AVL 树/红黑树问题

无 81 马啸阳 2018011054

2020 年 5 月 31 日

1 实验题目

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树，在 Linux 2.6 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。

2 数据结构

2.1 Linux 红黑树

参考 Linux 5.6.14 版本，其中红黑树数据结构在三个文件内。

- include/linux/rbtree.h
- include/linux/rbtree_augmented.h
- lib/rbtree.c

其中说明文档位于 Documentation/rbtree.txt。

红黑树结构位于 rbtree.h 中，节点与树如下。

```
struct rb_node {
 unsigned long __rb_parent_color;
 struct rb_node *rb_right;
 struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
   /* The alignment might seem pointless, but allegedly CRIS needs it */

struct rb_root {
```

```c
  struct rb_node *rb_node;
};
```

其中，__rb_parent_color 的末位存储节点颜色，前部存放父节点指针（由于指针末两位必定为 0），通过位操作可以单独读写父节点指针或节点颜色。rbtree.h 和 rbtree_augmented.h 中还定义了一些基本的宏与内联函数用以辅助获得节点信息，例如 rbtree_augmented.h 中的如下代码以宏定义了红色与黑色，并给出了读写节点颜色、父节点等的方法。

```c
#define RB_RED  0
#define RB_BLACK 1

#define __rb_parent(pc) ((struct rb_node *)(pc & ~3))

#define __rb_color(pc) ((pc) & 1)
#define __rb_is_black(pc) __rb_color(pc)
#define __rb_is_red(pc) (!__rb_color(pc))
#define rb_color(rb)   __rb_color((rb)->__rb_parent_color)
#define rb_is_red(rb)  __rb_is_red((rb)->__rb_parent_color)
#define rb_is_black(rb) __rb_is_black((rb)->__rb_parent_color)

static inline void rb_set_parent(struct rb_node *rb, struct rb_node *p)
{
  rb->__rb_parent_color = rb_color(rb) | (unsigned long)p;
}

static inline void rb_set_parent_color(struct rb_node *rb,
               struct rb_node *p, int color)
{
  rb->__rb_parent_color = (unsigned long)p | color;
}

static inline void
__rb_change_child(struct rb_node *old, struct rb_node *new,
     struct rb_node *parent, struct rb_root *root)
{
  if (parent) {
    if (parent->rb_left == old)
      WRITE_ONCE(parent->rb_left, new);
    else
      WRITE_ONCE(parent->rb_right, new);
  } else
    WRITE_ONCE(root->rb_node, new);
}
```

rbtree.c 中的 __rb_insert() 用以插入节点（实际上还有 rbtree.h 中的 rb_link_node() 先进行连接，__rb_insert() 实际上是平衡操作），rb_erase() 用以删除节点，代码略去，其中

还有一些函数用于辅助，如 ____rb_erase_color()。

rbtree_augmented.h 中如 rb_insert_augmented() 和 rb_erase_augmented() 等函数是一些操作的增强版本。这些函数主要帮助实现带有额外数据的增强红黑树，在我们的移植中，基本都不用实现，遇到 augment_rotate() 或其它增强操作通常忽略即可。

2.2 WRK AVL 树

WRK 的 AVL 树主要代码都在 base/ntos/mm/addrsup.c 中。AVL 树节点和树的结构定义位于 base/ntos/inc/ps.h 中，如下所示。

```c
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;

//
// A pair of macros to deal with the packing of parent & balance in the
// MMADDRESS_NODE.
//

#define SANITIZE_PARENT_NODE(Parent) ((PMMADDRESS_NODE)(((ULONG_PTR)(Parent)) & ~0x3))

//
// Macro to carefully preserve the balance while updating the parent.
//

#define MI_MAKE_PARENT(ParentNode,ExistingBalance) \
                (PMMADDRESS_NODE)((ULONG_PTR)(ParentNode) | ((ExistingBalance) & 0x3))

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;
```

其中 MMADDRESS_NODE 为节点，其中使用了联合体储存了父节点和 Balance 字段，与 Linux 实现红黑树类似。ps.h 中也定义了读写父节点的位操作宏。而 MM_AVL_TABLE 则是 AVL 树的结构，与 Linux 不同，它的根实际上不是真正的节点，而相当于一个辅助节点。

addrsup.c 中定义了 AVL 树的各种操作，代码略去，其中插入节点是 MiInsertNode()，删除节点是 MiRemoveNode()，而 MiRebalanceNode() 函数是平衡整棵树，是插入删除的辅助。

2.3 移植设计

将 WRK 中的 AVL 树改为红黑树时，由于它们都是平衡二叉树，节点和树的结构定义都可以通用。树的查找等操作，AVL 树与红黑树也没有区别。节点中，我们使用 MMADDRESS_NODE 中的 u1.Balance 直接作为红黑树的节点颜色，这样二者的节点定义基本没有区别。

