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. 从一个结点到一个 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：

```c

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

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

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）:

```c
\\在rbtree.h中
#define rb_parent(r)   ((struct rb_node *)((r)->__rb_parent_color &
    ~3))
\\在rbtree_augmented.c
#define RB_RED      0
#define RB_BLACK    1

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

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

static inline void rb_set_parent(struct rb_node *rb, struct rb_node *
    p)
{
    rb->__rb_parent_color = rb_color(rb) | (unsigned long)p;
}
//设置父节点的同时设置自己的颜色
```

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

static inline void
__rb_change_child(struct rb_node *old, struct rb_node *new,
                  struct rb_node *parent, struct rb_root *root)
{
    if (parent) {
        if (parent->rb_left == old)
            WRITE_ONCE(parent->rb_left, new);
            \\WRITE_ONCE(a,b)=>a=b;
        else
            WRITE_ONCE(parent->rb_right, new);
    } else
        WRITE_ONCE(root->rb_node, new);
}
\\在rbtree.c中，为旋转做准备
static inline void
__rb_rotate_set_parents(struct rb_node *old, struct rb_node *new,
                        struct rb_root *root, int color)
{
    struct rb_node *parent = rb_parent(old);
    new->__rb_parent_color = old->__rb_parent_color;
    rb_set_parent_color(old, new, color);
    __rb_change_child(old, new, parent, root);
}
```

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:

```
void rb_erase(struct rb_node *node, struct rb_root *root)
{
    struct rb_node *rebalance;
    rebalance = __rb_erase_augmented(node, root, &dummy_callbacks);
    if (rebalance)
```

```
        _____rb_erase_color(rebalance, root, dummy_rotate);
}
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,

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

其中的 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:

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

```
} MMADDRESS_NODE, *PMMADDRESS_NODE;
```

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

```
#define rb_black 0
#define rb_red 1

PMMADDRESS_NODE rb_parent(PMMADDRESS_NODE node)
{
    node =SANITIZE_PARENT_NODE(SANITIZE_PARENT_NODE(node)->u1.Parent)
        ;
    return node;
}
int rb_color(PMMADDRESS_NODE node)
{
    return node->u1.Balance;
}
int rb_is_red(PMMADDRESS_NODE node)
{
    return node->u1.Balance==rb_red;
}
int rb_is_black(PMMADDRESS_NODE node)
{
    return node->u1.Balance==rb_black;
}
```

```c
void rb_set_black(PMMADDRESS_NODE node)
{
    node->u1.Balance = rb_black;
}
void rb_set_red(PMMADDRESS_NODE node)
{
    node->u1.Balance = rb_red;
}
void rb_set_parent( PMMADDRESS_NODE rb,PMMADDRESS_NODE p)
{
    rb->u1.Parent =(PMMADDRESS_NODE)(((ULONG_PTR)(p)) + ((ULONG_PTR)(rb->u1.Balance)));
}

void rb_set_color( PMMADDRESS_NODE rb, int color)
{
    rb->u1.Balance = color;
}

void rb_set_parent_color(PMMADDRESS_NODE rb, PMMADDRESS_NODE p,int color)
{
    rb->u1.Parent =p;
    rb->u1.Balance=color;
}

void __rb_change_child(PMMADDRESS_NODE old, PMMADDRESS_NODE newer,
    PMMADDRESS_NODE parent, PMM_AVL_TABLE root)
{
    if (parent!=&root->BalancedRoot) {
        if (parent->LeftChild== old){
            parent->LeftChild = newer;
        }
        else
            parent->RightChild = newer;
    } else
        root->BalancedRoot.RightChild = newer;
}

```

```
void __rb_rotate_set_parents(PMMADDRESS_NODE old, PMMADDRESS_NODE
    newer,PMM_AVL_TABLE root, int color)
{
    PMMADDRESS_NODE parent = rb_parent(old);
     rb_set_color(newer,rb_color(old));
     rb_set_parent(newer,rb_parent(old));
     rb_set_parent_color(old, newer, color);
     __rb_change_child(old, newer, parent, root);
}
```

其中 SANITIZE_PARENT_NODE 的定义如下 (在 ntos/inc/ps.h 中)

```
#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

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

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：

```

        {
            PMMADDRESS_NODE NodeOrParent;
            PMMADDRESS_NODE parent,gparent,tmp;
            TABLE_SEARCH_RESULT SearchResult;
            //插入函数
            SearchResult = MiFindNodeOrParent (Table,
            NodeToInsert->StartingVpn,
            &NodeOrParent);

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

```

```
        Table->NumberGenericTableElements += 1;

        //
        // Insert the newer node in the tree.
        //

        if (SearchResult == TableEmptyTree)
        {
            Table->BalancedRoot.RightChild = NodeToInsert;
            rb_set_parent(NodeToInsert,&Table->BalancedRoot);
        }
        else
        {
            if (SearchResult == TableInsertAsLeft)
            {
                NodeOrParent->LeftChild = NodeToInsert;
            }
            else
            {
                NodeOrParent->RightChild = NodeToInsert;
            }
            rb_set_parent(NodeToInsert,NodeOrParent);
        }
        rb_set_red(NodeToInsert);
        parent=rb_parent(NodeToInsert);
        //平衡部分，直接由linux代码修改，需要注意根节点的性质
        while(1)  {
            if (parent==&Table->BalancedRoot) {
            Table->BalancedRoot.RightChild = NodeToInsert;
            rb_set_parent_color(NodeToInsert,&Table->BalancedRoot
                ,rb_black);
            break;
        } else if (rb_is_black(parent))
        break;
        gparent = rb_parent(parent);
        tmp = gparent->RightChild;

        if (parent != tmp) {    /* parent == gparent->rb_left */
            if (tmp && rb_is_red(tmp)) {
```

```
            /*
             * Case 1 - color flips
             *
             *       G            g
             *      / \          / \
             *     p   u  -->   P   U
             *    /            /
             *   n            n
             *
             * However, since g's parent might be red, and
             * 4) does not allow this, we need to recurse
             * at g.
             */
            rb_set_parent_color(tmp, gparent, rb_black);
            rb_set_parent_color(parent, gparent, rb_black);
            NodeToInsert = gparent;
            parent = rb_parent(NodeToInsert);
            rb_set_parent_color(NodeToInsert, parent, rb_red);
            continue;
        }

        tmp = parent->RightChild;
        if (NodeToInsert == tmp) {
            /*
             * Case 2 - left rotate at parent
             *
             *      G             G
             *     / \           / \
             *    p   U  -->    n   U
             *     \           /
             *      n         p
             *
             * This still leaves us in violation of 4), the
             * continuation into Case 3 will fix that.
             */
            parent->RightChild = tmp = NodeToInsert->LeftChild;
            NodeToInsert->LeftChild = parent;
            if (tmp)
            rb_set_parent_color(tmp, parent,rb_black);
```

```
                rb_set_parent_color(parent, NodeToInsert, rb_red);
                parent = NodeToInsert;
                tmp = NodeToInsert->RightChild;
            }

            /*
            * Case 3 - right rotate at gparent
            *
            *          G              P
            *         / \            / \
            *        p   U   -->   n    g
            *       /                    \
            *      n                      U
            */
            gparent->LeftChild= tmp;  /* == parent->rb_right */
            parent->RightChild= gparent;
            if (tmp)
            rb_set_parent_color(tmp, gparent, rb_black);
            __rb_rotate_set_parents(gparent, parent, Table, rb_red);
            break;
        } else {
            tmp = gparent->LeftChild;
            if (tmp && rb_is_red(tmp)) {
            /* Case 1 - color flips */
            rb_set_parent_color(tmp, gparent, rb_black);
            rb_set_parent_color(parent, gparent, rb_black);
            NodeToInsert = gparent;
            parent = rb_parent(NodeToInsert);
            rb_set_parent_color(NodeToInsert, parent, rb_red);
            continue;
        }

        tmp = parent->LeftChild;
        if (NodeToInsert == tmp) {
            /* Case 2 - right rotate at parent */
            parent->LeftChild = tmp = NodeToInsert->RightChild;
            NodeToInsert->RightChild = parent;
            if (tmp)
            rb_set_parent_color(tmp, parent,
```

```
                rb_black);
                rb_set_parent_color(parent, NodeToInsert, rb_red);
                parent = NodeToInsert;
                tmp = NodeToInsert->LeftChild;
            }

            /* Case 3 - left rotate at gparent */
            gparent->RightChild = tmp;   /* == parent->rb_left */
            parent->LeftChild = gparent;
            if (tmp)
            rb_set_parent_color(tmp, gparent, rb_black);
            __rb_rotate_set_parents(gparent, parent, Table, rb_red);
            break;
        }
}
return;
}
```

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：

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

使用 linux 代码前需要先移植 __rb_erase_augmente 和 ____rb_erase_color; 代码如下

```

static PMMADDRESS_NODE __rb_erase_augmented(PMMADDRESS_NODE node,
    PMM_AVL_TABLE root)
{
    PMMADDRESS_NODE  child = node->RightChild, tmp = node->LeftChild;
    PMMADDRESS_NODE  parent, rebalance;
    PMMADDRESS_NODE pc;

