

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。

```
1 struct rb_node {
2     unsigned long __rb_parent_color;
3     struct rb_node *rb_right;
4     struct rb_node *rb_left;
5 } __attribute__((aligned(sizeof(long))));
6     /* THE ALIGNMENT MIGHT SEEM POINTLESS, BUT ALLEGEDLY CRIS NEEDS IT */
7
8 struct rb_root {
```

```

9  struct rb_node *rb_node;
10 };

```

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。

```

1  #define RB_RED 0
2  #define RB_BLACK 1
3
4  #define __rb_parent(pc) ((struct rb_node *) (pc & ~3))
5
6  #define __rb_color(pc) ((pc) & 1)
7  #define __rb_is_black(pc) __rb_color(pc)
8  #define __rb_is_red(pc) (!__rb_color(pc))
9  #define rb_color(rb) __rb_color((rb)->__rb_parent_color)
10 #define rb_is_red(rb) __rb_is_red((rb)->__rb_parent_color)
11 #define rb_is_black(rb) __rb_is_black((rb)->__rb_parent_color)
12
13 static inline void rb_set_parent(struct rb_node *rb, struct rb_node *p)
14 {
15     rb->__rb_parent_color = rb_color(rb) | (unsigned long)p;
16 }
17
18 static inline void rb_set_parent_color(struct rb_node *rb,
19                                         struct rb_node *p, int color)
20 {
21     rb->__rb_parent_color = (unsigned long)p | color;
22 }
23
24 static inline void
25 __rb_change_child(struct rb_node *old, struct rb_node *new,
26                  struct rb_node *parent, struct rb_root *root)
27 {
28     if (parent) {
29         if (parent->rb_left == old)
30             WRITE_ONCE(parent->rb_left, new);
31         else
32             WRITE_ONCE(parent->rb_right, new);
33     } else
34         WRITE_ONCE(root->rb_node, new);
35 }

```

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。

```

1  typedef struct _MMADDRESS_NODE {
2      union {
3          LONG_PTR Balance : 2;
4          struct _MMADDRESS_NODE *Parent;
5      } u1;
6      struct _MMADDRESS_NODE *LeftChild;
7      struct _MMADDRESS_NODE *RightChild;
8      ULONG_PTR StartingVpn;
9      ULONG_PTR EndingVpn;
10 } MMADDRESS_NODE, *PMMADDRESS_NODE;
11
12 //
13 // A PAIR OF MACROS TO DEAL WITH THE PACKING OF PARENT & BALANCE IN THE
14 // MMADDRESS_NODE.
15 //
16
17 #define SANITIZE_PARENT_NODE(Parent) ((PMMADDRESS_NODE)((ULONG_PTR)(Parent)) & ~0x3)
18
19 //
20 // MACRO TO CAREFULLY PRESERVE THE BALANCE WHILE UPDATING THE PARENT.
21 //
22
23 #define MI_MAKE_PARENT(ParentNode, ExistingBalance) \
24     (PMMADDRESS_NODE)((ULONG_PTR)(ParentNode) | ((ExistingBalance) & 0x3))
25
26 typedef struct _MM_AVL_TABLE {
27     MMADDRESS_NODE BalancedRoot;
28     ULONG_PTR DepthOfTree: 5;
29     ULONG_PTR Unused: 3;
30 #if defined (_WIN64)
31     ULONG_PTR NumberGenericTableElements: 56;
32 #else
33     ULONG_PTR NumberGenericTableElements: 24;
34 #endif
35     PVOID NodeHint;

```

```
36     PVOID NodeFreeHint;  
37 } 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。

