

# AVL 树 → 红黑树问题

聂浩 无 31 2013011280

2015 年 12 月 26 日

## 1 实验内容

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 实验思路

红黑树与 AVL 的查找基本是一致的，不需要太多的修改。

插入与删除函数通过对 linux(v4.4-rc6) 与 WRK1.2 的分析。利用用 linux 的红黑树代码替换 WRK 中的 AVL 管理。

### a) Linux 中的红黑树

Linux 中，虚拟内存管理的 VAD 由红黑树实现。

红黑树是一种在插入或删除节点是都需要维持平衡的二叉查找树，且每个节点都具有颜色属性：

1. 一个结点要么是红色的，要么是黑色的。
2. 根结点是黑色的。
3. 如果一个结点是红色的，那么它的子结点必须是黑色的，也就是说在沿着从根结点出发的任何路径上都不会出现两个连续的红色结点。
4. 从一个结点到 NULL 指针的每条路径上必须包含相同数目的黑色结点。

本次实验使用 Linux Kernel 4.4-rc6 版本。

linux 内核源代码中，红黑树的定义在三个地方，分别是

- include/linux/rbtree.h
- include/linux/rbtree\_augmented.h
- lib/rbtree.c

其中红黑树节点的定义为:

```

1
2 struct rb_node {
3     unsigned long   __rb_parent_color;
4     struct rb_node *rb_right;
5     struct rb_node *rb_left;
6 } __attribute__((aligned(sizeof(long))));
7     /* The alignment might seem pointless, but allegedly CRIS needs
8        it */
9
10 struct rb_root {
11     struct rb_node *rb_node;
12 };

```

有意思的是, 这里使用了 `__attribute__((aligned(sizeof(long))))`, 使得结构体进行 4 字节大小对齐 (32 位系统), 因此地址最低两位始终是 00, linux 使用其中的最低一位表示红黑树节点的颜色。

基于此 linux 定义了一些基本操作 (在 `rbtree_augmented.c` 中):

```

1  \\ 在 rbtree.h 中
2 #define rb_parent(r)    ((struct rb_node *)((r)->__rb_parent_color &
3     ~3))
4
5  \\ 在 rbtree_augmented.c
6
7 #define __rb_parent(pc)  ((struct rb_node *) (pc & ~3))
8
9 #define __rb_color(pc)  ((pc) & 1)
10 #define __rb_is_black(pc)  __rb_color(pc)
11 #define __rb_is_red(pc)   (!__rb_color(pc))
12 #define rb_color(rb)     __rb_color((rb)->__rb_parent_color)
13 #define rb_is_red(rb)    __rb_is_red((rb)->__rb_parent_color)
14 #define rb_is_black(rb)  __rb_is_black((rb)->__rb_parent_color)
15
16 static inline void rb_set_parent(struct rb_node *rb, struct rb_node *
17     p)
18 {
19     rb->__rb_parent_color = rb_color(rb) | (unsigned long)p;
20 }
21 // 设置父节点的同时设置自己的颜色

```

```

21 static inline void rb_set_parent_color(struct rb_node *rb,
22                                     struct rb_node *p, int color)
23 {
24     rb->__rb_parent_color = (unsigned long)p | color;
25 }
26
27 static inline void
28 __rb_change_child(struct rb_node *old, struct rb_node *new,
29                 struct rb_node *parent, struct rb_root *root)
30 {
31     if (parent) {
32         if (parent->rb_left == old)
33             WRITE_ONCE(parent->rb_left, new);
34             \\WRITE_ONCE(a,b)=>a=b;
35         else
36             WRITE_ONCE(parent->rb_right, new);
37     } else
38         WRITE_ONCE(root->rb_node, new);
39 }
40 \\在rbtree.c中, 为旋转做准备
41 static inline void
42 __rb_rotate_set_parents(struct rb_node *old, struct rb_node *new,
43                       struct rb_root *root, int color)
44 {
45     struct rb_node *parent = rb_parent(old);
46     new->__rb_parent_color = old->__rb_parent_color;
47     rb_set_parent_color(old, new, color);
48     __rb_change_child(old, new, parent, root);
49 }

```

linux 红黑树的插入：具体调用为 `rb_insert_color`，然后再调用内部函数 `__rb_insert`，这里代码太长不在此处展示，具体代码可见 `rbtree.c` 函数。需要注意的是，linux 的插入函数处理的是节点已经添加在树上后的平衡过程。`root` 指向红黑树的根节点，但是根节点的 `parent` 指向 `NULL`。

linux 红黑树的删除：具体调用 `rb_erase` 函数，内容如下：

```

1 void rb_erase(struct rb_node *node, struct rb_root *root)
2 {
3     struct rb_node *rebalance;
4     rebalance = __rb_erase_augmented(node, root, &dummy_callbacks);
5     if (rebalance)

```

```

6         _____rb_erase_color(rebalance , root , dummy_rotate);
7     }
8     EXPORT_SYMBOL(rb_erase);

```

\_\_\_\_\_rb\_erase\_augmented 用于直接删除节点并判断是否破坏了红黑树的性质，在 rbtree\_augmented.c 中定义，\_\_\_\_\_rb\_erase\_color 用于修复被破坏的红黑树，其代码 rbtree.c

除此之外，linux 红黑树的实现还有替换 rb\_replace\_node，寻找前驱后继等函数，这些红黑树和 AVL 树没有区别，本实验不需要移植，在这里不再讨论。

## b) Windows 中的 AVL 树

Windows 中，虚拟内存管理的 VAD 是以 AVL 树的形式管理的。

Windows 的虚拟内存管理利用 MM\_AVL\_TABLE 型变量，其定义在 base/ntos/inc/ps.h,

```

1  typedef struct _MM_AVL_TABLE {
2      MMADDRESS_NODE  BalancedRoot;
3      ULONG_PTR DepthOfTree : 5;
4      ULONG_PTR Unused : 3;
5  #if defined (_WIN64)
6      ULONG_PTR NumberGenericTableElements : 56;
7  #else
8      ULONG_PTR NumberGenericTableElements : 24;
9  #endif
10     PVOID NodeHint;
11     PVOID NodeFreeHint;
12 } MM_AVL_TABLE, *PMM_AVL_TABLE;

```

其中的 BalancedRoot 保存了 AVL 树的根节点信息。NumberGenericTableElements 需要在插入和删除节点的时候进行修改。

通过相关资料查询以及对具体源代码的分析可知 BalancedRoot 的 RightChild 指向根节点，而根节点的 Parent 指向 BalancedRoot，这里和 linux 有较大的区别。

MMADDRESS\_NODE 的定义在在 base/ntos/mm/mi.h 里:

```

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;
```

和 linux 类似的是，Windows 也是利用存储的父地址的低两位存储节点的性质（平衡因子），不同的是，Windows 的实现方式是联合体。具体实践中发现，该联合体读取时是读出全部 4 字节，但写入时 Balance 和 Parent 需要分开写；红黑树不需要平衡因子，这里利用 Balance 变量存储颜色。后两个数是存储的页信息，和本实验无关。

Windows 中维护 AVL 树的函数主要有：

- 插入 MiInsertNode
- 平衡 MiRebalanceNode 和 MiPromoteNode
- 删除 MiRemoveNode 等等。

对外的接口有 MiInsertNode 和 MiRemoveNode，这也是我们在实验过程中需要保留并重新实现的，它们出现在/base/ntos/mm/addrsup.c 中。

### 3 实验过程

为了使用 linux 中的函数，先定义以下基础操作

```
1
2 #define rb_black 0
3 #define rb_red 1
4
5 PMMADDRESS_NODE rb_parent(PMMADDRESS_NODE node)
6 {
7     node = SANITIZE_PARENT_NODE(SANITIZE_PARENT_NODE(node)->u1.Parent)
8     ;
9     return node;
10 }
11
12 int rb_color(PMMADDRESS_NODE node)
13 {
14     return node->u1.Balance;
15 }
16
17 int rb_is_red(PMMADDRESS_NODE node)
18 {
19     return node->u1.Balance==rb_red;
20 }
21
22 int rb_is_black(PMMADDRESS_NODE node)
23 {
24     return node->u1.Balance==rb_black;
25 }
```

```
22 void rb_set_black(PMMADDRESS_NODE node)
23 {
24     node->u1.Balance = rb_black;
25 }
26 void rb_set_red(PMMADDRESS_NODE node)
27 {
28     node->u1.Balance = rb_red;
29 }
30 void rb_set_parent( PMMADDRESS_NODE rb,PMMADDRESS_NODE p)
31 {
32     rb->u1.Parent =(PMMADDRESS_NODE) (((ULONG_PTR)(p)) + ((ULONG_PTR)(
33         rb->u1.Balance))) );
34 }
35 void rb_set_color( PMMADDRESS_NODE rb, int color)
36 {
37     rb->u1.Balance = color;
38 }
39
40 void rb_set_parent_color(PMMADDRESS_NODE rb, PMMADDRESS_NODE p,int
41     color)
42 {
43     rb->u1.Parent =p;
44     rb->u1.Balance=color;
45 }
46 void __rb_change_child(PMMADDRESS_NODE old, PMMADDRESS_NODE newer,
47     PMMADDRESS_NODE parent, PMM_AVL_TABLE root)
48 {
49     if (parent!=&root->BalancedRoot) {
50         if (parent->LeftChild== old){
51             parent->LeftChild = newer;
52         }
53         else
54             parent->RightChild = newer;
55     } else
56         root->BalancedRoot.RightChild = newer;
57 }
58
```

```

59 void __rb_rotate_set_parents(PMMADDRESS_NODE old, PMMADDRESS_NODE
    newer, PMM_AVL_TABLE root, int color)
60 {
61     PMMADDRESS_NODE parent = rb_parent(old);
62     rb_set_color(newer, rb_color(old));
63     rb_set_parent(newer, rb_parent(old));
64     rb_set_parent_color(old, newer, color);
65     __rb_change_child(old, newer, parent, root);
66 }

```

其中 SANITIZE\_PARENT\_NODE 的定义如下 (在 ntos/inc/ps.h 中)

```

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

```

### a) 插入节点

插入节点的函数在 base/ntos/mm/addrsup.c 中, 接口为

```

1 VOID
2 FASTCALL
3 MiInsertNode(
4 IN PMMADDRESS_NODE NodeToInsert,
5 IN PMM_AVL_TABLE Table
6 )

```

与 linux 不同的是, windows 在这一过程中同时进行了插入和平衡的操作, linux 只有平衡, 即 MiInsertNode 中插入的一部分应该保留并稍作修改, 代码如下:

```

1
2     {
3         PMMADDRESS_NODE NodeOrParent;
4         PMMADDRESS_NODE parent, gparent, tmp;
5         TABLE_SEARCH_RESULT SearchResult;
6         // 插入函数
7         SearchResult = MiFindNodeOrParent (Table,
8         NodeToInsert->StartingVpn,
9         &NodeOrParent);
10
11         NodeToInsert->LeftChild = NULL;
12         NodeToInsert->RightChild = NULL;
13

```

```
14      Table->NumberGenericTableElements += 1;
15
16      //
17      // Insert the newer node in the tree.
18      //
19
20      if (SearchResult == TableEmptyTree)
21      {
22          Table->BalancedRoot.RightChild = NodeToInsert;
23          rb_set_parent(NodeToInsert, &Table->BalancedRoot);
24      }
25      else
26      {
27          if (SearchResult == TableInsertAsLeft)
28          {
29              NodeOrParent->LeftChild = NodeToInsert;
30          }
31          else
32          {
33              NodeOrParent->RightChild = NodeToInsert;
34          }
35          rb_set_parent(NodeToInsert, NodeOrParent);
36      }
37      rb_set_red(NodeToInsert);
38      parent = rb_parent(NodeToInsert);
39      // 平衡部分，直接由linux代码修改，需要注意根节点的性质
40      while(1) {
41          if (parent == &Table->BalancedRoot) {
42              Table->BalancedRoot.RightChild = NodeToInsert;
43              rb_set_parent_color(NodeToInsert, &Table->BalancedRoot,
44                                  rb_black);
45              break;
46          } else if (rb_is_black(parent))
47              break;
48          gparent = rb_parent(parent);
49          tmp = gparent->RightChild;
50
51          if (parent != tmp) { /* parent == gparent->rb_left */
52              if (tmp && rb_is_red(tmp)) {
```



```

52      /*
53      * Case 1 – color flips
54      *
55      *      G      g
56      *     /\    /\
57      *    p  u  →  P  U
58      *   /\    /\
59      *  n      n
60      *
61      * However, since g's parent might be red, and
62      * 4) does not allow this, we need to recurse
63      * at g.
64      */
65      rb_set_parent_color(tmp, gparent, rb_black);
66      rb_set_parent_color(parent, gparent, rb_black);
67      NodeToInsert = gparent;
68      parent = rb_parent(NodeToInsert);
69      rb_set_parent_color(NodeToInsert, parent, rb_red);
70      continue;
71  }
72
73  tmp = parent->RightChild;
74  if (NodeToInsert == tmp) {
75      /*
76      * Case 2 – left rotate at parent
77      *
78      *      G      G
79      *     /\    /\
80      *    p  U  →  n  U
81      *     \    /\
82      *      n  p
83      *
84      * This still leaves us in violation of 4), the
85      * continuation into Case 3 will fix that.
86      */
87      parent->RightChild = tmp = NodeToInsert->LeftChild;
88      NodeToInsert->LeftChild = parent;
89      if (tmp)
90          rb_set_parent_color(tmp, parent, rb_black);

```

```

91         rb_set_parent_color(parent, NodeToInsert, rb_red);
92         parent = NodeToInsert;
93         tmp = NodeToInsert->RightChild;
94     }
95
96     /*
97     * Case 3 - right rotate at gparent
98     *
99     *      G      P
100    *    / \    / \
101    *   p  U  -> n  g
102    *   /      \
103    *  n        U
104    */
105     gparent->LeftChild= tmp; /* == parent->rb_right */
106     parent->RightChild= gparent;
107     if (tmp)
108         rb_set_parent_color(tmp, gparent, rb_black);
109     __rb_rotate_set_parents(gparent, parent, Table, rb_red);
110     break;
111 } else {
112     tmp = gparent->LeftChild;
113     if (tmp && rb_is_red(tmp)) {
114         /* Case 1 - color flips */
115         rb_set_parent_color(tmp, gparent, rb_black);
116         rb_set_parent_color(parent, gparent, rb_black);
117         NodeToInsert = gparent;
118         parent = rb_parent(NodeToInsert);
119         rb_set_parent_color(NodeToInsert, parent, rb_red);
120         continue;
121     }
122
123     tmp = parent->LeftChild;
124     if (NodeToInsert == tmp) {
125         /* Case 2 - right rotate at parent */
126         parent->LeftChild = tmp = NodeToInsert->RightChild;
127         NodeToInsert->RightChild = parent;
128         if (tmp)
129             rb_set_parent_color(tmp, parent,

