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 1190200523

班　　级 1903002

学　　生 石翔宇

指 导 教 师 郑贵滨

实 验 地 点 G709

实 验 日 期 2021.6.11

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

- 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

- Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- 16GB RAM
- 1TB HDD + 512G SSD

1.2.2 软件环境

- Windows 10 21H1
- Ubuntu 20.04 LTS

1.2.3 开发工具

- VSCode，CodeBlocks，gcc+gdb

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。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，它紧接在未初始化的数据区域后开始，并向上生长。对于每个进程，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。

分配器有两种基本风格，两种风格都要求应用显式地分配块，它们的不同之处在于由哪个实体来负责释放已分配的块。

- 显式分配器：要求应用显式地释放任何已分配的块。
- 隐式分配器：要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。

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、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来指明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。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

我们称这种结构称为隐式空闲链表，是因为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所有的块，从而间接地遍历整个空闲块

的集合。注意：此时我们需要某种特殊标记的结束块，可以是一个设置了已分配位而大小为零的终止头部。

2.3 显式空间链表的基本原理（5 分）

显式空闲链表结构将堆组织成一个双向空闲链表，在每个空闲块的主体中，都包含一个前驱和后继指针。

双向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是线性的，也可能是个常数，这取决于空闲链表中块的排序策略。

一种方法是用后进先出的顺序维护链表，将新释放的块放置在链表的开始处。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。

2.4 红黑树的结构、查找、更新算法（5 分）

1. 结构

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

a) 节点是红色或黑色。
b) 根是黑色。
c) 所有叶子都是黑色（叶子是 NIL 节点）。
d) 每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）
e) 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

这些约束确保了红黑树的关键特性：从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

要知道为什么这些性质确保了这个结果，注意到性质 4 导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质 5 所有最长的路径都

有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

在很多树数据结构的表示中，一个节点有可能只有一个子节点，而叶子节点包含数据。用这种范例表示红黑树是可能的，但是这会改变一些性质并使算法复杂。为此，本文中我们使用"nil 叶子"或"空（null）叶子"，如上图所示，它不包含数据而只充当树在此结束的指示。这些节点在绘图中经常被省略，导致了这些树好像同上述原则相矛盾，而实际上不是这样。与此有关的结论是所有节点都有两个子节点，尽管其中的一个或两个可能是空叶子。

2. 查找

由于红黑树是二叉查找树，故其满足二叉查找树的性质，查询方式与之相同。

3. 插入

我们首先以二叉查找树的方法增加节点并标记它为红色。（如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换和树旋转来调整。）下面要进行什么操作取决于其他临近节点的颜色。注意：

- 性质 1 和性质 3 总是保持着。
- 性质 4 只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。
- 性质 5 只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。

在下面的示意图中，将要插入的节点标为 N，N 的父节点标为 P，N 的祖父节点标为 G，N 的叔父节点标为 U。在图中展示的任何颜色要么是由它所处情形这些所作的假定，要么是假定所暗含的。

对于每一种情形，我们将使用 C 示例代码来展示。通过下列函数，可以找到一个节点的叔父和祖父节点：

```
node* grandparent(node *n) {
    return n->parent->parent;
}

node* uncle(node *n) {
    if(n->parent == grandparent(n)->left)
        return grandparent (n)->right;
    else
        return grandparent (n)->left;
}
```

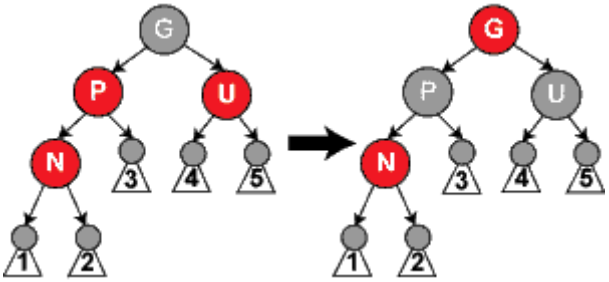
情形 1:新节点 N 位于树的根上，没有父节点。在这种情形下，我们把它重绘为黑色以满足性质 2。因为它在每个路径上对黑节点数目增加一，性质 5 符合。

```
void insert_case1(node *n) {
    if(n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2 (n);
}
```

