Project 3_3
AVL 树→红黑树问题

班级：无 53

姓名：陈相宁

学号：2015011033

Email：c.xiangning1997@gmail.com

问题描述：

在 Windows 的虚拟内存管理中，将 VAD 组织成 AVL 树。VAD 树是一种平衡二叉树。红黑树也是一种自平衡二叉查找，在 Linux2.6 及其以后版本的内核中，采用红黑树来维护内存块。

请尝试参考 Linux 源代码将 WRK 源代码中的 VAD 树由 AVL 树替换成红黑树。

一、 实验思路：

1. 红黑树（RB-Tree）学习：[1]

- 性质：
  1） 每个节点为红色或黑色
  2） 根节点是黑色的
  3） 每个叶子节点（树尾端 NULL 节点）是黑色的
  4） 如果一个节点是红色，那么其俩儿子都是黑色的
  5） 对任一节点，其到叶子节点的每一条路径都包含相同数目的黑色节点
- 红黑树的插入和插入修复：
  1） 插入根节点直接将根节点涂为黑色即可
  2） 插入节点父节点是黑色，则什么都不用做
  3） 插入节点父节点是红色，而祖父节点的另一儿子（叔叔节点）是红色，需插入修复
  4） 插入节点父节点是红色，叔叔节点是黑色，需插入修复
- 红黑树的删除和删除修复：
  涉及步骤较为复杂，此处不列举详细情况。

2. AVL 树与红黑树（RB-Tree）对比：

- 如果插入一个 node 引起了树的不平衡，AVL 和 RB-Tree 都是最多只需要 2 次旋转操作，即两者都是 O(1)；但是在删除 node 引起树的不平衡时，最坏情况下，AVL 需要维护从被删 node 到 root 这条路径上所有 node 的平衡性，因此需要旋转的量级 O(logN)，而 RB-Tree 最多只需 3 次旋转，只需要 O(1)的复杂度。

---

[1] 参考“The Art Of Programming” (https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md)

- AVL 的结构相较 RB-Tree 来说更为平衡，在插入和删除 node 更容易引起 Tree 的 unbalance，因此在大量数据需要插入或者删除时，AVL 需要 rebalance 的频率会更高。因此，RB-Tree 在需要大量插入和删除 node 的场景下，效率更高。但是，由于 AVL 高度平衡，因此 AVL 的 search 效率更高。

所以总体来说，AVL 树与红黑树（RB-Tree）各有利弊，但日常使用下总体统计性能 RB-Tree 应该是略高于 AVL 树的。

3. WRK 中的 AVL 树：

在 wrk-v1.2/base/inc/ps.h 中，AVL 树及节点定义如下：

```
typedef struct _MMADDRESS_NODE {
    union {
        LONG_PTR Balance : 2;
        struct _MMADDRESS_NODE *Parent;
    } u1;
    struct _MMADDRESS_NODE *LeftChild;
    struct _MMADDRESS_NODE *RightChild;
    ULONG_PTR StartingVpn;
    ULONG_PTR EndingVpn;
} MMADDRESS_NODE, *PMMADDRESS_NODE;
```

```
typedef struct _MM_AVL_TABLE {
    MMADDRESS_NODE  BalancedRoot;
    ULONG_PTR DepthOfTree: 5;
    ULONG_PTR Unused: 3;
#if defined (_WIN64)
    ULONG_PTR NumberGenericTableElements: 56;
#else
    ULONG_PTR NumberGenericTableElements: 24;
#endif
    PVOID NodeHint;
    PVOID NodeFreeHint;
} MM_AVL_TABLE, *PMM_AVL_TABLE;
```

函数具体实现在 wrk-v1.2/base/ntos/mm/addrsup.c 中。其中需要注意的是，AVL 树真正的 root 是 BalanceRoot 的 RightChild，而 LeftChild 是 NULL。

4. Linux 中的红黑树：

选取 Linux-2.6 版本。在 rbtree.c 以及 rbtree.h 中，红黑树有关操作如下：

红黑树节点定义如下：

```
100  struct rb_node
101  {
102      unsigned long  rb_parent_color;
103  #define RB_RED      0
104  #define RB_BLACK    1
105      struct rb_node *rb_right;
106      struct rb_node *rb_left;
107  } __attribute__((aligned(sizeof(long))));
108      /* The alignment might seem pointless, but allegedly CRIS needs it */
109
110  struct rb_root
111  {
112      struct rb_node *rb_node;
113  };
114
```