```

```
    if (!tmp) {
        /*
        * Case 1: node to erase has no more than 1 child (easy!)
        *
        * Note that if there is one child it must be red due to 5)
        * and node must be black due to 4). We adjust colors locally
        * so as to bypass __rb_erase_color() later on.
        */
        pc = node->u1.Parent;
        parent = rb_parent(node);
        __rb_change_child(node, child, parent, root);
        if (child) {
            child->u1.Parent = pc;
            child->u1.Balance=(((ULONG_PTR)(pc)) & 0x1);
            rebalance = &root->BalancedRoot;
        } else
            rebalance = (((ULONG_PTR)(pc)) & 0x1)==0 ? parent : &root->BalancedRoot;
    } else if (!child) {
        /* Still case 1, but this time the child is node->rb_left */
        tmp->u1.Parent =pc= node->u1.Parent;
        tmp->u1.Balance=node->u1.Balance;
        parent = rb_parent(node);
        __rb_change_child(node, tmp, parent, root);
        rebalance = &root->BalancedRoot;
    } else {
        PMMADDRESS_NODE successor = child, child2;
        tmp = child->LeftChild;
        if (!tmp) {
            /*
            * Case 2: node's successor is its right child
            *
            *    (n)          (s)
            *    / \          / \
            *  (x) (s)  ->  (x) (c)
            *        \
            *        (c)
            */
            parent = successor;
```

```
            child2 = successor->RightChild;
        } else {
            /*
            * Case 3: node's successor is leftmost under
            * node's right child subtree
            *
            *    (n)           (s)
            *    / \           / \
            *  (x) (y)  ->  (x) (y)
            *       /             /
            *    (p)           (p)
            *    /             /
            *  (s)           (c)
            *    \
            *    (c)
            */
            do {
                parent = successor;
                successor = tmp;
                tmp = tmp->LeftChild;
            } while (tmp);
            parent->LeftChild = child2 = successor->RightChild;
            successor->RightChild = child;
            rb_set_parent(child, successor);
        }

        successor->LeftChild = tmp = node->LeftChild;
        rb_set_parent(tmp, successor);
        pc = node->u1.Parent;
        tmp = SANITIZE_PARENT_NODE(pc);
        __rb_change_child(node, successor, tmp, root);
        if (child2) {
            successor->u1.Parent = pc;
            successor->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
            rb_set_parent_color(child2, parent, rb_black);
            rebalance = &root->BalancedRoot;
        } else {
            PMMADDRESS_NODE pc2 = successor->u1.Parent;
            successor->u1.Parent = pc;
```

```
                successor->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
                rebalance = (((ULONG_PTR)(pc2)) & 0x1)==0 ? parent : &
                    root->BalancedRoot;
            }
            tmp = successor;
        }
        return rebalance;
}

static void
    _____rb_erase_color(PMMADDRESS_NODE  parent, PMM_AVL_TABLE root)
{
    PMMADDRESS_NODE node =NULL, sibling, tmp1=NULL, tmp2=NULL;

        while (1) {
            /*
             * Loop invariants:
             * - node is black (or NULL on first iteration)
             * - node is not the root (parent is not NULL)
             * - All leaf paths going through parent and node have a
             *   black node count that is 1 lower than other leaf paths.
             */
            sibling = parent->RightChild;
            if (node != sibling) {  /* node == parent->rb_left */
                if (rb_is_red(sibling)) {
                    /*
                     * Case 1 - left rotate at parent
                     *
                     *       P               S
                     *      / \             / \
                     *     N   s    -->    p   Sr
                     *        / \         / \
                     *       Sl  Sr      N   Sl
                     */
                    parent->RightChild = tmp1 = sibling->LeftChild;
                    sibling->LeftChild = parent;
                    if(tmp1)
                        rb_set_parent_color(tmp1, parent, rb_black);
                    __rb_rotate_set_parents(parent, sibling, root,
```

```
                rb_red);
            sibling = tmp1;
        }
        tmp1 = sibling->RightChild;
        if (!tmp1 || rb_is_black(tmp1)) {
                tmp2 = sibling->LeftChild;
            if (!tmp2 || rb_is_black(tmp2)) {
                /*
                 * Case 2 - sibling color flip
                 * (p could be either color here)
                 *
                 *      (p)           (p)
                 *      / \           / \
                 *     N   S    -->  N   s
                 *        / \           / \
                 *       Sl  Sr        Sl  Sr
                 *
                 * This leaves us violating 5) which
                 * can be fixed by flipping p to black
                 * if it was red, or by recursing at p.
                 * p is red when coming from Case 1.
                 */
                    rb_set_parent_color(sibling, parent,
                    rb_red);
                if (rb_is_red(parent))
                    rb_set_black(parent);
                else {
                    node = parent;
                    parent = rb_parent(node);
                    if (parent!=&root->BalancedRoot)
                        continue;
                }
                break;
            }
            /*
             * Case 3 - right rotate at sibling
             * (p could be either color here)
             *
             *    (p)           (p)
```

```
                *   / \           / \
                *  N   S    -->  N    Sl
                *     / \              \
                *    sl  Sr             s
                *                        \
                *                         Sr
                */
                sibling->LeftChild = tmp1 = tmp2->RightChild;
                tmp2->RightChild = sibling;
                parent->RightChild = tmp2;
                if (tmp1)
                    rb_set_parent_color(tmp1, sibling,
                    rb_black);
                tmp1 = sibling;
                sibling = tmp2;
            }
            /*
            * Case 4 - left rotate at parent + color flips
            * (p and sl could be either color here.
            *  After rotation, p becomes black, s acquires
            *  p's color, and sl keeps its color)
            *
            *      (p)             (s)
            *      / \              / \
            *     N   S     -->   P   Sr
            *        / \         / \
            *      (sl) sr      N  (sl)
            */
            parent->RightChild = tmp2 = sibling->LeftChild;
            sibling->LeftChild = parent;
            rb_set_parent_color(tmp1, sibling, rb_black);
            if (tmp2)
                rb_set_parent(tmp2, parent);
            __rb_rotate_set_parents(parent, sibling, root,
                rb_black);
            break;
        } else {
            sibling = parent->LeftChild;
            if (rb_is_red(sibling)) {
```

```
            /* Case 1 - right rotate at parent */
            parent->LeftChild = tmp1 = sibling->RightChild;
            sibling->RightChild = parent;
            rb_set_parent_color(tmp1, parent, rb_black);
            __rb_rotate_set_parents(parent, sibling, root,
                rb_red);
            sibling = tmp1;
        }
        tmp1 = sibling->LeftChild;
        if (!tmp1 || rb_is_black(tmp1)) {
            tmp2 = sibling->RightChild;
            if (!tmp2 || rb_is_black(tmp2)) {
                /* Case 2 - sibling color flip */
                rb_set_parent_color(sibling, parent,
                    rb_red);
                if (rb_is_red(parent))
                    rb_set_black(parent);
                else {
                    node = parent;
                    parent = rb_parent(node);
                    if (parent!=&root->BalancedRoot)
                        continue;
                }
                break;
            }
            /* Case 3 - right rotate at sibling */
            sibling->RightChild = tmp1 = tmp2->LeftChild;
            tmp2->LeftChild = sibling;
            parent->LeftChild = tmp2;
            if (tmp1)
                rb_set_parent_color(tmp1, sibling,
                rb_black);
            tmp1 = sibling;
            sibling = tmp2;
        }
        /* Case 4 - left rotate at parent + color flips */
        parent->LeftChild = tmp2 = sibling->RightChild;
        sibling->RightChild = parent;
        rb_set_parent_color(tmp1, sibling, rb_black);
```

```
                if (tmp2)
                    rb_set_parent(tmp2, parent);
                __rb_rotate_set_parents(parent, sibling, root,
                    rb_black);
                break;
            }
        }
}
```

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,

```
{
    PMMADDRESS_NODE rebalance;
    Table->NumberGenericTableElements -= 1;
    rebalance = __rb_erase_augmented(NodeToDelete, Table);
    if (rebalance!=&Table->BalancedRoot&&rebalance)
    _____rb_erase_color(rebalance, Table);
}
```

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[1]调试器和虚拟机的启动引导信息[2]，然互选择 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，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；系统接口难以的定位；MS 的 symbol 服务器关机导致 WinDbg 难以进行复杂调试；