```

```

130         rb_black);
131         rb_set_parent_color(parent, NodeToInsert, rb_red);
132         parent = NodeToInsert;
133         tmp = NodeToInsert->LeftChild;
134     }
135
136     /* Case 3 - left rotate at gparent */
137     gparent->RightChild = tmp; /* == parent->rb_left */
138     parent->LeftChild = gparent;
139     if (tmp)
140         rb_set_parent_color(tmp, gparent, rb_black);
141     __rb_rotate_set_parents(gparent, parent, Table, rb_red);
142     break;
143 }
144 }
145 return;
146 }

```

有趣的是，linux 中给 `__rb_insert` 传入 `void (*augment_rotate)` 这一函数指针，但是该函数是空的（应该是给用户调用提供类似重载的特性），所以直接将相关语句删除即可。

## b) 删除节点

删除节点的函数也出现在 `base/ntos/mm/addrsup.c` 中，其接口为：

```

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

```

使用 linux 代码前需要先移植 `__rb_erase_augmented` 和 `__rb_erase_color`；代码如下

```

1
2 static PMMADDRESS_NODE __rb_erase_augmented(PMMADDRESS_NODE node,
3 PMM_AVL_TABLE root)
4 {
5     PMMADDRESS_NODE child = node->RightChild, tmp = node->LeftChild;
6     PMMADDRESS_NODE parent, rebalance;
7     PMMADDRESS_NODE pc;

```

```

8   if (!tmp) {
9       /*
10      * Case 1: node to erase has no more than 1 child (easy!)
11      *
12      * Note that if there is one child it must be red due to 5)
13      * and node must be black due to 4). We adjust colors locally
14      * so as to bypass __rb_erase_color() later on.
15      */
16      pc = node->u1.Parent;
17      parent = rb_parent(node);
18      __rb_change_child(node, child, parent, root);
19      if (child) {
20          child->u1.Parent = pc;
21          child->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
22          rebalance = &root->BalancedRoot;
23      } else
24          rebalance = (((ULONG_PTR)(pc)) & 0x1) == 0 ? parent : &root
25                      ->BalancedRoot;
26      } else if (!child) {
27          /* Still case 1, but this time the child is node->rb_left */
28          tmp->u1.Parent = pc = node->u1.Parent;
29          tmp->u1.Balance = node->u1.Balance;
30          parent = rb_parent(node);
31          __rb_change_child(node, tmp, parent, root);
32          rebalance = &root->BalancedRoot;
33      } else {
34          PMMADDRESS_NODE successor = child, child2;
35          tmp = child->LeftChild;
36          if (!tmp) {
37              /*
38              * Case 2: node's successor is its right child
39              *
40              *      (n)          (s)
41              *      / \        / \
42              *  (x) (s)  ->  (x) (c)
43              *              \
44              *              (c)
45              */
46              parent = successor;

```

```

46         child2 = successor->RightChild;
47     } else {
48         /*
49         * Case 3: node's successor is leftmost under
50         * node's right child subtree
51         *
52         *      (n)          (s)
53         *     / \         / \
54         *    (x) (y)  -> (x) (y)
55         *     /         /
56         *    (p)        (p)
57         *     /         /
58         *    (s)        (c)
59         *     \
60         *    (c)
61         */
62         do {
63             parent = successor;
64             successor = tmp;
65             tmp = tmp->LeftChild;
66         } while (tmp);
67         parent->LeftChild = child2 = successor->RightChild;
68         successor->RightChild = child;
69         rb_set_parent(child, successor);
70     }
71
72     successor->LeftChild = tmp = node->LeftChild;
73     rb_set_parent(tmp, successor);
74     pc = node->ul.Parent;
75     tmp = SANITIZE_PARENT_NODE(pc);
76     __rb_change_child(node, successor, tmp, root);
77     if (child2) {
78         successor->ul.Parent = pc;
79         successor->ul.Balance = (((ULONG_PTR)(pc)) & 0x1);
80         rb_set_parent_color(child2, parent, rb_black);
81         rebalance = &root->BalancedRoot;
82     } else {
83         PMMADDRESS_NODE pc2 = successor->ul.Parent;
84         successor->ul.Parent = pc;

```

```

85         successor->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
86         rebalance = (((ULONG_PTR)(pc2)) & 0x1)==0 ? parent : &
            root->BalancedRoot;
87     }
88     tmp = successor;
89 }
90 return rebalance;
91 }
92
93 static void
94 _____rb_erase_color(PMMADDRESS_NODE parent, PMM_AVL_TABLE root)
95 {
96     PMMADDRESS_NODE node =NULL, sibling, tmp1=NULL, tmp2=NULL;
97
98     while (1) {
99         /*
100          * Loop invariants:
101          * - node is black (or NULL on first iteration)
102          * - node is not the root (parent is not NULL)
103          * - All leaf paths going through parent and node have a
104          *   black node count that is 1 lower than other leaf paths.
105          */
106         sibling = parent->RightChild;
107         if (node != sibling) { /* node == parent->rb_left */
108             if (rb_is_red(sibling)) {
109                 /*
110                  * Case 1 - left rotate at parent
111                  *
112                  *      P              S
113                  *     / \            / \
114                  *    N  s      ->  p  Sr
115                  *     / \            / \
116                  *    Sl Sr        N  Sl
117                  */
118                 parent->RightChild = tmp1 = sibling->LeftChild;
119                 sibling->LeftChild = parent;
120                 if (tmp1)
121                     rb_set_parent_color(tmp1, parent, rb_black);
122                 __rb_rotate_set_parents(parent, sibling, root,

```

```

123         rb_red);
124     sibling = tmp1;
125 }
126 tmp1 = sibling->RightChild;
127 if (!tmp1 || rb_is_black(tmp1)) {
128     tmp2 = sibling->LeftChild;
129     if (!tmp2 || rb_is_black(tmp2)) {
130         /*
131          * Case 2 - sibling color flip
132          * (p could be either color here)
133          *
134          *      (p)          (p)
135          *     / \        / \
136          *    N  S  ->  N  s
137          *   / \        / \
138          *  Sl Sr      Sl Sr
139          *
140          * This leaves us violating 5) which
141          * can be fixed by flipping p to black
142          * if it was red, or by recursing at p.
143          * p is red when coming from Case 1.
144          */
145         rb_set_parent_color(sibling, parent,
146                             rb_red);
147         if (rb_is_red(parent))
148             rb_set_black(parent);
149         else {
150             node = parent;
151             parent = rb_parent(node);
152             if (parent != &root->BalancedRoot)
153                 continue;
154         }
155         break;
156     }
157     /*
158     * Case 3 - right rotate at sibling
159     * (p could be either color here)
160     *
161     *      (p)          (p)

```

```

162      *      / \      / \
163      *    N   S  —> N   Sl
164      *      / \      \
165      *     sl  Sr      s
166      *                      \
167      *                      Sr
168      */
169      sibling->LeftChild = tmp1 = tmp2->RightChild;
170      tmp2->RightChild = sibling;
171      parent->RightChild = tmp2;
172      if (tmp1)
173          rb_set_parent_color(tmp1, sibling,
174                               rb_black);
175      tmp1 = sibling;
176      sibling = tmp2;
177  }
178  /*
179  * Case 4 – left rotate at parent + color flips
180  * (p and sl could be either color here.
181  * After rotation, p becomes black, s acquires
182  * p's color, and sl keeps its color)
183  *
184  *      (p)          (s)
185  *      / \          / \
186  *     N   S  —>   P   Sr
187  *      / \          / \
188  *     (sl) sr     N   (sl)
189  */
190      parent->RightChild = tmp2 = sibling->LeftChild;
191      sibling->LeftChild = parent;
192      rb_set_parent_color(tmp1, sibling, rb_black);
193      if (tmp2)
194          rb_set_parent(tmp2, parent);
195      __rb_rotate_set_parents(parent, sibling, root,
196                              rb_black);
197      break;
198  } else {
199      sibling = parent->LeftChild;
200      if (rb_is_red(sibling)) {

```



```

201      /* Case 1 - right rotate at parent */
202      parent->LeftChild = tmp1 = sibling->RightChild;
203      sibling->RightChild = parent;
204      rb_set_parent_color(tmp1, parent, rb_black);
205      __rb_rotate_set_parents(parent, sibling, root,
206          rb_red);
207      sibling = tmp1;
208  }
209  tmp1 = sibling->LeftChild;
210  if (!tmp1 || rb_is_black(tmp1)) {
211      tmp2 = sibling->RightChild;
212      if (!tmp2 || rb_is_black(tmp2)) {
213          /* Case 2 - sibling color flip */
214          rb_set_parent_color(sibling, parent,
215              rb_red);
216          if (rb_is_red(parent))
217              rb_set_black(parent);
218          else {
219              node = parent;
220              parent = rb_parent(node);
221              if (parent != &root->BalancedRoot)
222                  continue;
223          }
224          break;
225      }
226      /* Case 3 - right rotate at sibling */
227      sibling->RightChild = tmp1 = tmp2->LeftChild;
228      tmp2->LeftChild = sibling;
229      parent->LeftChild = tmp2;
230      if (tmp1)
231          rb_set_parent_color(tmp1, sibling,
232              rb_black);
233      tmp1 = sibling;
234      sibling = tmp2;
235  }
236  /* Case 4 - left rotate at parent + color flips */
237  parent->LeftChild = tmp2 = sibling->RightChild;
238  sibling->RightChild = parent;
239  rb_set_parent_color(tmp1, sibling, rb_black);

```

```

240         if (tmp2)
241             rb_set_parent(tmp2, parent);
242         __rb_rotate_set_parents(parent, sibling, root,
243             rb_black);
244         break;
245     }
246 }
247 }

```

值得注意的是根节点的 Parent 值为 &Table->BalancedRoot, 以及记录颜色和父节点信息的联合体的特性。

之后再修改 MiRemoveNode 函数为,

```

1 {
2     PMMADDRESS_NODE rebalance;
3     Table->NumberGenericTableElements -= 1;
4     rebalance = __rb_erase_augmented(NodeToDelete, Table);
5     if (rebalance != &Table->BalancedRoot && rebalance)
6         __rb_erase_color(rebalance, Table);
7 }

```

即可完成移植

## 4 实验结果

到此为止, 代码移植工作已经完成, 编译好后 (nmake -nologo x86=) 复制到虚拟机 (Windows Server 2003 SP2) 的 C:/Windows/system32 文件夹, 设置好主机的 WinDbg<sup>1</sup> 调试器和虚拟机的启动引导信息<sup>2</sup>, 然后选择 Debug 引导启动, 如图1。

一切正常则能够正常进入系统, 并可以正常操作, 如图2。通过 WinDbg 可以发现, 在系统的初始或与平常使用中, 虚拟内存管理是一直使用的, 在对该红黑树在不断的插入与删除节点, 所以系统可以稳定运行就意味着没有 bug。

## 5 实验感想

本实验一开始思考比较简单, 因为无论是 WRK 还是 Linux 内核都有着大量的资料, 而且本实验的目的是一致的, 只需要利用原有的接口进行一定的修改即可。

但是随着任务的深入, 很多问题并不如想象的顺利: linux 在 3.x 版本起对红黑树部分进行了大改, 原有文档过于陈旧; 系统接口难以定位; MS 的 symbol 服务器关机导致 WinDbg 难以进行复杂调试;

<sup>1</sup> 见 debug.bat 文件, 需先导入 debug.WEW

<sup>2</sup> boot.in 增加信息 multi(0)disk(0)rdisk(0)partition(1) \WINDOWS="Debug" /kernel=wrkx86.exe /hal=halmacpi.dll /debug /debugport=com1 /baudrate=115200

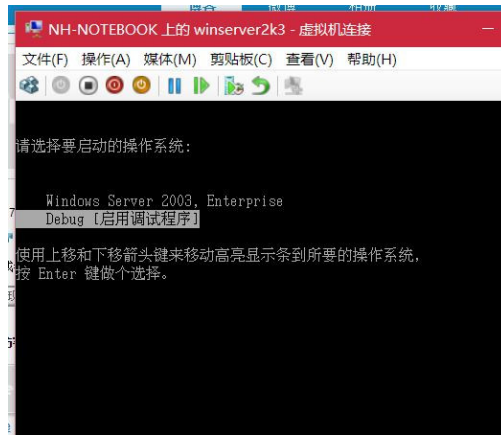


图 1: boot 选项

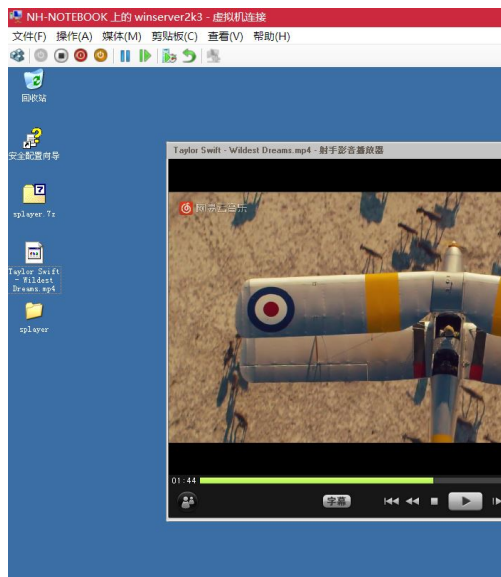


图 2: 正常工作

两者数据结构差异而导致的算法细节区别较大；联合体这一少见数据结构的陷阱。

这一个个问题很令我烦恼，也消耗了大量的时间在其中去解决，但我从中学到了很多。我学会了更好的利用 ctag 去阅读代码，学会了更好的从大量资料中寻找信息，学会了将大型项目中的函数及其依赖剥离出来进行测试<sup>3</sup>，了解了 Linux 内核和 Windows 内核迥异的代码风格以及他们在虚拟内存管上的异同，对操作系统有了更深一步的理解。