其中，rb_root 较特殊，该指针在移植时应特别注意。

一些基本操作接口如下：

```
116  #define rb_parent(r)   ((struct rb_node *)((r)->rb_parent_color & ~3))
117  #define rb_color(r)   ((r)->rb_parent_color & 1)
118  #define rb_is_red(r)   (!rb_color(r))
119  #define rb_is_black(r) rb_color(r)
120  #define rb_set_red(r)  do { (r)->rb_parent_color &= ~1; } while (0)
121  #define rb_set_black(r)  do { (r)->rb_parent_color |= 1; } while (0)
122
123  static inline void rb_set_parent(struct rb_node *rb, struct rb_node *p)
124  {
125      rb->rb_parent_color = (rb->rb_parent_color & 3) | (unsigned long)p;
126  }
127  static inline void rb_set_color(struct rb_node *rb, int color)
128  {
129      rb->rb_parent_color = (rb->rb_parent_color & ~1) | color;
130  }
131
132  #define RB_ROOT (struct rb_root) { NULL, }
133  #define rb_entry(ptr, type, member) container_of(ptr, type, member)
134
135  #define RB_EMPTY_ROOT(root) ((root)->rb_node == NULL)
136  #define RB_EMPTY_NODE(node) (rb_parent(node) == node)
137  #define RB_CLEAR_NODE(node) (rb_set_parent(node, node))
138
139  extern void rb_insert_color(struct rb_node *, struct rb_root *);
140  extern void rb_erase(struct rb_node *, struct rb_root *);
```

供外界使用的主要有获取父节点（rb_parent）、颜色（rb_color），判断节点颜色（rb_is_red、rb_is_black），设置节点颜色（rb_set_color、rb_set_black）以及对本次实验最为重要的 2 个外部函数 rb_insert_color 用于插入修复以及 rb_erase 用于删除修复。

二、 代码实现：

本次实验将 AVL 树转变为红黑树，因此只需将 rbtree.c 以及 rbtree.h 中的方法移植到 addrsup.c 中即可。

为了便于移植做出的几处更改：

```
117  #define rb_color(r)   ((r)->rb_parent_color & 1)
```

1. 由 rb_color 可见，rb_node 最低位表示颜色，为了避免指针带来的不必要错误，用 PMMADDRERSS 的 Balance 表示节点颜色。
2. AVL 树初始化时，Balance 为 0，但是红黑树初始化时根节点为黑色。因此为了尽量降低出错概率，将红色定义为 1，黑色定义为 0，这与 rb_tree.h 中的定义正好相反。

针对若干简单函数，只需将：

1. rb_node *替换为 PMMADDRESS_NODE
2. rb_root *替换为 PMM_AVL_TABLE
3. rb_left 替换为 LeftChild
4. rb_right 替换为 RightChild
5. root 替换为 root→BalancedRoot.RightChild 即可

具体修改代码如下：

```
47  //
48  // define the structure & method of rbtree
49  //
50
51  #define rb_black 0
52  #define rb_red 1
53
54  # define rb_parent(node) (SANITIZE_PARENT_NODE((node)->u1.Parent))
55  # define rb_color(node) ((node)->u1.Balance)
56  # define rb_is_red(node) (rb_color(node))
57  # define rb_is_black(node) (!rb_color(node))
58  # define rb_set_red(node) ((node)->u1.Balance = rb_red)
59  # define rb_set_black(node) ((node)->u1.Balance = rb_black)
60
61  static void
62  rb_set_parent(
63      PMMADDRESS_NODE rb, PMMADDRESS_NODE p)
64  {
65      rb->u1.Parent = (PMMADDRESS_NODE) (rb_color(rb) + (unsigned long)p);
66  }
67
```

rb_tree 左旋函数__rb_rotate_left：

```
68  static void
69  __rb_rotate_left(
70      PMMADDRESS_NODE node,
71      PMM_AVL_TABLE Table)
72  {
73      PMMADDRESS_NODE right = node->RightChild;
74      PMMADDRESS_NODE parent = rb_parent(node);
75
76      if ((node->RightChild = right->LeftChild))
77          rb_set_parent(right->LeftChild, node);
78      right->LeftChild = node;
79
80      rb_set_parent(right, parent);
81
82      if (parent)
83      {
84          if (node == parent->LeftChild)
85              parent->LeftChild = right;
86          else
87              parent->RightChild = right;
88      }
89      else
90          Table->BalancedRoot.RightChild = right;
91      rb_set_parent(node, right);
92  }
```

rb_tree 右旋函数__rb_rotate_right：

```
94  static void
95  __rb_rotate_right(
96      PMMADDRESS_NODE node,
97      PMM_AVL_TABLE Table)
98  {
99      PMMADDRESS_NODE left = node->LeftChild;
100     PMMADDRESS_NODE parent = rb_parent(node);
101
102     if ((node->LeftChild = left->RightChild))
103         rb_set_parent(left->RightChild, node);
104     left->RightChild = node;
105
106     rb_set_parent(left, parent);
107
108     if (parent)
109     {
110         if (node == parent->RightChild)
111             parent->RightChild = left;
112         else
113             parent->LeftChild = left;
114     }
115     else
116         Table->BalancedRoot.RightChild = left;
117     rb_set_parent(node, left);
118 }
```