[1] 见 debug.bat 文件, 需先导入 debug.WEW

[2] 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

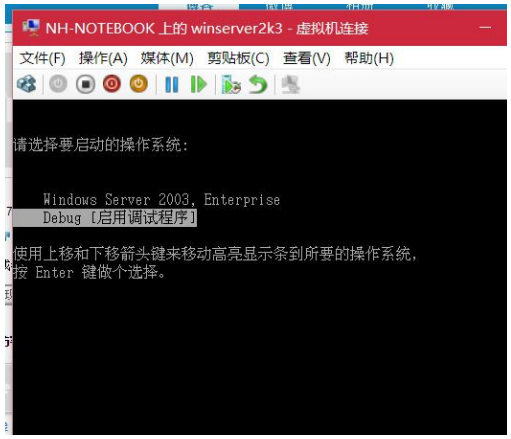

图 1: boot 选项

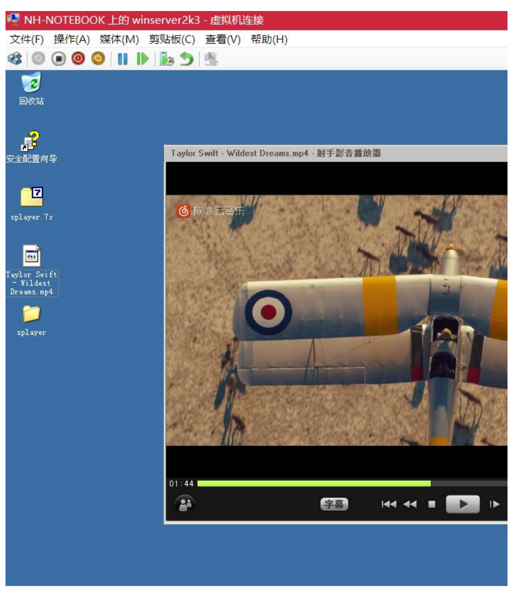

图 2: 正常工作

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去寻找信息，学会了将大型项目中的函数及其依赖剥离出来进行测试[3]，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，但这一过程除去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。

```
/*++

Copyright (c) Microsoft Corporation. All rights reserved.

You may only use this code if you agree to the terms of the Windows
    Research Kernel Source Code License agreement (see License.txt).
If you do not agree to the terms, do not use the code.


Module Name:

    addrsup.c

Abstract:

    This module implements a new version of the generic table package
    based on balanced binary trees (later named AVL), as described in
    Knuth, "The Art of Computer Programming, Volume 3, Sorting and
        Searching",
    and refers directly to algorithms as they are presented in the
        second
    edition Copyrighted in 1973.

    Used rtl\avltable.c as a starting point, adding the following:

```

[3] 见测试文件中的测试项目

```
    - Use less memory for structures as these are nonpaged & heavily
      used.
    - Caller allocates the pool to reduce mutex hold times.
    - Various VAD-specific customizations/optimizations.
    - Hints.

Environment:

    Kernel mode only, working set mutex held, APCs disabled.

--*/

#include "mi.h"

#define rb_black 0
#define rb_red 1

PMMADDRESS_NODE rb_parent(PMMADDRESS_NODE node)
{
    node =SANITIZE_PARENT_NODE(SANITIZE_PARENT_NODE(node)->u1.Parent)
        ;
    return node;
}
int rb_color(PMMADDRESS_NODE node)
{
    return node->u1.Balance;
}
int rb_is_red(PMMADDRESS_NODE node)
{
    return node->u1.Balance==rb_red;
}
int rb_is_black(PMMADDRESS_NODE node)
{
    return node->u1.Balance==rb_black;
}
void rb_set_black(PMMADDRESS_NODE node)
{
    node->u1.Balance = rb_black;
}
```

```
void rb_set_red(PMMADDRESS_NODE node)
{
    node->u1.Balance = rb_red;
}
void rb_set_parent( PMMADDRESS_NODE rb,PMMADDRESS_NODE p)
{
    rb->u1.Parent =(PMMADDRESS_NODE)(((ULONG_PTR)(p)) + ((ULONG_PTR)(
        rb->u1.Balance)));
}

void rb_set_color( PMMADDRESS_NODE rb, int color)
{
    rb->u1.Balance = color;
}

void rb_set_parent_color(PMMADDRESS_NODE rb, PMMADDRESS_NODE p,int
    color)
{
    rb->u1.Parent =p;
    rb->u1.Balance=color;
}

void
    __rb_change_child(PMMADDRESS_NODE old, PMMADDRESS_NODE newer,
    PMMADDRESS_NODE parent, PMM_AVL_TABLE root)
{
    if (parent!=&root->BalancedRoot) {
        if (parent->LeftChild== old){
            parent->LeftChild = newer;
        }
        else
            parent->RightChild = newer;
    } else
        root->BalancedRoot.RightChild = newer;
}

void __rb_rotate_set_parents(PMMADDRESS_NODE old, PMMADDRESS_NODE
    newer,
                                PMM_AVL_TABLE root, int color)
```

```
{
    PMMADDRESS_NODE parent = rb_parent(old);
    rb_set_color(newer,rb_color(old));
    rb_set_parent(newer,rb_parent(old));
    rb_set_parent_color(old, newer, color);
    __rb_change_child(old, newer, parent, root);
}

static PMMADDRESS_NODE __rb_erase_augmented(PMMADDRESS_NODE node,
   PMM_AVL_TABLE root)
{
    PMMADDRESS_NODE  child = node->RightChild, tmp = node->LeftChild;
    PMMADDRESS_NODE  parent, rebalance;
    PMMADDRESS_NODE pc;

    if (!tmp) {
        /*
        * Case 1: node to erase has no more than 1 child (easy!)
        *
        * Note that if there is one child it must be red due to 5)
        * and node must be black due to 4). We adjust colors locally
        * so as to bypass __rb_erase_color() later on.
        */
        pc = node->u1.Parent;
        parent = rb_parent(node);
        __rb_change_child(node, child, parent, root);
        if (child) {
            child->u1.Parent = pc;
            child->u1.Balance=(((ULONG_PTR)(pc)) & 0x1);
            rebalance = &root->BalancedRoot;
        } else
            rebalance = (((ULONG_PTR)(pc)) & 0x1)==0 ? parent : &root
              ->BalancedRoot;
    } else if (!child) {
        /* Still case 1, but this time the child is node->rb_left */
        tmp->u1.Parent =pc= node->u1.Parent;
        tmp->u1.Balance=node->u1.Balance;
        parent = rb_parent(node);
        __rb_change_child(node, tmp, parent, root);
```

```c
        rebalance = &root->BalancedRoot;
    } else {
        PMMADDRESS_NODE successor = child, child2;
        tmp = child->LeftChild;
        if (!tmp) {
            /*
             * Case 2: node's successor is its right child
             *
             *    (n)          (s)
             *    / \          / \
             *  (x) (s)  ->  (x) (c)
             *        \
             *        (c)
             */
            parent = successor;
            child2 = successor->RightChild;
        } else {
            /*
             * Case 3: node's successor is leftmost under
             * node's right child subtree
             *
             *    (n)          (s)
             *    / \          / \
             *  (x) (y)  ->  (x) (y)
             *      /            /
             *    (p)          (p)
             *    /            /
             *  (s)          (c)
             *    \
             *    (c)
             */
            do {
                parent = successor;
                successor = tmp;
                tmp = tmp->LeftChild;
            } while (tmp);
            parent->LeftChild = child2 = successor->RightChild;
            successor->RightChild = child;
            rb_set_parent(child, successor);
```

```
        }

        successor->LeftChild = tmp = node->LeftChild;
        rb_set_parent(tmp, successor);
        pc = node->u1.Parent;
        tmp = SANITIZE_PARENT_NODE(pc);
        __rb_change_child(node, successor, tmp, root);
        if (child2) {
            successor->u1.Parent = pc;
            successor->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
            rb_set_parent_color(child2, parent, rb_black);
            rebalance = &root->BalancedRoot;
        } else {
            PMMADDRESS_NODE pc2 = successor->u1.Parent;
            successor->u1.Parent = pc;
            successor->u1.Balance = (((ULONG_PTR)(pc)) & 0x1);
            rebalance = (((ULONG_PTR)(pc2)) & 0x1)==0 ? parent : &
                root->BalancedRoot;
        }
        tmp = successor;
    }
    return rebalance;
}

static void
    _____rb_erase_color(PMMADDRESS_NODE  parent, PMM_AVL_TABLE root)
{
    PMMADDRESS_NODE node =NULL, sibling, tmp1=NULL, tmp2=NULL;

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

```
            if (rb_is_red(sibling)) {
                /*
                 * Case 1 - left rotate at parent
                 *
                 *      P               S
                 *     / \             / \
                 *    N   s    -->    p   Sr
                 *       / \         / \
                 *      Sl  Sr      N   Sl
                 */
                parent->RightChild = tmp1 = sibling->LeftChild;
                sibling->LeftChild = parent;
                if(tmp1)
                    rb_set_parent_color(tmp1, parent, rb_black);
                __rb_rotate_set_parents(parent, sibling, root,
                    rb_red);
                sibling = tmp1;
            }
            tmp1 = sibling->RightChild;
            if (!tmp1 || rb_is_black(tmp1)) {
                    tmp2 = sibling->LeftChild;
                if (!tmp2 || rb_is_black(tmp2)) {
                    /*
                     * Case 2 - sibling color flip
                     * (p could be either color here)
                     *
                     *     (p)           (p)
                     *     / \           / \
                     *    N   S    -->  N   s
                     *       / \           / \
                     *      Sl  Sr        Sl  Sr
                     *
                     * This leaves us violating 5) which
                     * can be fixed by flipping p to black
                     * if it was red, or by recursing at p.
                     * p is red when coming from Case 1.
                     */
                        rb_set_parent_color(sibling, parent,
                        rb_red);
```