虽然本次实验所涉及的 VAD 修改只是操作系统的一小部分，它仅仅完成是内存的组织、分配、调度和回收，但这一过程去除了操作系统的神秘感，使我能更好的去探索这两份代码中的秘密，也为以后的学习工作带来很好的基础。

## 6 实验代码

系统原有代码见 origin 文件夹，修改后的部分都存储在 addrsup.c 文件中，使用时直接用 addrsup.c 替换 wrk 中的同名文件编译即可。

```
1  /*++
2
3  Copyright (c) Microsoft Corporation. All rights reserved.
4
5  You may only use this code if you agree to the terms of the Windows
   Research Kernel Source Code License agreement (see License.txt).
6  If you do not agree to the terms, do not use the code.
7
8
9  Module Name:
10
11     addrsup.c
12
13  Abstract:
14
15     This module implements a new version of the generic table package
16     based on balanced binary trees (later named AVL), as described in
17     Knuth, "The Art of Computer Programming, Volume 3, Sorting and
       Searching",
18     and refers directly to algorithms as they are presented in the
       second
19     edition Copyrighted in 1973.
20
21     Used rtl\avltable.c as a starting point, adding the following:
22
```

<sup>3</sup>见测试文件中的测试项目

```

23     - Use less memory for structures as these are nonpaged & heavily
        used.
24     - Caller allocates the pool to reduce mutex hold times.
25     - Various VAD-specific customizations/optimizations.
26     - Hints.
27
28 Environment:
29
30     Kernel mode only, working set mutex held, APCs disabled.
31
32 ---*/
33
34 #include "mi.h"
35
36 #define rb_black 0
37 #define rb_red 1
38
39 PMMADDRESS_NODE rb_parent(PMMADDRESS_NODE node)
40 {
41     node = SANITIZE_PARENT_NODE(SANITIZE_PARENT_NODE(node)->u1.Parent)
42     ;
43     return node;
44 }
45
46 int rb_color(PMMADDRESS_NODE node)
47 {
48     return node->u1.Balance;
49 }
50
51 int rb_is_red(PMMADDRESS_NODE node)
52 {
53     return node->u1.Balance==rb_red;
54 }
55
56 int rb_is_black(PMMADDRESS_NODE node)
57 {
58     return node->u1.Balance==rb_black;
59 }
60
61 void rb_set_black(PMMADDRESS_NODE node)
62 {
63     node->u1.Balance = rb_black;
64 }

```

```
60 void rb_set_red(PMMADDRESS_NODE node)
61 {
62     node->u1.Balance = rb_red;
63 }
64 void rb_set_parent( PMMADDRESS_NODE rb ,PMMADDRESS_NODE p)
65 {
66     rb->u1.Parent =(PMMADDRESS_NODE) (((ULONG_PTR)(p)) + ((ULONG_PTR)(
67         rb->u1.Balance))) );
68 }
69 void rb_set_color( PMMADDRESS_NODE rb, int color)
70 {
71     rb->u1.Balance = color;
72 }
73
74 void rb_set_parent_color(PMMADDRESS_NODE rb, PMMADDRESS_NODE p, int
75     color)
76 {
77     rb->u1.Parent =p;
78     rb->u1.Balance=color;
79 }
80 void
81 __rb_change_child(PMMADDRESS_NODE old, PMMADDRESS_NODE newer,
82 PMMADDRESS_NODE parent, PMM_AVL_TABLE root)
83 {
84     if (parent!=&root->BalancedRoot) {
85         if (parent->LeftChild== old){
86             parent->LeftChild = newer;
87         }
88         else
89             parent->RightChild = newer;
90     } else
91         root->BalancedRoot.RightChild = newer;
92 }
93
94 void __rb_rotate_set_parents(PMMADDRESS_NODE old, PMMADDRESS_NODE
95     newer ,
96                             PMM_AVL_TABLE root, int color)
```

```

96 {
97     PMMADDRESS_NODE parent = rb_parent(old);
98     rb_set_color(newer, rb_color(old));
99     rb_set_parent(newer, rb_parent(old));
100     rb_set_parent_color(old, newer, color);
101     __rb_change_child(old, newer, parent, root);
102 }
103
104 static PMMADDRESS_NODE __rb_erase_augmented(PMMADDRESS_NODE node,
105     PMM_AVL_TABLE root)
106 {
107     PMMADDRESS_NODE child = node->RightChild, tmp = node->LeftChild;
108     PMMADDRESS_NODE parent, rebalance;
109     PMMADDRESS_NODE pc;
110
111     if (!tmp) {
112         /*
113          * Case 1: node to erase has no more than 1 child (easy!)
114          *
115          * Note that if there is one child it must be red due to 5)
116          * and node must be black due to 4). We adjust colors locally
117          * so as to bypass __rb_erase_color() later on.
118          */
119         pc = node->u1.Parent;
120         parent = rb_parent(node);
121         __rb_change_child(node, child, parent, root);
122         if (child) {
123             child->u1.Parent = pc;
124             child->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
125             rebalance = &root->BalancedRoot;
126         } else
127             rebalance = (((ULONG_PTR)(pc)) & 0x1) == 0 ? parent : &root
128                 ->BalancedRoot;
129     } else if (!child) {
130         /* Still case 1, but this time the child is node->rb_left */
131         tmp->u1.Parent = pc = node->u1.Parent;
132         tmp->u1.Balance = node->u1.Balance;
133         parent = rb_parent(node);
134         __rb_change_child(node, tmp, parent, root);

```

```

133     rebalance = &root->BalancedRoot;
134 } else {
135     PMMADDRESS_NODE successor = child, child2;
136     tmp = child->LeftChild;
137     if (!tmp) {
138         /*
139          * Case 2: node's successor is its right child
140          *
141          *      (n)          (s)
142          *      / \          / \
143          *    (x) (s)  ->  (x) (c)
144          *                \
145          *                (c)
146          */
147         parent = successor;
148         child2 = successor->RightChild;
149     } else {
150         /*
151          * Case 3: node's successor is leftmost under
152          * node's right child subtree
153          *
154          *      (n)          (s)
155          *      / \          / \
156          *    (x) (y)  ->  (x) (y)
157          *      /          /
158          *    (p)          (p)
159          *      /          /
160          *    (s)          (c)
161          *      \
162          *      (c)
163          */
164         do {
165             parent = successor;
166             successor = tmp;
167             tmp = tmp->LeftChild;
168         } while (tmp);
169         parent->LeftChild = child2 = successor->RightChild;
170         successor->RightChild = child;
171         rb_set_parent(child, successor);

```



```

172     }
173
174     successor->LeftChild = tmp = node->LeftChild;
175     rb_set_parent(tmp, successor);
176     pc = node->u1.Parent;
177     tmp = SANITIZE_PARENT_NODE(pc);
178     __rb_change_child(node, successor, tmp, root);
179     if (child2) {
180         successor->u1.Parent = pc;
181         successor->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
182         rb_set_parent_color(child2, parent, rb_black);
183         rebalance = &root->BalancedRoot;
184     } else {
185         PMMADDRESS_NODE pc2 = successor->u1.Parent;
186         successor->u1.Parent = pc;
187         successor->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
188         rebalance = (((ULONG_PTR)(pc2)) & 0x1)==0 ? parent : &
            root->BalancedRoot;
189     }
190     tmp = successor;
191 }
192 return rebalance;
193 }
194
195 static void
196 _____rb_erase_color(PMMADDRESS_NODE parent, PMM_AVL_TABLE root)
197 {
198     PMMADDRESS_NODE node = NULL, sibling, tmp1=NULL, tmp2=NULL;
199
200     while (1) {
201         /*
202          * Loop invariants:
203          * - node is black (or NULL on first iteration)
204          * - node is not the root (parent is not NULL)
205          * - All leaf paths going through parent and node have a
206          *   black node count that is 1 lower than other leaf paths.
207          */
208         sibling = parent->RightChild;
209         if (node != sibling) { /* node == parent->rb_left */

```

```

210     if (rb_is_red(sibling)) {
211         /*
212          * Case 1 – left rotate at parent
213          *
214          *      P              S
215          *     / \            / \
216          *    N  s  —>   p  Sr
217          *           / \   / \
218          *          Sl  Sr N  Sl
219          */
220         parent->RightChild = tmp1 = sibling->LeftChild;
221         sibling->LeftChild = parent;
222         if(tmp1)
223             rb_set_parent_color(tmp1, parent, rb_black);
224         __rb_rotate_set_parents(parent, sibling, root,
225                                 rb_red);
226         sibling = tmp1;
227     }
228     tmp1 = sibling->RightChild;
229     if (!tmp1 || rb_is_black(tmp1)) {
230         tmp2 = sibling->LeftChild;
231         if (!tmp2 || rb_is_black(tmp2)) {
232             /*
233              * Case 2 – sibling color flip
234              * (p could be either color here)
235              *
236              *      (p)          (p)
237              *     / \          / \
238              *    N  S  —>   N  s
239              *           / \   / \
240              *          Sl  Sr Sl  Sr
241              *
242              * This leaves us violating 5) which
243              * can be fixed by flipping p to black
244              * if it was red, or by recursing at p.
245              * p is red when coming from Case 1.
246              */
247             rb_set_parent_color(sibling, parent,
248                                 rb_red);

```

```

249         if (rb_is_red(parent))
250             rb_set_black(parent);
251         else {
252             node = parent;
253             parent = rb_parent(node);
254             if (parent != &root->BalancedRoot)
255                 continue;
256         }
257         break;
258     }
259     /*
260     * Case 3 - right rotate at sibling
261     * (p could be either color here)
262     *
263     *      (p)          (p)
264     *     / \        / \
265     *    N  S  ->  N  Sl
266     *      / \      \
267     *     sl Sr      s
268     *                  \
269     *                   Sr
270     */
271     sibling->LeftChild = tmp1 = tmp2->RightChild;
272     tmp2->RightChild = sibling;
273     parent->RightChild = tmp2;
274     if (tmp1)
275         rb_set_parent_color(tmp1, sibling,
276                             rb_black);
277     tmp1 = sibling;
278     sibling = tmp2;
279 }
280 /*
281 * Case 4 - left rotate at parent + color flips
282 * (p and sl could be either color here.
283 * After rotation, p becomes black, s acquires
284 * p's color, and sl keeps its color)
285 *
286 *      (p)          (s)
287 *     / \        / \

```

```

288 *      N   S      —>   P   Sr
289 *      /  \      /  \
290 *      (sl) sr    N  (sl)
291 */
292 parent->RightChild = tmp2 = sibling->LeftChild;
293 sibling->LeftChild = parent;
294 rb_set_parent_color(tmp1, sibling, rb_black);
295 if (tmp2)
296     rb_set_parent(tmp2, parent);
297 __rb_rotate_set_parents(parent, sibling, root,
298     rb_black);
299 break;
300 } else {
301     sibling = parent->LeftChild;
302     if (rb_is_red(sibling)) {
303         /* Case 1 - right rotate at parent */
304         parent->LeftChild = tmp1 = sibling->RightChild;
305         sibling->RightChild = parent;
306         rb_set_parent_color(tmp1, parent, rb_black);
307         __rb_rotate_set_parents(parent, sibling, root,
308             rb_red);
309         sibling = tmp1;
310     }
311     tmp1 = sibling->LeftChild;
312     if (!tmp1 || rb_is_black(tmp1)) {
313         tmp2 = sibling->RightChild;
314         if (!tmp2 || rb_is_black(tmp2)) {
315             /* Case 2 - sibling color flip */
316             rb_set_parent_color(sibling, parent,
317                 rb_red);
318             if (rb_is_red(parent))
319                 rb_set_black(parent);
320             else {
321                 node = parent;
322                 parent = rb_parent(node);
323                 if (parent != &root->BalancedRoot)
324                     continue;
325             }
326             break;

```

```

327         }
328         /* Case 3 - right rotate at sibling */
329         sibling->RightChild = tmp1 = tmp2->LeftChild;
330         tmp2->LeftChild = sibling;
331         parent->LeftChild = tmp2;
332         if (tmp1)
333             rb_set_parent_color(tmp1, sibling,
334             rb_black);
335         tmp1 = sibling;
336         sibling = tmp2;
337     }
338     /* Case 4 - left rotate at parent + color flips */
339     parent->LeftChild = tmp2 = sibling->RightChild;
340     sibling->RightChild = parent;
341     rb_set_parent_color(tmp1, sibling, rb_black);
342     if (tmp2)
343         rb_set_parent(tmp2, parent);
344     __rb_rotate_set_parents(parent, sibling, root,
345         rb_black);
346     break;
347 }
348 }
349 }
350
351 #if !defined (__USERMODE)
352 #define PRINT
353 #define COUNT_BALANCE_MAX(a)
354 #else
355 extern MM_AVL_TABLE MmSectionBasedRoot;
356 #endif
357
358 #if (_MSC_VER >= 800)
359 #pragma warning(disable:4010)           // Allow pretty pictures without
360                                         the noise
361 #endif
362
363 TABLE_SEARCH_RESULT
364 MiFindNodeOrParent (
365     IN PMM_AVL_TABLE Table,

