，一个节点的左儿子的数据一定小于等于它自身的数值，右儿子的数据一定大于它自身的数值。因此在查找时，根据数值大小关系递归查找左子树/右子树即可。算法如下：

```
Node* find(ElementType target, Node* node){
    if(node -> data == target)
```

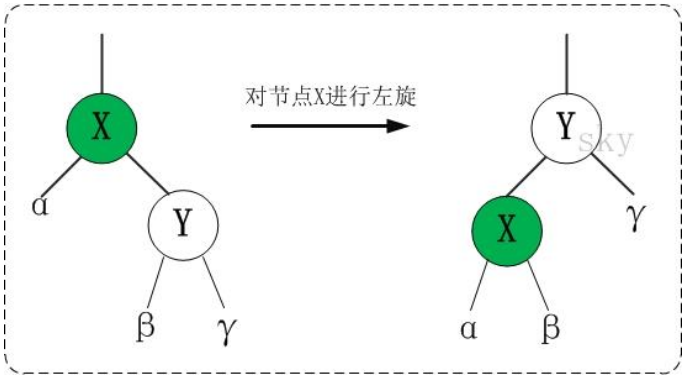
```
        return node;
    else if(target < node -> data)
        Return find(target, node -> left);
    else
        Return find(target, node -> right);
}
```

3) 红黑树的旋转

红黑树的基本操作是添加、删除。在对红黑树进行添加或删除之后，可能这棵树就不再满足红黑树的五条性质，这个时候就需要利用旋转，来将这棵树的结构重新构造为一个红黑树。旋转包括两种：左旋和右旋。下面分别对它们进行介绍。

左旋：

对 x 进行左旋，意味着“将 x 变成一个左节点”。

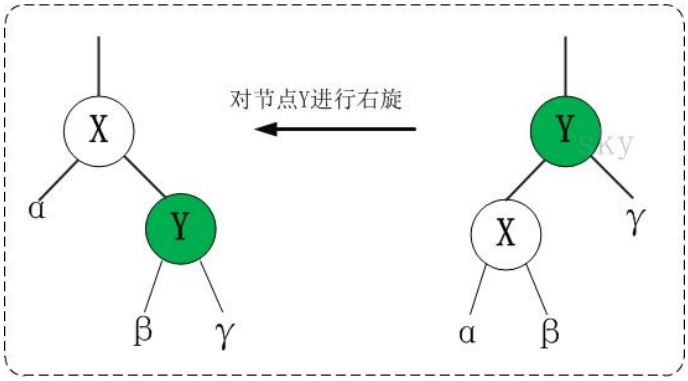


左旋的伪代码如下：

```
LEFT-ROTATE(T, x)
01  y ← right[x]                  // 前提：这里假设 x 的右孩子为 y。下面开始正式操作
02  right[x] ← left[y]            // 将 “y 的左孩子” 设为 “x 的右孩子”，即 将 β 设为 x 的右孩子
03  p[left[y]] ← x                // 将 “x” 设为 “y 的左孩子的父亲”，即 将 β 的父亲设为 x
04  p[y] ← p[x]                   // 将 “x 的父亲” 设为 “y 的父亲”
05  if p[x] = nil[T]
06  then root[T] ← y              // 情况 1：如果 “x 的父亲” 是空节点，则将 y 设为根节点
07  else if x = left[p[x]]
08            then left[p[x]] ← y  // 情况 2：如果 x 是它父节点的左孩子，则将 y 设为“x 的父节点的左孩子”
09            else right[p[x]] ← y // 情况 3：(x 是它父节点的右孩子) 将 y 设为“x 的父节点的右孩子”
10  left[y] ← x                   // 将 “x” 设为 “y 的左孩子”
11  p[x] ← y                      // 将 “x 的父节点” 设为 “y”
```