情形 2:新节点的父节点 P 是黑色，所以性质 4 没有失效（新节点是红色的）。在这种情形下，树仍是有效的。性质 5 也未受到威胁，尽管新节点 N 有两个黑色叶子子节点；但由于新节点 N 是红色，通过它的每个子节点的路径就都有同通过它所取代的黑色的叶子的路径同样数目的黑色节点，所以依然满足这个性质。

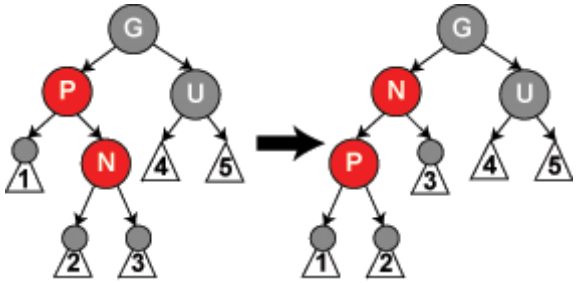
```
void insert_case2(node *n) {
    if(n->parent->color == BLACK)
        return; /* 树仍旧有效*/
    else
        insert_case3 (n);
}
```

情形 3:如果父节点 P 和叔父节点 U 二者都是红色，（此时新插入节点 N 做为 P 的左子节点或右子节点都属于情形 3，这里右图仅显示 N 做为 P 左子的情形）则我们可以将它们两个重绘为黑色并重绘祖父节点 G 为红色（用来保持性质 5）。现在我们的新节点 N 有了一个黑色的父节点 P。因为通过父节点 P 或叔父节点 U 的任何路径都必定通过祖父节点 G，在这些路径上的黑节点数目没有改变。但是，红色的祖父节点 G 可能是根节点，这就违反了性质 2，也有可能祖父节点 G 的父节点是红色的，这就违反了性质 4。为了解决这个问题，我们在祖父节点 G 上递归地进行情形 1 的整个过程。（把 G 当成是新加入的节点进行各种情形的检查）




```
void insert_case3(node *n){
    if(uncle(n) != NULL && uncle (n)->color == RED) {
        n->parent->color = BLACK;
        uncle (n)->color = BLACK;
        grandparent (n)->color = RED;
        insert_case1(grandparent(n));
    }
    else
        insert_case4 (n);
}
```

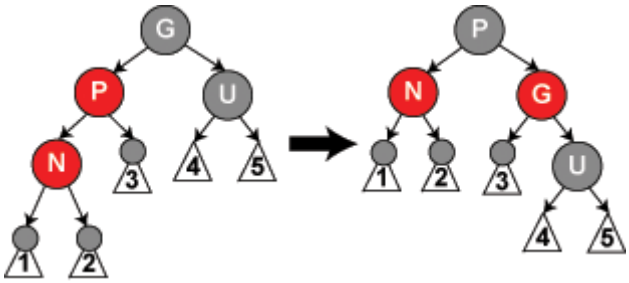
情形 4:父节点 P 是红色而叔父节点 U 是黑色或缺少，并且新节点 N 是其父节点 P 的右子节点而父节点 P 又是其父节点的左子节点。在这种情形下，我们进行一次左旋转调换新节点和其父节点的角色;接着，我们按情形 5 处理以前的父节点 P 以解决仍然失效的性质 4。注意这个改变会导致某些路径通过它们以前不通过的新节点 N（比如图中 1 号叶子节点）或不通过节点 P（比如图中 3 号叶子节点），但由于这两个节点都是红色的，所以性质 5 仍有效。




```
void insert_case4(node *n){
    if(n == n->parent->right && n->parent == grandparent(n)->left) {
        rotate_left(n);
        n = n->left;
    } else if(n == n->parent->left && n->parent == grandparent(n)->right) {
        rotate_right(n);
        n = n->right;
    }
    insert_case5 (n);
}
```