```
        if (rb_is_red(parent))
            rb_set_black(parent);
        else {
            node = parent;
            parent = rb_parent(node);
            if (parent!=&root->BalancedRoot)
                continue;
        }
        break;
    }
    /*
    * Case 3 - right rotate at sibling
    * (p could be either color here)
    *
    *   (p)           (p)
    *   / \           / \
    *  N   S    -->  N   Sl
    *     / \             \
    *    sl  Sr            s
    *                       \
    *                        Sr
    */
    sibling->LeftChild = tmp1 = tmp2->RightChild;
    tmp2->RightChild = sibling;
    parent->RightChild = tmp2;
    if (tmp1)
        rb_set_parent_color(tmp1, sibling,
        rb_black);
    tmp1 = sibling;
    sibling = tmp2;
}
/*
* Case 4 - left rotate at parent + color flips
* (p and sl could be either color here.
*  After rotation, p becomes black, s acquires
*  p's color, and sl keeps its color)
*
*      (p)             (s)
*      / \             / \
```

```
             *      N    S      -->    P    Sr
             *          / \           / \
             *       (sl) sr         N  (sl)
             */
            parent->RightChild = tmp2 = sibling->LeftChild;
            sibling->LeftChild = parent;
            rb_set_parent_color(tmp1, sibling, rb_black);
            if (tmp2)
                rb_set_parent(tmp2, parent);
            __rb_rotate_set_parents(parent, sibling, root,
                rb_black);
            break;
        } else {
            sibling = parent->LeftChild;
            if (rb_is_red(sibling)) {
                /* Case 1 - right rotate at parent */
                parent->LeftChild = tmp1 = sibling->RightChild;
                sibling->RightChild = parent;
                rb_set_parent_color(tmp1, parent, rb_black);
                __rb_rotate_set_parents(parent, sibling, root,
                    rb_red);
                sibling = tmp1;
            }
            tmp1 = sibling->LeftChild;
            if (!tmp1 || rb_is_black(tmp1)) {
                tmp2 = sibling->RightChild;
                if (!tmp2 || rb_is_black(tmp2)) {
                    /* Case 2 - sibling color flip */
                    rb_set_parent_color(sibling, parent,
                        rb_red);
                    if (rb_is_red(parent))
                        rb_set_black(parent);
                    else {
                        node = parent;
                        parent = rb_parent(node);
                        if (parent!=&root->BalancedRoot)
                            continue;
                    }
                    break;
```

```
                    }
                    /* Case 3 - right rotate at sibling */
                    sibling->RightChild = tmp1 = tmp2->LeftChild;
                    tmp2->LeftChild = sibling;
                    parent->LeftChild = tmp2;
                    if (tmp1)
                         rb_set_parent_color(tmp1, sibling,
                         rb_black);
                    tmp1 = sibling;
                    sibling = tmp2;
               }
               /* Case 4 - left rotate at parent + color flips */
               parent->LeftChild = tmp2 = sibling->RightChild;
               sibling->RightChild = parent;
               rb_set_parent_color(tmp1, sibling, rb_black);
               if (tmp2)
                    rb_set_parent(tmp2, parent);
               __rb_rotate_set_parents(parent, sibling, root,
                    rb_black);
               break;
          }
     }
}

#if !defined (_USERMODE)
#define PRINT
#define COUNT_BALANCE_MAX(a)
#else
extern MM_AVL_TABLE MmSectionBasedRoot;
#endif

#if (_MSC_VER >= 800)
#pragma warning(disable:4010)         // Allow pretty pictures without
     the noise
#endif

TABLE_SEARCH_RESULT
MiFindNodeOrParent (
     IN PMM_AVL_TABLE Table,
```

```c
    IN ULONG_PTR StartingVpn,
    OUT PMMADDRESS_NODE *NodeOrParent
    );

VOID
MiPromoteNode (
    IN PMMADDRESS_NODE C
    );

ULONG
MiRebalanceNode (
    IN PMMADDRESS_NODE S
    );

PMMADDRESS_NODE
MiRealSuccessor (
    IN PMMADDRESS_NODE Links
    );

PMMADDRESS_NODE
MiRealPredecessor (
    IN PMMADDRESS_NODE Links
    );

VOID
MiInitializeVadTableAvl (
    IN PMM_AVL_TABLE Table
    );

PVOID
MiEnumerateGenericTableWithoutSplayingAvl (
    IN PMM_AVL_TABLE Table,
    IN PVOID *RestartKey
    );

#ifdef ALLOC_PRAGMA
#pragma alloc_text(PAGE, MiCheckForConflictingNode)
#pragma alloc_text(PAGE, MiRealSuccessor)
#pragma alloc_text(PAGE, MiRealPredecessor)
```

```c
#pragma alloc_text(PAGE, MiInitializeVadTableAvl)
#pragma alloc_text(PAGE, MiFindEmptyAddressRangeInTree)
#pragma alloc_text(PAGE, MiFindEmptyAddressRangeDownTree)
#pragma alloc_text(PAGE, MiFindEmptyAddressRangeDownBasedTree)
#endif

//
// Various Rtl macros that reference Parent use private versions here
    since
// Parent is overloaded with Balance.
//

//
//  The macro function Parent takes as input a pointer to a splay
    link in a
//  tree and returns a pointer to the splay link of the parent of the
    input
//  node.  If the input node is the root of the tree the return value
    is
//  equal to the input value.
//
//  PRTL_SPLAY_LINKS
//  MiParent (
//      PRTL_SPLAY_LINKS Links
//      );
//

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

//
//  The macro function IsLeftChild takes as input a pointer to a
    splay link
//  in a tree and returns TRUE if the input node is the left child of
    its
//  parent, otherwise it returns FALSE.
//
//  BOOLEAN
```

```
//   MiIsLeftChild (
//       PRTL_SPLAY_LINKS Links
//       );
//

#define MiIsLeftChild(Links) (                                  \
    (RtlLeftChild(MiParent(Links)) == (PRTL_SPLAY_LINKS)(Links)) \
    )

//
//  The macro function IsRightChild takes as input a pointer to a
    splay link
//  in a tree and returns TRUE if the input node is the right child
    of its
//  parent, otherwise it returns FALSE.
//
//  BOOLEAN
//  MiIsRightChild (
//       PRTL_SPLAY_LINKS Links
//       );
//

#define MiIsRightChild(Links) (                                  \
    (RtlRightChild(MiParent(Links)) == (PRTL_SPLAY_LINKS)(Links)) \
    )



#if DBG

//
// Build a table of the best case efficiency of a balanced binary
    tree,
// holding the most possible nodes that can possibly be held in a
    binary
// tree with a given number of levels.  The answer is always (2**n) -
    1.
//
// (Used for debug only.)
```

```
//

ULONG MiBestCaseFill[33] = {
        0,          1,          3,          7,
        0xf,        0x1f,       0x3f,       0x7f,
        0xff,       0x1ff,      0x3ff,      0x7ff,
        0xfff,      0x1fff,     0x3fff,     0x7fff,
        0xffff,     0x1ffff,    0x3ffff,    0x7ffff,
        0xfffff,    0x1fffff,   0x3fffff,   0x7fffff,
        0xffffff,   0x1ffffff,  0x3ffffff,  0x7ffffff,
        0xfffffff,  0x1fffffff, 0x3fffffff, 0x7fffffff,
        0xffffffff
};

//
// Build a table of the worst case efficiency of a balanced binary tree,
// holding the fewest possible nodes that can possibly be contained in a
// balanced binary tree with the given number of levels.  After the first
// two levels, each level n is obviously occupied by a root node, plus
// one subtree the size of level n-1, and another subtree which is the
// size of n-2, i.e.:
//
//      MiWorstCaseFill[n] = 1 + MiWorstCaseFill[n-1] + MiWorstCaseFill[n-2]
//
// The efficiency of a typical balanced binary tree will normally fall
// between the two extremes, typically closer to the best case.  Note
// however that even with the worst case, it only takes 32 compares to
// find an element in a worst case tree populated with ~3.5M nodes.
//
// Unbalanced trees and splay trees, on the other hand, can and will sometimes
```

```
// degenerate to a straight line, requiring on average n/2 compares
   to
// find a node.
//
// A specific case is one where the nodes are inserted in collated
   order.
// In this case an unbalanced or a splay tree will generate a
   straight
// line, yet the balanced binary tree will always create a perfectly
// balanced tree (best-case fill) in this situation.
//
// (Used for debug only.)
//

ULONG MiWorstCaseFill[33] = {
        0,          1,          2,          4,
        7,          12,         20,         33,
        54,         88,         143,        232,
        376,        609,        986,        1596,
        2583,       4180,       6764,       10945,
        17710,      28656,      46367,      75024,
        121392,     196417,     317810,     514228,
        832039,     1346268,    2178308,    3524577,
        5702886
};

#endif


TABLE_SEARCH_RESULT
MiFindNodeOrParent (
    IN PMM_AVL_TABLE Table,
    IN ULONG_PTR StartingVpn,
    OUT PMMADDRESS_NODE *NodeOrParent
    )

/*++

Routine Description:
```

```

    This routine is used by all of the routines of the generic
    table package to locate the a node in the tree.  It will
    find and return (via the NodeOrParent parameter) the node
    with the given key, or if that node is not in the tree it
    will return (via the NodeOrParent parameter) a pointer to
    the parent.

Arguments:

    Table - The generic table to search for the key.

    StartingVpn - The starting virtual page number.

    NodeOrParent - Will be set to point to the node containing the
                   the key or what should be the parent of the node
                   if it were in the tree.  Note that this will *NOT*
                   be set if the search result is TableEmptyTree.