右旋：

对 x 进行右旋，意味着“将 x 变成一个右节点”。



右旋的伪代码如下：

```
RIGHT-ROTATE(T, y)
01  x ← left[y]                  // 前提：这里假设 y 的左孩子为 x。下面开始正式操作
02  left[y] ← right[x]           // 将 “x 的右孩子” 设为 “y 的左孩子”，即 将 β 设为 y 的左孩子
03  p[right[x]] ← y              // 将 “y” 设为 “x 的右孩子的父亲”，即 将 β 的父亲设为 y
04  p[x] ← p[y]                  // 将 “y 的父亲” 设为 “x 的父亲”
05  if p[y] = nil[T]
06  then root[T] ← x                    // 情况 1：如果 “y 的父亲” 是空节点，则将 x 设为根节点
07  else if y = right[p[y]]
08            then right[p[y]] ← x      // 情况 2：如果 y 是它父节点的右孩子，则将 x 设为“y 的父节点的左孩子”
09            else left[p[y]] ← x       // 情况 3：(y 是它父节点的左孩子) 将 x 设为“y 的父节点的左孩子”
10  right[x] ← y                 // 将 “y” 设为 “x 的右孩子”
11  p[y] ← x                     // 将 “y 的父节点” 设为 “x”
```

4) 红黑树的插入

插入时，首先将红黑树当作一颗二叉查找树，将节点插入(根据节点数据和插入数据的大小关系，递归地寻找一个合适的位置插入数据)；然后，将节点着色为红色；最后，通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。将节点插入后的操作详细描述如下：

先将插入的节点着色为红色，不会违背红黑树的第 5 条性质“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”。

然后根据不同情况采取不同策略：

a. 当前节点的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。策略：将“父节点”设为黑色；将“叔叔节点”设为黑色；将“祖父节点”设为“红色”；将“祖父节点”设为“当前节点”(红色节点)；即，之后继续对“当前节点”进行操作；

b. 当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子。策略：将“父节点”作为“新的当前节点”；以“新的当前节点”为支点进行左旋。

c. 当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子。策略：将“父节点”设为“黑色”；将“祖父节点”设为“红色”；以“祖父节点”为支点进行右旋。

整个算法的伪代码如下：

```
RB-INSERT-FIXUP(T, z)
01 while color[p[z]] = RED
02       do if p[z] = left[p[p[z]]]
03             then y ← right[p[p[z]]]
04                  if color[y] = RED
05                     then color[p[z]] ← BLACK
06                          color[y] ← BLACK
07                          color[p[p[z]]] ← RED
08                          z ← p[p[z]]
09                     else if z = right[p[z]]
10                             then z ← p[z]
11                                  LEFT-ROTATE(T, z)
12                             color[p[z]] ← BLACK
13                             color[p[p[z]]] ← RED
14                             RIGHT-ROTATE(T, p[p[z]])
15          else (same as then clause with "right" and "left" exchanged)
16 color[root[T]] ← BLACK
```

5) 红黑树的删除

首先，将红黑树当作一颗二叉查找树，将该节点从二叉查找树中删除；然后，通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。

删除节点后分 4 种情况处理：

a) x 是"黑+黑"节点，x 的兄弟节点是红色。(此时 x 的父节点和 x 的兄弟节点的子节点都是黑节点)。策略：将 x 的兄弟节点设为“黑色”；将 x 的父节点设为“红色”；对 x 的父节点进行左旋；左旋后，重新设置 x 的兄弟节点。

b) x 是“黑+黑”节点，x 的兄弟节点是黑色，x 的兄弟节点的两个孩子都是黑色。策略：将 x 的兄弟节点设为“红色”；设置“x 的父节点”为“新的 x 节点”。

c) x 是“黑+黑”节点，x 的兄弟节点是黑色；x 的兄弟节点的左孩子是红色，右孩子是黑色的。策略：将 x 兄弟节点的左孩子设为“黑色”；将 x 兄弟

节点设为“红色”；对 x 的兄弟节点进行右旋；右旋后，重新设置 x 的兄弟节点。

d) x 是“黑+黑”节点，x 的兄弟节点是黑色；x 的兄弟节点的右孩子是红色的，x 的兄弟节点的左孩子任意颜色。策略：将 x 父节点颜色 赋值给 x 的兄弟节点；将 x 父节点设为“黑色”；将 x 兄弟节点的右子节设为“黑色”；对 x 的父节点进行左旋；设置“x”为“根节点”。

