AVL 树/红黑树问题

无 36　李思涵　2013011187　lisihan969@gmail.com

2016 年 1 月 20 日

目录

1 问题描述 1

2 现有结构 1

　2.1 Linux 1

　2.2 WRK 3

3 模块设计 4

4 代码实现 4

5 实验结果 24

6 实验感想 25

1 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 现有结构

2.1 Linux

我们选取的 Linux Kernel 版本为 4.4-rc8。其中和红黑树有关的代码共有以下三个文件：

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

有关其使用方法的介绍可以在 `Documentation/rbtree.txt` 中找到。

`rbtree.h` 包含了红黑树的结构体的定义，以及对其数据的基本访问和操作。其中结点的定义如下：

```
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 */
```

红黑树根的定义如下:

```
struct rb_root {
    struct rb_node *rb_node;
};
```

一些基本的访问和操作如下:

```
#define rb_parent(r)   ((struct rb_node *)((r)->__rb_parent_color & ~3))
#define rb_entry(ptr, type, member) container_of(ptr, type, member)

extern void rb_insert_color(struct rb_node *, struct rb_root *);
extern void rb_erase(struct rb_node *, struct rb_root *);

/* Find logical next and previous nodes in a tree */
extern struct rb_node *rb_next(const struct rb_node *);
extern struct rb_node *rb_prev(const struct rb_node *);
extern struct rb_node *rb_first(const struct rb_root *);
extern struct rb_node *rb_last(const struct rb_root *);
```

在 rbtree_augmented.h 中还定义了更多的内部访问和操作，如下所示:

```
#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;
}
```

```c
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;
}
```

我们总结如下：

- 红黑树结点结构体是以 long 的长度对齐的，在现在的计算机上这个值一般是 4 字节或是 8 字节。
- 红黑树结点中，父节点地址和颜色被储存在了同一个 long 变量中。其中低两位保存的是结点的颜色，高位保存的是结点的地址（以 4 字节为单位）。这么做可行是建立在 C 语言标准中，long 的长度至少为 4 字节的基础上的，故结构体在以 long 对其后最低 2 位一定为 00。
- 该红黑树用 0 代表红结点，用 1 代表黑结点。
- 外界通过在合适的结点调用 `rb_link_node` 插入新结点，然后调用 `rb_insert_color` 使树平衡。
- 外界使用 `rb_erase` 删除结点。

同时，需要注意到的是，`rbtree_augmented.h` 中还提供了一些操作的增强版本，如 `rb_insert_augmented` 和 `rb_erase_augmented`。这些函数主要是为了实现 “增强红黑树”，也就是每个结点保存了一些额外信息的红黑树。由于我们的实现中不需要增强红黑树，我们将忽略那些带有 `augmented` 后缀的函数版本。同时，我们需要实现的对外接口也应只限于 `rbtree.h` 中的对外接口。`rbtree_augmented.h` 中的对外接口不用移植，但其中的一些内部定义和访问需要移植。

2.2 WRK

为了完成移植工作，我们还需要了解 WRK 中 VAD 树。我们的目标是找到 VAD 树所使用的 AVL 树，并在不修改接口的情况下对 AVL 树的实现进行更改，将其内部更改成红黑树。

首先，VAD 所使用的 AVL 树定义在 `base/ntos/mm/addrsup.c` 中。根据该文件顶部的描述，该模块是基于 Knuth 的 “The Art of Computer Programming, Volume 3, Sorting and Searching” 第二版中的 AVL 树实现的。

从 /WRK-v1.2/base/ntos/inc/ps.h 我们可以找到 AVL 树结构的定义。

```c
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;

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;
```

3 模块设计

需要注意到的是，和我们一般编程时实现的不同，Linux 和 WRK 中的结点都是内嵌在真正的数据结构当中的。例如，若想使用 Linux 中的红黑树，则应先定义类似下面数据结构：

```
struct mytype {
  struct rb_node node;
  char *keystring;
};
```