```
1  #define RB_RED 0
2  #define RB_BLACK 1
3
4  #define MiIsBlack(Links) ( \
5      (Links->u1.Balance == RB_BLACK) \
6      )
7
8  #define MiIsRed(Links) ( \
9      (Links->u1.Balance == RB_RED) \
10     )
11
12 #define MiRBParent(Links) ( \
13     (SANITIZE_PARENT_NODE((Links)->u1.Parent)) \
14     )
15
16 VOID
17 MiSetParentColor(
18     IN PMMADDRESS_NODE Links,
19     IN PMMADDRESS_NODE Parent,
20     IN ULONG Color
21 )
22 {
23     Links->u1.Parent = MI_MAKE_PARENT(Parent, Color);
24 }
25
26 VOID
27 MiSetParent(
28     IN PMMADDRESS_NODE Links,
29     IN PMMADDRESS_NODE Parent
30 )
31 {
32     ULONG Color = Links->u1.Balance;
33     MiSetParentColor(Links, Parent, Color);
34 }
35
36 VOID
37 MiChangeChild(
38     IN PMMADDRESS_NODE Old,
39     IN PMMADDRESS_NODE New,
40     IN PMMADDRESS_NODE Parent
41 )
42 {
43     if (Parent->LeftChild == Old)
44         Parent->LeftChild = New;
45     else
46         Parent->RightChild = New;
```

```
47 }
48
49 VOID
50 MiRotateSetParents(
51     IN PMMADDRESS_NODE Old,
52     IN PMMADDRESS_NODE New,
53     IN ULONG Color
54 )
55 {
56     PMMADDRESS_NODE Parent = MiRBPParent(Old);
57     New->u1 = Old->u1;
58     MiSetParentColor(Old, New, Color);
59     MiChangeChild(Old, New, Parent);
60 }
61
62 VOID
63 MiSetBlack(
64     IN PMMADDRESS_NODE Links
65 )
66 {
67     Links->u1.Parent = MI_MAKE_PARENT(MiRBPParent(Links), RB_BLACK);
68 }
69
70 VOID
71 MiPrintTreePreOrder(
72     IN PMMADDRESS_NODE Links
73 )
74 {
75     DbgPrint("Node %u, Parent %u, %u - %u, Color %d\n",
76             (ULONG)Links,
77             (ULONG)Links->u1.Parent,
78             Links->StartingVpn,
79             Links->EndingVpn,
80             Links->u1.Balance);
81     if (Links->LeftChild)
82         MiPrintTreePreOrder(Links->LeftChild);
83     if (Links->RightChild)
84         MiPrintTreePreOrder(Links->RightChild);
85 }
```

MiRemoveNode() 修改如下，其中添加了一些调试语句。

```
1 VOID
2 FASTCALL
3 MiRemoveNode (
4     IN PMMADDRESS_NODE NodeToDelete,
5     IN PMM_AVL_TABLE Table
6 )
```

```

7
8  /**
9
10 ROUTINE DESCRIPTION:
11
12     THIS ROUTINE DELETES THE SPECIFIED NODE FROM THE BALANCED TREE, REBALANCING
13     AS NECESSARY. IF THE NODEToDelete HAS AT LEAST ONE NULL CHILD POINTERS,
14     THEN IT IS CHOSEN AS THE EASYDELETE, OTHERWISE A SUBTREE PREDECESSOR OR
15     SUCCESSOR IS FOUND AS THE EASYDELETE. IN EITHER CASE THE EASYDELETE IS
16     DELETED AND THE TREE IS REBALANCED. FINALLY IF THE NODEToDelete WAS
17     DIFFERENT THAN THE EASYDELETE, THEN THE EASYDELETE IS LINKED BACK INTO THE
18     TREE IN PLACE OF THE NODEToDelete.
19
20 ARGUMENTS:
21
22     NODEToDelete - POINTER TO THE NODE WHICH THE CALLER WISHES TO DELETE.
23
24     TABLE - THE GENERIC TABLE IN WHICH THE DELETE IS TO OCCUR.
25
26 RETURN VALUE:
27
28     NONE.
29
30 ENVIRONMENT:
31
32     KERNEL MODE. THE PFN LOCK IS HELD FOR SOME OF THE TABLES.
33
34 --*/
35
36 {
37     PMMADDRESS_NODE Node = NodeToDelete;
38     PMMADDRESS_NODE Child = NodeToDelete->RightChild;
39     PMMADDRESS_NODE Temp = NodeToDelete->LeftChild;
40     PMMADDRESS_NODE Parent, Rebalance;
41     ULONG PC;
42
43     DbgPrint("Remove node begin, node %u, tree %u.\n", (ULONG)NodeToDelete, (ULONG>(&Table->
44         BalancedRoot));
45     MiPrintTreePreOrder(&Table->BalancedRoot);
46
47     if (!Temp) {
48         /*
49          * CASE 1: NODE TO ERASE HAS NO MORE THAN 1 CHILD (EASY!)
50          *
51          * NOTE THAT IF THERE IS ONE CHILD IT MUST BE RED DUE TO 5)
52          * AND NODE MUST BE BLACK DUE TO 4). WE ADJUST COLORS LOCALLY
53          * SO AS TO BYPASS __RB_ERASE_COLOR() LATER ON.

```