整个算法伪代码如下：

```
RB-DELETE-FIXUP(T, x)
01 while x ≠ root[T] and color[x] = BLACK
02       do if x = left[p[x]]
03             then w ← right[p[x]]
04                  if color[w] = RED
05                     then color[w] ← BLACK
06                          color[p[x]] ← RED
07                          LEFT-ROTATE(T, p[x])
08                          w ← right[p[x]]
09                  if color[left[w]] = BLACK and color[right[w]] = BLACK
10                     then color[w] ← RED
11                          x ← p[x]
12                     else if color[right[w]] = BLACK
13                             then color[left[w]] ← BLACK
14                                  color[w] ← RED
15                                  RIGHT-ROTATE(T, w)
16                                  w ← right[p[x]]
17                           color[w] ← color[p[x]]
18                           color[p[x]] ← BLACK
19                           color[right[w]] ← BLACK
20                           LEFT-ROTATE(T, p[x])
21                           x ← root[T]
22          else (same as then clause with "right" and "left" exchanged)
23 color[x] ← BLACK
```

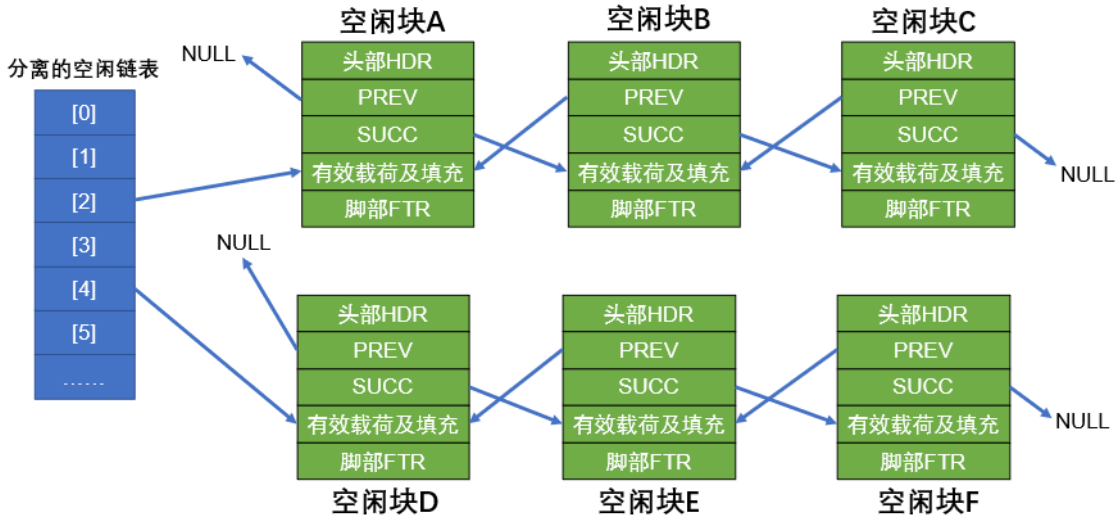
第 3 章 分配器的设计与实现

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。

总体采用了分离的显式空闲链表来实现分配器。在每个链表中采用“分离适配”组织块的大小结构以提高内存利用率，即不同链表中的块对应不同的大小类，每一个链表中的块又具有不同的大小而不是取这个大小类中的最大值。此外，每一个链表中的不同大小块是按从小到大的顺序排列的，这样可以在一定程度上提高查找效率。



具体来说，链表的设计如上图所示。用一个数组来保存所有链表的所有头节点地址。每一个链表是一个大小类，链表[0]是 1 字节的大小类，链表[1]是 2 字节的大小类，链表 2 是 3~4 字节的大小类，链表 3 是 5~8 字节的大小类，链表 i 是大小 $2^{i-1}-1 \sim 2^{i}$ 的大小类。

在每一个大小类内部，也就是每一个链表内部有若干个空闲块，空闲块的有效载荷的前两个字用于存放它前一块和后一块的指针，指针分别指向它在链表中的前驱块和后继块的有效载荷的第一个字节。一个链表中的块的大小都属于这个链表的大小类，但是链表中块的序列是按块的大小排序的，例如在上图中，空闲块 A 的大小小于空闲块 B 的大小，空闲块 B 的大小小于空闲块 C 的大小；空闲块 D 小于空闲块 E 的大小，空闲块 E 小于空闲块 F 的大小，但空闲库 D、E、F 都属于 9~16 字节的大小类。

搜索空闲块时，根据申请内存的大小找到对应的大小类，然后遍历这个大小类对应的空闲链表，找到一个满足大小要求的空闲块之后，如果空闲块比申请的空间多余的字节足以成为一个新的空闲块，则进行分割，将分割出的新块插入到