同样的，从 WRK 的 AVL 树中的 `MMADDRESS_NODE` 也是与 `rb_node` 类似的内嵌结构。而 `MMVAD` 则是包含内嵌结点的结构，即 AVL 树的真正结点。需要注意的是，在 Linux 中用户需要针对真正结点定义自己的插入和删除函数；而 WRK 中则更进一步，用 `MM_AVL_TABLE` 对 AVL 树进行了封装，集成了插入和删除函数。

所以，我们只需要修改 WRK 中与 `MM_AVL_TABLE` 有关的操作，即 `/WRK-v1.2/base/ntos/mm/addrsup.c` 中的 `MiInsertNode` 和 `MiRemoveNode`，便可以达到修改 AVL 树的效果。为了尽量减小修改，我们沿用原内嵌结点定义，并将其中的 `Balance` 域作为红黑树结点颜色。

这里需要注意的是，由于 AVL 树初始化时 Balance 为 0，而红黑树初始化时根节点应为黑色，我们将红色的定义改为 1，黑色的定义改为 0。这样做是为了尽可能少地更改已有的 WRK 代码，从而减小出错的可能性。

4 代码实现

```
diff --git a/3-file-system/WRK-v1.2/base/ntos/mm/addrsup.c b/3-file-system/WRK-v1.2/base/ntc
index 5842f18..f18ac49 100644
--- a/3-file-system/WRK-v1.2/base/ntos/mm/addrsup.c
+++ b/3-file-system/WRK-v1.2/base/ntos/mm/addrsup.c
```

```
@@ -33,8 +33,331 @@ Environment:

 #include "mi.h"

+
+// Ported definations.
+#define RB_RED      1
+#define RB_BLACK    0
+
+#define __rb_parent(pc)    ((PMMADDRESS_NODE)((long)(pc) & ~3))
+#define rb_parent(rb)      (SANITIZE_PARENT_NODE((rb)->u1.Parent))
+#define rb_red_parent(rb)  (rb_parent(rb))
+
+#define __rb_color(pc)     ((long)(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)->u1.Parent)
+#define rb_is_red(rb)      __rb_is_red((rb)->u1.Parent)
+#define rb_is_black(rb)    __rb_is_black((rb)->u1.Parent)
+#define rb_set_red(rb)     ((rb)->u1.Balance = RB_RED)
+#define rb_set_black(rb)   ((rb)->u1.Balance = RB_BLACK)
+
+
+
+static void rb_set_parent(PMMADDRESS_NODE rb, PMMADDRESS_NODE p)
+{
+    rb->u1.Parent = (PMMADDRESS_NODE)(rb_color(rb) | (long)p);
+}
+
+static void rb_set_parent_color(PMMADDRESS_NODE rb,
+                                        PMMADDRESS_NODE p, int color)
+{
+    rb->u1.Parent = p;
+    rb->u1.Balance = color;
+}
+
+static void
+__rb_change_child(PMMADDRESS_NODE old, PMMADDRESS_NODE new_,
+                   PMMADDRESS_NODE parent, PMMADDRESS_NODE root)
+{
+    if (parent) {
+        if (parent->LeftChild == old)
```

```
+            parent->LeftChild = new_;
+        else
+            parent->RightChild = new_;
+    } else
+        root->RightChild = new_;
+}
+
+/*
+ * Helper function for rotations:
+ * - old's parent and color get assigned to new
+ * - old gets assigned new as a parent and 'color' as a color.
+ */
+static void
+__rb_rotate_set_parents(PMMADDRESS_NODE old, PMMADDRESS_NODE new_,
+                        PMMADDRESS_NODE root, int color)
+{
+    PMMADDRESS_NODE parent = rb_parent(old);
+    new_->u1.Parent = old->u1.Parent;
+    rb_set_parent_color(old, new_, color);
+    __rb_change_child(old, new_, parent, root);
+}
+
+static PMMADDRESS_NODE
+__rb_erase_augmented(PMMADDRESS_NODE node, PMMADDRESS_NODE root)
+{
+    PMMADDRESS_NODE child = node->RightChild;
+    PMMADDRESS_NODE tmp = node->LeftChild;
+    PMMADDRESS_NODE parent, rebalance;
+    PMMADDRESS_NODE pc;
+
+    if (!tmp) {
+        /*
+         * 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.
+         */
+        pc = node->u1.Parent;
+        parent = __rb_parent(pc);
+        __rb_change_child(node, child, parent, root);
+        if (child) {
```

```
+           child->u1.Parent = pc;
+           rebalance = NULL;
+       } else
+           rebalance = __rb_is_black(pc) ? parent : NULL;
+       tmp = parent;
+   } else if (!child) {
+       /* Still case 1, but this time the child is node->LeftChild */
+       tmp->u1.Parent = pc = node->u1.Parent;
+       parent = __rb_parent(pc);
+       __rb_change_child(node, tmp, parent, root);
+       rebalance = NULL;
+       tmp = parent;
+   } else {
+       PMMADDRESS_NODE successor = child, child2;
+
+       tmp = child->LeftChild;
+       if (!tmp) {
+           /*
+            * Case 2: node's successor is its right child
+            *
+            *    (n)          (s)
+            *    / \          / \
+            *  (x) (s)  ->  (x) (c)
+            *        \
+            *        (c)
+            */
+           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)
```

```
+             */
+            do {
+                parent = successor;
+                successor = tmp;
+                tmp = tmp->LeftChild;
+            } while (tmp);
+            child2 = successor->RightChild;
+            parent->LeftChild = child2;
+            successor->RightChild = child;
+            rb_set_parent(child, successor);
+        }
+
+        tmp = node->LeftChild;
+        successor->LeftChild = tmp;