```

53  */
54  DbgPrint("Remove case 1 right.\n");
55  PC = (ULONG)Node->u1.Parent;
56  Parent = MiRBPParent(Node);
57  MiChangeChild(Node, Child, Parent);
58  if (Child) {
59      Child->u1.Parent = (PMADDRESS_NODE)PC;
60      Rebalance = NULL;
61  } else
62      Rebalance = MiIsBlack(Node) ? Parent : NULL;
63  Temp = Parent;
64  } else if (!Child) {
65      /* STILL CASE 1, BUT THIS TIME THE CHILD IS NODE->LEFTCHILD */
66      DbgPrint("Remove case 1 left.\n");
67      PC = (ULONG)Node->u1.Parent;
68      Temp->u1.Parent = (PMADDRESS_NODE)PC;
69      Parent = MiRBPParent(Node);
70      MiChangeChild(Node, Temp, Parent);
71      Rebalance = NULL;
72      Temp = Parent;
73  } else {
74      PMADDRESS_NODE Successor = Child, Child2;
75
76      Temp = Child->LeftChild;
77      if (!Temp) {
78          /*
79           * CASE 2: NODE'S SUCCESSOR IS ITS RIGHT CHILD
80           *
81           *      (N)          (S)
82           *    / \          / \
83           * (X) (S) -> (X) (C)
84           *      \
85           *      (C)
86           */
87          DbgPrint("Remove case 2.\n");
88          Parent = Successor;
89          Child2 = Successor->RightChild;
90      } else {
91          /*
92           * CASE 3: NODE'S SUCCESSOR IS LEFTMOST UNDER
93           * NODE'S RIGHT CHILD SUBTREE
94           *
95           *      (N)          (S)
96           *    / \          / \
97           * (X) (Y) -> (X) (Y)
98           *   /          /
99           *  (P)          (P)

```



```

100      *      /      /
101      * (s)      (c)
102      *      \
103      *      (c)
104      */
105      DbgPrint("Remove case 3.\n");
106      do {
107          Parent = Successor;
108          Successor = Temp;
109          Temp = Temp->LeftChild;
110      } while (Temp);
111      Child2 = Successor->RightChild;
112      Parent->LeftChild = Child2;
113      Successor->RightChild = Child;
114      MiSetParent(Child, Successor);
115  }
116
117      Temp = Node->LeftChild;
118      Successor->LeftChild = Temp;
119      MiSetParent(Temp, Successor);
120
121      PC = (ULONG)Node->u1.Parent;
122      Temp = MiRBPParent(Node);
123      MiChangeChild(Node, Successor, Temp);
124
125      if (Child2) {
126          MiSetParentColor(Child2, Parent, RB_BLACK);
127          Rebalance = NULL;
128      } else {
129          Rebalance = MiIsBlack(Successor) ? Parent : NULL;
130      }
131      Successor->u1.Parent = (PMMADDRESS_NODE)PC;
132      Temp = Successor;
133  }
134
135      DbgPrint("Remove rebalance %u.\n", (ULONG)Rebalance);
136      if (Rebalance && Rebalance != &Table->BalancedRoot) {
137          PMMADDRESS_NODE Node = NULL, Sibling, Temp1, Temp2;
138
139          while (TRUE) {
140              /*
141              * LOOP INVARIANTS:
142              * - NODE IS BLACK (OR NULL ON FIRST ITERATION)
143              * - NODE IS NOT THE ROOT (PARENT IS NOT NULL)
144              * - ALL LEAF PATHS GOING THROUGH PARENT AND NODE HAVE A
145              *   BLACK NODE COUNT THAT IS 1 LOWER THAN OTHER LEAF PATHS.
146              */

```