Return Value:

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

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

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

                          TableInsertAsRight: Node with key was not
```

```
                                            found.
                                                        NodeOrParent points to
                                                           what would
                                                        be parent.  The node
                                                           would be
                                                        the right child.

Environment:

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

--*/

{
#if DBG
    ULONG NumberCompares = 0;
#endif
    PMMADDRESS_NODE Child;
    PMMADDRESS_NODE NodeToExamine;

    if (Table->NumberGenericTableElements == 0) {
        return TableEmptyTree;
    }

    NodeToExamine = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;

    do {

        //
        // Make sure the depth of tree is correct.
        //

       // ASSERT(++NumberCompares <= Table->DepthOfTree);

        //
        // Compare the buffer with the key in the tree element.
        //

        if (StartingVpn < NodeToExamine->StartingVpn) {
```

```c

            Child = NodeToExamine->LeftChild;

            if (Child != NULL) {
                NodeToExamine = Child;
            }
            else {

                //
                // Node is not in the tree.  Set the output
                // parameter to point to what would be its
                // parent and return which child it would be.
                //

                *NodeOrParent = NodeToExamine;
                return TableInsertAsLeft;
            }
        }
        else if (StartingVpn <= NodeToExamine->EndingVpn) {

            //
            // This is the node.
            //

            *NodeOrParent = NodeToExamine;
            return TableFoundNode;
        }
        else {

            Child = NodeToExamine->RightChild;

            if (Child != NULL) {
                NodeToExamine = Child;
            }
            else {

                //
                // Node is not in the tree.  Set the output
                // parameter to point to what would be its
```

```
                    // parent and return which child it would be.
                    //

                    *NodeOrParent = NodeToExamine;
                    return TableInsertAsRight;
               }
          }

     } while (TRUE);
}


PMMADDRESS_NODE
MiCheckForConflictingNode (
     IN ULONG_PTR StartVpn,
     IN ULONG_PTR EndVpn,
     IN PMM_AVL_TABLE Table
     )

/*++

Routine Description:

     The function determines if any addresses between a given starting
          and
     ending address is contained within a virtual address descriptor.

Arguments:

     StartVpn - Supplies the virtual address to locate a containing
                              descriptor.

     EndVpn - Supplies the virtual address to locate a containing
                              descriptor.

Return Value:

     Returns a pointer to the first conflicting virtual address
         descriptor
```

```
    if one is found, otherwise a NULL value is returned.

--*/

{
    PMMADDRESS_NODE Node;

    if (Table->NumberGenericTableElements == 0) {
        return NULL;
    }

    Node = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;
    ASSERT (Node != NULL);

    do {

        if (Node == NULL) {
            return NULL;
        }

        if (StartVpn > Node->EndingVpn) {
            Node = Node->RightChild;
        }
        else if (EndVpn < Node->StartingVpn) {
            Node = Node->LeftChild;
        }
        else {

            //
            // The starting address is less than or equal to the end VA
            // and the ending address is greater than or equal to the
            // start va. Return this node.
            //

            return Node;
        }

    } while (TRUE);
```

```
}


PMMADDRESS_NODE
FASTCALL
MiGetFirstNode (
    IN PMM_AVL_TABLE Table
    )

/*++

Routine Description:

    This function locates the virtual address descriptor which contains
    the address range which logically is first within the address space.

Arguments:

    None.

Return Value:

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

--*/

{
    PMMADDRESS_NODE First;

    if (Table->NumberGenericTableElements == 0) {
        return NULL;
    }

    First = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;

```

```
    ASSERT (First != NULL);

    while (First->LeftChild != NULL) {
        First = First->LeftChild;
    }

    return First;
}


VOID
MiPromoteNode (
    IN PMMADDRESS_NODE C
    )

/*++

Routine Description:

    This routine performs the fundamental adjustment required for balancing
    the binary tree during insert and delete operations.  Simply put, the
    designated node is promoted in such a way that it rises one level in
    the tree and its parent drops one level in the tree, becoming now the
    child of the designated node.  Generally the path length to the subtree
    "opposite" the original parent.  Balancing occurs as the caller chooses
    which nodes to promote according to the balanced tree algorithms from
    Knuth.

    This is not the same as a splay operation, typically a splay "promotes"
    a designated node twice.

```

```
    Note that the pointer to the root node of the tree is assumed to be
    contained in a MMADDRESS_NODE structure itself, to allow the
    algorithms below to change the root of the tree without checking
    for special cases.  Note also that this is an internal routine,
    and the caller guarantees that it never requests to promote the
    root itself.

    This routine only updates the tree links; the caller must update
    the balance factors as appropriate.

Arguments:

    C - pointer to the child node to be promoted in the tree.

Return Value:

    None.

--*/

{
    PMMADDRESS_NODE P;
    PMMADDRESS_NODE G;

    //
    // Capture the current parent and grandparent (may be the root).
    //

    P = SANITIZE_PARENT_NODE (C->u1.Parent);
    G = SANITIZE_PARENT_NODE (P->u1.Parent);

    //
    // Break down the promotion into two cases based upon whether C
    // is a left or right child.
    //

    if (P->LeftChild == C) {

```

```
            //
            // This promotion looks like this:
            //
            //          G            G
            //          |            |
            //          P            C
            //         / \    =>    / \
            //        C   z        x   P
            //       / \              / \
            //      x   y            y   z
            //

            P->LeftChild = C->RightChild;

            if (P->LeftChild != NULL) {

                P->LeftChild->u1.Parent = MI_MAKE_PARENT (P, P->LeftChild->u1.Balance);
            }

            C->RightChild = P;

            //
            // Fall through to update parent and G <-> C relationship in
            // common code.
            //

        }
        else {

            ASSERT(P->RightChild == C);

            //
            // This promotion looks like this:
            //
            //        G                G
            //        |                |
            //        P                C
            //       / \     =>       / \
```

```
            //        x    C              P    z
            //            / \            / \
            //           y   z          x   y
            //

            P->RightChild = C->LeftChild;

            if (P->RightChild != NULL) {
                P->RightChild->u1.Parent = MI_MAKE_PARENT (P, P->
                    RightChild->u1.Balance);
            }

            C->LeftChild = P;
        }

        //
        // Update parent of P, for either case above.
        //

        P->u1.Parent = MI_MAKE_PARENT (C, P->u1.Balance);

        //
        // Finally update G <-> C links for either case above.
        //

        if (G->LeftChild == P) {
            G->LeftChild = C;
        }
        else {
            ASSERT(G->RightChild == P);
            G->RightChild = C;
        }
        C->u1.Parent = MI_MAKE_PARENT (G, C->u1.Balance);
}


ULONG
MiRebalanceNode (
    IN PMMADDRESS_NODE S
```

```
    )

/*++

Routine Description:

    This routine performs a rebalance around the input node S, for which the
    Balance factor has just effectively become +2 or -2.  When called, the
    Balance factor still has a value of +1 or -1, but the respective longer
    side has just become one longer as the result of an insert or delete
    operation.

    This routine effectively implements steps A7.iii (test for Case 1 or
    Case 2) and steps A8 and A9 of Knuth's balanced insertion algorithm,
    plus it handles Case 3 identified in the delete section, which can
    only happen on deletes.

    The trick is, to convince yourself that while traveling from the
    insertion point at the bottom of the tree up, that there are only
    these two cases, and that when traveling up from the deletion point,
    that there are just these three cases.  Knuth says it is obvious!

Arguments:

    S - pointer to the node which has just become unbalanced.

Return Value:

    TRUE if Case 3 was detected (causes delete algorithm to terminate).

```

```
Environment:

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

--*/

{
    PMMADDRESS_NODE R, P;
    SCHAR a;

    PRINT("rebalancing node %p bal=%x start=%x end=%x\n",
                    S,
                    S->u1.Balance,
                    S->StartingVpn,
                    S->EndingVpn);

    //
    // The parent node is never the argument node.
    //

    ASSERT (SANITIZE_PARENT_NODE(S->u1.Parent) != S);

    //
    // Capture which side is unbalanced.
    //

    a = (SCHAR) S->u1.Balance;

    if (a == +1) {
        R = S->RightChild;
    }
    else {
        R = S->LeftChild;
    }

    //
    // If the balance of R and S are the same (Case 1 in Knuth) then a single
    // promotion of R will do the single rotation.  (Step A8, A10)
```