操作上，我们用 ___rb_insert() 和 rb_erase() 为原型替换 MiInsertNode() 和 MiRemoveNode()，需要注意 WRK 中判定是否为根不能像 Linux 中简单地判定父指针是否为 NULL，而需要将指针与 &Table->BalancedRoot 比较，因为 WRK 中有一个辅助根节点。因此需要区分 Linux 中判断一个指针是否为 NULL 是判断其是否为叶子的孩子还是判断其是否为根的父亲。其它诸如 MiRebalanceNode() 与 MiPromoteNode() 等函数，只是辅助插入删除节点，替换了插入与删除后实际上已经失去作用，不必修改（但为了保险起见，没有删除这些方法）。替换时注意 MiFindNodeOrParent() 返回的 TABLE_SEARCH_RESULT 信息含义。这一部分可以参考 AVL 树原先的写法。

另外仿照 Linux 的红黑树补上若干用于读写节点颜色、父节点的宏与函数即可。

3 程序代码

代码仅修改了 addrsup.c，附上的 addrsup.c 中以 MODIFY 注释标注了更改了的地方，可以直接搜索，主要是以下三部分。代码移植时参考了 WRK 的命名规范采用大驼峰式命名法并以 Mi 开头，均与 Linux 的代码有相应对应（以 rb 开头的下划线命名法）。

首先是参考 Linux 添加了一些宏定义与基本的函数操作。这里除了 MiPrintTreePreOrder() 外所有宏与函数均有 Linux 对应（MiRBParent() 和已有的 MiParent() 效果一样，但 MiParent() 返回的类型为 PRTL_SPLAY_LINKS，不兼容，且不宜直接修改，因此添加了 MiRBParent()）。MiPrintTreePreOrder() 是一个调试函数，打印出整个树的结构，使用了 DbgPrint() 先序遍历树。

```
#define RB_RED  0
#define RB_BLACK 1

#define MiIsBlack(Links) (                              \
    (Links->u1.Balance == RB_BLACK)                     \
    )

#define MiIsRed(Links) (                                \
    (Links->u1.Balance == RB_RED)                       \
    )

#define MiRBParent(Links) (                             \
    (SANITIZE_PARENT_NODE((Links)->u1.Parent))          \
    )

VOID
MiSetParentColor(
  IN PMMADDRESS_NODE Links,
  IN PMMADDRESS_NODE Parent,
  IN ULONG Color
)
{
  Links->u1.Parent = MI_MAKE_PARENT(Parent, Color);
}

VOID
MiSetParent(
    IN PMMADDRESS_NODE Links,
    IN PMMADDRESS_NODE Parent
)
{
    ULONG Color = Links->u1.Balance;
  MiSetParentColor(Links, Parent, Color);
}

VOID
MiChangeChild(
    IN PMMADDRESS_NODE Old,
    IN PMMADDRESS_NODE New,
    IN PMMADDRESS_NODE Parent
)
{
  if (Parent->LeftChild == Old)
    Parent->LeftChild = New;
  else
    Parent->RightChild = New;
```

```
}

VOID
MiRotateSetParents(
    IN PMMADDRESS_NODE Old,
    IN PMMADDRESS_NODE New,
    IN ULONG Color
)
{
    PMMADDRESS_NODE Parent = MiRBParent(Old);
  New->u1 = Old->u1;
  MiSetParentColor(Old, New, Color);
    MiChangeChild(Old, New, Parent);
}

VOID
MiSetBlack(
    IN PMMADDRESS_NODE Links
)
{
    Links->u1.Parent = MI_MAKE_PARENT(MiRBParent(Links), RB_BLACK);
}

VOID
MiPrintTreePreOrder(
    IN PMMADDRESS_NODE Links
)
{
    DbgPrint("Node %u, Parent %u, %u - %u, Color %d\n",
             (ULONG)Links,
             (ULONG)Links->u1.Parent,
             Links->StartingVpn,
             Links->EndingVpn,
             Links->u1.Balance);
    if (Links->LeftChild)
    MiPrintTreePreOrder(Links->LeftChild);
  if (Links->RightChild)
    MiPrintTreePreOrder(Links->RightChild);
}
```