+        rb_set_parent(tmp, successor);
+
+        pc = node->u1.Parent;
+        tmp = __rb_parent(pc);
+        __rb_change_child(node, successor, tmp, root);
+
+        if (child2) {
+            successor->u1.Parent = pc;
+            rb_set_parent_color(child2, parent, RB_BLACK);
+            rebalance = NULL;
+        } else {
+            PMMADDRESS_NODE pc2 = successor->u1.Parent;
+            successor->u1.Parent = pc;
+            rebalance = __rb_is_black(pc2) ? parent : NULL;
+        }
+        tmp = successor;
+    }
+
+    return rebalance;
+}
+
+/*
+ * Inline version for rb_erase() use - we want to be able to inline
+ * and eliminate the dummy_rotate callback there
+ */
+static void
+____rb_erase_color(PMMADDRESS_NODE parent, PMMADDRESS_NODE root)
+{
```

```
+    PMMADDRESS_NODE node = NULL, sibling, tmp1, tmp2;
+
+    while (1) {
+        /*
+         * 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.
+         */
+        sibling = parent->RightChild;
+        if (node != sibling) {  /* node == parent->LeftChild */
+            if (rb_is_red(sibling)) {
+                /*
+                 * Case 1 - left rotate at parent
+                 *
+                 *     P               S
+                 *    / \             / \
+                 *   N   s    -->    p   Sr
+                 *      / \         / \
+                 *     Sl  Sr      N   Sl
+                 */
+                tmp1 = sibling->LeftChild;
+                parent->RightChild = tmp1;
+                sibling->LeftChild = parent;
+                rb_set_parent_color(tmp1, parent, RB_BLACK);
+                __rb_rotate_set_parents(parent, sibling, root,
+                            RB_RED);
+                sibling = tmp1;
+            }
+            tmp1 = sibling->RightChild;
+            if (!tmp1 || rb_is_black(tmp1)) {
+                tmp2 = sibling->LeftChild;
+                if (!tmp2 || rb_is_black(tmp2)) {
+                    /*
+                     * 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.
+                     */
+                    rb_set_parent_color(sibling, parent,
+                                RB_RED);
+                    if (rb_is_red(parent))
+                        rb_set_black(parent);
+                    else {
+                        node = parent;
+                        parent = rb_parent(node);
+                        if (parent)
+                            continue;
+                    }
+                    break;
+                }
+                /*
+                 * Case 3 - right rotate at sibling
+                 * (p could be either color here)
+                 *
+                 *   (p)           (p)
+                 *   / \           / \
+                 *  N   S    -->  N   Sl
+                 *     / \             \
+                 *    sl  Sr            s
+                 *                       \
+                 *                        Sr
+                 */
+                tmp1 = tmp2->RightChild;
+                sibling->LeftChild = tmp1;
+                tmp2->RightChild = sibling;
+                parent->RightChild = tmp2;
+                if (tmp1)
+                    rb_set_parent_color(tmp1, sibling,
+                                RB_BLACK);
+                tmp1 = sibling;
+                sibling = tmp2;
+            }
+            /*
```

```
+              * 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)
+              */
+             tmp2 = sibling->LeftChild;
+             parent->RightChild = tmp2;
+             sibling->LeftChild = parent;
+             rb_set_parent_color(tmp1, sibling, RB_BLACK);
+             if (tmp2)
+                 rb_set_parent(tmp2, parent);
+             __rb_rotate_set_parents(parent, sibling, root,
+                         RB_BLACK);
+             break;
+         } else {
+             sibling = parent->LeftChild;
+             if (rb_is_red(sibling)) {
+                 /* Case 1 - right rotate at parent */
+                 tmp1 = sibling->RightChild;
+                 parent->LeftChild = tmp1;
+                 sibling->RightChild = parent;
+                 rb_set_parent_color(tmp1, parent, RB_BLACK);
+                 __rb_rotate_set_parents(parent, sibling, root,
+                             RB_RED);
+                 sibling = tmp1;
+             }