```
    //
    // Here is a diagram of the Case 1 transformation, for a == +1 (a mirror
    // image transformation occurs when a == -1), and where the subtree
    // heights are h and h+1 as shown (++ indicates the node out of balance):
    //
    //                  |                   |
    //                  S++                 R
    //                 / \                 / \
    //               (h)  R+     ==>      S  (h+1)
    //                   / \             / \
    //                 (h) (h+1)       (h) (h)
    //
    // Note that on an insert we can hit this case by inserting an item in the
    // right subtree of R.  The original height of the subtree before the insert
    // was h+2, and it is still h+2 after the rebalance, so insert rebalancing
    // may terminate.
    //
    // On a delete we can hit this case by deleting a node from the left subtree
    // of S.  The height of the subtree before the delete was h+3, and after the
    // rebalance it is h+2, so rebalancing must continue up the tree.
    //

    if ((SCHAR) R->u1.Balance == a) {

        MiPromoteNode (R);
        R->u1.Balance = 0;
        S->u1.Balance = 0;

        return FALSE;
    }

```

```
//
// Otherwise, we have to promote the appropriate child of R twice (Case 2
// in Knuth).  (Step A9, A10)
//
// Here is a diagram of the Case 2 transformation, for a == +1 (a mirror
// image transformation occurs when a == -1), and where the subtree
// heights are h and h-1 as shown.  There are actually two minor subcases,
// differing only in the original balance of P (++ indicates the node out
// of balance).
//
//                   |                   |
//                  S++                  P
//                  / \                 / \
//                 /   \               /   \
//                /     \             /     \
//             (h)       R-   ==>   S-       R
//                      / \        / \      / \
//                    P+   (h)   (h)(h-1)(h) (h)
//                   / \
//               (h-1) (h)
//
//
//                   |                   |
//                  S++                  P
//                  / \                 / \
//                 /   \               /   \
//                /     \             /     \
//             (h)       R-   ==>   S        R+
//                      / \        / \      / \
//                    P-   (h)   (h) (h)(h-1)(h)
//                   / \
//                 (h) (h-1)
//
// Note that on an insert we can hit this case by inserting an
```

```
        item in the
    // left subtree of R.  The original height of the subtree before
        the insert
    // was h+2, and it is still h+2 after the rebalance, so insert
        rebalancing
    // may terminate.
    //
    // On a delete we can hit this case by deleting a node from the
        left subtree
    // of S.  The height of the subtree before the delete was h+3,
        and after the
    // rebalance it is h+2, so rebalancing must continue up the tree.
    //

    if ((SCHAR) R->u1.Balance == -a) {

        //
        // Pick up the appropriate child P for the double rotation (
            Link(-a,R)).
        //

        if (a == 1) {
            P = R->LeftChild;
        }
        else {
            P = R->RightChild;
        }

        //
        // Promote him twice to implement the double rotation.
        //

        MiPromoteNode (P);
        MiPromoteNode (P);

        //
        // Now adjust the balance factors.
        //

```

```
            S->u1.Balance = 0;
            R->u1.Balance = 0;
            if ((SCHAR) P->u1.Balance == a) {
                PRINT("REBADJ A: Node %p, Bal %x -> %x\n", S, S->u1.
                    Balance, -a);
                COUNT_BALANCE_MAX ((SCHAR)-a);
                S->u1.Balance = (ULONG_PTR) -a;
            }
            else if ((SCHAR) P->u1.Balance == -a) {
                PRINT("REBADJ B: Node %p, Bal %x -> %x\n", R, R->u1.
                    Balance, a);
                COUNT_BALANCE_MAX ((SCHAR)a);
                R->u1.Balance = (ULONG_PTR) a;
            }

            P->u1.Balance = 0;
            return FALSE;
        }

        //
        // Otherwise this is Case 3 which can only happen on Delete (
            identical
        // to Case 1 except R->u1.Balance == 0).  We do a single rotation
            , adjust
        // the balance factors appropriately, and return TRUE.  Note that
             the
        // balance of S stays the same.
        //
        // Here is a diagram of the Case 3 transformation, for a == +1 (a
             mirror
        // image transformation occurs when a == -1), and where the
            subtree
        // heights are h and h+1 as shown (++ indicates the node out of
            balance):
        //
        //                  |                      |
        //                 S++                    R-
        //                / \                    / \
        //             (h)   R       ==>       S+  (h+1)
```

```
    //                         / \               / \
    //                    (h+1)(h+1)          (h)  (h+1)
    //
    // This case can not occur on an insert, because it is impossible for
    // a single insert to balance R, yet somehow grow the right subtree of
    // S at the same time.  As we move up the tree adjusting balance factors
    // after an insert, we terminate the algorithm if a node becomes balanced,
    // because that means the subtree length did not change!
    //
    // On a delete we can hit this case by deleting a node from the left
    // subtree of S.  The height of the subtree before the delete was h+3,
    // and after the rebalance it is still h+3, so rebalancing may terminate
    // in the delete path.
    //

    MiPromoteNode (R);
    PRINT("REBADJ C: Node %p, Bal %x -> %x\n", R, R->u1.Balance, -a);
    COUNT_BALANCE_MAX ((SCHAR)-a);
    R->u1.Balance = -a;
    return TRUE;
}


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

/*++

```

```
Routine Description:

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

Arguments:

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

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

Return Value:

    None.

Environment:

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

--*/

{
   PMMADDRESS_NODE rebalance;
    Table->NumberGenericTableElements -= 1;
    rebalance = __rb_erase_augmented(NodeToDelete, Table);
    if (rebalance!=&Table->BalancedRoot&&rebalance)
        __rb_erase_color(rebalance, Table);
```

```
}


PMMADDRESS_NODE
MiRealSuccessor (
    IN PMMADDRESS_NODE Links
    )

/*++

Routine Description:

    This function takes as input a pointer to a balanced link
    in a tree and returns a pointer to the successor of the input
       node within
    the entire tree.  If there is not a successor, the return value
       is NULL.

Arguments:

    Links - Supplies a pointer to a balanced link in a tree.

Return Value:

    PMMADDRESS_NODE - returns a pointer to the successor in the
       entire tree

--*/

{
    PMMADDRESS_NODE Ptr;

    /*
        First check to see if there is a right subtree to the input
           link
        if there is then the real successor is the left most node in
        the right subtree.  That is find and return S in the
           following diagram

```

```
                  Links
                     \
                      .
                     .
                    .
                   /
                  S
                   \
    */

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

        while (Ptr->LeftChild != NULL) {
            Ptr = Ptr->LeftChild;
        }

        return Ptr;
    }

    /*
        We do not have a right child so check to see if have a parent and if
        so find the first ancestor that we are a left decendant of. That
        is find and return S in the following diagram

                          S
                         /
                        .
                         .
                          .
                         Links

        Note that this code depends on how the BalancedRoot is initialized,
        which is Parent points to self, and the RightChild points to an
        actual node which is the root of the tree, and LeftChild does not
```

```
            point to self.
        */

        Ptr = Links;
        while (MiIsRightChild(Ptr)) {
            Ptr = SANITIZE_PARENT_NODE (Ptr->u1.Parent);
        }

        if (MiIsLeftChild(Ptr)) {
            return SANITIZE_PARENT_NODE (Ptr->u1.Parent);
        }

        //
        // Otherwise we are do not have a real successor so we simply
            return NULL.
        //
        // This can only occur when we get back to the root, and we can
            tell
        // that since the Root is its own parent.
        //

        ASSERT (SANITIZE_PARENT_NODE(Ptr->u1.Parent) == Ptr);

        return NULL;
}


PMMADDRESS_NODE
MiRealPredecessor (
        IN PMMADDRESS_NODE Links
        )

/*++

Routine Description:

        The RealPredecessor function takes as input a pointer to a
            balanced link
        in a tree and returns a pointer to the predecessor of the input
```

```
        node
    within the entire tree.  If there is not a predecessor, the return value
    is NULL.

Arguments:

    Links - Supplies a pointer to a balanced link in a tree.

Return Value:

    PMMADDRESS_NODE - returns a pointer to the predecessor in the entire tree

--*/

{
    PMMADDRESS_NODE Ptr;
    PMMADDRESS_NODE Parent;
    PMMADDRESS_NODE GrandParent;

    /*
      First check to see if there is a left subtree to the input link
      if there is then the real predecessor is the right most node in
      the left subtree.  That is find and return P in the following diagram

                    Links
                     /
                    .
                     .
                      .
                       P
                      /
    */

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

        while (Ptr->RightChild != NULL) {
```

```
            Ptr = Ptr->RightChild;
        }

        return Ptr;

    }

    /*
      We do not have a left child so check to see if have a parent and if
      so find the first ancestor that we are a right decendant of. That
      is find and return P in the following diagram

                          P
                           \
                            .
                           .
                          .
                       Links

        Note that this code depends on how the BalancedRoot is initialized,
        which is Parent points to self, and the RightChild points to an
        actual node which is the root of the tree.
    */

    Ptr = Links;
    while (MiIsLeftChild(Ptr)) {
        Ptr = SANITIZE_PARENT_NODE (Ptr->u1.Parent);
    }

    if (MiIsRightChild(Ptr)) {
        Parent = SANITIZE_PARENT_NODE (Ptr->u1.Parent);
        GrandParent = SANITIZE_PARENT_NODE (Parent->u1.Parent);
        if (GrandParent != Parent) {
            return Parent;
        }
```

```
    }

    //
    // Otherwise we are do not have a real predecessor so we simply
       return
    // NULL.
    //

    return NULL;
}


VOID
MiInitializeVadTableAvl (
    IN PMM_AVL_TABLE Table
    )

/*++

Routine Description:

    This routine initializes a table.

Arguments:

    Table - Pointer to the generic table to be initialized.

Return Value:

    None.

--*/

{

#if DBG
    ULONG i;

    for (i = 2; i < 33; i += 1) {
```

```
        ASSERT(MiWorstCaseFill[i] == (1 + MiWorstCaseFill[i - 1] +
            MiWorstCaseFill[i - 2]));
    }
#endif

    //
    // Initialize each field in the argument Table.
    //

    RtlZeroMemory (Table, sizeof(MM_AVL_TABLE));

    Table->BalancedRoot.u1.Parent = MI_MAKE_PARENT (&Table->
        BalancedRoot, 0);
}


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

/*++

Routine Description:

    This function inserts a new element in a table.