移植时，只需更改源码中的 MiInsertNode 以及 MiRemoveNode 函数。

在 wrk-v1.2/base/ntos/mm/mi.h 中定义了这两个函数的外部接口：

```
5301  VOID
5302  FASTCALL
5303  MiInsertNode (
5304      IN PMMADDRESS_NODE Node,
5305      IN PMM_AVL_TABLE Root
5306      );
5307
5308  VOID
5309  FASTCALL
5310  MiRemoveNode (
5311      IN PMMADDRESS_NODE Node,
5312      IN PMM_AVL_TABLE Root
5313      );
5314
```

1. MiInsertNode 函数：

Linux rbtree.h 源码中用于插入节点的函数是 rb_link_node：

```
152 static inline void rb_link_node(struct rb_node * node, struct rb_node * parent,
153                 struct rb_node ** rb_link)
154 {
155     node->rb_parent_color = (unsigned long )parent;
156     node->rb_left = node->rb_right = NULL;
157
158     *rb_link = node;
159 }
```

但是如图所示，该函数输入为待删除节点、该节点的父节点以及根节点；而 MiInsertNode 函数输入只有待删除节点以及根节点，所以不能使用该函数。正确的做法应该是利用 MiInsertNode 中本来的插入节点部分，只是需要在插入节点后使用 rb_insert_color 以保持 rbtree 的结构。

判断节点插入位置代码如下：

```
{
    PMMADDRESS_NODE NodeOrParent;
    TABLE_SEARCH_RESULT SearchResult;

    ASSERT((Table->NumberGenericTableElements >= MiWorstCaseFill[Table->DepthOfTree]) &&
           (Table->NumberGenericTableElements <= MiBestCaseFill[Table->DepthOfTree]));

    SearchResult = MiFindNodeOrParent (Table,
                                       NodeToInsert->StartingVpn,
                                       &NodeOrParent);

    ASSERT (SearchResult != TableFoundNode);

    //
    // The node wasn't in the (possibly empty) tree.
    //
    // We just check that the table isn't getting too big.
    //

    ASSERT (Table->NumberGenericTableElements != (MAXULONG-1));

    NodeToInsert->LeftChild = NULL;
    NodeToInsert->RightChild = NULL;

    Table->NumberGenericTableElements += 1;
```

首先利用源码中的寻找符合节点函数 MiFindNodeOrParent，返回值为 TableEmptyTree、TableFoundNode、TableInsertAsLeft、TableInsertAsRight。具体意义如下：

```
    TABLE_SEARCH_RESULT - TableEmptyTree: The tree was empty.  NodeOrParent
                                          is *not* altered.

                          TableFoundNode: A node with the key is in the tree.
                                          NodeOrParent points to that node.

                          TableInsertAsLeft: Node with key was not found.
                                             NodeOrParent points to what would
                                             be parent.  The node would be the
                                             left child.

                          TableInsertAsRight: Node with key was not found.
                                              NodeOrParent points to what would
                                              be parent.  The node would be
                                              the right child.
```

依据这个函数便得到了节点的插入位置。

之后，根据 SearchResult 的结果分别处理：

```
1408    if (SearchResult == TableEmptyTree) {
1409        // The root of rbtree is Balance.RightChild actually
1410        Table->BalancedRoot.RightChild = NodeToInsert;
1411        rb_set_parent(NodeToInsert, &Table->BalancedRoot)
1412        ASSERT (NodeToInsert->u1.Balance == 0);
1413        ASSERT(Table->DepthOfTree == 0);
1414        Table->DepthOfTree = 1;
1415
1416    ASSERT((Table->NumberGenericTableElements >= MiWorstCaseFill[Table->DepthOfTree]) &&
1417           (Table->NumberGenericTableElements <= MiBestCaseFill[Table->DepthOfTree]));
1418
1419    }
1420    else {
1421
1422        PMMADDRESS_NODE R = NodeToInsert;
1423        PMMADDRESS_NODE S = NodeOrParent;
1424
1425        if (SearchResult == TableInsertAsLeft) {
1426            NodeOrParent->LeftChild = NodeToInsert;
1427        }
1428        else {
1429            NodeOrParent->RightChild = NodeToInsert;
1430        }
1431
1432        rb_set_parent(NodeToInsert, NodeOrParent);
1433
1434        //
1435        // Modify color to ensure the property of rbtree
1436        //
1437
1438        rb_insert_color(NodeToInsert, Table);
1439    }
1440
1441    //
1442    // Sanity check tree size and depth.
1443    //
1444
1445    ASSERT((Table->NumberGenericTableElements >= MiWorstCaseFill[Table->DepthOfTree]) &&
1446           (Table->NumberGenericTableElements <= MiBestCaseFill[Table->DepthOfTree]));
1447
1448    return;
1449 }
1450
```