+             tmp1 = sibling->LeftChild;
+             if (!tmp1 || rb_is_black(tmp1)) {
+                 tmp2 = sibling->RightChild;
+                 if (!tmp2 || rb_is_black(tmp2)) {
+                     /* Case 2 - sibling color flip */
+                     rb_set_parent_color(sibling, parent,
+                                 RB_RED);
+                     if (rb_is_red(parent))
+                         rb_set_black(parent);
+                     else {
```

```
+                       node = parent;
+                       parent = rb_parent(node);
+                       if (parent)
+                           continue;
+                   }
+                   break;
+               }
+               /* Case 3 - right rotate at sibling */
+               tmp1 = tmp2->LeftChild;
+               sibling->RightChild = tmp1;
+               tmp2->LeftChild = sibling;
+               parent->LeftChild = tmp2;
+               if (tmp1)
+                   rb_set_parent_color(tmp1, sibling,
+                                 RB_BLACK);
+               tmp1 = sibling;
+               sibling = tmp2;
+           }
+           /* Case 4 - left rotate at parent + color flips */
+           tmp2 = sibling->RightChild;
+           parent->LeftChild = tmp2;
+           sibling->RightChild = parent;
+           rb_set_parent_color(tmp1, sibling, RB_BLACK);
+           if (tmp2)
+               rb_set_parent(tmp2, parent);
+           __rb_rotate_set_parents(parent, sibling, root,
+                       RB_BLACK);
+           break;
+       }
+   }
+}
+
+
 #if !defined (_USERMODE)
-#define PRINT
+#define PRINT
 #define COUNT_BALANCE_MAX(a)
 #else
 extern MM_AVL_TABLE MmSectionBasedRoot;
@@ -839,231 +1162,11 @@ Environment:
 --*/
```

```
{
-     PMMADDRESS_NODE Parent;
-     PMMADDRESS_NODE EasyDelete;
-     PMMADDRESS_NODE P;
-     SCHAR a;
-
-     //
-     // If the NodeToDelete has at least one NULL child pointer, then we can
-     // delete it directly.
-     //
-
-     if ((NodeToDelete->LeftChild == NULL) ||
-         (NodeToDelete->RightChild == NULL)) {
-
-         EasyDelete = NodeToDelete;
-     }
-
-     //
-     // Otherwise, we may as well pick the longest side to delete from (if one is
-     // is longer), as that reduces the probability that we will have to
-     // rebalance.
-     //
-
-     else if ((SCHAR) NodeToDelete->u1.Balance >= 0) {
-
-         //
-         // Pick up the subtree successor.
-         //
-
-         EasyDelete = NodeToDelete->RightChild;
-         while (EasyDelete->LeftChild != NULL) {
-             EasyDelete = EasyDelete->LeftChild;
-         }
-     }
-     else {
-
-         //
-         // Pick up the subtree predecessor.
-         //
-
-         EasyDelete = NodeToDelete->LeftChild;
-         while (EasyDelete->RightChild != NULL) {
```

```
-            EasyDelete = EasyDelete->RightChild;
-        }
-    }
-
-    //
-    // Rebalancing must know which side of the first parent the delete occurred
-    // on.  Assume it is the left side and otherwise correct below.
-    //
-
-    a = -1;
-
-    //
-    // Now we can do the simple deletion for the no left child case.
-    //
-
-    if (EasyDelete->LeftChild == NULL) {
-
-        Parent = SANITIZE_PARENT_NODE (EasyDelete->u1.Parent);
-
-        if (MiIsLeftChild(EasyDelete)) {
-            Parent->LeftChild = EasyDelete->RightChild;
-        }
-        else {
-            Parent->RightChild = EasyDelete->RightChild;
-            a = 1;
-        }
-
-        if (EasyDelete->RightChild != NULL) {
-            EasyDelete->RightChild->u1.Parent = MI_MAKE_PARENT (Parent, EasyDelete->RightCh
-        }
-
-    //
-    // Now we can do the simple deletion for the no right child case,
-    // plus we know there is a left child.
-    //
-
-    }
-    else {
-
-        Parent = SANITIZE_PARENT_NODE (EasyDelete->u1.Parent);
-
-        if (MiIsLeftChild(EasyDelete)) {
```

```
-           Parent->LeftChild = EasyDelete->LeftChild;
-       }
-       else {
-           Parent->RightChild = EasyDelete->LeftChild;
-           a = 1;
-       }
-
-       EasyDelete->LeftChild->u1.Parent = MI_MAKE_PARENT (Parent,
-                                           EasyDelete->LeftChild->u1.Balance);
-   }
-
-   //
-   // For delete rebalancing, set the balance at the root to 0 to properly
-   // terminate the rebalance without special tests, and to be able to detect