```

```
365     IN ULONG_PTR StartingVpn ,
366     OUT PMMADDRESS_NODE *NodeOrParent
367 );
368
369 VOID
370 MiPromoteNode (
371     IN PMMADDRESS_NODE C
372 );
373
374 ULONG
375 MiRebalanceNode (
376     IN PMMADDRESS_NODE S
377 );
378
379 PMMADDRESS_NODE
380 MiRealSuccessor (
381     IN PMMADDRESS_NODE Links
382 );
383
384 PMMADDRESS_NODE
385 MiRealPredecessor (
386     IN PMMADDRESS_NODE Links
387 );
388
389 VOID
390 MiInitializeVadTableAvl (
391     IN PMM_AVL_TABLE Table
392 );
393
394 PVOID
395 MiEnumerateGenericTableWithoutSplayingAvl (
396     IN PMM_AVL_TABLE Table ,
397     IN PVOID *RestartKey
398 );
399
400 #ifdef ALLOC_PRAGMA
401 #pragma alloc_text (PAGE, MiCheckForConflictingNode)
402 #pragma alloc_text (PAGE, MiRealSuccessor)
403 #pragma alloc_text (PAGE, MiRealPredecessor)
```

```

404 #pragma alloc_text (PAGE, MiInitializeVadTableAvl)
405 #pragma alloc_text (PAGE, MiFindEmptyAddressRangeInTree)
406 #pragma alloc_text (PAGE, MiFindEmptyAddressRangeDownTree)
407 #pragma alloc_text (PAGE, MiFindEmptyAddressRangeDownBasedTree)
408 #endif
409
410 //
411 // Various Rtl macros that reference Parent use private versions here
    since
412 // Parent is overloaded with Balance.
413 //
414
415 //
416 // The macro function Parent takes as input a pointer to a splay
    link in a
417 // tree and returns a pointer to the splay link of the parent of the
    input
418 // node. If the input node is the root of the tree the return value
    is
419 // equal to the input value.
420 //
421 // PRTL_SPLAY_LINKS
422 // MiParent (
423 //     PRTL_SPLAY_LINKS Links
424 // );
425 //
426
427 #define MiParent(Links) ( \
428     (PRTL_SPLAY_LINKS)(SANITIZE_PARENT_NODE((Links)->u1.Parent)) \
429 )
430
431 //
432 // The macro function IsLeftChild takes as input a pointer to a
    splay link
433 // in a tree and returns TRUE if the input node is the left child of
    its
434 // parent, otherwise it returns FALSE.
435 //
436 // BOOLEAN

```

```

437 //  MiIsLeftChild (
438 //      PRTL_SPLAY_LINKS Links
439 //  );
440 //
441
442 #define MiIsLeftChild(Links) ( \
443     (RtlLeftChild(MiParent(Links)) == (PRTL_SPLAY_LINKS)(Links)) \
444     )
445
446 //
447 //  The macro function IsRightChild takes as input a pointer to a
448 //  splay link
449 //  in a tree and returns TRUE if the input node is the right child
450 //  of its
451 //  parent, otherwise it returns FALSE.
452 //
453 //  BOOLEAN
454 //  MiIsRightChild (
455 //      PRTL_SPLAY_LINKS Links
456 //  );
457 //
458 #define MiIsRightChild(Links) ( \
459     (RtlRightChild(MiParent(Links)) == (PRTL_SPLAY_LINKS)(Links)) \
460     )
461
462
463 #if DBG
464
465 //
466 //  Build a table of the best case efficiency of a balanced binary
467 //  tree,
468 //  holding the most possible nodes that can possibly be held in a
469 //  binary
470 //  tree with a given number of levels. The answer is always (2**n) -
471 //  1.
472 //
473 //  (Used for debug only.)

```



```

471 //
472
473 ULONG MiBestCaseFill[33] = {
474     0,          1,          3,          7,
475     0xf,        0x1f,        0x3f,        0x7f,
476     0xff,        0x1ff,        0x3ff,        0x7ff,
477     0xfff,        0x1fff,        0x3fff,        0x7fff,
478     0xffff,        0x1ffff,        0x3ffff,        0x7ffff,
479     0xfffff,        0x1fffff,        0x3fffff,        0x7fffff,
480     0xfffff,        0x1fffff,        0x3fffff,        0x7fffff,
481     0xfffffff,        0x1fffffff,        0x3fffffff,        0x7fffffff,
482     0xffffffff
483 };
484
485 //
486 // Build a table of the worst case efficiency of a balanced binary
487 // tree,
488 // holding the fewest possible nodes that can possibly be contained
489 // in a
490 // balanced binary tree with the given number of levels. After the
491 // first
492 // two levels, each level n is obviously occupied by a root node,
493 // plus
494 // one subtree the size of level n-1, and another subtree which is
495 // the
496 // size of n-2, i.e.:
497 //
498 //      MiWorstCaseFill[n] = 1 + MiWorstCaseFill[n-1] +
499 //      MiWorstCaseFill[n-2]
500 //
501 // The efficiency of a typical balanced binary tree will normally
502 // fall
503 // between the two extremes, typically closer to the best case. Note
504 // however that even with the worst case, it only takes 32 compares
505 // to
506 // find an element in a worst case tree populated with ~3.5M nodes.
507 //
508 // Unbalanced trees and splay trees, on the other hand, can and will
509 // sometimes

```

```

501 // degenerate to a straight line , requiring on average n/2 compares
      to
502 // find a node.
503 //
504 // A specific case is one where the nodes are inserted in collated
      order.
505 // In this case an unbalanced or a splay tree will generate a
      straight
506 // line , yet the balanced binary tree will always create a perfectly
507 // balanced tree (best-case fill) in this situation.
508 //
509 // (Used for debug only.)
510 //
511
512 ULONG MiWorstCaseFill[33] = {
513     0,          1,          2,          4,
514     7,          12,         20,         33,
515     54,         88,        143,        232,
516     376,        609,        986,       1596,
517     2583,       4180,       6764,      10945,
518     17710,      28656,      46367,     75024,
519     121392,     196417,     317810,    514228,
520     832039,     1346268,    2178308,   3524577,
521     5702886
522 };
523
524 #endif
525
526
527 TABLE_SEARCH_RESULT
528 MiFindNodeOrParent (
529     IN PMM_AVL_TABLE Table ,
530     IN ULONG_PTR StartingVpn ,
531     OUT PMMADDRESS_NODE *NodeOrParent
532 )
533
534 /*++
535
536 Routine Description:

```

```
537
538 This routine is used by all of the routines of the generic
539 table package to locate the a node in the tree. It will
540 find and return (via the NodeOrParent parameter) the node
541 with the given key, or if that node is not in the tree it
542 will return (via the NodeOrParent parameter) a pointer to
543 the parent.
544
545 Arguments:
546
547     Table – The generic table to search for the key.
548
549     StartingVpn – The starting virtual page number.
550
551     NodeOrParent – Will be set to point to the node containing the
552                    the key or what should be the parent of the node
553                    if it were in the tree. Note that this will *NOT*
554                    be set if the search result is TableEmptyTree.
555
556 Return Value:
557
558     TABLE_SEARCH_RESULT – TableEmptyTree: The tree was empty.
559                            NodeOrParent
560
561                                is *not* altered.
562
563                                TableFoundNode: A node with the key is in
564                                the tree.
565
566                                NodeOrParent points to that
567                                node.
568
569                                TableInsertAsLeft: Node with key was not
570                                found.
571
572                                NodeOrParent points to
573                                what would
574                                be parent. The node
575                                would be the
576                                left child.
577
578                                TableInsertAsRight: Node with key was not
```

```

                    found.
570                                     NodeOrParent points to
571                                     what would
572                                     be parent. The node
573                                     would be
574                                     the right child.
575
576 Environment:
577
578     Kernel mode. The PFN lock is held for some of the tables.
579
580 ---*/
581 {
582 #if DBG
583     ULONG NumberCompares = 0;
584 #endif
585     PMMADDRESS_NODE Child;
586     PMMADDRESS_NODE NodeToExamine;
587
588     if (Table->NumberGenericTableElements == 0) {
589         return TableEmptyTree;
590     }
591
592     NodeToExamine = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;
593
594     do {
595
596         //
597         // Make sure the depth of tree is correct.
598         //
599
600         // ASSERT(++NumberCompares <= Table->DepthOfTree);
601
602         //
603         // Compare the buffer with the key in the tree element.
604         //
605
606         if (StartingVpn < NodeToExamine->StartingVpn) {
```

```
606
607     Child = NodeToExamine->LeftChild;
608
609     if (Child != NULL) {
610         NodeToExamine = Child;
611     }
612     else {
613
614         //
615         // Node is not in the tree. Set the output
616         // parameter to point to what would be its
617         // parent and return which child it would be.
618         //
619
620         *NodeOrParent = NodeToExamine;
621         return TableInsertAsLeft;
622     }
623 }
624 else if (StartingVpn <= NodeToExamine->EndingVpn) {
625
626     //
627     // This is the node.
628     //
629
630     *NodeOrParent = NodeToExamine;
631     return TableFoundNode;
632 }
633 else {
634
635     Child = NodeToExamine->RightChild;
636
637     if (Child != NULL) {
638         NodeToExamine = Child;
639     }
640     else {
641
642         //
643         // Node is not in the tree. Set the output
644         // parameter to point to what would be its
```

```
645         // parent and return which child it would be.
646         //
647
648         *NodeOrParent = NodeToExamine;
649         return TableInsertAsRight;
650     }
651 }
652
653 } while (TRUE);
654 }
655
656
657 PMMADDRESS_NODE
658 MiCheckForConflictingNode (
659     IN ULONG_PTR StartVpn,
660     IN ULONG_PTR EndVpn,
661     IN PMM_AVL_TABLE Table
662 )
663
664 /*++
665
666 Routine Description:
667
668     The function determines if any addresses between a given starting
669     and
670     ending address is contained within a virtual address descriptor.
671
672 Arguments:
673
674     StartVpn - Supplies the virtual address to locate a containing
675                descriptor.
676
677     EndVpn - Supplies the virtual address to locate a containing
678               descriptor.
679
680 Return Value:
681
682     Returns a pointer to the first conflicting virtual address
683     descriptor
```

```
682     if one is found, otherwise a NULL value is returned.
683
684 ---*/
685
686 {
687     PMMADDRESS_NODE Node;
688
689     if (Table->NumberGenericTableElements == 0) {
690         return NULL;
691     }
692
693     Node = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;
694     ASSERT (Node != NULL);
695
696     do {
697
698         if (Node == NULL) {
699             return NULL;
700         }
701
702         if (StartVpn > Node->EndingVpn) {
703             Node = Node->RightChild;
704         }
705         else if (EndVpn < Node->StartingVpn) {
706             Node = Node->LeftChild;
707         }
708         else {
709
710             //
711             // The starting address is less than or equal to the end
712             // VA
713             // and the ending address is greater than or equal to the
714             // start va. Return this node.
715             //
716
717             return Node;
718         }
719     } while (TRUE);
```

```
720 }
721
722
723 PMMADDRESS_NODE
724 FASTCALL
725 MiGetFirstNode (
726     IN PMM_AVL_TABLE Table
727 )
728
729 /*++
730
731 Routine Description:
732
733     This function locates the virtual address descriptor which
734     contains
735     the address range which logically is first within the address
736     space.
737
738 Arguments:
739
740     None.
741
742 Return Value:
743
744     Returns a pointer to the virtual address descriptor containing
745     the
746     first address range, NULL if none.
747
748 --*/
749
750 {
751     PMMADDRESS_NODE First;
752
753     if (Table->NumberGenericTableElements == 0) {
754         return NULL;
755     }
756
757     First = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;
```



```
756     ASSERT (First != NULL);
757
758     while (First->LeftChild != NULL) {
759         First = First->LeftChild;
760     }
761
762     return First;
763 }
764
765
766 VOID
767 MiPromoteNode (
768     IN PMMADDRESS_NODE C
769 )
770
771 /*++
772
773 Routine Description:
774
775     This routine performs the fundamental adjustment required for
776     balancing
777     the binary tree during insert and delete operations.  Simply put,
778     the
779     designated node is promoted in such a way that it rises one level
780     in
781     the tree and its parent drops one level in the tree, becoming now
782     the
783     child of the designated node.  Generally the path length to the
784     subtree
785     "opposite" the original parent.  Balancing occurs as the caller
786     chooses
787     which nodes to promote according to the balanced tree algorithms
788     from
789     Knuth.
790
791     This is not the same as a splay operation, typically a splay "
792     promotes"
793     a designated node twice.
```

```
787     Note that the pointer to the root node of the tree is assumed to
788         be
789     contained in a MMADDRESS_NODE structure itself, to allow the
790     algorithms below to change the root of the tree without checking
791     for special cases. Note also that this is an internal routine,
792     and the caller guarantees that it never requests to promote the
793     root itself.
794
795     This routine only updates the tree links; the caller must update
796     the balance factors as appropriate.
797
798 Arguments:
799
800     C - pointer to the child node to be promoted in the tree.
801
802 Return Value:
803
804     None.
805
806 ---*/
807 {
808     PMMADDRESS_NODE P;
809     PMMADDRESS_NODE G;
810
811     //
812     // Capture the current parent and grandparent (may be the root).
813     //
814
815     P = SANITIZE_PARENT_NODE (C->u1.Parent);
816     G = SANITIZE_PARENT_NODE (P->u1.Parent);
817
818     //
819     // Break down the promotion into two cases based upon whether C
820     // is a left or right child.
821     //
822
823     if (P->LeftChild == C) {
824
```

```

825 //
826 // This promotion looks like this:
827 //
828 //          G          G
829 //          |          |
830 //          P          C
831 //        / \      =>  / \
832 //       C   z      x   P
833 //      / \      / \
834 //     x   y     y   z
835 //
836
837 P->LeftChild = C->RightChild;
838
839 if (P->LeftChild != NULL) {
840
841     P->LeftChild->u1.Parent = MI_MAKE_PARENT (P, P->LeftChild
842         ->u1.Balance);
843
844     C->RightChild = P;
845
846 //
847 // Fall through to update parent and G <-> C relationship in
848 // common code.
849 //
850
851 }
852 else {
853
854     ASSERT(P->RightChild == C);
855
856 //
857 // This promotion looks like this:
858 //
859 //          G          G
860 //          |          |
861 //          P          C
862 //        / \      =>  / \

```

```

863      //      x   C           P   z
864      //      / \         / \
865      //      y   z         x   y
866      //
867
868      P->RightChild = C->LeftChild;
869
870      if (P->RightChild != NULL) {
871          P->RightChild->u1.Parent = MI_MAKE_PARENT (P, P->
872              RightChild->u1.Balance);
873      }
874
875      C->LeftChild = P;
876  }
877
878  //
879  // Update parent of P, for either case above.
880  //
881
882      P->u1.Parent = MI_MAKE_PARENT (C, P->u1.Balance);
883
884  //
885  // Finally update G <-> C links for either case above.
886  //
887
888      if (G->LeftChild == P) {
889          G->LeftChild = C;
890      }
891      else {
892          ASSERT(G->RightChild == P);
893          G->RightChild = C;
894      }
895      C->u1.Parent = MI_MAKE_PARENT (G, C->u1.Balance);
896  }
897
898  ULONG
899  MiRebalanceNode (
900      IN PMMADDRESS_NODE S