- 如为 TableEmptyTree，则将将待插入点置于 Table→BalancedRoot.RightChild（rb_tree 实际 root 节点）
- 如为 TableInsertAsLeft 或 TableInsertAsRight，则根据 MiFindNodeOrParent 返回的 NodeOrParent 值插入到相应位置。之后核心的一步便是调用 rb_insert_color 函数调整颜色以保持红黑树的性质。

rb_insert_color 代码如下：

```
120 static void
121 rb_insert_color(
122     PMMADDRESS_NODE node,
123     PMM_AVL_TABLE Table)
124 {
125     PMMADDRESS_NODE parent, grandparent, judge_node
126     register PMMADDRESS_NODE uncle;
127
128     rb_set_red(node);
129     while ((parent = SANITIZE_PARENT_NODE(node->u1.Parent)) && rb_is_red(parent))
130     {
131         grandparent = rb_parent(parent);
132         if (parent == grandparent->LeftChild)
133         {
134             uncle = grandparent->RightChild;
135             judge_node = parent->RightChild;
136         }
137         else
138         {
139             uncle = grandparent->LeftChild;
140             judge_node = parent->LeftChild;
141         }
142         if (uncle && rb_is_red(uncle))
143         {
144             rb_set_black(uncle);
145             rb_set_black(parent);
146             rb_set_red(grandparent);
147             node = grandparent;
148             continue;
149         }
150         if (judge_node == node)
151         {
152             register PMMADDRESS_NODE tmp;
153             __rb_rotate_left(parent, Table);
154             tmp = parent;
155             parent = node;
156             node = tmp;
157         }
158         rb_set_black(parent);
159         rb_set_red(grandparent);
160         __rb_rotate_right(grandparent, Table);
161     }
162     rb_set_black(Table->BalancedRoot.RightChild);
163
164     return;
165 }
```

该函数实现逻辑与 linux-2.6 中逻辑一致，并分别对应实验思路红黑树学习插入和插入修复中的几种情况。

2. MiRemoveNode 函数：

Linux rbtree.h 源码中用于插入节点的函数是 rb_erase：

```
140  extern void rb_erase(struct rb_node *, struct rb_root *);
```

函数输入为待删除节点以及根节点；相应的，MiRemoveNode 函数输入为待删除节点以及根节点。这两个函数输入完全一致，因此可以直接利用 rb_erase。只不过需要调整用于删除节点后保持 rbtree 性质的__rb_erase_color 函数。

修改后__rb_erase_color 函数代码如下：

```
120  static void
121  __rb_erase_color(
122      PMMADDRESS_NODE node,
123      PMMADDRESS_NODE parent,
124      PMM_AVL_TABLE root)
125  {
126      PMMADDRESS_NODE other;
127
128      while ((!node || rb_is_black(node)) && node != root->BalancedRoot.RightChild)
129      {
130          if (parent->LeftChild == node)
131          {
132              other = parent->RightChild;
133              if (rb_is_red(other))
134              {
135                  rb_set_black(other);
136                  rb_set_red(parent);
137                  __rb_rotate_left(parent, root);
138                  other = parent->RightChild;
139              }
140              if ((!other->LeftChild || rb_is_black(other->LeftChild)) &&
141                  (!other->RightChild || rb_is_black(other->RightChild)))
142              {
143                  rb_set_red(other);
144                  node = parent;
145                  parent = rb_parent(node);
146              }
147              else
148              {
149                  if (!other->RightChild || rb_is_black(other->RightChild))
150                  {
151                      rb_set_black(other->LeftChild);
152                      rb_set_red(other);
153                      __rb_rotate_right(other, root);
154                      other = parent->RightChild;
155                  }
156                  other->u1.Balance = rb_color(parent);
157                  rb_set_black(parent);
158                  rb_set_black(other->RightChild);
159                  __rb_rotate_left(parent, root);
160                  node = root->BalancedRoot.RightChild;
161                  break;
162              }
163          }
```

```
        else
        {
            other = parent->LeftChild;
            if (rb_is_red(other))
            {
                rb_set_black(other);
                rb_set_red(parent);
                __rb_rotate_right(parent, root);
                other = parent->LeftChild;
            }
            if ((!other->LeftChild || rb_is_black(other->LeftChild)) &&
                (!other->RightChild || rb_is_black(other->RightChild)))
            {
                rb_set_red(other);
                node = parent;
                parent = rb_parent(node);
            }
            else
            {
                if (!other->LeftChild || rb_is_black(other->LeftChild))
                {
                    rb_set_black(other->RightChild);
                    rb_set_red(other);
                    __rb_rotate_left(other, root);
                    other = parent->LeftChild;
                }
                other->u1.Balance=parent->u1.Balance;
                rb_set_black(parent);
                rb_set_black(other->LeftChild);
                __rb_rotate_right(parent, root);
                node = root->BalancedRoot.RightChild;
                break;
            }
        }
    }
    if (node)
        rb_set_black(node);

    return;
}
```