-   // if the depth of the tree actually decreased.
-   //
-
-   Table->BalancedRoot.u1.Balance = 0;
-   P = SANITIZE_PARENT_NODE (EasyDelete->u1.Parent);
-
-   //
-   // Loop until the tree is balanced.
-   //
+   PMMADDRESS_NODE rebalance, root = &Table->BalancedRoot;
+   rebalance = __rb_erase_augmented(NodeToDelete, root);
+   if (rebalance)
+       ____rb_erase_color(rebalance, root);

-   while (TRUE) {
-
-       //
-       // First handle the case where the tree became more balanced.  Zero
-       // the balance factor, calculate a for the next loop and move on to
-       // the parent.
-       //
-
-       if ((SCHAR) P->u1.Balance == a) {
-
-           P->u1.Balance = 0;
-
-       //
-       // If this node is curently balanced, we can show it is now unbalanced
```

```
-        // and terminate the scan since the subtree length has not changed.
-        // (This may be the root, since we set Balance to 0 above!)
-        //
-
-        }
-        else if (P->u1.Balance == 0) {
-
-            PRINT("REBADJ D: Node %p, Bal %x -> %x\n", P, P->u1.Balance, -a);
-            COUNT_BALANCE_MAX ((SCHAR)-a);
-            P->u1.Balance = -a;
-
-            //
-            // If we shortened the depth all the way back to the root, then
-            // the tree really has one less level.
-            //
-
-            if (Table->BalancedRoot.u1.Balance != 0) {
-                Table->DepthOfTree -= 1;
-            }
-
-            break;
-
-        //
-        // Otherwise we made the short side 2 levels less than the long side,
-        // and rebalancing is required.  On return, some node has been promoted
-        // to above node P.  If Case 3 from Knuth was not encountered, then we
-        // want to effectively resume rebalancing from P's original parent which
-        // is effectively its grandparent now.
-        //
-
-        }
-        else {
-
-            //
-            // We are done if Case 3 was hit, i.e., the depth of this subtree is
-            // now the same as before the delete.
-            //
-
-            if (MiRebalanceNode(P)) {
-                break;
-            }
-
```

```
-           P = SANITIZE_PARENT_NODE (P->u1.Parent);
-       }
-
-       a = -1;
-       if (MiIsRightChild(P)) {
-           a = 1;
-       }
-       P = SANITIZE_PARENT_NODE (P->u1.Parent);
-   }
-
-   //
-   // Finally, if we actually deleted a predecessor/successor of the
-   // NodeToDelete, we will link him back into the tree to replace
-   // NodeToDelete before returning.  Note that NodeToDelete did have
-   // both child links filled in, but that may no longer be the case
-   // at this point.
-   //
-
-   if (NodeToDelete != EasyDelete) {
-
-       //
-       // Note carefully - VADs are of differing sizes therefore it is not safe
-       // to just overlay the EasyDelete node with the NodeToDelete like the
-       // rtl avl code does.
-       //
-       // Copy just the links, preserving the rest of the original EasyDelete
-       // VAD.
-       //
-
-       EasyDelete->u1.Parent = NodeToDelete->u1.Parent;
-       EasyDelete->LeftChild = NodeToDelete->LeftChild;
-       EasyDelete->RightChild = NodeToDelete->RightChild;
-
-       if (MiIsLeftChild(NodeToDelete)) {
-           Parent = SANITIZE_PARENT_NODE (EasyDelete->u1.Parent);
-           Parent->LeftChild = EasyDelete;
-       }
-       else {
-           ASSERT(MiIsRightChild(NodeToDelete));
-           Parent = SANITIZE_PARENT_NODE (EasyDelete->u1.Parent);
-           Parent->RightChild = EasyDelete;
-       }
```

```
-         if (EasyDelete->LeftChild != NULL) {
-             EasyDelete->LeftChild->u1.Parent = MI_MAKE_PARENT (EasyDelete,
-                                                 EasyDelete->LeftChild->u1.Balance);
-         }
-         if (EasyDelete->RightChild != NULL) {
-             EasyDelete->RightChild->u1.Parent = MI_MAKE_PARENT (EasyDelete,
-                                                 EasyDelete->RightChild->u1.Balance);
-         }
-     }
-
-     Table->NumberGenericTableElements -= 1;
-
-     //
-     // Sanity check tree size and depth.
-     //
-
-     ASSERT((Table->NumberGenericTableElements >= MiWorstCaseFill[Table->DepthOfTree]) &&
-            (Table->NumberGenericTableElements <= MiBestCaseFill[Table->DepthOfTree]));
-
-     return;
 }

 ^L
@@ -1328,22 +1431,12 @@ Environment:
     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;
@@ -1357,19 +1450,15 @@ Environment:
     if (SearchResult == TableEmptyTree) {

         Table->BalancedRoot.RightChild = NodeToInsert;
-        NodeToInsert->u1.Parent = &Table->BalancedRoot;
-        ASSERT (NodeToInsert->u1.Balance == 0);
-        ASSERT(Table->DepthOfTree == 0);
+        rb_set_parent(NodeToInsert, &Table->BalancedRoot);
         Table->DepthOfTree = 1;

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

         PMMADDRESS_NODE R = NodeToInsert;
         PMMADDRESS_NODE S = NodeOrParent;
+        PMMADDRESS_NODE node, root, parent, gparent, tmp;

         if (SearchResult == TableInsertAsLeft) {
             NodeOrParent->LeftChild = NodeToInsert;
@@ -1378,8 +1467,7 @@ Environment:
             NodeOrParent->RightChild = NodeToInsert;
         }

-        NodeToInsert->u1.Parent = NodeOrParent;
-        ASSERT (NodeToInsert->u1.Balance == 0);
+        rb_set_parent(NodeToInsert, NodeOrParent);

         //
         // The above completes the standard binary tree insertion, which
@@ -1392,101 +1480,127 @@ Environment:
         // to simplify loop control.
         //

-        PRINT("REBADJ E: Table %p, Bal %x -> %x\n", Table, Table->BalancedRoot.u1.Balance,
-        COUNT_BALANCE_MAX ((SCHAR)-1);
```

```
-         Table->BalancedRoot.u1.Balance = (ULONG_PTR) -1;
-
-         //
-         // Now loop to adjust balance factors and see if any balance operations
-         // must be performed, using NodeOrParent to ascend the tree.
-         //
-
-         do {
-
-             SCHAR a;
-
-             //
-             // Calculate the next adjustment.
-             //
-
-             a = 1;
-             if (MiIsLeftChild (R)) {
-                 a = -1;
-             }
-
-             PRINT("LW 0: Table %p, Bal %x, %x\n", Table, Table->BalancedRoot.u1.Balance, a)
-             PRINT("LW 0: R Node %p, Bal %x, %x\n", R, R->u1.Balance, 1);
-             PRINT("LW 0: S Node %p, Bal %x, %x\n", S, S->u1.Balance, 1);
-
-             //
-             // If this node was balanced, show that it is no longer and
-             // keep looping.  This is essentially A6 of Knuth's algorithm,
-             // where he updates all of the intermediate nodes on the
-             // insertion path which previously had balance factors of 0.
-             // We are looping up the tree via Parent pointers rather than
-             // down the tree as in Knuth.
-             //
-
-             if (S->u1.Balance == 0) {
-
-                 PRINT("REBADJ F: Node %p, Bal %x -> %x\n", S, S->u1.Balance, a);
-                 COUNT_BALANCE_MAX ((SCHAR)a);
-                 S->u1.Balance = a;
-                 R = S;
-                 S = SANITIZE_PARENT_NODE (S->u1.Parent);
-             }
-             else if ((SCHAR) S->u1.Balance != a) {
```

```
-
-                   PRINT("LW 1: Table %p, Bal %x, %x\n", Table, Table->BalancedRoot.u1.Balance
-
-                   //
-                   // If this node has the opposite balance, then the tree got
-                   // more balanced (or we hit the root) and we are done.
-                   //
-                   // Step A7.ii
-                   //
-
-                   S->u1.Balance = 0;
+         // Beginning of ported code.
+         // PMMADDRESS_NODE node = NodeToInsert;
+         node = NodeToInsert;
+         root = &Table->BalancedRoot;
+         parent = rb_red_parent(node);
+
+         while (1) {
+             /*
+              * Loop invariant: node is red
+              *
+              * If there is a black parent, we are done.
+              * Otherwise, take some corrective action as we don't
+              * want a red root or two consecutive red nodes.
+              */
+             if (!parent) {
+                 rb_set_parent_color(node, NULL, RB_BLACK);
+                 break;
+             } else if (rb_is_black(parent))
+                 break;

-                   //
-                   // If S is actually the root, then this means the depth
-                   // of the tree just increased by 1!  (This is essentially
-                   // A7.i, but we just initialized the root balance to force
-                   // it through here.)
-                   //
+             gparent = rb_red_parent(parent);
+
+             tmp = gparent->RightChild;