MiRemoveNode() 修改如下，其中添加了一些调试语句。

```
VOID
FASTCALL
MiRemoveNode (
    IN PMMADDRESS_NODE NodeToDelete,
    IN PMM_AVL_TABLE Table
    )
```

```

/*++

Routine Description:

    This routine deletes the specified node from the balanced tree, rebalancing
    as necessary. If the NodeToDelete has at least one NULL child pointers,
    then it is chosen as the EasyDelete, otherwise a subtree predecessor or
    successor is found as the EasyDelete. In either case the EasyDelete is
    deleted and the tree is rebalanced. Finally if the NodeToDelete was
    different than the EasyDelete, then the EasyDelete is linked back into the
    tree in place of the NodeToDelete.

Arguments:

    NodeToDelete - Pointer to the node which the caller wishes to delete.

    Table - The generic table in which the delete is to occur.

Return Value:

    None.

Environment:

    Kernel mode. The PFN lock is held for some of the tables.

--*/

{
    PMMADDRESS_NODE Node = NodeToDelete;
    PMMADDRESS_NODE Child = NodeToDelete->RightChild;
  PMMADDRESS_NODE Temp = NodeToDelete->LeftChild;
  PMMADDRESS_NODE Parent, Rebalance;
  ULONG PC;

    DbgPrint("Remove node begin, node %u, tree %u.\n", (ULONG)NodeToDelete, (ULONG)(&Table->BalancedRoot));
    MiPrintTreePreOrder(&Table->BalancedRoot);

  if (!Temp) {
    /*
     * Case 1: node to erase has no more than 1 child (easy!)
     *
     * Note that if there is one child it must be red due to 5)
     * and node must be black due to 4). We adjust colors locally
     * so as to bypass __rb_erase_color() later on.
```

```
     */
       DbgPrint("Remove case 1 right.\n");
    PC = (ULONG)Node->u1.Parent;
    Parent = MiRBParent(Node);
    MiChangeChild(Node, Child, Parent);
    if (Child) {
      Child->u1.Parent = (PMMADDRESS_NODE)PC;
      Rebalance = NULL;
    } else
      Rebalance = MiIsBlack(Node) ? Parent : NULL;
    Temp = Parent;
  } else if (!Child) {
    /* Still case 1, but this time the child is Node->LeftChild */
       DbgPrint("Remove case 1 left.\n");
    PC = (ULONG)Node->u1.Parent;
       Temp->u1.Parent = (PMMADDRESS_NODE)PC;
    Parent = MiRBParent(Node);
    MiChangeChild(Node, Temp, Parent);
    Rebalance = NULL;
    Temp = Parent;
  } else {
    PMMADDRESS_NODE Successor = Child, Child2;

    Temp = Child->LeftChild;
    if (!Temp) {
      /*
       * Case 2: node's successor is its right child
       *
       *    (n)        (s)
       *    / \        / \
       *  (x) (s) -> (x) (c)
       *        \
       *        (c)
       */
          DbgPrint("Remove case 2.\n");
      Parent = Successor;
      Child2 = Successor->RightChild;
    } else {
      /*
       * Case 3: node's successor is leftmost under
       * node's right child subtree
       *
       *    (n)        (s)
       *    / \        / \
       *  (x) (y) -> (x) (y)
       *      /          /
       *    (p)        (p)
```

```
     *   /          /
     *  (s)        (c)
     *   \
     *   (c)
     */
        DbgPrint("Remove case 3.\n");
    do {
      Parent = Successor;
      Successor = Temp;
      Temp = Temp->LeftChild;
    } while (Temp);
    Child2 = Successor->RightChild;
    Parent->LeftChild = Child2;
        Successor->RightChild = Child;
    MiSetParent(Child, Successor);
  }

  Temp = Node->LeftChild;
  Successor->LeftChild = Temp;
  MiSetParent(Temp, Successor);

  PC = (ULONG)Node->u1.Parent;
  Temp = MiRBParent(Node);
  MiChangeChild(Node, Successor, Temp);

  if (Child2) {
    MiSetParentColor(Child2, Parent, RB_BLACK);
    Rebalance = NULL;
  } else {
    Rebalance = MiIsBlack(Successor) ? Parent : NULL;
  }
  Successor->u1.Parent = (PMMADDRESS_NODE)PC;
  Temp = Successor;
}

  DbgPrint("Remove rebalance %u.\n", (ULONG)Rebalance);
  if (Rebalance && Rebalance != &Table->BalancedRoot) {
    PMMADDRESS_NODE Node = NULL, Sibling, Temp1, Temp2;

    while (TRUE) {
        /*
        * Loop invariants:
        * - node is black (or NULL on first iteration)
        * - node is not the root (parent is not NULL)
        * - All leaf paths going through parent and node have a
        *   black node count that is 1 lower than other leaf paths.
        */
```

```c
Sibling = Parent->RightChild;
if (Node != Sibling) { /* Node == Parent->LeftChild */
    if (MiIsRed(Sibling)) {
        /*
        * Case 1 - left rotate at parent
        *
        *    P             S
        *   / \           / \
        *  N   s    -->  p   Sr
        *     / \       / \
        *    Sl  Sr    N   Sl
        */
        DbgPrint("Remove rebalance case 1.\n");
        Temp1 = Sibling->LeftChild;
        Parent->RightChild = Temp1;
        Sibling->LeftChild = Parent;
        MiSetParentColor(Temp1, Parent, RB_BLACK);
        MiRotateSetParents(Parent, Sibling, RB_RED);
        Sibling = Temp1;
    }
    Temp1 = Sibling->RightChild;
    if (!Temp1 || MiIsBlack(Temp1)) {
        Temp2 = Sibling->LeftChild;
        if (!Temp2 || MiIsBlack(Temp2)) {
            /*
            * Case 2 - sibling color flip
            * (p could be either color here)
            *
            *    (p)           (p)
            *    / \           / \