Arguments:

    NodeToInsert - The initialized address node to insert.

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

Return Value:

    None.

```

```
Environment:

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

--*/

{
   PMMADDRESS_NODE NodeOrParent;
    PMMADDRESS_NODE parent,gparent,tmp;
    TABLE_SEARCH_RESULT SearchResult;

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

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

    Table->NumberGenericTableElements += 1;

    //
    // Insert the newer node in the tree.
    //

    if (SearchResult == TableEmptyTree)
    {
        Table->BalancedRoot.RightChild = NodeToInsert;
        rb_set_parent(NodeToInsert,&Table->BalancedRoot);
    }
    else
    {
        if (SearchResult == TableInsertAsLeft)
        {
            NodeOrParent->LeftChild = NodeToInsert;
        }
        else
        {
            NodeOrParent->RightChild = NodeToInsert;
        }
```

```
        rb_set_parent(NodeToInsert,NodeOrParent);
    }
    rb_set_red(NodeToInsert);
    parent=rb_parent(NodeToInsert);
    while(1)  {
        if (parent==&Table->BalancedRoot) {
            Table->BalancedRoot.RightChild = NodeToInsert;
            rb_set_parent_color(NodeToInsert,&Table->BalancedRoot,rb_black);
            break;
        } else if (rb_is_black(parent))
            break;
        gparent = rb_parent(parent);
        tmp = gparent->RightChild;

        if (parent != tmp) {    /* parent == gparent->rb_left */
            if (tmp && rb_is_red(tmp)) {
                /*
                 * Case 1 - color flips
                 *
                 *       G            g
                 *      / \          / \
                 *     p   u  -->   P   U
                 *    /            /
                 *   n            n
                 *
                 * However, since g's parent might be red, and
                 * 4) does not allow this, we need to recurse
                 * at g.
                 */
                rb_set_parent_color(tmp, gparent, rb_black);
                rb_set_parent_color(parent, gparent, rb_black);
                NodeToInsert = gparent;
                parent = rb_parent(NodeToInsert);
                rb_set_parent_color(NodeToInsert, parent, rb_red);
                continue;
            }

            tmp = parent->RightChild;
```

```
            if (NodeToInsert == tmp) {
                /*
                 * Case 2 - left rotate at parent
                 *
                 *        G               G
                 *       / \             / \
                 *      p   U  -->      n   U
                 *       \             /
                 *        n           p
                 *
                 * This still leaves us in violation of 4), the
                 * continuation into Case 3 will fix that.
                 */
                parent->RightChild = tmp = NodeToInsert->LeftChild;
                NodeToInsert->LeftChild = parent;
                if (tmp)
                    rb_set_parent_color(tmp, parent,rb_black);
                rb_set_parent_color(parent, NodeToInsert, rb_red);
                parent = NodeToInsert;
                tmp = NodeToInsert->RightChild;
            }

            /*
             * Case 3 - right rotate at gparent
             *
             *          G             P
             *         / \           / \
             *        p   U  -->    n   g
             *       /                   \
             *      n                     U
             */
            gparent->LeftChild= tmp;  /* == parent->rb_right */
            parent->RightChild= gparent;
            if (tmp)
                rb_set_parent_color(tmp, gparent, rb_black);
            __rb_rotate_set_parents(gparent, parent, Table, rb_red);
            break;
        } else {
            tmp = gparent->LeftChild;
```

```
            if (tmp && rb_is_red(tmp)) {
                /* Case 1 - color flips */
                rb_set_parent_color(tmp, gparent, rb_black);
                rb_set_parent_color(parent, gparent, rb_black);
                NodeToInsert = gparent;
                parent = rb_parent(NodeToInsert);
                rb_set_parent_color(NodeToInsert, parent, rb_red);
                continue;
            }

            tmp = parent->LeftChild;
            if (NodeToInsert == tmp) {
                /* Case 2 - right rotate at parent */
                parent->LeftChild = tmp = NodeToInsert->RightChild;
                NodeToInsert->RightChild = parent;
                if (tmp)
                    rb_set_parent_color(tmp, parent,
                    rb_black);
                rb_set_parent_color(parent, NodeToInsert, rb_red);
                parent = NodeToInsert;
                tmp = NodeToInsert->LeftChild;
            }

            /* Case 3 - left rotate at gparent */
            gparent->RightChild = tmp;  /* == parent->rb_left */
            parent->LeftChild = gparent;
            if (tmp)
                rb_set_parent_color(tmp, gparent, rb_black);
            __rb_rotate_set_parents(gparent, parent, Table, rb_red);
            break;
        }
    }
    return;
}


PVOID
MiEnumerateGenericTableWithoutSplayingAvl (
    IN PMM_AVL_TABLE Table,
```

```
    IN PVOID *RestartKey
    )

/*++

Routine Description:

    The function EnumerateGenericTableWithoutSplayingAvl will return
        to the
    caller one-by-one the elements of of a table.  The return value
        is a
    pointer to the user defined structure associated with the element
        .
    The input parameter RestartKey indicates if the enumeration
        should
    start from the beginning or should return the next element.  If
        the
    are no more new elements to return the return value is NULL.  As
        an
    example of its use, to enumerate all of the elements in a table
        the
    user would write:

        *RestartKey = NULL;

        for (ptr = EnumerateGenericTableWithoutSplayingAvl(Table, &
            RestartKey);
             ptr != NULL;
             ptr = EnumerateGenericTableWithoutSplayingAvl(Table, &
                RestartKey)) {
                :
        }

Arguments:

    Table - Pointer to the generic table to enumerate.

    RestartKey - Pointer that indicates if we should restart or
        return the next
```

```
                    element.  If the contents of RestartKey is NULL, the
                        search
                    will be started from the beginning.

Return Value:

    PVOID - Pointer to the user data.

--*/

{
    PMMADDRESS_NODE NodeToReturn;

    if (Table->NumberGenericTableElements == 0) {

        //
        // Nothing to do if the table is empty.
        //

        return NULL;

    }

    //
    // If the restart flag is true then go to the least element
    // in the tree.
    //

    if (*RestartKey == NULL) {

        //
        // Loop until we find the leftmost child of the root.
        //

        for (NodeToReturn = Table->BalancedRoot.RightChild;
             NodeToReturn->LeftChild;
             NodeToReturn = NodeToReturn->LeftChild) {

            NOTHING;
```

```
        }

        *RestartKey = NodeToReturn;

    }
    else {

        //
        // The caller has passed in the previous entry found
        // in the table to enable us to continue the search.  We call
        // RealSuccessor to step to the next element in the tree.
        //

        NodeToReturn = MiRealSuccessor (*RestartKey);

        if (NodeToReturn) {
            *RestartKey = NodeToReturn;
        }
    }

    //
    // Return the found element.
    //

    return NodeToReturn;
}


PMMADDRESS_NODE
FASTCALL
MiGetNextNode (
    IN PMMADDRESS_NODE Node
    )

/*++

Routine Description:

    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.
```

```
            //

            if (Parent->LeftChild == Next) {
                return Parent;
            }

            Next = Parent;

        } while (TRUE);
    }

    //
    // A right child exists, locate the left most child of that right
        child.
    //

    Next = Next->RightChild;

    do {

        Left = Next->LeftChild;

        if (Left == NULL) {
            break;
        }

        Next = Left;

    } while (TRUE);

    return Next;

}

PMMADDRESS_NODE
FASTCALL
MiGetPreviousNode (
    IN PMMADDRESS_NODE Node
    )
```

```c

/*++

Routine Description:

    This function locates the virtual address descriptor which
        contains
    the address range which logically precedes the specified virtual
    address descriptor.

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 Previous;
    PMMADDRESS_NODE Parent;

    Previous = Node;

    if (Previous->LeftChild == NULL) {

        ASSERT (Previous->u1.Parent != NULL);

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

        while (Parent != Previous) {

            //
            // Locate the first ancestor of this node of which this
            // node is the right child of and return that node as the
```

```
                // Previous element.
                //

                if (Parent->RightChild == Previous) {

                    if (Parent == SANITIZE_PARENT_NODE (Parent->u1.Parent
                        )) {
                        return NULL;
                    }

                    return Parent;
                }

                Previous = Parent;
                Parent = SANITIZE_PARENT_NODE (Previous->u1.Parent);
            }
            return NULL;
        }

        //
        // A left child exists, locate the right most child of that left
            child.
        //

        Previous = Previous->LeftChild;

        while (Previous->RightChild != NULL) {
            Previous = Previous->RightChild;
        }

        return Previous;
}


PMMADDRESS_NODE
FASTCALL
MiLocateAddressInTree (
        IN ULONG_PTR Vpn,
        IN PMM_AVL_TABLE Table
```

```
    )

/*++

Routine Description:

    The function locates the virtual address descriptor which
        describes
    a given address.

Arguments:

    Vpn - Supplies the virtual page number to locate a descriptor for
        .

Return Value:

    Returns a pointer to the virtual address descriptor which
        contains
    the supplied virtual address or NULL if none was located.