```

147     Sibling = Parent->RightChild;
148     if (Node != Sibling) { /* NODE == PARENT->LEFTCHILD */
149         if (MiIsRed(Sibling)) {
150             /*
151              * CASE 1 - LEFT ROTATE AT PARENT
152              *
153              *      P          S
154              *     / \        / \
155              *    N  S  -->  P  SR
156              *   / \        / \
157              *  SL SR      N  SL
158              */
159             DbgPrint("Remove rebalance case 1.\n");
160             Temp1 = Sibling->LeftChild;
161             Parent->RightChild = Temp1;
162             Sibling->LeftChild = Parent;
163             MiSetParentColor(Temp1, Parent, RB_BLACK);
164             MiRotateSetParents(Parent, Sibling, RB_RED);
165             Sibling = Temp1;
166         }
167         Temp1 = Sibling->RightChild;
168         if (!Temp1 || MiIsBlack(Temp1)) {
169             Temp2 = Sibling->LeftChild;
170             if (!Temp2 || MiIsBlack(Temp2)) {
171                 /*
172                  * CASE 2 - SIBLING COLOR FLIP
173                  * (P COULD BE EITHER COLOR HERE)
174                  *
175                  *      (P)          (P)
176                  *     / \        / \
177                  *    N  S  -->  N  S
178                  *   / \        / \
179                  *  SL SR      SL SR
180                  *
181                  * THIS LEAVES US VIOLATING 5) WHICH
182                  * CAN BE FIXED BY FLIPPING P TO BLACK
183                  * IF IT WAS RED, OR BY RECURSING AT P.
184                  * P IS RED WHEN COMING FROM CASE 1.
185                  */
186                 DbgPrint("Remove rebalance case 2.\n");
187                 MiSetParentColor(Sibling, Parent, RB_RED);
188                 if (MiIsRed(Parent))
189                     MiSetBlack(Parent);
190                 else {
191                     Node = Parent;
192                     Parent = MiRBPParent(Node);
193                     if (Parent && Parent != &Table->BalancedRoot)

```

```

194         continue;
195     }
196     break;
197 }
198 /*
199  * CASE 3 - RIGHT ROTATE AT SIBLING
200  * (P COULD BE EITHER COLOR HERE)
201  *
202  *      (P)          (P)
203  *    / \          / \
204  *  N  S  --> N  SL
205  *    / \          \
206  *   SL SR          S
207  *                   \
208  *                   SR
209  *
210  * NOTE: P MIGHT BE RED, AND THEN BOTH
211  * P AND SL ARE RED AFTER ROTATION(WHICH
212  * BREAKS PROPERTY 4). THIS IS FIXED IN
213  * CASE 4 (IN __RB_ROTATE_SET_PARENTS())
214  * WHICH SET SL THE COLOR OF P
215  * AND SET P RB_BLACK)
216  *
217  *      (P)          (SL)
218  *    / \          / \
219  *  N  SL  --> P  S
220  *    \      / \
221  *   S      N  SR
222  *    \
223  *   SR
224  */
225 DbgPrint("Remove rebalance case 3.\n");
226 Temp1 = Temp2->RightChild;
227 Sibling->LeftChild = Temp1;
228 Temp2->RightChild = Sibling;
229 Parent->RightChild = Temp2;
230 if (Temp1)
231     MiSetParentColor(Temp1, Sibling, RB_BLACK);
232 Temp1 = Sibling;
233 Sibling = Temp2;
234 }
235 /*
236  * CASE 4 - LEFT ROTATE AT PARENT + COLOR FLIPS
237  * (P AND SL COULD BE EITHER COLOR HERE.
238  * AFTER ROTATION, P BECOMES BLACK, S ACQUIRES
239  * P'S COLOR, AND SL KEEPS ITS COLOR)
240  *

```