            *   N   S    -->  N   s
            *      / \           / \
            *     Sl  Sr        Sl  Sr
            *
            * This leaves us violating 5) which
            * can be fixed by flipping p to black
            * if it was red, or by recursing at p.
            * p is red when coming from Case 1.
            */
            DbgPrint("Remove rebalance case 2.\n");
            MiSetParentColor(Sibling, Parent, RB_RED);
            if (MiIsRed(Parent))
                MiSetBlack(Parent);
            else {
                Node = Parent;
                Parent = MiRBParent(Node);
                if (Parent && Parent != &Table->BalancedRoot)
```

```
                continue;
            }
            break;
        }
        /*
        * Case 3 - right rotate at sibling
        * (p could be either color here)
        *
        *   (p)           (p)
        *   / \           / \
        * N   S    --> N   sl
        *    / \             \
        *   sl  Sr            S
        *                      \
        *                       Sr
        *
        * Note: p might be red, and then both
        * p and sl are red after rotation(which
        * breaks property 4). This is fixed in
        * Case 4 (in __rb_rotate_set_parents()
        *        which set sl the color of p
        *        and set p RB_BLACK)
        *
        *   (p)            (sl)
        *   / \            /  \
        * N   sl   -->   P    S
        *      \        /      \
        *       S      N       Sr
        *        \
        *         Sr
        */
        DbgPrint("Remove rebalance case 3.\n");
        Temp1 = Temp2->RightChild;
        Sibling->LeftChild = Temp1;
        Temp2->RightChild = Sibling;
        Parent->RightChild = Temp2;
        if (Temp1)
            MiSetParentColor(Temp1, Sibling, RB_BLACK);
        Temp1 = Sibling;
        Sibling = Temp2;
    }
    /*
    * Case 4 - left rotate at parent + color flips
    * (p and sl could be either color here.
    *  After rotation, p becomes black, s acquires
    *  p's color, and sl keeps its color)
    *
```

```
            *     (p)            (s)
            *      / \           / \
            *     N   S    -->  P   Sr
            *        / \       / \
            *     (sl) sr     N  (sl)
            */
            DbgPrint("Remove rebalance case 4.\n");
            Temp2 = Sibling->LeftChild;
            Parent->RightChild = Temp2;
            Sibling->LeftChild = Parent;
            MiSetParentColor(Temp1, Sibling, RB_BLACK);
            if (Temp2)
                MiSetParent(Temp2, Parent);
            MiRotateSetParents(Parent, Sibling, RB_BLACK);
            break;
        } else {
            Sibling = Parent->LeftChild;
            if (MiIsRed(Sibling)) {
                /* Case 1 - right rotate at parent */
                DbgPrint("Remove rebalance case 1\'.\n");
                Temp1 = Sibling->RightChild;
                Parent->LeftChild = Temp1;
                Sibling->RightChild = Parent;
                MiSetParentColor(Temp1, Parent, RB_BLACK);
                MiRotateSetParents(Parent, Sibling, RB_RED);
                Sibling = Temp1;
            }
            Temp1 = Sibling->LeftChild;
            if (!Temp1 || MiIsBlack(Temp1)) {
                Temp2 = Sibling->RightChild;
                if (!Temp2 || MiIsBlack(Temp2)) {
                    /* Case 2 - sibling color flip */
                    DbgPrint("Remove rebalance case 2\'.\n");
                    MiSetParentColor(Sibling, Parent, RB_RED);
                    if (MiIsRed(Parent))
                        MiSetBlack(Parent);
                    else {
                        Node = Parent;
                        Parent = MiRBParent(Node);
                        if (Parent && Parent != &Table->BalancedRoot)
                            continue;
                    }
                    break;
                }
                /* Case 3 - left rotate at sibling */
                DbgPrint("Remove rebalance case 3\'.\n");
                Temp1 = Temp2->LeftChild;
```

```
                    Sibling->RightChild = Temp1;
                    Temp2->LeftChild = Sibling;
                    Parent->LeftChild = Temp2;
                    if (Temp1)
                        MiSetParentColor(Temp1, Sibling, RB_BLACK);
                    Temp1 = Sibling;
                    Sibling = Temp2;
                }
                /* Case 4 - right rotate at parent + color flips */
                DbgPrint("Remove rebalance case 4\'.\n");
                Temp2 = Sibling->RightChild;
                Parent->LeftChild = Temp2;
                Sibling->RightChild = Parent;
                MiSetParentColor(Temp1, Sibling, RB_BLACK);
                if (Temp2)
                    MiSetParent(Temp2, Parent);
                MiRotateSetParents(Parent, Sibling, RB_BLACK);
                break;
            }
        }
    }

    Table->NumberGenericTableElements -= 1;
    DbgPrint("Remove node finish.\n");
    MiPrintTreePreOrder(&Table->BalancedRoot);

    return;
}
```