情形 5：父节点 P 是红色而叔父节点 U 是黑色或缺少，新节点 N 是其父节点的左子节点，而父节点 P 又是其父节点 G 的左子节点。在这种情形

下，我们进行针对祖父节点 G 的一次右旋转；在旋转产生的树中，以前的父节点 P 现在是新节点 N 和以前的祖父节点 G 的父节点。我们知道以前的祖父节点 G 是黑色，否则父节点 P 就不可能是红色（如果 P 和 G 都是红色就违反了性质 4，所以 G 必须是黑色）。我们切换以前的父节点 P 和祖父节点 G 的颜色，结果的树满足性质 4。性质 5 也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过祖父节点 G，现在它们都通过以前的父节点 P。在各自的情形下，这都是三个节点中唯一的黑色节点。




```
void insert_case5(node *n) {
    n->parent->color = BLACK;
    grandparent (n)->color = RED;
    if(n == n->parent->left && n->parent == grandparent(n)->left) {
        rotate_right(n->parent);
    } else {
        /* Here, n == n->parent->right && n->parent == grandparent (n)->right */
        rotate_left(n->parent);
    }
}
```

4. 删除

如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题（为了表述方便，这里所指的儿子，为非叶子节点的儿子）。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们要么找到它左子树中的最大元素、要么找到它右子树中的最小元素，并把它的值转移到要删除的节点中（如在这里所展示的那样）。我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子。因为只是复制了一个值（没有复制颜色），不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

在本文余下的部分中，我们只需要讨论删除只有一个儿子的节点（如果它两个儿子都为空，即均为叶子，我们任意将其中一个看作它的儿子）。如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父

亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它，并不会破坏性质 3 和性质 4。通过被删除节点的所有路径只是少了一个红色节点，这样可以继续保证性质 5。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏性质 5，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质 5。

需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况（这种情况下该节点的两个儿子都是叶子节点，否则若其中一个儿子是黑色非叶子节点，另一个儿子是叶子节点，那么从该节点通过非叶子节点儿子的路径上的黑色节点数最小为 2，而从该节点到另一个叶子节点儿子的路径上的黑色节点数为 1，违反了性质 5）。我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为 N（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为 S。在下面的示意图中，我们还是使用 P 称呼 N 的父亲，SL 称呼 S 的左儿子，SR 称呼 S 的右儿子。我们将使用下述函数找到兄弟节点：

```
struct node *
sibling(struct node *n)
{
        if(n == n->parent->left)
                return n->parent->right;
        else
                return n->parent->left;
}
```

我们可以使用下列代码进行上述的概要步骤，这里的函数 replace_node 替换 child 到 n 在树中的位置。出于方便，在本章节中的代码将假定空叶子被用不是 NULL 的实际节点对象来表示（在插入章节中的代码可以同任何一种表示一起工作）。

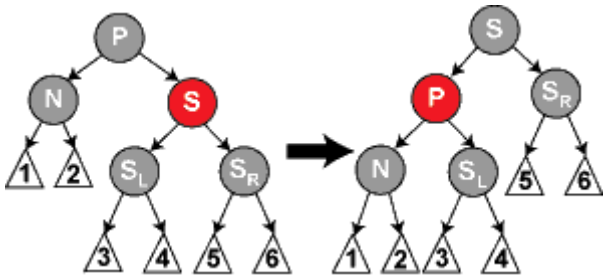
```
void
delete_one_child(struct node *n)
{
        /*
         * Precondition: n has at most one non-null child.
         */
        struct node *child = is_leaf(n->right)? n->left : n->right;

        replace_node(n, child);
        if(n->color == BLACK){
                if(child->color == RED)
                        child->color = BLACK;
                else
                        delete_case1 (child);
        }
        free (n);
}
```

如果 N 和它初始的父亲是黑色，则删除它的父亲导致通过 N 的路径都比不通过它的路径少了一个黑色节点。因为这违反了性质 5，树需要被重新平衡。有几种情形需要考虑：

情形 1: N 是新的根。在这种情形下，我们就做完了。我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

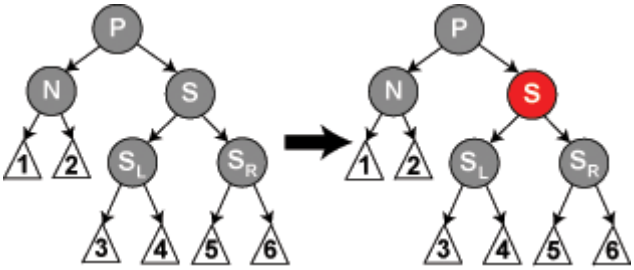
```
void
delete_case1(struct node *n)
{
        if(n->parent != NULL)
                delete_case2 (n);
}
```