```

```

901     )
902
903     /*++
904
905 Routine Description:
906
907     This routine performs a rebalance around the input node S, for
908     which the
909     Balance factor has just effectively become +2 or -2. When called
910     , the
911     Balance factor still has a value of +1 or -1, but the respective
912     longer
913     side has just become one longer as the result of an insert or
914     delete
915     operation.
916
917     This routine effectively implements steps A7.iii (test for Case 1
918     or
919     Case 2) and steps A8 and A9 of Knuth's balanced insertion
920     algorithm,
921     plus it handles Case 3 identified in the delete section, which
922     can
923     only happen on deletes.
924
925     The trick is, to convince yourself that while traveling from the
926     insertion point at the bottom of the tree up, that there are only
927     these two cases, and that when traveling up from the deletion
928     point,
929     that there are just these three cases. Knuth says it is obvious!
930
931 Arguments:
932
933     S - pointer to the node which has just become unbalanced.
934
935 Return Value:
936
937     TRUE if Case 3 was detected (causes delete algorithm to terminate
938     ).
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

```
931 Environment:
932
933     Kernel mode.  The PFN lock is held for some of the tables.
934
935 ---*/
936
937 {
938     PMMADDRESS_NODE R, P;
939     SCHAR a;
940
941     PRINT("rebalancing_node%p_bal=%x_start=%x_end=%x\n",
942          S,
943          S->u1.Balance,
944          S->StartingVpn,
945          S->EndingVpn);
946
947     //
948     // The parent node is never the argument node.
949     //
950
951     ASSERT (SANITIZE_PARENT_NODE(S->u1.Parent) != S);
952
953     //
954     // Capture which side is unbalanced.
955     //
956
957     a = (SCHAR) S->u1.Balance;
958
959     if (a == +1) {
960         R = S->RightChild;
961     }
962     else {
963         R = S->LeftChild;
964     }
965
966     //
967     // If the balance of R and S are the same (Case 1 in Knuth) then
968     // a single
969     // promotion of R will do the single rotation.  (Step A8, A10)
```

```

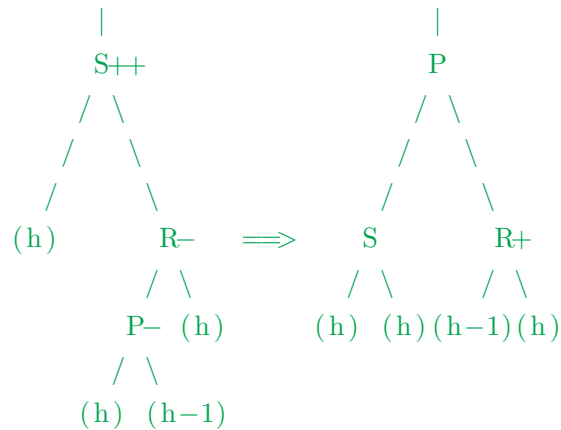
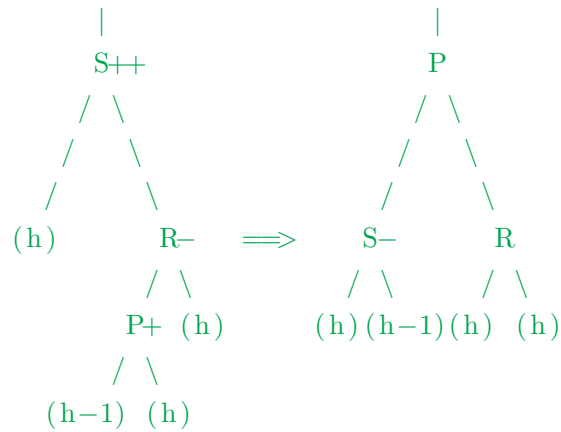
969 //
970 // Here is a diagram of the Case 1 transformation, for a == +1 (a
    mirror
971 // image transformation occurs when a == -1), and where the
    subtree
972 // heights are h and h+1 as shown (++ indicates the node out of
    balance):
973 //
974 //
975 //           |                |
976 //         S++              R
977 //        / \             / \
978 //      (h) R+  ==>    S  (h+1)
979 //           / \             / \
980 //         (h) (h+1)       (h) (h)
981 //
982 // Note that on an insert we can hit this case by inserting an
    item in the
983 // right subtree of R. The original height of the subtree before
    the insert
984 // was h+2, and it is still h+2 after the rebalance, so insert
    rebalancing
985 // may terminate.
986 //
987 // On a delete we can hit this case by deleting a node from the
    left subtree
988 // of S. The height of the subtree before the delete was h+3,
    and after the
989 // rebalance it is h+2, so rebalancing must continue up the tree.
990 //
991 if ((SCHAR) R->u1.Balance == a) {
992
993     MiPromoteNode (R);
994     R->u1.Balance = 0;
995     S->u1.Balance = 0;
996
997     return FALSE;
998 }
999

```

```

1000 //
1001 // Otherwise, we have to promote the appropriate child of R twice
      (Case 2
1002 // in Knuth). (Step A9, A10)
1003 //
1004 // Here is a diagram of the Case 2 transformation, for a == +1 (a
      mirror
1005 // image transformation occurs when a == -1), and where the
      subtree
1006 // heights are h and h-1 as shown. There are actually two minor
      subcases,
1007 // differing only in the original balance of P (++ indicates the
      node out
1008 // of balance).

```



```

1032 //
1033 // Note that on an insert we can hit this case by inserting an

```



```

    item in the
1034 // left subtree of R. The original height of the subtree before
    the insert
1035 // was h+2, and it is still h+2 after the rebalance, so insert
    rebalancing
1036 // may terminate.
1037 //
1038 // On a delete we can hit this case by deleting a node from the
    left subtree
1039 // of S. The height of the subtree before the delete was h+3,
    and after the
1040 // rebalance it is h+2, so rebalancing must continue up the tree.
1041 //
1042
1043 if ((SCHAR) R->u1.Balance == -a) {
1044
1045     //
1046     // Pick up the appropriate child P for the double rotation (
        Link(-a,R)).
1047     //
1048
1049     if (a == 1) {
1050         P = R->LeftChild;
1051     }
1052     else {
1053         P = R->RightChild;
1054     }
1055
1056     //
1057     // Promote him twice to implement the double rotation.
1058     //
1059
1060     MiPromoteNode (P);
1061     MiPromoteNode (P);
1062
1063     //
1064     // Now adjust the balance factors.
1065     //
1066
```

```

1067     S->u1.Balance = 0;
1068     R->u1.Balance = 0;
1069     if ((SCHAR) P->u1.Balance == a) {
1070         PRINT("REBADJ_A: _Node%p, _Bal_%x->_%x\n", S, S->u1.
            Balance, -a);
1071         COUNT_BALANCE_MAX ((SCHAR)-a);
1072         S->u1.Balance = (ULONG_PTR) -a;
1073     }
1074     else if ((SCHAR) P->u1.Balance == -a) {
1075         PRINT("REBADJ_B: _Node%p, _Bal_%x->_%x\n", R, R->u1.
            Balance, a);
1076         COUNT_BALANCE_MAX ((SCHAR)a);
1077         R->u1.Balance = (ULONG_PTR) a;
1078     }
1079
1080     P->u1.Balance = 0;
1081     return FALSE;
1082 }
1083
1084 //
1085 // Otherwise this is Case 3 which can only happen on Delete (
    identical
1086 // to Case 1 except R->u1.Balance == 0). We do a single rotation
    , adjust
1087 // the balance factors appropriately, and return TRUE. Note that
    the
1088 // balance of S stays the same.
1089 //
1090 // Here is a diagram of the Case 3 transformation, for a == +1 (a
    mirror
1091 // image transformation occurs when a == -1), and where the
    subtree
1092 // heights are h and h+1 as shown (++ indicates the node out of
    balance):
1093 //
1094 //           |
1095 //         S++
1096 //        / \
1097 //       (h) R    ==>           |
                                R-
                               / \
                              S+ (h+1)

```

```

1098 //          / \          / \
1099 //          (h+1)(h+1)      (h) (h+1)
1100 //
1101 // This case can not occur on an insert, because it is impossible
1102 // for
1103 // a single insert to balance R, yet somehow grow the right
1104 // subtree of
1105 // S at the same time. As we move up the tree adjusting balance
1106 // factors
1107 // after an insert, we terminate the algorithm if a node becomes
1108 // balanced,
1109 // because that means the subtree length did not change!
1110 //
1111 // On a delete we can hit this case by deleting a node from the
1112 // left
1113 // subtree of S. The height of the subtree before the delete was
1114 // h+3,
1115 // and after the rebalance it is still h+3, so rebalancing may
1116 // terminate
1117 // in the delete path.
1118 //
1119
1120 MiPromoteNode (R);
1121 PRINT( "REBADJ_C: Node%p, Bal %x->%x\n", R, R->u1.Balance, -a);
1122 COUNT_BALANCE_MAX ((SCHAR)-a);
1123 R->u1.Balance = -a;
1124 return TRUE;
1125 }
1126
1127 VOID
1128 FASTCALL
1129 MiRemoveNode (
1130     IN PMMADDRESS_NODE NodeToDelete,
1131     IN PMM_AVL_TABLE Table
1132 )
1133
1134 /*++

```

### 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 rebalance;

Table->NumberGenericTableElements -= 1;

rebalance = \_\_rb\_erase\_augmented(NodeToDelete, Table);

if (rebalance != &Table->BalancedRoot && rebalance)

\_\_\_\_rb\_erase\_color(rebalance, Table);

```

1162 }
1163
1164
1165 PMMADDRESS_NODE
1166 MiRealSuccessor (
1167     IN PMMADDRESS_NODE Links
1168 )
1169
1170 /*++
1171
1172 Routine Description:
1173
1174     This function takes as input a pointer to a balanced link
1175     in a tree and returns a pointer to the successor of the input
1176     node within
1177     the entire tree.  If there is not a successor, the return value
1178     is NULL.
1179
1180 Arguments:
1181
1182     Links - Supplies a pointer to a balanced link in a tree.
1183
1184 Return Value:
1185
1186     PMMADDRESS_NODE - returns a pointer to the successor in the
1187     entire tree
1188
1189 ---*/
1190
1191 {
1192     PMMADDRESS_NODE Ptr;
1193
1194     /*
1195         First check to see if there is a right subtree to the input
1196         link
1197         if there is then the real successor is the left most node in
1198         the right subtree.  That is find and return S in the
1199         following diagram
1200

```

```

1196         Links
1197         \
1198         .
1199         .
1200         .
1201         /
1202         S
1203         \
1204     */
1205
1206     if ((Ptr = Links->RightChild) != NULL) {
1207
1208         while (Ptr->LeftChild != NULL) {
1209             Ptr = Ptr->LeftChild;
1210         }
1211
1212         return Ptr;
1213     }
1214
1215     /*
1216     We do not have a right child so check to see if have a parent
1217     and if
1218     so find the first ancestor that we are a left descendant of.
1219     That
1220     is find and return S in the following diagram
1221
1222         S
1223         /
1224         .
1225         .
1226         .
1227         Links
1228
1229     Note that this code depends on how the BalancedRoot is
1230     initialized ,
1231     which is Parent points to self , and the RightChild points to
1232     an
1233     actual node which is the root of the tree , and LeftChild does
1234     not

```

```

1230         point to self.
1231     */
1232
1233     Ptr = Links;
1234     while (MiIsRightChild(Ptr)) {
1235         Ptr = SANITIZE_PARENT_NODE (Ptr->u1.Parent);
1236     }
1237
1238     if (MiIsLeftChild(Ptr)) {
1239         return SANITIZE_PARENT_NODE (Ptr->u1.Parent);
1240     }
1241
1242     //
1243     // Otherwise we are do not have a real successor so we simply
1244     return NULL.
1245
1246     //
1247     // This can only occur when we get back to the root, and we can
1248     tell
1249     // that since the Root is its own parent.
1250     //
1251
1252     ASSERT (SANITIZE_PARENT_NODE(Ptr->u1.Parent) == Ptr);
1253
1254     return NULL;
1255 }
1256
1257 PMMADDRESS_NODE
1258 MiRealPredecessor (
1259     IN PMMADDRESS_NODE Links
1260 )
1261
1262 /*++
1263
1264 Routine Description:
1265
1266     The RealPredecessor function takes as input a pointer to a
1267     balanced link
1268     in a tree and returns a pointer to the predecessor of the input

```

```

1266     node
1267     within the entire tree. If there is not a predecessor, the
1268     return value
1269     is NULL.
1270
1271 Arguments:
1272
1273 Links - Supplies a pointer to a balanced link in a tree.
1274
1275 Return Value:
1276
1277 PMMADDRESS_NODE - returns a pointer to the predecessor in the
1278     entire tree
1279
1280 ---*/
1281 {
1282     PMMADDRESS_NODE Ptr;
1283     PMMADDRESS_NODE Parent;
1284     PMMADDRESS_NODE GrandParent;
1285
1286     /*
1287     First check to see if there is a left subtree to the input link
1288     if there is then the real predecessor is the right most node in
1289     the left subtree. That is find and return P in the following
1290     diagram
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300

```

```

Links
 /
.
.
.
P
 /

```

```

*/

if ((Ptr = Links->LeftChild) != NULL) {

    while (Ptr->RightChild != NULL) {

```



```

1301         Ptr = Ptr->RightChild;
1302     }
1303
1304     return Ptr;
1305
1306 }
1307
1308 /*
1309     We do not have a left child so check to see if have a parent
1310     and if
1311     so find the first ancestor that we are a right descendant of.
1312     That
1313     is find and return P in the following diagram
1314
1315         P
1316         \
1317         .
1318         .
1319         .
1320         Links
1321
1322     Note that this code depends on how the BalancedRoot is
1323     initialized ,
1324     which is Parent points to self , and the RightChild points to
1325     an
1326     actual node which is the root of the tree .
1327 */
1328
1329 Ptr = Links;
1330 while (MiIsLeftChild(Ptr)) {
1331     Ptr = SANITIZE_PARENT_NODE (Ptr->u1.Parent);
1332 }
1333
1334 if (MiIsRightChild(Ptr)) {
1335     Parent = SANITIZE_PARENT_NODE (Ptr->u1.Parent);
1336     GrandParent = SANITIZE_PARENT_NODE (Parent->u1.Parent);
1337     if (GrandParent != Parent) {
1338         return Parent;
1339     }
1340 }

```