MiInsertNode() 修改如下。

```
VOID
FASTCALL
MiInsertNode (
    IN PMMADDRESS_NODE NodeToInsert,
    IN PMM_AVL_TABLE Table
    )

/*++

Routine Description:

    This function inserts a new element in a table.

Arguments:

    NodeToInsert - The initialized address node to insert.

```

```
    Table - Pointer to the table in which to insert the new node.

Return Value:

    None.

Environment:

    Kernel mode. The PFN lock is held for some of the tables.

--*/

{
    //
    // Holds a pointer to the node in the table or what would be the
    // parent of the node.
    //
    PMMADDRESS_NODE NodeOrParent;
    TABLE_SEARCH_RESULT SearchResult;

    DbgPrint("Insert node begin, node %u, tree %u.\n", (ULONG)NodeToInsert, (ULONG)(&Table->BalancedRoot));
    MiPrintTreePreOrder(&Table->BalancedRoot);

    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;

    //
    // Insert the new node in the tree.
    //

```

```
    if (SearchResult == TableEmptyTree) {
        DbgPrint("Insert empty tree.\n");
        Table->BalancedRoot.RightChild = NodeToInsert;
        NodeToInsert->u1.Parent = MI_MAKE_PARENT(&Table->BalancedRoot, RB_BLACK);
        ASSERT (NodeToInsert->u1.Balance == RB_BLACK);
        ASSERT(Table->DepthOfTree == 0);
        Table->DepthOfTree = 1;

    }
    else {
        PMMADDRESS_NODE Node = NodeToInsert;
        PMMADDRESS_NODE Parent = NodeOrParent;
        PMMADDRESS_NODE GParent, Temp;

        DbgPrint("Insert non-empty tree.\n");

        if (SearchResult == TableInsertAsLeft) {
            NodeOrParent->LeftChild = NodeToInsert;
        }
        else {
            NodeOrParent->RightChild = NodeToInsert;
        }

        Node->u1.Parent = MI_MAKE_PARENT(Parent, RB_RED);;
        ASSERT (NodeToInsert->u1.Balance == RB_RED);

        while (TRUE) {
            /*
            * Loop invariant: node is red.
            */
            if (MiRBParent(Node) == &Table->BalancedRoot) {
                /*
                * The inserted node is root. Either this is the
                * first node, or we recursed at Case 1 below and
                * are no longer violating 4).
                */
                DbgPrint("Insert root.\n");
                MiSetParentColor(Node, &Table->BalancedRoot, RB_BLACK);
                break;
            }

            /*
            * If there is a black parent, we are done.
            * Otherwise, take some corrective action as,
            * per 4), we don't want a red root or two
            * consecutive red nodes.
            */
```

```
        if (MiIsBlack(Parent))
            break;

        GParent = MiRBParent(Parent);

        Temp = GParent->RightChild;
        if (Parent != Temp) { /* Parent == GParent->LeftChild */
            if (Temp && MiIsRed(Temp)) {
                /*
                 * Case 1 - node's uncle is red (color flips).
                 *
                 *       G            g
                 *      / \          / \
                 *     p   u  -->   P   U
                 *    /            /
                 *   n            n
                 *
                 * However, since g's parent might be red, and
                 * 4) does not allow this, we need to recurse
                 * at g.
                 */
                DbgPrint("Insert case 1.\n");
                MiSetParentColor(Temp, GParent, RB_BLACK);
                MiSetParentColor(Parent, GParent, RB_BLACK);
                Node = GParent;
                Parent = MiRBParent(Node);
                MiSetParentColor(Node, Parent, RB_RED);
                continue;
            }

            Temp = Parent->RightChild;
            if (Node == Temp) {
                /*
                 * Case 2 - node's uncle is black and node is
                 * the parent's right child (left rotate at parent).
                 *
                 *      G             G
                 *     / \           / \
                 *    p   U  -->    n   U
                 *     \           /
                 *      n         p
                 *
                 * This still leaves us in violation of 4), the
                 * continuation into Case 3 will fix that.
                 */
                DbgPrint("Insert case 2.\n");
                Temp = Node->LeftChild;
```

```
                Parent->RightChild = Temp;
                Node->LeftChild = Parent;
                if (Temp)
                    MiSetParentColor(Temp, Parent, RB_BLACK);
                MiSetParentColor(Parent, Node, RB_RED);
                Parent = Node;
                Temp = Node->RightChild;
            }

            /*
             * Case 3 - node's uncle is black and node is
             * the parent's left child (right rotate at gparent).
             *
             *         G           P
             *        / \         / \
             *       p   U  -->  n   g
             *      /                 \
             *     n                   U
             */
            DbgPrint("Insert case 3.\n");
            GParent->LeftChild = Temp; /* == Parent->RightChild */
            Parent->RightChild = GParent;
            if (Temp)
                MiSetParentColor(Temp, GParent, RB_BLACK);
            MiRotateSetParents(GParent, Parent, RB_RED);
            break;
        } else {
            Temp = GParent->LeftChild;
            if (Temp && MiIsRed(Temp)) {
                /* Case 1 - color flips */
                DbgPrint("Insert case 1\'.\n");
                MiSetParentColor(Temp, GParent, RB_BLACK);
                MiSetParentColor(Parent, GParent, RB_BLACK);
                Node = GParent;
                Parent = MiRBParent(Node);
                MiSetParentColor(Node, Parent, RB_RED);
                continue;
            }

            Temp = Parent->LeftChild;
            if (Node == Temp) {
                /* Case 2 - right rotate at parent */
                DbgPrint("Insert case 2\'.\n");
                Temp = Node->RightChild;
                Parent->LeftChild = Temp;
                Node->RightChild = Parent;
                if (Temp)
```