该函数实现逻辑与 rbtree.c 中完全一致，只是替换为相应的数据结构。

而 MiRemoveNode 函数可以直接按照 rb_erase 的方式实现：

```
{
    PMMADDRESS_NODE child, parent;
    int color;

    if (!NodeToDelete->LeftChild)
        child = NodeToDelete->RightChild;
    else if(!NodeToDelete->RightChild)
        child = NodeToDelete->LeftChild;
    else
    {
        PMMADDRESS_NODE old = NodeToDelete, left;
        NodeToDelete = NodeToDelete->RightChild;

        while((left = NodeToDelete->LeftChild) != NULL)
            NodeToDelete = left;

        if(rb_parent(old))
        {
            if(rb_parent(old)->LeftChild == old)
                rb_parent(old)->LeftChild = NodeToDelete;
            else
                rb_parent(old)->RightChild = NodeToDelete;
        }
        else
            Table->BalancedRoot.RightChild = NodeToDelete;
```

```
1076            child = NodeToDelete->RightChild;
1077            parent = rb_parent(NodeToDelete);
1078            color = rb_color(NodeToDelete);
1079
1080            if (parent == old)
1081                parent = NodeToDelete;
1082            else
1083            {
1084                if (child)
1085                    rb_set_parent(child, parent);
1086                parent->LeftChild = child;
1087
1088                NodeToDelete->RightChild = old->RightChild;
1089                rb_set_parent(old->RightChild, NodeToDelete);
1090            }
1091
1092            NodeToDelete->u1.Parent = old->u1.Parent;
1093            NodeToDelete->LeftChild = old->LeftChild;
1094            rb_set_parent(old->LeftChild, NodeToDelete);
1095
1096            goto color;
1097        }
1098
1099        parent = rb_parent(NodeToDelete);
1100        color = rb_color(NodeToDelete);
1101
1102        if (child)
1103            rb_set_parent(child, parent);
1104        if (parent)
1105        {
1106            if (parent->LeftChild == NodeToDelete)
1107                parent->LeftChild = child;
1108            else
1109                parent->RightChild = child;
1110        }
1111        else
1112            Table->BalancedRoot.RightChild = child;
1113
1114    color:
1115        if (color == rb_black)
1116            __rb_erase_color(child, parent, Table);
1117
1118        Table->NumberGenericTableElements -= 1;
1119
1120        return;
1121    }
```

至此，代码移植工作完成。

接下来，在虚拟机上使用脚本将新的代码移植到操作系统内核即可。

```
1  path \wrk-v1.2\tools\x86;%path%
2  cd \wrk-v1.2\base\ntos
3  nmake -nologo x86=
4  cp C:\wrk\WRK-v1.2\base\ntos\BUILD\EXE\wrkx86.exe C:\WINDOWS\system32\wrkx86.exe
5  pause
6
```

三、 实验感想：

本次实验我阅读了大量 WRK 及 linux 源码，并成功将 AVL 树改为红黑树。实验原理其实不难，难度主要在找到并理解 windows 和 linux 文件系统的相关接口。本次实验收获主要在两方面：其一，这是我第一次接触操作系统编程及内核调试，算是打开一扇新世界的大门；其二，在阅读操作系统源码的过程中，我感受到了专业工程师们严谨的代码风格以及详实的注释，对于 debug 及他人阅读带来了很大的便利性。所以，我在以后的编程中也应自我鞭策，多加注意。