```
1336     }
1337
1338     //
1339     // Otherwise we are do not have a real predecessor so we simply
1340     // return
1341     // NULL.
1342     //
1343     return NULL;
1344 }
1345
1346
1347 VOID
1348 MiInitializeVadTableAvl (
1349     IN PMM_AVL_TABLE Table
1350 )
1351
1352 /*++
1353
1354 Routine Description:
1355
1356     This routine initializes a table.
1357
1358 Arguments:
1359
1360     Table - Pointer to the generic table to be initialized.
1361
1362 Return Value:
1363
1364     None.
1365
1366 --*/
1367 {
1368
1369 #if DBG
1370     ULONG i;
1371
1372     for (i = 2; i < 33; i += 1) {
```

```

1374     ASSERT(MiWorstCaseFill[i] == (1 + MiWorstCaseFill[i - 1] +
1375         MiWorstCaseFill[i - 2]));
1376 }
1377 #endif
1378
1379 //
1380 // Initialize each field in the argument Table.
1381 //
1382 RtlZeroMemory (Table, sizeof(MM_AVL_TABLE));
1383
1384 Table->BalancedRoot.u1.Parent = MI_MAKE_PARENT (&Table->
1385     BalancedRoot, 0);
1386 }
1387
1388 VOID
1389 FASTCALL
1390 MiInsertNode (
1391     IN PMMADDRESS_NODE NodeToInsert,
1392     IN PMM_AVL_TABLE Table
1393 )
1394
1395 /*++
1396
1397 Routine Description:
1398
1399     This function inserts a new element in a table.
1400
1401 Arguments:
1402
1403     NodeToInsert - The initialized address node to insert.
1404
1405     Table - Pointer to the table in which to insert the new node.
1406
1407 Return Value:
1408
1409     None.
1410

```

```
1411 Environment:
1412
1413     Kernel mode.  The PFN lock is held for some of the tables.
1414
1415 ---*/
1416
1417 {
1418     PMMADDRESS_NODE NodeOrParent;
1419     PMMADDRESS_NODE parent, gparent, tmp;
1420     TABLE_SEARCH_RESULT SearchResult;
1421
1422     SearchResult = MiFindNodeOrParent (Table,
1423         NodeToInsert->StartingVpn,
1424         &NodeOrParent);
1425
1426     NodeToInsert->LeftChild = NULL;
1427     NodeToInsert->RightChild = NULL;
1428
1429     Table->NumberGenericTableElements += 1;
1430
1431     //
1432     // Insert the newer node in the tree.
1433     //
1434
1435     if (SearchResult == TableEmptyTree)
1436     {
1437         Table->BalancedRoot.RightChild = NodeToInsert;
1438         rb_set_parent(NodeToInsert, &Table->BalancedRoot);
1439     }
1440     else
1441     {
1442         if (SearchResult == TableInsertAsLeft)
1443         {
1444             NodeOrParent->LeftChild = NodeToInsert;
1445         }
1446         else
1447         {
1448             NodeOrParent->RightChild = NodeToInsert;
1449         }
1450     }
1451 }
```

```

1450     rb_set_parent(NodeToInsert, NodeOrParent);
1451 }
1452 rb_set_red(NodeToInsert);
1453 parent = rb_parent(NodeToInsert);
1454 while(1) {
1455     if (parent == &Table->BalancedRoot) {
1456         Table->BalancedRoot.RightChild = NodeToInsert;
1457         rb_set_parent_color(NodeToInsert, &Table->BalancedRoot,
1458                             rb_black);
1458         break;
1459     } else if (rb_is_black(parent))
1460         break;
1461     gparent = rb_parent(parent);
1462     tmp = gparent->RightChild;
1463
1464     if (parent != tmp) { /* parent == gparent->rb_left */
1465         if (tmp && rb_is_red(tmp)) {
1466             /*
1467              * Case 1 - color flips
1468              *
1469              *      G      g
1470              *     / \    / \
1471              *    p  u  -> P  U
1472              *   /      /
1473              *  n      n
1474              *
1475              * However, since g's parent might be red, and
1476              * 4) does not allow this, we need to recurse
1477              * at g.
1478              */
1479             rb_set_parent_color(tmp, gparent, rb_black);
1480             rb_set_parent_color(parent, gparent, rb_black);
1481             NodeToInsert = gparent;
1482             parent = rb_parent(NodeToInsert);
1483             rb_set_parent_color(NodeToInsert, parent, rb_red);
1484             continue;
1485         }
1486
1487         tmp = parent->RightChild;

```

```

1488     if (NodeToInsert == tmp) {
1489         /*
1490          * Case 2 - left rotate at parent
1491          *
1492          *      G          G
1493          *     / \        / \
1494          *    p  U  ->  n  U
1495          *     \        /
1496          *      n      p
1497          *
1498          * This still leaves us in violation of 4), the
1499          * continuation into Case 3 will fix that.
1500          */
1501         parent->RightChild = tmp = NodeToInsert->LeftChild;
1502         NodeToInsert->LeftChild = parent;
1503         if (tmp)
1504             rb_set_parent_color(tmp, parent, rb_black);
1505         rb_set_parent_color(parent, NodeToInsert, rb_red);
1506         parent = NodeToInsert;
1507         tmp = NodeToInsert->RightChild;
1508     }
1509
1510     /*
1511     * Case 3 - right rotate at gparent
1512     *
1513     *      G          P
1514     *     / \        / \
1515     *    p  U  ->  n  g
1516     *     /        \
1517     *    n          U
1518     */
1519     gparent->LeftChild = tmp; /* == parent->rb_right */
1520     parent->RightChild = gparent;
1521     if (tmp)
1522         rb_set_parent_color(tmp, gparent, rb_black);
1523     __rb_rotate_set_parents(gparent, parent, Table, rb_red);
1524     break;
1525 } else {
1526     tmp = gparent->LeftChild;

```

```

1527         if (tmp && rb_is_red(tmp)) {
1528             /* Case 1 - color flips */
1529             rb_set_parent_color(tmp, gparent, rb_black);
1530             rb_set_parent_color(parent, gparent, rb_black);
1531             NodeToInsert = gparent;
1532             parent = rb_parent(NodeToInsert);
1533             rb_set_parent_color(NodeToInsert, parent, rb_red);
1534             continue;
1535         }
1536
1537         tmp = parent->LeftChild;
1538         if (NodeToInsert == tmp) {
1539             /* Case 2 - right rotate at parent */
1540             parent->LeftChild = tmp = NodeToInsert->RightChild;
1541             NodeToInsert->RightChild = parent;
1542             if (tmp)
1543                 rb_set_parent_color(tmp, parent,
1544                                     rb_black);
1545             rb_set_parent_color(parent, NodeToInsert, rb_red);
1546             parent = NodeToInsert;
1547             tmp = NodeToInsert->LeftChild;
1548         }
1549
1550         /* Case 3 - left rotate at gparent */
1551         gparent->RightChild = tmp; /* == parent->rb_left */
1552         parent->LeftChild = gparent;
1553         if (tmp)
1554             rb_set_parent_color(tmp, gparent, rb_black);
1555         __rb_rotate_set_parents(gparent, parent, Table, rb_red);
1556         break;
1557     }
1558 }
1559 return;
1560 }
1561
1562
1563 PVOID
1564 MiEnumerateGenericTableWithoutSplayingAvl (
1565     IN PMM_AVL_TABLE Table,

```

```

1566     IN PVOID *RestartKey
1567 )

```

```

1568
1569 /*++

```

```

1570

```

```

1571 Routine Description:

```

```

1572

```

```

1573     The function EnumerateGenericTableWithoutSplayingAvl will return
1574     to the caller one-by-one the elements of of a table. The return value
1575     is a pointer to the user defined structure associated with the element
1576     .

```

```

1576     The input parameter RestartKey indicates if the enumeration
1577     should start from the beginning or should return the next element. If
1578     the are no more new elements to return the return value is NULL. As
1579     an example of its use, to enumerate all of the elements in a table
1580     the user would write:

```

```

1581
1582     *RestartKey = NULL;
1583
1584     for (ptr = EnumerateGenericTableWithoutSplayingAvl(Table, &
1585     RestartKey);
1586     ptr != NULL;
1587     ptr = EnumerateGenericTableWithoutSplayingAvl(Table, &
1588     RestartKey)) {
1589     :
1590     }

```

```

1590 Arguments:

```

```

1591

```

```

1592     Table – Pointer to the generic table to enumerate.

```

```

1593

```

```

1594     RestartKey – Pointer that indicates if we should restart or
1595     return the next

```



```
1595         element.  If the contents of RestartKey is NULL, the
1596             search
1597             will be started from the beginning.
1598 Return Value:
1599
1600     PVOID – Pointer to the user data.
1601
1602 ---*/
1603
1604 {
1605     PMMADDRESS_NODE NodeToReturn;
1606
1607     if (Table->NumberGenericTableElements == 0) {
1608
1609         //
1610         // Nothing to do if the table is empty.
1611         //
1612
1613         return NULL;
1614
1615     }
1616
1617     //
1618     // If the restart flag is true then go to the least element
1619     // in the tree.
1620     //
1621
1622     if (*RestartKey == NULL) {
1623
1624         //
1625         // Loop until we find the leftmost child of the root.
1626         //
1627
1628         for (NodeToReturn = Table->BalancedRoot.RightChild;
1629             NodeToReturn->LeftChild;
1630             NodeToReturn = NodeToReturn->LeftChild) {
1631
1632             NOTHING;
```

```
1633     }
1634
1635     *RestartKey = NodeToReturn;
1636
1637 }
1638 else {
1639
1640     //
1641     // The caller has passed in the previous entry found
1642     // in the table to enable us to continue the search. We call
1643     // RealSuccessor to step to the next element in the tree.
1644     //
1645
1646     NodeToReturn = MiRealSuccessor (*RestartKey);
1647
1648     if (NodeToReturn) {
1649         *RestartKey = NodeToReturn;
1650     }
1651 }
1652
1653 //
1654 // Return the found element.
1655 //
1656
1657 return NodeToReturn;
1658 }
1659
1660
1661 PMMADDRESS_NODE
1662 FASTCALL
1663 MiGetNextNode (
1664     IN PMMADDRESS_NODE Node
1665 )
1666
1667 /*++
1668
1669 Routine Description:
1670
1671     This function locates the virtual address descriptor which
```

```
contains
the address range which logically follows the specified address
range.

Arguments:

Node – Supplies a pointer to a virtual address descriptor.

Return Value:

Returns a pointer to the virtual address descriptor containing
the
next address range, NULL if none.

---*/

{
    PMMADDRESS_NODE Next;
    PMMADDRESS_NODE Parent;
    PMMADDRESS_NODE Left;

    Next = Node;

    if (Next->RightChild == NULL) {

        do {

            Parent = SANITIZE_PARENT_NODE (Next->u1.Parent);

            ASSERT (Parent != NULL);

            if (Parent == Next) {
                return NULL;
            }

            //
            // Locate the first ancestor of this node of which this
            // node is the left child of and return that node as the
            // next element.
```

```
1708         //
1709
1710         if (Parent->LeftChild == Next) {
1711             return Parent;
1712         }
1713
1714         Next = Parent;
1715
1716     } while (TRUE);
1717 }
1718
1719 //
1720 // A right child exists, locate the left most child of that right
1721 // child.
1722 //
1723 Next = Next->RightChild;
1724
1725 do {
1726
1727     Left = Next->LeftChild;
1728
1729     if (Left == NULL) {
1730         break;
1731     }
1732
1733     Next = Left;
1734
1735 } while (TRUE);
1736
1737 return Next;
1738
1739 }
1740
1741 PMMADDRESS_NODE
1742 FASTCALL
1743 MiGetPreviousNode (
1744     IN PMMADDRESS_NODE Node
1745 )
```

```
1746
1747 /*++
1748
1749 Routine Description:
1750
1751     This function locates the virtual address descriptor which
1752     contains
1753     the address range which logically precedes the specified virtual
1754     address descriptor.
1755
1756 Arguments:
1757
1758     Node – Supplies a pointer to a virtual address descriptor.
1759
1760 Return Value:
1761
1762     Returns a pointer to the virtual address descriptor containing
1763     the
1764     next address range, NULL if none.
1765
1766 —*/
1767 {
1768     PMMADDRESS_NODE Previous;
1769     PMMADDRESS_NODE Parent;
1770
1771     Previous = Node;
1772
1773     if (Previous->LeftChild == NULL) {
1774
1775         ASSERT (Previous->u1.Parent != NULL);
1776
1777         Parent = SANITIZE_PARENT_NODE (Previous->u1.Parent);
1778
1779         while (Parent != Previous) {
1780
1781             //
1782             // Locate the first ancestor of this node of which this
1783             // node is the right child of and return that node as the
```

```
1783         // Previous element.
1784         //
1785
1786         if (Parent->RightChild == Previous) {
1787
1788             if (Parent == SANITIZE_PARENT_NODE (Parent->u1.Parent
1789             )) {
1790                 return NULL;
1791             }
1792             return Parent;
1793         }
1794
1795         Previous = Parent;
1796         Parent = SANITIZE_PARENT_NODE (Previous->u1.Parent);
1797     }
1798     return NULL;
1799 }
1800
1801 //
1802 // A left child exists, locate the right most child of that left
1803 // child.
1804 //
1805 Previous = Previous->LeftChild;
1806
1807 while (Previous->RightChild != NULL) {
1808     Previous = Previous->RightChild;
1809 }
1810
1811 return Previous;
1812 }
1813
1814
1815 PMMADDRESS_NODE
1816 FASTCALL
1817 MiLocateAddressInTree (
1818     IN ULONG_PTR Vpn,
1819     IN PMM_AVL_TABLE Table
```

```
1820     )
1821
1822     /*++
1823
1824     Routine Description:
1825
1826         The function locates the virtual address descriptor which
1827         describes
1828         a given address.
1829
1830     Arguments:
1831
1832         Vpn - Supplies the virtual page number to locate a descriptor for
1833         .
1834
1835     Return Value:
1836
1837         Returns a pointer to the virtual address descriptor which
1838         contains
1839         the supplied virtual address or NULL if none was located.
1840
1841     --*/
1842
1843     {
1844         PVOID NodeOrParent;
1845         TABLE_SEARCH_RESULT SearchResult;
1846
1847         //
1848         // Lookup the element and save the result.
1849         //
1850
1851         SearchResult = MiFindNodeOrParent (Table,
1852                                           Vpn,
1853                                           (PMMADDRESS_NODE *) &
1854                                           NodeOrParent);
1855
1856         if (SearchResult == TableFoundNode) {
1857
1858             //
```