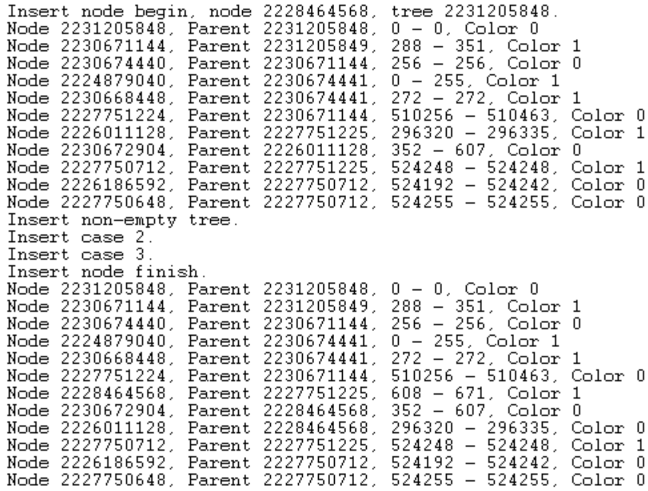
```
                        MiSetParentColor(Temp, Parent, RB_BLACK);
                    MiSetParentColor(Parent, Node, RB_RED);
                    Parent = Node;
                    Temp = Node->LeftChild;
                }

                /* Case 3 - left rotate at gparent */
                DbgPrint("Insert case 3\'.\n");
                GParent->RightChild = Temp; /* == Parent->LeftChild */
                Parent->LeftChild = GParent;
                if (Temp)
                    MiSetParentColor(Temp, GParent, RB_BLACK);
                MiRotateSetParents(GParent, Parent, RB_RED);
                break;
            }
        }
  }

    DbgPrint("Insert node finish.\n");
    MiPrintTreePreOrder(&Table->BalancedRoot);

    return;
}
```

4 实验结果

首先，编译后能正常开机运行诸如记事本、IE 浏览器等应用程序而不崩溃。

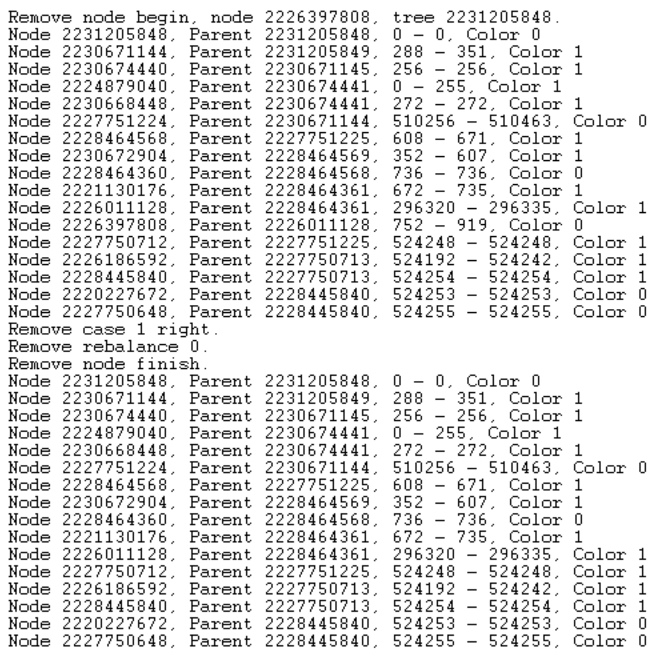
连接上调试端口后，会打印各插入/删除操作开始前和结束后的红黑树先序遍历。先序遍历可确定整个红黑树，图 1、图 2给出某两个时刻的插入与删除操作前后红黑树先序遍历结果（实际上是两个较简单情况，但由于操作过多，画红黑树亦不易，故难以找到较为复杂的红黑树操作进行验证）。需要注意的是，每次打印的第一个节点（0-0）是辅助根节点而不是实际的节点。