+             if (parent != tmp) {    /* parent == gparent->LeftChild */
+                 if (tmp && rb_is_red(tmp)) {
```

```
+                       /*
+                        * Case 1 - 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.
+                        */
+                       rb_set_parent_color(tmp, gparent, RB_BLACK);
+                       rb_set_parent_color(parent, gparent, RB_BLACK);
+                       node = gparent;
+                       parent = rb_parent(node);
+                       rb_set_parent_color(node, parent, RB_RED);
+                       continue;
+                   }

-                   if (Table->BalancedRoot.u1.Balance == 0) {
-                       Table->DepthOfTree += 1;
+                   tmp = parent->RightChild;
+                   if (node == tmp) {
+                       /*
+                        * Case 2 - 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.
+                        */
+                       tmp = node->LeftChild;
+                       parent->RightChild = tmp;
+                       node->LeftChild = parent;
+                       if (tmp)
+                           rb_set_parent_color(tmp, parent, RB_BLACK);
```

```
+                    rb_set_parent_color(parent, node, RB_RED);
+                    parent = node;
+                    tmp = node->RightChild;
                }

+               /*
+                * Case 3 - right rotate at gparent
+                *
+                *         G           P
+                *        / \         / \
+                *       p   U  -->  n   g
+                *      /                 \
+                *     n                   U
+                */
+               gparent->LeftChild = tmp; /* == parent->RightChild */
+               parent->RightChild = gparent;
+               if (tmp)
+                   rb_set_parent_color(tmp, gparent, RB_BLACK);
+               __rb_rotate_set_parents(gparent, parent, root, RB_RED);
                break;
-           }
-           else {
-
-               PRINT("LW 2: Table %p, Bal %x, %x\n", Table, Table->BalancedRoot.u1.Balance
+           } else {
+               tmp = gparent->LeftChild;
+               if (tmp && rb_is_red(tmp)) {
+                   /* Case 1 - color flips */
+                   rb_set_parent_color(tmp, gparent, RB_BLACK);
+                   rb_set_parent_color(parent, gparent, RB_BLACK);
+                   node = gparent;
+                   parent = rb_parent(node);
+                   rb_set_parent_color(node, parent, RB_RED);
+                   continue;
+               }

-               //
-               // The tree became unbalanced (path length differs
-               // by 2 below us) and we need to do one of the balancing
-               // operations, and then we are done.  The RebalanceNode routine
-               // does steps A7.iii, A8 and A9.
-               //
```

```
+                tmp = parent->LeftChild;
+                if (node == tmp) {
+                    /* Case 2 - right rotate at parent */
+                    tmp = node->RightChild;
+                    parent->LeftChild = tmp;
+                    node->RightChild = parent;
+                    if (tmp)
+                        rb_set_parent_color(tmp, parent, RB_BLACK);
+                    rb_set_parent_color(parent, node, RB_RED);
+                    parent = node;
+                    tmp = node->LeftChild;
+                }

-                MiRebalanceNode (S);
+                /* Case 3 - left rotate at gparent */
+                gparent->RightChild = tmp; /* == parent->LeftChild */
+                parent->LeftChild = gparent;
+                if (tmp)
+                    rb_set_parent_color(tmp, gparent, RB_BLACK);
+                __rb_rotate_set_parents(gparent, parent, root, RB_RED);
                 break;
             }
-            PRINT("LW 3: Table %p, Bal %x, %x\n", Table, Table->BalancedRoot.u1.Balance, -1
-        } while (TRUE);
-        PRINT("LW 4: Table %p, Bal %x, %x\n", Table, Table->BalancedRoot.u1.Balance, -1);
+        }  // End of ported code.
     }

-    //
-    // Sanity check tree size and depth.
-    //
-
-    ASSERT((Table->NumberGenericTableElements >= MiWorstCaseFill[Table->DepthOfTree]) &&
-           (Table->NumberGenericTableElements <= MiBestCaseFill[Table->DepthOfTree]));
-
     return;
 }
```