```

241      *      (P)      (S)
242      *      / \      / \
243      *      N  S      --> P  SR
244      *      / \      / \
245      *      (SL) SR      N  (SL)
246      */
247      DbgPrint("Remove rebalance case 4.\n");
248      Temp2 = Sibling->LeftChild;
249      Parent->RightChild = Temp2;
250      Sibling->LeftChild = Parent;
251      MiSetParentColor(Temp1, Sibling, RB_BLACK);
252      if (Temp2)
253          MiSetParent(Temp2, Parent);
254      MiRotateSetParents(Parent, Sibling, RB_BLACK);
255      break;
256  } else {
257      Sibling = Parent->LeftChild;
258      if (MiIsRed(Sibling)) {
259          /* CASE 1 - RIGHT ROTATE AT PARENT */
260          DbgPrint("Remove rebalance case 1'\n");
261          Temp1 = Sibling->RightChild;
262          Parent->LeftChild = Temp1;
263          Sibling->RightChild = Parent;
264          MiSetParentColor(Temp1, Parent, RB_BLACK);
265          MiRotateSetParents(Parent, Sibling, RB_RED);
266          Sibling = Temp1;
267      }
268      Temp1 = Sibling->LeftChild;
269      if (!Temp1 || MiIsBlack(Temp1)) {
270          Temp2 = Sibling->RightChild;
271          if (!Temp2 || MiIsBlack(Temp2)) {
272              /* CASE 2 - SIBLING COLOR FLIP */
273              DbgPrint("Remove rebalance case 2'\n");
274              MiSetParentColor(Sibling, Parent, RB_RED);
275              if (MiIsRed(Parent))
276                  MiSetBlack(Parent);
277              else {
278                  Node = Parent;
279                  Parent = MiRBPParent(Node);
280                  if (Parent && Parent != &Table->BalancedRoot)
281                      continue;
282              }
283              break;
284          }
285          /* CASE 3 - LEFT ROTATE AT SIBLING */
286          DbgPrint("Remove rebalance case 3'\n");
287          Temp1 = Temp2->LeftChild;

```

```

288         Sibling->RightChild = Temp1;
289         Temp2->LeftChild = Sibling;
290         Parent->LeftChild = Temp2;
291         if (Temp1)
292             MiSetParentColor(Temp1, Sibling, RB_BLACK);
293         Temp1 = Sibling;
294         Sibling = Temp2;
295     }
296     /* CASE 4 - RIGHT ROTATE AT PARENT + COLOR FLIPS */
297     DbgPrint("Remove rebalance case 4\'.\\n");
298     Temp2 = Sibling->RightChild;
299     Parent->LeftChild = Temp2;
300     Sibling->RightChild = Parent;
301     MiSetParentColor(Temp1, Sibling, RB_BLACK);
302     if (Temp2)
303         MiSetParent(Temp2, Parent);
304     MiRotateSetParents(Parent, Sibling, RB_BLACK);
305     break;
306 }
307 }
308 }
309
310 Table->NumberGenericTableElements -= 1;
311 DbgPrint("Remove node finish.\\n");
312 MiPrintTreePreOrder(&Table->BalancedRoot);
313
314 return;
315 }

```

MiInsertNode() 修改如下。

```

1 VOID
2 FASTCALL
3 MiInsertNode (
4     IN PMMADDRESS_NODE NodeToInsert,
5     IN PMM_AVL_TABLE Table
6 )
7
8 /**+
9
10 ROUTINE DESCRIPTION:
11
12     THIS FUNCTION INSERTS A NEW ELEMENT IN A TABLE.
13
14 ARGUMENTS:
15
16     NODETOINSERT - THE INITIALIZED ADDRESS NODE TO INSERT.
17

```