```
1855     // Return the VAD.
1856     //
1857
1858     return (PMMADDRESS_NODE) NodeOrParent;
1859 }
1860
1861 return NULL;
1862 }
1863
1864
1865 NTSTATUS
1866 MiFindEmptyAddressRangeInTree (
1867     IN SIZE_T SizeOfRange ,
1868     IN ULONG_PTR Alignment ,
1869     IN PMM_AVL_TABLE Table ,
1870     OUT PMMADDRESS_NODE *PreviousVad ,
1871     OUT PVOID *Base
1872 )
1873
1874 /*++
1875
1876 Routine Description:
1877
1878     The function examines the virtual address descriptors to locate
1879     an unused range of the specified size and returns the starting
1880     address of the range.
1881
1882 Arguments:
1883
1884     SizeOfRange - Supplies the size in bytes of the range to locate.
1885
1886     Alignment - Supplies the alignment for the address. Must be
1887                 a power of 2 and greater than the page_size.
1888
1889     Table - Supplies the root of the tree to search through.
1890
1891     PreviousVad - Supplies the Vad which is before this the found
1892                   address range.
1893
```



```
1894     Base – Receives the starting address of a suitable range on
1895         success.
1896 Return Value:
1897
1898     NTSTATUS.
1899
1900 ---*/
1901
1902 {
1903     PMMADDRESS_NODE Node;
1904     PMMADDRESS_NODE NextNode;
1905     ULONG_PTR AlignmentVpn;
1906     ULONG_PTR SizeOfRangeVpn;
1907
1908     AlignmentVpn = Alignment >> PAGE_SHIFT;
1909
1910     //
1911     // Locate the node with the lowest starting address.
1912     //
1913
1914     ASSERT (SizeOfRange != 0);
1915     SizeOfRangeVpn = (SizeOfRange + (PAGE_SIZE - 1)) >> PAGE_SHIFT;
1916     ASSERT (SizeOfRangeVpn != 0);
1917
1918     if (Table->NumberGenericTableElements == 0) {
1919         *Base = MM_LOWEST_USER_ADDRESS;
1920         return STATUS_SUCCESS;
1921     }
1922
1923     Node = Table->BalancedRoot.RightChild;
1924
1925     while (Node->LeftChild != NULL) {
1926         Node = Node->LeftChild;
1927     }
1928
1929     //
1930     // Check to see if a range exists between the lowest address VAD
1931     // and lowest user address.
```

```

1932 //
1933
1934 if (Node->StartingVpn > MI_VA_TO_VPN (MM_LOWEST_USER_ADDRESS)) {
1935
1936     if (SizeOfRangeVpn <
1937         (Node->StartingVpn - MI_VA_TO_VPN (MM_LOWEST_USER_ADDRESS
1938             ))) {
1939
1940         *PreviousVad = NULL;
1941         *Base = MM_LOWEST_USER_ADDRESS;
1942         return STATUS_SUCCESS;
1943     }
1944 }
1945 do {
1946
1947     NextNode = MiGetNextNode (Node);
1948
1949     if (NextNode != NULL) {
1950
1951         if (SizeOfRangeVpn <=
1952             ((ULONG_PTR)NextNode->StartingVpn -
1953              MI_ROUND_TO_SIZE(1 + Node->EndingVpn,
1954                  AlignmentVpn))) {
1955
1956             //
1957             // Check to ensure that the ending address aligned
1958             // upwards
1959             // is not greater than the starting address.
1960             //
1961
1962             if ((ULONG_PTR)NextNode->StartingVpn >
1963                 MI_ROUND_TO_SIZE(1 + Node->EndingVpn,
1964                     AlignmentVpn)) {
1965
1966                 *PreviousVad = Node;
1967                 *Base = (PVOID) MI_ROUND_TO_SIZE(
1968                     (ULONG_PTR)MI_VPN_TO_VA_ENDING(Node->
1969                         EndingVpn) ,

```

```

1968                                     Alignment);
1969         return STATUS_SUCCESS;
1970     }
1971 }
1972
1973 } else {
1974
1975     //
1976     // No more descriptors, check to see if this fits into
1977     // the remainder
1978     // of the address space.
1979     //
1980     if (((ULONG_PTR)Node->EndingVpn + MI_VA_TO_VPN(X64K)) <
1981         MI_VA_TO_VPN (MM_HIGHEST_VAD_ADDRESS))
1982         &&
1983         (SizeOfRange <=
1984          ((ULONG_PTR)MM_HIGHEST_VAD_ADDRESS -
1985           (ULONG_PTR)MI_ROUND_TO_SIZE(
1986            (ULONG_PTR)MI_VPN_TO_VA(Node->EndingVpn),
1987             Alignment)))) {
1988
1989         *PreviousVad = Node;
1990         *Base = (PVOID) MI_ROUND_TO_SIZE(
1991             (ULONG_PTR)MI_VPN_TO_VA_ENDING(Node->
1992              EndingVpn),
1993             Alignment);
1994         return STATUS_SUCCESS;
1995     }
1996     return STATUS_NO_MEMORY;
1997 }
1998 Node = NextNode;
1999
2000 } while (TRUE);
2001 }
2002
2003 NTSTATUS
2004 MiFindEmptyAddressRangeDownTree (
2005     IN SIZE_T SizeOfRange,

```

```
IN PVOID HighestAddressToEndAt ,
IN ULONG_PTR Alignment ,
IN PMM_AVL_TABLE Table ,
OUT PVOID *Base
)
```

```
/*++
```

#### Routine Description:

The function examines the virtual address descriptors to locate an unused range of the specified size and returns the starting address of the range. The function examines from the high addresses down and ensures that starting address is less than the specified address.

Note this cannot be used for the based section tree because only the nodes in that tree are stored as VAs instead of VPNs.

#### Arguments:

SizeOfRange – Supplies the size in bytes of the range to locate.

HighestAddressToEndAt – Supplies the virtual address that limits the value of the ending address. The ending address of the located range must be less than this address.

Alignment – Supplies the alignment for the address. Must be a power of 2 and greater than the page\_size.

Table – Supplies the root of the tree to search through.

Base – Receives the starting address of a suitable range on success.

#### Return Value:

```

2041     NTSTATUS.
2042
2043     /*
2044
2045     {
2046         PMMADDRESS_NODE Node;
2047         PMMADDRESS_NODE PreviousNode;
2048         ULONG_PTR AlignedEndingVa;
2049         PVOID OptimalStart;
2050         ULONG_PTR OptimalStartVpn;
2051         ULONG_PTR HighestVpn;
2052         ULONG_PTR AlignmentVpn;
2053
2054         //
2055         // Note this cannot be used for the based section tree because
2056         // only
2057         // the nodes in that tree are stored as VAs instead of VPNs.
2058         //
2059         ASSERT (Table != &MmSectionBasedRoot);
2060
2061         SizeOfRange = MI_ROUND_TO_SIZE (SizeOfRange, PAGE_SIZE);
2062
2063         if (((ULONG_PTR) HighestAddressToEndAt + 1) < SizeOfRange) {
2064             return STATUS_NO_MEMORY;
2065         }
2066
2067         ASSERT (HighestAddressToEndAt != NULL);
2068         ASSERT (HighestAddressToEndAt <= (PVOID) ((ULONG_PTR)
2069             MM_HIGHEST_VAD_ADDRESS + 1));
2070
2071         HighestVpn = MI_VA_TO_VPN (HighestAddressToEndAt);
2072
2073         //
2074         // Locate the Node with the highest starting address.
2075         //
2076         OptimalStart = (PVOID) (MI_ALIGN_TO_SIZE(
2077             (((ULONG_PTR) HighestAddressToEndAt + 1) -

```

```
2078         SizeOfRange),
2079         Alignment));
2080     if (Table->NumberGenericTableElements == 0) {
2081
2082         //
2083         // The tree is empty, any range is okay.
2084         //
2085
2086         *Base = OptimalStart;
2087         return STATUS_SUCCESS;
2088     }
2089
2090     Node = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;
2091
2092     //
2093     // See if an empty slot exists to hold this range, locate the
2094     // largest
2095     // element in the tree.
2096     //
2097     while (Node->RightChild != NULL) {
2098         Node = Node->RightChild;
2099     }
2100
2101     //
2102     // Check to see if a range exists between the highest address VAD
2103     // and the highest address to end at.
2104     //
2105
2106     AlignedEndingVa = (ULONG_PTR)MI_ROUND_TO_SIZE ((ULONG_PTR)
2107         MI_VPN_TO_VA_ENDING (Node->EndingVpn),
2108         Alignment);
2109
2110     if (AlignedEndingVa < (ULONG_PTR)HighestAddressToEndAt) {
2111
2112         if (SizeOfRange < ((ULONG_PTR)HighestAddressToEndAt -
2113             AlignedEndingVa)) {
```

```

2113         *Base = MI_ALIGN_TO_SIZE(
2114             ((ULONG_PTR) HighestAddressToEndAt -
2115              SizeOfRange),
2116             Alignment);
2117     return STATUS_SUCCESS;
2118 }
2119
2120 //
2121 // Walk the tree backwards looking for a fit.
2122 //
2123
2124 OptimalStartVpn = MI_VA_TO_VPN (OptimalStart);
2125 AlignmentVpn = MI_VA_TO_VPN (Alignment);
2126
2127 do {
2128
2129     PreviousNode = MiGetPreviousNode (Node);
2130
2131     if (PreviousNode != NULL) {
2132
2133         //
2134         // Is the ending Va below the top of the address to end
2135         // at.
2136
2137         if (PreviousNode->EndingVpn < OptimalStartVpn) {
2138             if ((SizeOfRange >> PAGE_SHIFT) <=
2139                 ((ULONG_PTR) Node->StartingVpn -
2140                  (ULONG_PTR) MI_ROUND_TO_SIZE(1 + PreviousNode->
2141                      EndingVpn,
2142                      AlignmentVpn))) {
2143
2144                 //
2145                 // See if the optimal start will fit between
2146                 // these
2147                 // two VADs.

```

```

2148         if ((OptimalStartVpn > PreviousNode->EndingVpn)
2149             &&
2150             (HighestVpn < Node->StartingVpn)) {
2151             *Base = OptimalStart;
2152             return STATUS_SUCCESS;
2153         }
2154
2155         //
2156         // Check to ensure that the ending address
2157         // aligned upwards
2158         // is not greater than the starting address.
2159         //
2160         if ((ULONG_PTR)Node->StartingVpn >
2161             (ULONG_PTR)MI_ROUND_TO_SIZE(1 +
2162             PreviousNode->EndingVpn,
2163             AlignmentVpn)) {
2164
2165             *Base = MI_ALIGN_TO_SIZE(
2166                 (ULONG_PTR)MI_VPN_TO_VA (
2167                     Node->StartingVpn) -
2168                     SizeOfRange,
2169                     Alignment);
2170             return STATUS_SUCCESS;
2171         }
2172     }
2173 } else {
2174
2175     //
2176     // No more descriptors, check to see if this fits into
2177     // the remainder
2178     // of the address space.
2179     //
2180
2181     if (Node->StartingVpn > MI_VA_TO_VPN (
2182         MM_LOWEST_USER_ADDRESS)) {
2183         if ((SizeOfRange >> PAGE_SHIFT) <=
2184             ((ULONG_PTR)Node->StartingVpn - MI_VA_TO_VPN (

```



```

2180         MM_LOWEST_USER_ADDRESS))) {
2181         //
2182         // See if the optimal start will fit between
2183         // these
2184         // two VADs.
2185         //
2186         if (HighestVpn < Node->StartingVpn) {
2187             *Base = OptimalStart;
2188             return STATUS_SUCCESS;
2189         }
2190
2191         *Base = MI_ALIGN_TO_SIZE(
2192             (ULONG_PTR)MI_VPN_TO_VA (Node->
2193                 StartingVpn) - SizeOfRange,
2194             Alignment);
2195         return STATUS_SUCCESS;
2196     }
2197     return STATUS_NO_MEMORY;
2198 }
2199 Node = PreviousNode;
2200
2201 } while (TRUE);
2202 }
2203
2204
2205 NTSTATUS
2206 MiFindEmptyAddressRangeDownBasedTree (
2207     IN SIZE_T SizeOfRange,
2208     IN PVOID HighestAddressToEndAt,
2209     IN ULONG_PTR Alignment,
2210     IN PMM_AVL_TABLE Table,
2211     OUT PVOID *Base
2212 )
2213
2214 /*++
2215

```

### Routine Description:

The function examines the virtual address descriptors to locate an unused range of the specified size and returns the starting address of the range. The function examines from the high addresses down and ensures that starting address is less than the specified address.

Note this is only used for the based section tree because only the nodes in that tree are stored as VAs instead of VPNs.

### Arguments:

SizeOfRange – Supplies the size in bytes of the range to locate.

HighestAddressToEndAt – Supplies the virtual address that limits the value of the ending address. The ending address of the located range must be less than this address.

Alignment – Supplies the alignment for the address. Must be a power of 2 and greater than the page\_size.

Table – Supplies the root of the tree to search through.

Base – Receives the starting address of a suitable range on success.

### Return Value:

NTSTATUS.

—\*/

{

```
PMMADDRESS_NODE Node;  
PMMADDRESS_NODE PreviousNode;  
ULONG_PTR AlignedEndingVa;
```

```
2253     ULONG_PTR OptimalStart;
2254
2255     //
2256     // Note this is only used for the based section tree because only
2257     // the nodes in that tree are stored as VAs instead of VPNs.
2258     //
2259
2260     ASSERT (Table == &MmSectionBasedRoot);
2261
2262     SizeOfRange = MI_ROUND_TO_SIZE (SizeOfRange, PAGE_SIZE);
2263
2264     if (((ULONG_PTR) HighestAddressToEndAt + 1) < SizeOfRange) {
2265         return STATUS_NO_MEMORY;
2266     }
2267
2268     ASSERT (HighestAddressToEndAt != NULL);
2269     ASSERT (HighestAddressToEndAt <= (PVOID) ((ULONG_PTR)
        MM_HIGHEST_VAD_ADDRESS + 1));
2270
2271     //
2272     // Locate the node with the highest starting address.
2273     //
2274
2275     OptimalStart = (ULONG_PTR) MI_ALIGN_TO_SIZE (
2276         ((ULONG_PTR) HighestAddressToEndAt + 1) -
2277         SizeOfRange,
2278         Alignment);
2279
2280     if (Table->NumberGenericTableElements == 0) {
2281         //
2282         // The tree is empty, any range is okay.
2283         //
2284
2285         *Base = (PVOID) OptimalStart;
2286         return STATUS_SUCCESS;
2287     }
2288
2289     Node = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;
```