情形 2： S 是红色。在这种情形下我们在 N 的父亲上做左旋转，把红色兄弟转换成 N 的祖父，我们接着对调 N 的父亲和祖父的颜色。完成这两个操作后，尽管所有路径上黑色节点的数目没有改变，但现在 N 有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色 S 的一个儿子），所以我们可以接下去按情形 4、情形 5 或情形 6 来处理。



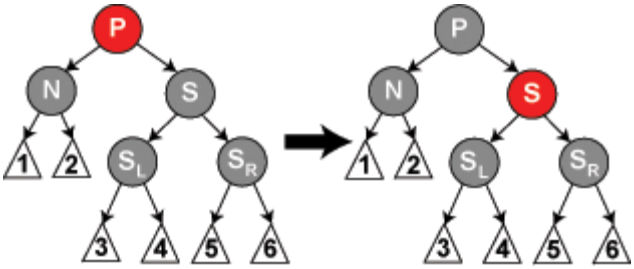
```
void
delete_case2(struct node *n)
{
        struct node *s = sibling (n);

        if(s->color == RED) {
                n->parent->color = RED;
                s->color = BLACK;
                if(n == n->parent->left)
                        rotate_left(n->parent);
                else
                        rotate_right(n->parent);
        }
        delete_case3 (n);
}
```

情形 3： N 的父亲、S 和 S 的儿子都是黑色的。在这种情形下，我们简单的重绘 S 为红色。结果是通过 S 的所有路径，它们就是以前不通过 N 的那些路径，都少了一个黑色节点。因为删除 N 的初始的父亲使通过 N 的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过 P 的所有路径现在比不通过 P 的路径少了一个黑色节点，所以仍然违反性质 5。要修正这个问题，我们要从情形 1 开始，在 P 上做重新平衡处理。



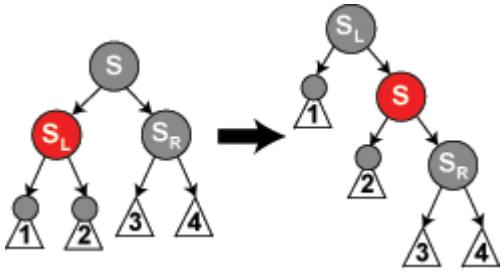

```
void
delete_case3(struct node *n)
{
        struct node *s = sibling (n);

        if((n->parent->color == BLACK)&&
(s->color == BLACK)&&
(s->left->color == BLACK)&&
(s->right->color == BLACK)) {
                s->color = RED;
                delete_case1(n->parent);
        } else
                delete_case4 (n);
}
```

情形 4： S 和 S 的儿子都是黑色，但是 N 的父亲是红色。在这种情形下，我们简单的交换 N 的兄弟和父亲的颜色。这不影响不通过 N 的路径的黑色节点的数目，但是它在通过 N 的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。




```
void
delete_case4(struct node *n)
{
        struct node *s = sibling (n);

        if ((n->parent->color == RED)&&
(s->color == BLACK)&&
(s->left->color == BLACK)&&
(s->right->color == BLACK)) {
                s->color = RED;
                n->parent->color = BLACK;
        } else
                delete_case5 (n);
}
```

情形 5： S 是黑色，S 的左儿子是红色，S 的右儿子是黑色，而 N 是它父亲的左儿子。在这种情形下我们在 S 上做右旋转，这样 S 的左儿子成为 S 的父亲和 N 的新兄弟。我们接着交换 S 和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在 N 有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了情形 6。N 和它的父亲都不受这个变换的影响。



```
void
delete_case5(struct node *n)
{
        struct node *s = sibling (n);

        if (s->color == BLACK) { /* this if statement is trivial,
due to Case 2(even though Case two changed the sibling to a sibling's child,
the sibling's child can't be red, since no red parent can have a red child). */
// the following statements just force the red to be on the left of the left of the parent,
// or right of the right, so case six will rotate correctly.
                if((n == n->parent->left)&&
(s->right->color == BLACK)&&
(s->left->color == RED)) { // this last test is trivial too due to cases 2-4.
                        s->color = RED;
                        s->left->color = BLACK;
                        rotate_right (s);
                } else if((n == n->parent->right)&&
(s->left->color == BLACK)&&
(s->right->color == RED)) {// this last test is trivial too due to cases 2-4.
                        s->color = RED;
                        s->right->color = BLACK;
                        rotate_left (s);
                }
        }
        delete_case6 (n);
}
```