```
18     TABLE - POINTER TO THE TABLE IN WHICH TO INSERT THE NEW NODE.
19
20 RETURN VALUE:
21
22     NONE.
23
24 ENVIRONMENT:
25
26     KERNEL MODE. THE PFN LOCK IS HELD FOR SOME OF THE TABLES.
27
28 --*/
29
30 {
31     //
32     // HOLDS A POINTER TO THE NODE IN THE TABLE OR WHAT WOULD BE THE
33     // PARENT OF THE NODE.
34     //
35     PMMADDRESS_NODE NodeOrParent;
36     TABLE_SEARCH_RESULT SearchResult;
37
38     DbgPrint("Insert node begin, node %u, tree %u.\n", (ULONG)NodeToInsert, (ULONG)(&Table->
        BalancedRoot));
39     MiPrintTreePreOrder(&Table->BalancedRoot);
40
41     SearchResult = MiFindNodeOrParent (Table,
42                                     NodeToInsert->StartingVpn,
43                                     &NodeOrParent);
44
45     ASSERT (SearchResult != TableFoundNode);
46
47     //
48     // THE NODE WASN'T IN THE (POSSIBLY EMPTY) TREE.
49     //
50     // WE JUST CHECK THAT THE TABLE ISN'T GETTING TOO BIG.
51     //
52
53     ASSERT (Table->NumberGenericTableElements != (MAXULONG-1));
54
55     NodeToInsert->LeftChild = NULL;
56     NodeToInsert->RightChild = NULL;
57
58     Table->NumberGenericTableElements += 1;
59
60     //
61     // INSERT THE NEW NODE IN THE TREE.
62     //
63
```

```
64     if (SearchResult == TableEmptyTree) {
65         DbgPrint("Insert empty tree.\n");
66         Table->BalancedRoot.RightChild = NodeToInsert;
67         NodeToInsert->u1.Parent = MI_MAKE_PARENT(&Table->BalancedRoot, RB_BLACK);
68         ASSERT (NodeToInsert->u1.Balance == RB_BLACK);
69         ASSERT(Table->DepthOfTree == 0);
70         Table->DepthOfTree = 1;
71     }
72 }
73 else {
74     PMMADDRESS_NODE Node = NodeToInsert;
75     PMMADDRESS_NODE Parent = NodeOrParent;
76     PMMADDRESS_NODE GParent, Temp;
77
78     DbgPrint("Insert non-empty tree.\n");
79
80     if (SearchResult == TableInsertAsLeft) {
81         NodeOrParent->LeftChild = NodeToInsert;
82     }
83     else {
84         NodeOrParent->RightChild = NodeToInsert;
85     }
86
87     Node->u1.Parent = MI_MAKE_PARENT(Parent, RB_RED);
88     ASSERT (NodeToInsert->u1.Balance == RB_RED);
89
90     while (TRUE) {
91         /*
92          * LOOP INVARIANT: NODE IS RED.
93          */
94         if (MiRBParent(Node) == &Table->BalancedRoot) {
95             /*
96              * THE INSERTED NODE IS ROOT. EITHER THIS IS THE
97              * FIRST NODE, OR WE RECURSED AT CASE 1 BELOW AND
98              * ARE NO LONGER VIOLATING 4).
99              */
100             DbgPrint("Insert root.\n");
101             MiSetParentColor(Node, &Table->BalancedRoot, RB_BLACK);
102             break;
103         }
104
105         /*
106          * IF THERE IS A BLACK PARENT, WE ARE DONE.
107          * OTHERWISE, TAKE SOME CORRECTIVE ACTION AS,
108          * PER 4), WE DON'T WANT A RED ROOT OR TWO
109          * CONSECUTIVE RED NODES.
110          */
```