```
2290
2291 //
2292 // See if an empty slot exists to hold this range, locate the
2293 // largest
2294 // element in the tree.
2295 //
2296 while (Node->RightChild != NULL) {
2297     Node = Node->RightChild;
2298 }
2299
2300 //
2301 // Check to see if a range exists between the highest address VAD
2302 // and the highest address to end at.
2303 //
2304
2305 AlignedEndingVa = MI_ROUND_TO_SIZE (Node->EndingVpn, Alignment);
2306
2307 PRINT("search_down0: %p_%p_%p\n", AlignedEndingVa,
2308     HighestAddressToEndAt, SizeOfRange);
2309
2310 if ((AlignedEndingVa < (ULONG_PTR) HighestAddressToEndAt) &&
2311     (SizeOfRange < ((ULONG_PTR) HighestAddressToEndAt -
2312         AlignedEndingVa))) {
2313
2314     *Base = MI_ALIGN_TO_SIZE(
2315         ((ULONG_PTR) HighestAddressToEndAt -
2316             SizeOfRange),
2317         Alignment);
2318     return STATUS_SUCCESS;
2319 }
2320
2321 //
2322 // Walk the tree backwards looking for a fit.
2323 //
2324
2325 do {
2326
2327     PreviousNode = MiGetPreviousNode (Node);
```

```

2325
2326 PRINT("search_down1: %p %p %p %p\n", PreviousNode, Node,
        OptimalStart, Alignment);
2327
2328 if (PreviousNode == NULL) {
2329     break;
2330 }
2331
2332 //
2333 // Is the ending Va below the top of the address to end at.
2334 //
2335
2336 if (PreviousNode->EndingVpn < OptimalStart) {
2337
2338     if (SizeOfRange <= (Node->StartingVpn -
2339         MI_ROUND_TO_SIZE(1 + PreviousNode->EndingVpn,
2340             Alignment))) {
2341
2342         //
2343         // See if the optimal start will fit between these
2344         // two VADs.
2345
2346         if ((OptimalStart > PreviousNode->EndingVpn) &&
2347             ((ULONG_PTR) HighestAddressToEndAt < Node->
2348                 StartingVpn)) {
2349             *Base = (PVOID) OptimalStart;
2350             return STATUS_SUCCESS;
2351         }
2352
2353         //
2354         // Check to ensure that the ending address aligned
2355         // upwards
2356         // is not greater than the starting address.
2357         //
2358         if (Node->StartingVpn >
2359             MI_ROUND_TO_SIZE(1 + PreviousNode->EndingVpn,
2360                 Alignment)) {

```

```
2358
2359         *Base = MI_ALIGN_TO_SIZE (Node->StartingVpn -
2360                                     SizeOfRange ,
2361                                     Alignment);
2362
2363         return STATUS_SUCCESS;
2364     }
2365 }
2366
2367 Node = PreviousNode;
2368
2369 } while (TRUE);
2370
2371
2372 //
2373 // No more descriptors , check to see if this fits into the
2374 // remainder
2375 // of the address space.
2376 //
2377 if (Node->StartingVpn > (ULONG_PTR) MM_LOWEST_USER_ADDRESS) {
2378
2379     if (SizeOfRange <= (Node->StartingVpn - (ULONG_PTR)
2380         MM_LOWEST_USER_ADDRESS)) {
2381
2382         //
2383         // See if the optimal start will fit between these two
2384         // VADs.
2385
2386         if ((ULONG_PTR) HighestAddressToEndAt < Node->StartingVpn
2387             ) {
2388             *Base = (PVOID) OptimalStart;
2389             return STATUS_SUCCESS;
2390         }
2391
2392         *Base = MI_ALIGN_TO_SIZE (Node->StartingVpn - SizeOfRange
2393                                     ,
```

```
2391                                     Alignment);
2392
2393         return STATUS_SUCCESS;
2394     }
2395 }
2396 return STATUS_NO_MEMORY;
2397 }
2398
2399 #if !defined (_USERMODE)
2400
2401 PMMVAD
2402 FASTCALL
2403 MiLocateAddress (
2404     IN PVOID VirtualAddress
2405 )
2406
2407 /*++
2408
2409 Routine Description:
2410
2411     The function locates the virtual address descriptor which
2412     describes
2413     a given address.
2414
2415 Arguments:
2416
2417     VirtualAddress – Supplies the virtual address to locate a
2418                     descriptor for.
2419
2420     Table – Supplies the table describing the tree.
2421
2422 Return Value:
2423
2424     Returns a pointer to the virtual address descriptor which
2425     contains
2426     the supplied virtual address or NULL if none was located.
2427
2428 --*/
```

```
2427 {
2428     PMMVAD FoundVad;
2429     ULONG_PTR Vpn;
2430     PMM_AVL_TABLE Table;
2431     TABLE_SEARCH_RESULT SearchResult;
2432
2433     Table = &PsGetCurrentProcess ()->VadRoot;
2434
2435     //
2436     // Note the NodeHint *MUST* be captured locally – see the
2437     // synchronization
2438     // comment below for details.
2439     //
2440     FoundVad = (PMMVAD) Table->NodeHint;
2441
2442     if (FoundVad == NULL) {
2443         return NULL;
2444     }
2445
2446     Vpn = MI_VA_TO_VPN (VirtualAddress);
2447
2448     if ((Vpn >= FoundVad->StartingVpn) && (Vpn <= FoundVad->EndingVpn
2449         )) {
2450         return FoundVad;
2451     }
2452
2453     //
2454     // Lookup the element and save the result.
2455     //
2456     SearchResult = MiFindNodeOrParent (Table,
2457                                         Vpn,
2458                                         (PMMADDRESS_NODE *) &FoundVad)
2459                                         ;
2460
2461     if (SearchResult != TableFoundNode) {
2462         return NULL;
2463     }
```



```
2463
2464     ASSERT (FoundVad != NULL);
2465
2466     ASSERT ((Vpn >= FoundVad->StartingVpn) && (Vpn <= FoundVad->
2467         EndingVpn));
2468
2469     //
2470     // Note the NodeHint field update is not synchronized in all
2471     // cases, ie:
2472     // some callers hold the address space mutex and others hold the
2473     // working
2474     // set pushlock. It is ok that the update is not synchronized -
2475     // as long
2476     // as care is taken above that it is read into a local variable
2477     // and then
2478     // referenced. Because no VAD can be removed from the tree
2479     // without holding
2480     // both the address space & working set.
2481     //
2482
2483     Table->NodeHint = (PVOID) FoundVad;
2484
2485     //
2486     // Return the VAD.
2487     //
2488
2489     return FoundVad;
2490 }
2491 #endif
2492
2493 #if DBG
2494 VOID
2495 MiNodeTreeWalk (
2496     IN PMM_AVL_TABLE Table
2497 )
2498 {
2499     PVOID RestartKey;
2500     PMMADDRESS_NODE NewNode;
2501     PMMADDRESS_NODE PrevNode;
```

```

2496 PMMADDRESS_NODE NextNode;
2497
2498 RestartKey = NULL;
2499
2500 do {
2501
2502     NewNode = MiEnumerateGenericTableWithoutSplayingAvl (Table ,
2503                                                         &
                                                         RestartKey
                                                         );
2504
2505     if (NewNode == NULL) {
2506         break;
2507     }
2508
2509     PrevNode = MiGetPreviousNode (NewNode);
2510     NextNode = MiGetNextNode (NewNode);
2511
2512     PRINT ( "Node_□p_□x_□x\n" ,
2513            NewNode ,
2514            NewNode->StartingVpn ,
2515            NewNode->EndingVpn );
2516
2517     if (PrevNode != NULL) {
2518         PRINT ( "\tPrevNode_□p_□x_□x\n" ,
2519                PrevNode ,
2520                PrevNode->StartingVpn ,
2521                PrevNode->EndingVpn );
2522     }
2523
2524     if (NextNode != NULL) {
2525         PRINT ( "\tNextNode_□p_□x_□x\n" ,
2526                NextNode ,
2527                NextNode->StartingVpn ,
2528                NextNode->EndingVpn );
2529     }
2530
2531 } while (TRUE);
2532

```

```
2533     PRINT ( "NumberGenericTableElements_0x%x, Depth_0x%x\n" ,
2534           Table->NumberGenericTableElements ,
2535           Table->DepthOfTree );
2536
2537     return ;
2538 }
2539 #endif
2540
2541 #if defined (_USERMODE)
2542
2543 MMADDRESS_NODE MiBalancedLinks ;
2544
2545 MM_AVL_TABLE MiAvlTable ;
2546 MM_AVL_TABLE MmSectionBasedRoot ;
2547
2548 ULONG DeleteRandom = 1 ;
2549
2550 #if RANDOM
2551 #define NUMBER_OF_VADS 32
2552 #else
2553 #define NUMBER_OF_VADS 4
2554 #endif
2555
2556 int __cdecl
2557 main(
2558     int argc ,
2559     PCHAR argv []
2560 )
2561 {
2562     ULONG i ;
2563     PVOID StartingAddress ;
2564     PVOID EndingAddress ;
2565     NTSTATUS Status ;
2566     PMMADDRESS_NODE NewNode ;
2567 #if RANDOM
2568     PMMADDRESS_NODE PrevNode ;
2569     ULONG RandomNumber = 0x99887766 ;
2570     ULONG_PTR DeleteVpn = 0 ;
2571 #endif
```

```

2572     PMM_AVL_TABLE Table;
2573     SIZE_T CapturedRegionSize;
2574
2575     UNREFERENCED_PARAMETER (argc);
2576     UNREFERENCED_PARAMETER (argv);
2577
2578     #if RANDOM
2579         Table = &MiAvlTable;
2580     #else
2581         Table = &MmSectionBasedRoot;
2582     #endif
2583
2584     MiInitializeVadTableAvl (Table);
2585
2586     for (i = 0; i < NUMBER_OF_VADS; i += 1) {
2587         NewNode = malloc (sizeof (MMADDRESS_NODE));
2588         ASSERT (((ULONG_PTR)NewNode & 0x3) == 0);
2589
2590         if (NewNode == NULL) {
2591             PRINT ("Malloc failed\n");
2592             exit (1);
2593         }
2594
2595         NewNode->u1.Parent = NULL;
2596         NewNode->LeftChild = NULL;
2597         NewNode->RightChild = NULL;
2598         NewNode->u1.Balance = 0;
2599
2600     #if RANDOM
2601         RandomNumber = RtlRandom (&RandomNumber);
2602
2603         CapturedRegionSize = (SIZE_T) (RandomNumber & 0x1FFFFFF);
2604
2605         Status = MiFindEmptyAddressRangeInTree (CapturedRegionSize,
2606                                                  64 * 1024, //
2607                                                  align
2608                                                  Table,
2609                                                  &PrevNode,
2610                                                  &StartingAddress);

```

```

2610
2611 #else
2612     CapturedRegionSize = 0x800000;
2613
2614     Status = MiFindEmptyAddressRangeDownBasedTree (
2615         CapturedRegionSize ,
2616         (PVOID) 0x7f7effff ,
2617             // highest
2618             addr
2619             64 * 1024, //
2620             align
2621             Table ,
2622             &StartingAddress);
2623 #endif
2624
2625     if (!NT_SUCCESS (Status)) {
2626         PRINT ("Could not find empty addr range in tree for size
2627             %p\n", CapturedRegionSize);
2628         free (NewNode);
2629         continue;
2630     }
2631
2632 #if RANDOM
2633     EndingAddress = (PVOID) (((ULONG_PTR) StartingAddress +
2634         CapturedRegionSize - 1L) | (PAGE_SIZE -
2635             1L));
2636 #else
2637     EndingAddress = (PVOID) (((ULONG_PTR) StartingAddress +
2638         CapturedRegionSize - 1L));
2639 #endif
2640
2641     printf ("Inserting addr range in tree @ %p\n",
2642         StartingAddress, EndingAddress);
2643
2644 #if RANDOM
2645     NewNode->StartingVpn = MI_VA_TO_VPN (StartingAddress);
2646     NewNode->EndingVpn = MI_VA_TO_VPN (EndingAddress);
2647 #else
2648     NewNode->StartingVpn = (ULONG_PTR) StartingAddress;

```

```

2642     NewNode->EndingVpn = (ULONG_PTR) EndingAddress;
2643 #endif
2644
2645     MiInsertNode (NewNode, Table);
2646
2647 #if RANDOM
2648     RandomNumber = RtlRandom (&RandomNumber);
2649
2650     if (RandomNumber & 0x3) {
2651         DeleteVpn = NewNode->StartingVpn;
2652     }
2653
2654     if (DeleteRandom && ((i & 0x3) == 0)) {
2655         NewNode = MiLocateAddressInTree (DeleteVpn, Table);
2656         printf ("Located node for random deletion - vpn %p @ %p\n",
2657             DeleteVpn, NewNode);
2658
2659         if (NewNode != NULL) {
2660             MiRemoveNode (NewNode, Table);
2661             printf ("Removed random node for vpn %p @ %p %p %p\n",
2662                 DeleteVpn, NewNode, NewNode->StartingVpn, NewNode->EndingVpn);
2663         }
2664     }
2665 #endif
2666     printf ("\n");
2667 }
2668
2669 MiNodeTreeWalk (Table);
2670
2671 NewNode = MiLocateAddressInTree (5, Table);
2672 printf ("Located node for vpn 5 @ %p\n", NewNode);
2673
2674 if (NewNode != NULL) {
2675     MiRemoveNode (NewNode, Table);
2676     printf ("Removed node for vpn 5 @ %p\n", NewNode);
2677 }

```

```
2678     NewNode = MiLocateAddressInTree (5, Table);
2679     printf("Located node for vpn 5 @ %p\n", NewNode);
2680
2681     printf("all done, balmin=%x, balmax=%x\n", BalMin, BalMax);
2682
2683     return 0;
2684 }
2685
2686 #endif
```