它对应的空闲链表当中；否则的话不分割，直接将这一块分配出去。如果一整个链表都找不到一个足够大的块，则调用 sbrk()函数申请额外堆内存，并插入到对应的链表中，再将这一个新块分配出去。

在释放已分配块时，先进行合并，再将合并后的空闲块插入到对应的空闲链表当中即可。

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：初始化空闲链表及堆空间
处理流程：

先将空闲链表中的头指针赋为 NULL。然后初始化 heap_listp 指针以标记堆的开始位置，之后就可以初始化序言块和尾言块。然后申请一个初始的堆内存，即将堆拓展到一个最小片的大小 minChunkSize。

要点分析：

初始化链表采用一个 for 循环即可：

```
for (i = 0; i < MAX_LEN; i++)
{
    sepFreeLists[i] = NULL;
}
```

初始化堆空间时，需要调用 mem_sbrk，申请 4 个字的空间，然后返回给 heap_lisp 标记。

```
heap_listp = mem_sbrk(4 * WSIZE)) == NULL
```

初始化序言块和尾言块时，根据它们的定义，赋予不同的值即可。

```
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 */
```

初始化堆中的最小片时，无法确定最小片的大小给多少字节是最优的，我一开始选择了 4KB，然后在其他代码都写完之后，逐步调整最小片的内存大小，发现最小片给予 64 字节是最优的。

```
size_t minChunkSize = 1 << 6; //经过多次调整最小的 CHUNKSIZE 是 64B
extend_heap(minChunkSize / WSIZE)
```

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放一个已经申请过的空间
参　　数：void *ptr，指向要释放的空间的起始位置
处理流程：

先修改 ptr 指向的块的头部和脚部，将这个块由已分配修改为未分配标记。然

后把这个空闲块插入到空闲链表当中。最后再合并这个块前后可能的空闲块。

要点分析：

由于良好的封装性，这个函数只需要 5 行代码即可完成。

```
size_t size = GET_SIZE(HDRP(bp));
PUT(HDRP(bp), PACK(size, 0));
PUT(FTRP(bp), PACK(size, 0));
insertNode(bp, size);
coalesce(bp);
```

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

函数功能：给一片已分配空间重新分配空间，保持空间内原有内容不变

参　　数：void *ptr，指向之前的那一片空间，size_t size，重新分配空间后的大小。

处理流程：

先将 size 加上两个字(头部和脚部)之后向双字对齐，如果 size 小于 ptr 原来指向的块的大小，则直接返回，不进行任何操作。否则的话进行下面的操作

检查 ptr 指向的块在堆中的下一个块是否是空闲块(即 NEXT_BLKP 宏定义的下一块，地址连续的下一块)，如果是空闲块并且加上这个空闲块之后空间大小满足 size 的条件，则将两块合并，然后返回 ptr，否则进行拓展块。如果 ptr 后面没有可以利用的连续的空闲块，则调用 malloc 申请新的块，进行内容复制之后返回新块的首地址。

要点分析：

size 向双字对齐：

```
if (size <= DSIZE)
    size = 2 * DSIZE;
else
    size = ALIGN(size + DSIZE);
```

检查 ptr 块在堆中的下一块并完成潜在的堆扩展：

```
else if (!GET_ALLOC(HDRP(NEXT_BLKP(ptr))) || !GET_SIZE(HDRP(NEXT_BLKP(ptr))))
{
    /* 如果加上后面连续地址上的未分配块空间也不够，那么需要扩展块 */
    if ((remaining = GET_SIZE(HDRP(ptr)) + GET_SIZE(HDRP(NEXT_BLKP(ptr))) - size) < 0)
    {
        if (extend_heap(MAX(-remaining, CHUNKSIZE) / WSIZE) == NULL)
            return NULL;
        remaining += MAX(-remaining, CHUNKSIZE);
    }

    /* 从分离空闲链表中删除刚刚利用的未分配块并设置新块的头尾 */
    deleteNode(NEXT_BLKP(ptr));
    PUT(HDRP(ptr), PACK(size + remaining, 1));
    PUT(FTRP(ptr), PACK(size + remaining, 1));
}
```

必须申请新块的情况，先 malloc，然后调用 memcpy 完成内容的复制，最后还要记得 free 旧的空间：

```
else
{
    new_p = mm_malloc(size);
    memcpy(new_p, ptr, GET_SIZE(HDRP(ptr)));
    mm_free(ptr);
}
```