```

111         if (MiIsBlack(Parent))
112             break;
113
114         GParent = MiRBPParent(Parent);
115
116         Temp = GParent->RightChild;
117         if (Parent != Temp) { /* PARENT == GPARENT->LEFTCHILD */
118             if (Temp && MiIsRed(Temp)) {
119                 /*
120                  * CASE 1 - NODE'S UNCLE IS RED (COLOR FLIPS).
121                  *
122                  *      G          G
123                  *     / \      / \
124                  *    P  U  --> P  U
125                  *   /      /
126                  *  N      N
127                  *
128                  * HOWEVER, SINCE G'S PARENT MIGHT BE RED, AND
129                  * 4) DOES NOT ALLOW THIS, WE NEED TO RECURSE
130                  * AT G.
131                  */
132                 DbgPrint("Insert case 1.\n");
133                 MiSetParentColor(Temp, GParent, RB_BLACK);
134                 MiSetParentColor(Parent, GParent, RB_BLACK);
135                 Node = GParent;
136                 Parent = MiRBPParent(Node);
137                 MiSetParentColor(Node, Parent, RB_RED);
138                 continue;
139             }
140
141             Temp = Parent->RightChild;
142             if (Node == Temp) {
143                 /*
144                  * CASE 2 - NODE'S UNCLE IS BLACK AND NODE IS
145                  * THE PARENT'S RIGHT CHILD (LEFT ROTATE AT PARENT).
146                  *
147                  *      G          G
148                  *     / \      / \
149                  *    P  U  --> N  U
150                  *   \      /
151                  *    N      P
152                  *
153                  * THIS STILL LEAVES US IN VIOLATION OF 4), THE
154                  * CONTINUATION INTO CASE 3 WILL FIX THAT.
155                  */
156                 DbgPrint("Insert case 2.\n");
157                 Temp = Node->LeftChild;

```



```

158         Parent->RightChild = Temp;
159         Node->LeftChild = Parent;
160         if (Temp)
161             MiSetParentColor(Temp, Parent, RB_BLACK);
162             MiSetParentColor(Parent, Node, RB_RED);
163             Parent = Node;
164             Temp = Node->RightChild;
165     }
166
167     /*
168     * CASE 3 - NODE'S UNCLE IS BLACK AND NODE IS
169     * THE PARENT'S LEFT CHILD (RIGHT ROTATE AT GPARENT).
170     *
171     *      G      P
172     *    / \    / \
173     *   P  U --> N  G
174     *  /      \
175     * N        U
176     */
177     DbgPrint("Insert case 3.\n");
178     GParent->LeftChild = Temp; /* == PARENT->RIGHTCHILD */
179     Parent->RightChild = GParent;
180     if (Temp)
181         MiSetParentColor(Temp, GParent, RB_BLACK);
182         MiRotateSetParents(GParent, Parent, RB_RED);
183         break;
184     } else {
185         Temp = GParent->LeftChild;
186         if (Temp && MiIsRed(Temp)) {
187             /* CASE 1 - COLOR FLIPS */
188             DbgPrint("Insert case 1\'.n");
189             MiSetParentColor(Temp, GParent, RB_BLACK);
190             MiSetParentColor(Parent, GParent, RB_BLACK);
191             Node = GParent;
192             Parent = MiRBPParent(Node);
193             MiSetParentColor(Node, Parent, RB_RED);
194             continue;
195         }
196
197         Temp = Parent->LeftChild;
198         if (Node == Temp) {
199             /* CASE 2 - RIGHT ROTATE AT PARENT */
200             DbgPrint("Insert case 2\'.n");
201             Temp = Node->RightChild;
202             Parent->LeftChild = Temp;
203             Node->RightChild = Parent;
204             if (Temp)

```