5 实验结果

为了简化编译的流程，我们使用如下批处理文件来编译内核，并将编译出的二进制文件复制到合适的地方：

```
path \wrk-v1.2\tools\x86;%path%
cd \wrk-v1.2\base\ntos
nmake -nologo x86=
copy /y \WRK-v1.2\base\ntos\BUILD\EXE\wrkx86.exe \WINDOWS\system32\
```

然后在重启后选择进入 WRK 内核，系统能正常开机和运行，同时在正常的操作下不过崩溃，这便说明我们成功地将红黑树移植到了 WRK 内存管理中。

6 实验感想

本次实验的工作量确实比较大。一来需要阅读大量的 WRK 和 Linux 源代码，并理解二者的设计思路。二来移植的时候需要考虑到二者的结构差异，做到在影响最小的情况下将算法替换。这就要求我们能尽量复用 WRK 中原有的 AVL 树的结构，并在其原有结构上移植红黑树。

同时，我也第一次接触到了内核调试这一高大上的概念。与一般的程序调试不同的是，内核调试是在两个操作系统之间进行的，所以需要通过串口等方式传递信息。在本次实验中，我们是在虚拟机中虚拟了一个串口，从而在一台物理主机上实现了调试。

最后，我再次明白了这样一个道理：

> If you do it once, great. If you do it twice, frown. If you do it three times, automate it.

开始时，我每次编译都是手动输入那些指令，然后将生成的二进制文件手动复制到 system32 文件夹的，在经历了近十次痛苦的重复工作之后，我终于写了一个批处理文件。世界瞬间清静了……

总之，这次实验从许多方面来说都对我有很大帮助。读代码，移植，调试……真的是锻炼了综合能力呢！