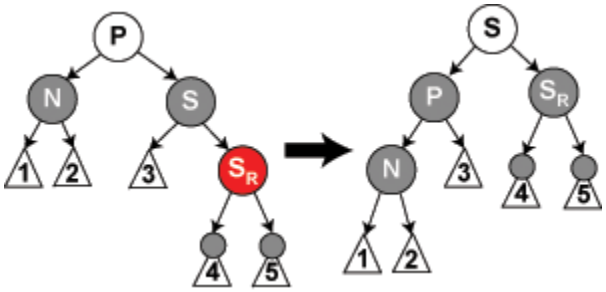
情形 6： S 是黑色，S 的右儿子是红色，而 N 是它父亲的左儿子。在这种情形下我们在 N 的父亲上做左旋转，这样 S 成为 N 的父亲（P）和 S 的右儿子的父亲。我们接着交换 N 的父亲和 S 的颜色，并使 S 的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以性质 3 没有被违反。但是，N 现在增加了一个黑色祖先：要么 N 的父亲变成黑色，要么它是黑色而 S 被增加为一个黑色祖父。所以，通过 N 的路径都增加了一个黑色节点。

此时，如果一个路径不通过 N，则有两种可能性：

它通过 N 的新兄弟。那么它以前和现在都必定通过 S 和 N 的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。

它通过 N 的新叔父，S 的右儿子。那么它以前通过 S、S 的父亲和 S 的右儿子，但是现在只通过 S，它被假定为它以前的父亲的颜色，和 S 的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。

在任何情况下，在这些路径上的黑色节点数目都没有改变。所以我们恢复了性质 4。在示意图中的白色节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。




```
void
delete_case6(struct node *n)
{
        struct node *s = sibling (n);

        s->color = n->parent->color;
        n->parent->color = BLACK;

        if(n == n->parent->left){
                s->right->color = BLACK;
                rotate_left(n->parent);
        } else {
                s->left->color = BLACK;
                rotate_right(n->parent);
        }
}
```

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。假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长。对于每个进程，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. 堆中内存块的组织结构：

   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，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来指明这个块是已分配的还是空闲的。

3. 采用的空闲块、分配块链表/树结构：

   采用分离的空闲链表。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. 相应算法：

   a) 插入：

   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。按照不同的情况来插入。

b) 删除：

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，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 关键函数设计（40 分）

3.2.1 int mm_init(void)函数（5 分）

函数功能：

初始化内存管理程序。

处理流程：

1. 调用 mem_sbrk 从内存中得到四个字，并将它们初始化。
2. 第一个字是一个双字边界对齐不使用的填充字。
3. 填充后面紧跟着一个特殊的序言块。
4. 堆的结尾以一个特殊的结尾块来结束。
5. 调用 extend_heap 函数创建初始的空闲块。

要点分析：

要对堆结构有足够的了解。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：

释放一个块。

参　　数：

void *ptr

处理流程：

1. 调用 GET_SIZE 获取块大小。
2. 调用 PUT 将块的头部和尾部的表示分配的位置设置为 0。
3. 调用 coalesce 将块与前后块合并（如果可以合并的话）。

要点分析：

注意释放的块需要和与之相邻的空闲块使用边界标记合并。

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

函数功能：

向 ptr 所指的块重新分配一个具有至少 size 字节的块。

参　　数：

void *ptr, size_t size

处理流程：

1. 调用 mm_malloc 申请一个具有至少 size 字节的块。
2. 调用 GET_SIZE 获取原来块的内容大小。
3. 调用 memcpy 将原来块的内容复制到新的块中。
4. 调用 mm_free 将原来的块释放。