```
205         MiSetParentColor(Temp, Parent, RB_BLACK);
206         MiSetParentColor(Parent, Node, RB_RED);
207         Parent = Node;
208         Temp = Node->LeftChild;
209     }
210
211     /* CASE 3 - LEFT ROTATE AT GPARENT */
212     DbgPrint("Insert case 3\'.\\n");
213     GParent->RightChild = Temp; /* == PARENT->LEFTCHILD */
214     Parent->LeftChild = GParent;
215     if (Temp)
216         MiSetParentColor(Temp, GParent, RB_BLACK);
217     MiRotateSetParents(GParent, Parent, RB_RED);
218     break;
219 }
220 }
221 }
222
223     DbgPrint("Insert node finish.\\n");
224     MiPrintTreePreOrder(&Table->BalancedRoot);
225
226     return;
227 }
```

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

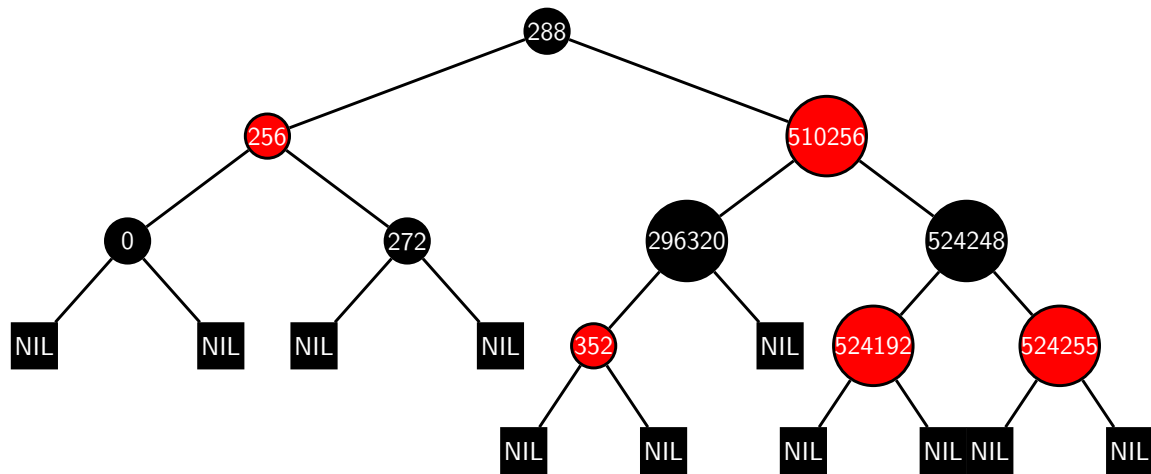


图 3: 插入前红黑树

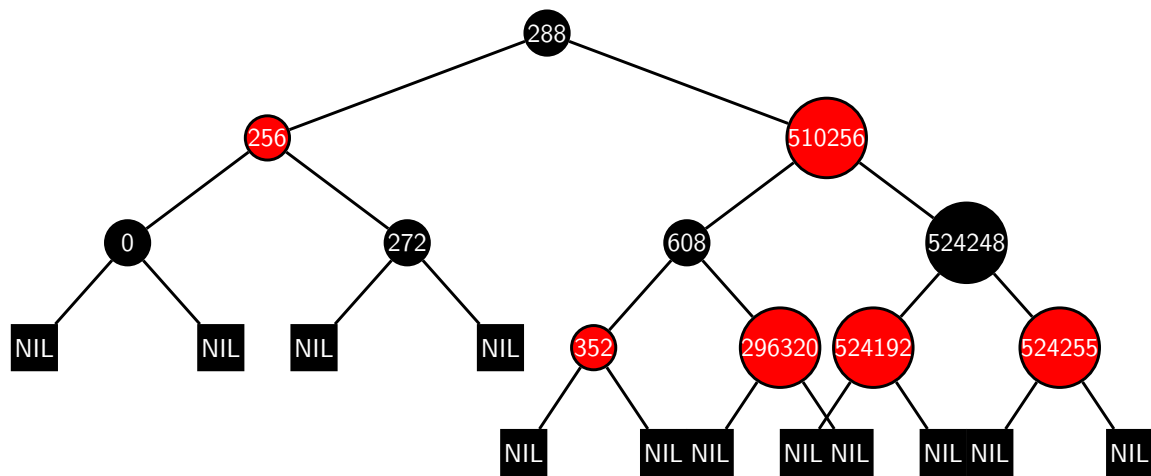


图 4: 插入后红黑树

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。