将图 1所示的插入操作前后红黑树作出如图 3、图 4所示，可以验证确实正确。图 2所示的删除操作删除了一个最底层的红节点，只是单纯的删除了该节点而没有进行平衡操作。也可以验证其所示红黑树确实是红黑树，此处略去不作图。这两次操作事实上已经有充分多的节点验证了移植的正确性。



```
Insert node begin, node 2228464568, tree 2231205848.
Node 2231205848, Parent 2231205848, 0 - 0, Color 0
Node 2230671144, Parent 2231205849, 288 - 351, Color 1
Node 2230674440, Parent 2230671144, 256 - 256, Color 0
Node 2224879040, Parent 2230674441, 0 - 255, Color 1
Node 2230668448, Parent 2230674441, 272 - 272, Color 1
Node 2227751224, Parent 2230671144, 510256 - 510463, Color 0
Node 2226011128, Parent 2227751225, 296320 - 296335, Color 1
Node 2230672904, Parent 2226011128, 352 - 607, Color 0
Node 2227750712, Parent 2227751225, 524248 - 524248, Color 1
Node 2226186592, Parent 2227750712, 524192 - 524242, Color 0
Node 2227750648, Parent 2227750712, 524255 - 524255, Color 0
Insert non-empty tree.
Insert case 2.
Insert case 3.
Insert node finish.
Node 2231205848, Parent 2231205848, 0 - 0, Color 0
Node 2230671144, Parent 2231205849, 288 - 351, Color 1
Node 2230674440, Parent 2230671144, 256 - 256, Color 0
Node 2224879040, Parent 2230674441, 0 - 255, Color 1
Node 2230668448, Parent 2230674441, 272 - 272, Color 1
Node 2227751224, Parent 2230671144, 510256 - 510463, Color 0
Node 2228464568, Parent 2227751225, 608 - 671, Color 1
Node 2230672904, Parent 2228464568, 352 - 607, Color 0
Node 2226011128, Parent 2228464568, 296320 - 296335, Color 0
Node 2227750712, Parent 2227751225, 524248 - 524248, Color 1
Node 2226186592, Parent 2227750712, 524192 - 524242, Color 0
Node 2227750648, Parent 2227750712, 524255 - 524255, Color 0
```

图 1: 红黑树插入操作



```
Remove node begin, node 2226397808, tree 2231205848.
Node 2231205848, Parent 2231205848, 0 - 0, Color 0
Node 2230671144, Parent 2231205849, 288 - 351, Color 1
Node 2230674440, Parent 2230671145, 256 - 256, Color 1
Node 2224879040, Parent 2230674441, 0 - 255, Color 1
Node 2230668448, Parent 2230674441, 272 - 272, Color 1
Node 2227751224, Parent 2230671144, 510256 - 510463, Color 0
Node 2228464568, Parent 2227751225, 608 - 671, Color 1
Node 2230672904, Parent 2228464569, 352 - 607, Color 1
Node 2228464360, Parent 2228464568, 736 - 736, Color 0
Node 2221130176, Parent 2228464361, 672 - 735, Color 1
Node 2226011128, Parent 2228464361, 296320 - 296335, Color 1
Node 2226397808, Parent 2226011128, 752 - 919, Color 0
Node 2227750712, Parent 2227751225, 524248 - 524248, Color 1
Node 2226186592, Parent 2227750713, 524192 - 524242, Color 1
Node 2228445840, Parent 2227750713, 524254 - 524254, Color 1
Node 2220227672, Parent 2228445840, 524253 - 524253, Color 0
Node 2227750648, Parent 2228445840, 524255 - 524255, Color 0
Remove case 1 right.
Remove rebalance 0.
Remove node finish.
Node 2231205848, Parent 2231205848, 0 - 0, Color 0
Node 2230671144, Parent 2231205849, 288 - 351, Color 1
Node 2230674440, Parent 2230671145, 256 - 256, Color 1
Node 2224879040, Parent 2230674441, 0 - 255, Color 1
Node 2230668448, Parent 2230674441, 272 - 272, Color 1
Node 2227751224, Parent 2230671144, 510256 - 510463, Color 0
Node 2228464568, Parent 2227751225, 608 - 671, Color 1
Node 2230672904, Parent 2228464569, 352 - 607, Color 1
Node 2228464360, Parent 2228464568, 736 - 736, Color 0
Node 2221130176, Parent 2228464361, 672 - 735, Color 1
Node 2226011128, Parent 2228464361, 296320 - 296335, Color 1
Node 2227750712, Parent 2227751225, 524248 - 524248, Color 1
Node 2226186592, Parent 2227750713, 524192 - 524242, Color 1
Node 2228445840, Parent 2227750713, 524254 - 524254, Color 1
Node 2220227672, Parent 2228445840, 524253 - 524253, Color 0
Node 2227750648, Parent 2228445840, 524255 - 524255, Color 0
```