要点分析：

注意在复制块内容之前，需要将新旧块的大小进行比较，若新的块大小小于旧的块，需要将复制块大小改为新的块的大小。

3.2.4 int mm_checkheap(int verbose)函数（5 分）

函数功能：

检查堆的一致性。

处理流程：

1. 调用 GET_SIZE 和 GET_ALLOC 检查开始块，若序言块不是 8 字节的已分配块，则输出 Bad prologue header。
2. 对于链表中的每一个块，调用 checkblock 函数。
3. 调用 GET_SIZE 和 GET_ALLOC 检查结尾块，若结尾块不是大小为 0 的已分配块，则输出 Bad epilogue header。

要点分析：

记得要检查 verbose，若不为 0，则调用 printblock 函数。

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

函数功能：

申请一个具有至少 size 字节的块。

参　　数：

size_t size

处理流程：

1. 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。
2. 调用 find_fit 搜索空闲链表，找到一个合适的空闲块。
3. 若找到了则调用 place 放置这个请求块。
4. 若没有找到，则调用 extend_heap 来扩展堆。

要点分析：

开始时要判断检查请求的真假。要考虑到没有合适的空闲块的情况。

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

函数功能：

合并相邻的空闲块。

处理流程：

1. 调用 GET_ALLOC 获得相邻块的已分配位。

2. 调用 GET_SIZE 获得当前块的大小。
3. 根据相邻块的分配情况，可以得到 4 种策略，分别进行处理：
   a) 前后都分配了，直接返回；
   b) 前一个块没有分配，与前一个块进行合并；
   c) 后一个块没有分配，与后一个块进行合并；
   d) 前后块都没有分配，与前后块进行合并。

要点分析：

注意要根据前后块的分配情况分别进行讨论。

第 4 章测试

总分 10 分

4.1 测试方法与测试结果(3 分)

- 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 mm_malloc, mm_realloc 和 mm_free
- 获得测试总分：linux>./mdriver -av -t traces/

测试结果：

```
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.006592   864
 1       yes  100%    5848  0.006303   928
 2       yes   99%    6648  0.010169   654
 3       yes  100%    5380  0.007529   715
 4       yes   97%   14400  0.000093154672
 5       yes   92%    4800  0.006554   732
 6       yes   92%    4800  0.006207   773
 7       yes   55%   12000  0.137069    88
 8       yes   51%   24000  0.267410    90
 9       yes   30%   14401  0.057216   252
10       yes   44%   14401  0.002049  7028
Total          78%  112372  0.507191   222

Perf index = 47 (util) + 15 (thru) = 62/100
```

4.2 测试结果分析与评价（3 分）

由于使用了隐式空闲链表，所以测试结果不甚理想。其中利用率（util）和速度（thru）与放置策略有关，对于分配操作，要求对空闲链表进行搜索，该搜索所需时间与堆中已分配块和空闲块的总数呈线性关系；同时，该策略会选择第一个满足要求的块，这也使得空间利用率不高。但也达到了预期的目标结果，完成了动态内存分配的基本操作要求。

4.4 性能瓶颈与改进方法分析（4 分）

性能瓶颈：

速度（thru）：

我们需要关注 malloc、free、realloc 每次操作的复杂度。

- malloc 的复杂度是与已分配块和空闲块的总数呈线性关系；
- free 的复杂度是常数时间的；
- realloc 的复杂度是与已分配块和空闲块的总数呈线性关系。

利用率（util）：

我们需要关注内部碎片和外部碎片，即我们 free 和 malloc 的时候要注意整体大块利用。但首次适配的放置策略使得我们无法很好地提高内存利用率。

改进方法：

我们可以选择下一次适配策略或者最佳适配。其中下一次适配策略比首次适配运行起来明显要快一些，且内存利用率要比首次适配低得多。最佳适配比上述两个适配策略的利用率都要高，但是速度会很慢。

我们也可以改变分配器结构。使用分离的空闲链表，包括简单分离存储和分离适配。

第 5 章 总结

5.1 请总结本次实验的收获

- 对内存管理有了更深的理解。

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.