3.2.4 int mm_check(void)函数（5 分）

函数功能：堆的一致性检查

处理流程：

先检查序言块，然后遍历堆中的每一块，最后检查尾言块。对每个块的检查是通过调用 checkblock 函数来完成的。在检查的过程中如果 verbose 为 1 的话就会打印一些冗余的信息。

要点分析：

检查序言块：

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

用一个 for 循环遍历每一个块并检查：

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

最后检查尾言块：

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

其中，checkblock()函数会检查块的地址是否对齐到了 8 字节，还会检查块的头部和尾部是否一致。

```
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：分配一个大小至少为 size 字节的块，返回块的有效载荷的首地址。
参　　数：size_t size，目标分配块的大小，单位是字节。
处理流程：

先将 size 加上双字(块的头部和尾部)之后向 8 字节对齐，得到目标块的实际大小 asize。然后根据 asize 的大小找到它对应的大小类，从而确定目标块位于哪一个空闲链表当中，然后在这个空闲链表中寻找一个满足 asize 大小要求的空闲块，找到之后调用 place()函数将对应的块标记为已分配并完成潜在的分割需求。

如果 asize 的大小类对应的空闲链表中没有满足 asize 大小要求的空闲块，则调用 extend_heap 函数申请扩展堆，然后在新申请到的块上放置这个已分配块。

要点分析：

通过下面的代码将 size 加上 overhead 并向 8 字节对齐：

```
if (size <= DSIZE)
    size = DSIZE + OVERHEAD;
else
    size = ALIGN(size + DSIZE);
asize = size;
```

通过两个嵌套的 while 循环找到 asize 的大小类对应的链表，并在链表中找到合适的块：(如果没有找到合适的空闲块，则经过两层 while 循环之后，bp 指针仍为 NULL)

```
while (i < MAX_LEN)
{
    /* 先找合适的空闲链表 */
    if (((asize <= 1) && (sepFreeLists[i] != NULL)))
    {
        //到这里时原先的asize大于等于2的i次方小于2的i+1次方
        bp = sepFreeLists[i];
        //遍历列表寻找大小合适的块，链表中的块是按大小升序排列的
        while ((bp != NULL) && ((size > GET_SIZE(HDRP(bp)))))
            bp = SUCC(bp);
        /* 找到对应的未分配的块 */
        if (bp != NULL)
            break;
    }
    asize >>= 1;
    i++;
}
```

最后处理没有找到合适的块的情况，并调用 place 函数完成分配块的放置。

```
/* 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) / WSIZE)) == NULL)
        return NULL;
}
/* 在未分配块中放置size大小的块 */
bp = place(bp, size);
```

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

函数功能：合并相邻空闲块，并返回合并后的指针。

处理流程：

先查到 bp 的前一块和后一块的分配情况，根据它们分配情况的四种可能组合分别采取不同的策略。

情况 1：bp 的前一块和后一块都已经分配。则无需合并，直接返回 bp 即可。

情况 2：bp 的前一块已分配，后一块空闲。则先将 bp 块和后一块都从空闲链表中删除，然后修改 bp 块和后一块的头部和脚部完成合并，最后将合并后的新块添加到空闲链表。

情况 3：bp 的前一块空闲，后一块已分配。则先将 bp 块和前一块都从空闲链表中删除，然后修改 bp 块和前一块的头部和脚部完成合并，最后将合并后的新块添加到空闲链表。

情况 4：bp 的前一块和后一块都空闲。则将 bp 块、前一块、后一块三个块都从空闲链表中删除，然后修改三个块的头部脚部完成合并，最后将合并后的新块添加到空闲链表。

要点分析：

每一种情况都采用相同的处理模式：先从空闲链表删除合并前的块，然后修改要合并块的头部和脚部就相当于完成了合并，最后将合并后的新块添加到空闲链表。这个过程中可能需要更新 bp 指针的值。

情况 1：bp 的前一块和后一块都已经分配。直接返回 bp。

```
if (prev_alloc && next_alloc)
    return bp;
```

情况 2：bp 的前一块已分配，后一块空闲。合并后新块的大小是 bp 块和后一块的大小之和，合并后的新块的有效载荷的起始地址仍为 bp 指针所指向，因此无需修改 bp 的值。