图 2: 红黑树删除操作

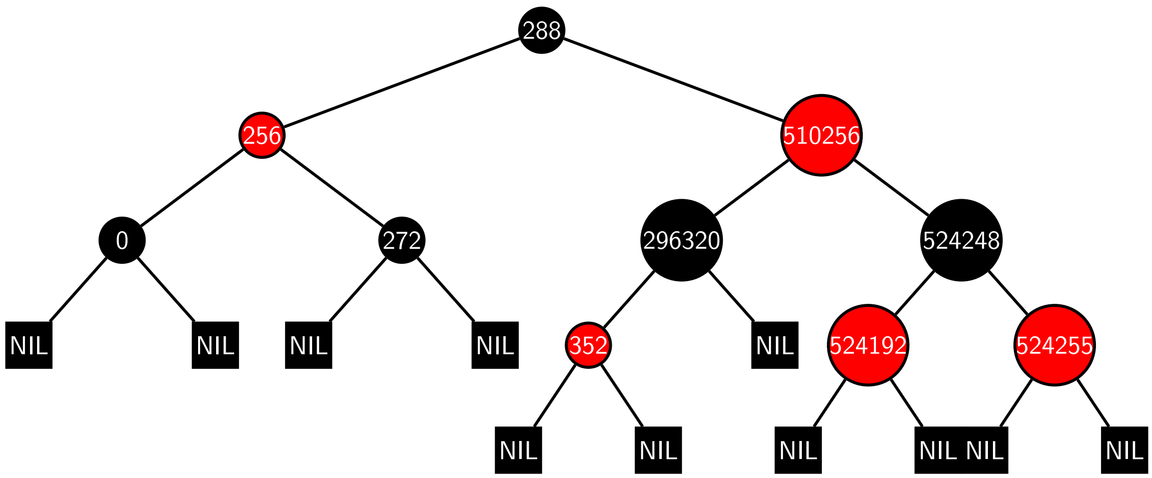


图 3: 插入前红黑树

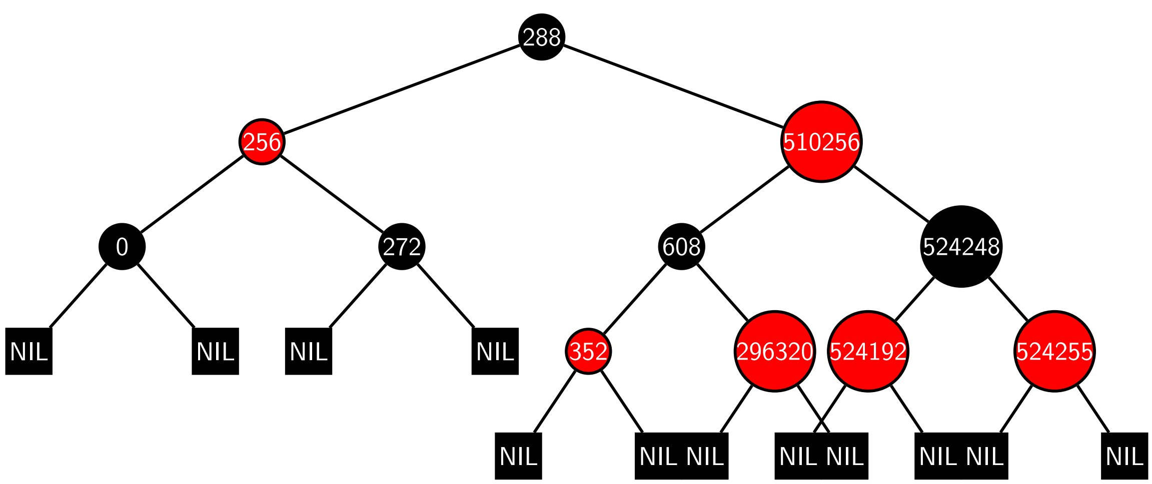


图 4: 插入后红黑树

5 实验感想

本次实验涉及一个实际操作系统内核的修改，锻炼的能力方面非常之多。首先读懂 WRK 和 Linux 相应数据结构的代码，找到需要移植的部分就比较困难。其次操作系统内核中对 C 语言的运用也使我加深了很多语法结构的认识，例如宏与联合体的使用。最后，操作系统内核的调试方法也是此次实验的一大收获。通过串口连接主机，在内核中进行调试的方法也是第一次学会。调试时开始遇到了非常多的困难，但最终也一一解决，使我收获颇丰。