```
else if (prev_alloc && !next_alloc)
{
    //bp的前一块分配了，后一块未分配，与后一块合并
    deleteNode(bp);
    deleteNode(NEXT_BLKP(bp));
    size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
    //合并后，bp仍指向合并块的有效载荷的第一个字节
    PUT(HDRP(bp), PACK(size, 0));
    //将bp头部的size修改后，FTRP(bp)已经指向修改后的脚部
    PUT(FTRP(bp), PACK(size, 0));
}
```

情况 3：bp 的前一块空闲，后一块已分配。合并后新块的大小是 bp 块和前一块的大小之和，合并后的新块的有效载荷的起始地址应该是 bp 的前一块的有效载荷地址，因此需要更新 bp 的值。

```
else if (!prev_alloc && next_alloc)
{
    //bp的后一块分配了，前一块未分配，与前一块合并
    deleteNode(bp);
    deleteNode(PREV_BLKP(bp));
    size += GET_SIZE(HDRP(PREV_BLKP(bp)));
    //合并后的块的有效载荷的第一个字节位于bp的前一块当中
    bp = PREV_BLKP(bp);
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
}
```

情况 4：bp 的前一块和后一块都空闲。合并后新块的大小是 bp、前一块、后一块的大小之和，合并后的新块的有效载荷的起始地址应该是 bp 的前一块的有效载荷地址，因此需要更新 bp 的值。

```
else
{
    //prev 和 next都未分配
    deleteNode(bp);
    deleteNode(PREV_BLKP(bp));
    deleteNode(NEXT_BLKP(bp));
    size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(HDRP(NEXT_BLKP(bp)));
    //合并后的块的有效载荷的第一个字节位于bp的前一块当中
    bp = PREV_BLKP(bp);
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
}
```

第 4 章测试

总分 10 分

4.1 测试方法

在 handout 文件夹下使用 make 命令编译源文件。然后调用下面的命令进行测试：

./mdriver -t traces -a -v

4.2 测试结果评价

如果将堆中初始片的大小设为常见的 4KB，则测试效果能达到 95 分，主要是在 4 号测试轨迹上跑分较低。

经过堆初始片的大小进行多次调优，可以发现初始片大小为 64B 时，可以跑到综合的最高分 98 分。其中，吞吐率单项得分已经达到了 40 分，已经是这一项得分的上限，因此性能提高的瓶颈就在于内存利用率。

对每一个轨迹文件进行单独分析，发现跑分最低的两个测试点是 6 号测试点和 8 号测试点，内存空间利用率得分分别是 91%和 88%，两个测试点分别是 **random2** 和 **binary2**。推测可能是由于采取了带边界标签的空闲链表造成了额外的内存开销，但是这样的内存开销可以换来程序运行速度的效益，这样的折中显然是值得的。

4.3 自测试结果

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab8/malloclab-handout ./mdriver -t traces -a -v
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000224 25397
 1       yes   97%    5848  0.000240 24377
 2       yes   99%    6648  0.000253 26318
 3       yes   99%    5380  0.000208 25878
 4       yes   99%   14400  0.000336 42895
 5       yes   94%    4800  0.000287 16754
 6       yes   91%    4800  0.000299 16054
 7       yes   95%   12000  0.000308 38974
 8       yes   88%   24000  0.002222 10802
 9       yes   99%   14401  0.000176 81777
10       yes   98%   14401  0.000152 94433
Total          96%  112372  0.004704 23888

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

空间利用率 58 分+吞吐量得分 40 分=98 分

第 5 章 总结

5.1 请总结本次实验的收获

- 深入理解了分离的、带边界标签的显式空闲链表的底层实现机理。
- 经历了一次艰难的软件外部参数调优的过程，锻炼了自身的耐力。
- 所设计的分配器已经很接近 C 语言库函数的 malloc 性能，获得了成就感。

5.2 请给出对本次实验内容的建议

- 实验总体设计难度较大，老师可以多给一些提示
- 链表中红黑树的设计如果不用结构体很难实现，我在红黑树的构造上浪费了大量时间，做了很多无用功，最后也没有成功，希望老师能够放宽对红黑树数据结构的使用限制。

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.

[7] https://blog.csdn.net/weixin_45406155/article/details/112513139