--*/

{
    PVOID NodeOrParent;
    TABLE_SEARCH_RESULT SearchResult;

    //
    // Lookup the element and save the result.
    //

    SearchResult = MiFindNodeOrParent (Table,
                                       Vpn,
                                       (PMMADDRESS_NODE *) &
                                           NodeOrParent);

    if (SearchResult == TableFoundNode) {

        //
```

```
            // Return the VAD.
            //

            return (PMMADDRESS_NODE) NodeOrParent;
        }

        return NULL;
}


NTSTATUS
MiFindEmptyAddressRangeInTree (
        IN SIZE_T SizeOfRange,
        IN ULONG_PTR Alignment,
        IN PMM_AVL_TABLE Table,
        OUT PMMADDRESS_NODE *PreviousVad,
        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.

Arguments:

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

        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.

        PreviousVad - Supplies the Vad which is before this the found
                      address range.

```

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

Return Value:

    NTSTATUS.

--*/

{
    PMMADDRESS_NODE Node;
    PMMADDRESS_NODE NextNode;
    ULONG_PTR AlignmentVpn;
    ULONG_PTR SizeOfRangeVpn;

    AlignmentVpn = Alignment >> PAGE_SHIFT;

    //
    // Locate the node with the lowest starting address.
    //

    ASSERT (SizeOfRange != 0);
    SizeOfRangeVpn = (SizeOfRange + (PAGE_SIZE - 1)) >> PAGE_SHIFT;
    ASSERT (SizeOfRangeVpn != 0);

    if (Table->NumberGenericTableElements == 0) {
        *Base = MM_LOWEST_USER_ADDRESS;
        return STATUS_SUCCESS;
    }

    Node = Table->BalancedRoot.RightChild;

    while (Node->LeftChild != NULL) {
        Node = Node->LeftChild;
    }

    //
    // Check to see if a range exists between the lowest address VAD
    // and lowest user address.
```

```
    //

    if (Node->StartingVpn > MI_VA_TO_VPN (MM_LOWEST_USER_ADDRESS)) {

        if (SizeOfRangeVpn <
            (Node->StartingVpn - MI_VA_TO_VPN (MM_LOWEST_USER_ADDRESS
                ))) {

            *PreviousVad = NULL;
            *Base = MM_LOWEST_USER_ADDRESS;
            return STATUS_SUCCESS;
        }
    }

    do {

        NextNode = MiGetNextNode (Node);

        if (NextNode != NULL) {

            if (SizeOfRangeVpn <=
                ((ULONG_PTR)NextNode->StartingVpn -
                                MI_ROUND_TO_SIZE(1 + Node->EndingVpn,
                                                AlignmentVpn))) {

                //
                // Check to ensure that the ending address aligned
                    upwards
                // is not greater than the starting address.
                //

                if ((ULONG_PTR)NextNode->StartingVpn >
                        MI_ROUND_TO_SIZE(1 + Node->EndingVpn,
                                        AlignmentVpn)) {

                    *PreviousVad = Node;
                    *Base = (PVOID) MI_ROUND_TO_SIZE(
                                (ULONG_PTR)MI_VPN_TO_VA_ENDING(Node->
                                    EndingVpn),
```

```
                                                Alignment);
                        return STATUS_SUCCESS;
                    }
                }

            } else {

                //
                // No more descriptors, check to see if this fits into
                    the remainder
                // of the address space.
                //

                if ((((ULONG_PTR)Node->EndingVpn + MI_VA_TO_VPN(X64K)) <
                        MI_VA_TO_VPN (MM_HIGHEST_VAD_ADDRESS))
                            &&
                    (SizeOfRange <=
                        ((ULONG_PTR)MM_HIGHEST_VAD_ADDRESS -
                            (ULONG_PTR)MI_ROUND_TO_SIZE(
                            (ULONG_PTR)MI_VPN_TO_VA(Node->EndingVpn),
                                Alignment)))) {

                    *PreviousVad = Node;
                    *Base = (PVOID) MI_ROUND_TO_SIZE(
                                (ULONG_PTR)MI_VPN_TO_VA_ENDING(Node->
                                    EndingVpn),
                                    Alignment);
                    return STATUS_SUCCESS;
                }
                return STATUS_NO_MEMORY;
            }
            Node = NextNode;

    } while (TRUE);
}

NTSTATUS
MiFindEmptyAddressRangeDownTree (
    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:

```

```
    NTSTATUS.

--*/

{
    PMMADDRESS_NODE Node;
    PMMADDRESS_NODE PreviousNode;
    ULONG_PTR AlignedEndingVa;
    PVOID OptimalStart;
    ULONG_PTR OptimalStartVpn;
    ULONG_PTR HighestVpn;
    ULONG_PTR AlignmentVpn;

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

    ASSERT (Table != &MmSectionBasedRoot);

    SizeOfRange = MI_ROUND_TO_SIZE (SizeOfRange, PAGE_SIZE);

    if (((ULONG_PTR)HighestAddressToEndAt + 1) < SizeOfRange) {
        return STATUS_NO_MEMORY;
    }

    ASSERT (HighestAddressToEndAt != NULL);
    ASSERT (HighestAddressToEndAt <= (PVOID)((ULONG_PTR)
        MM_HIGHEST_VAD_ADDRESS + 1));

    HighestVpn = MI_VA_TO_VPN (HighestAddressToEndAt);

    //
    // Locate the Node with the highest starting address.
    //

    OptimalStart = (PVOID)(MI_ALIGN_TO_SIZE(
                                    (((ULONG_PTR)HighestAddressToEndAt + 1) -
```

```
                                   SizeOfRange),
                                 Alignment));

    if (Table->NumberGenericTableElements == 0) {

        //
        // The tree is empty, any range is okay.
        //

        *Base = OptimalStart;
        return STATUS_SUCCESS;
    }

    Node = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;

    //
    // See if an empty slot exists to hold this range, locate the
       largest
    // element in the tree.
    //

    while (Node->RightChild != NULL) {
        Node = Node->RightChild;
    }

    //
    // Check to see if a range exists between the highest address VAD
    // and the highest address to end at.
    //

    AlignedEndingVa = (ULONG_PTR)MI_ROUND_TO_SIZE ((ULONG_PTR)
       MI_VPN_TO_VA_ENDING (Node->EndingVpn),
                                                   Alignment);

    if (AlignedEndingVa < (ULONG_PTR)HighestAddressToEndAt) {

        if ( SizeOfRange < ((ULONG_PTR)HighestAddressToEndAt -
           AlignedEndingVa)) {

```

```
                *Base = MI_ALIGN_TO_SIZE(
                                        ((ULONG_PTR)HighestAddressToEndAt - SizeOfRange),
                                        Alignment);
                return STATUS_SUCCESS;
            }
        }

        //
        // Walk the tree backwards looking for a fit.
        //

        OptimalStartVpn = MI_VA_TO_VPN (OptimalStart);
        AlignmentVpn = MI_VA_TO_VPN (Alignment);

        do {

            PreviousNode = MiGetPreviousNode (Node);

            if (PreviousNode != NULL) {

                //
                // Is the ending Va below the top of the address to end at.
                //

                if (PreviousNode->EndingVpn < OptimalStartVpn) {
                    if ((SizeOfRange >> PAGE_SHIFT) <=
                        ((ULONG_PTR)Node->StartingVpn -
                        (ULONG_PTR)MI_ROUND_TO_SIZE(1 + PreviousNode->EndingVpn,
                                                    AlignmentVpn))) {

                        //
                        // See if the optimal start will fit between these
                        // two VADs.
                        //

```

```
                        if ((OptimalStartVpn > PreviousNode->EndingVpn) &&
                            (HighestVpn < Node->StartingVpn)) {
                            *Base = OptimalStart;
                            return STATUS_SUCCESS;
                        }

                        //
                        // Check to ensure that the ending address aligned upwards
                        // is not greater than the starting address.
                        //

                        if ((ULONG_PTR)Node->StartingVpn >
                                (ULONG_PTR)MI_ROUND_TO_SIZE(1 + PreviousNode->EndingVpn,
                                                            AlignmentVpn)) {

                            *Base = MI_ALIGN_TO_SIZE(
                                                (ULONG_PTR)MI_VPN_TO_VA (Node->StartingVpn) - SizeOfRange,
                                                Alignment);
                            return STATUS_SUCCESS;
                        }
                    }
                }
            } else {

                //
                // No more descriptors, check to see if this fits into the remainder
                // of the address space.
                //

                if (Node->StartingVpn > MI_VA_TO_VPN (MM_LOWEST_USER_ADDRESS)) {
                    if ((SizeOfRange >> PAGE_SHIFT) <=
                        ((ULONG_PTR)Node->StartingVpn - MI_VA_TO_VPN (
```

```
                              MM_LOWEST_USER_ADDRESS))) {

                    //
                    // See if the optimal start will fit between these
                    // two VADs.
                    //

                    if (HighestVpn < Node->StartingVpn) {
                        *Base = OptimalStart;
                        return STATUS_SUCCESS;
                    }

                    *Base = MI_ALIGN_TO_SIZE(
                                    (ULONG_PTR)MI_VPN_TO_VA (Node->StartingVpn) - SizeOfRange,
                                    Alignment);
                    return STATUS_SUCCESS;
                }
            }
            return STATUS_NO_MEMORY;
        }
        Node = PreviousNode;

    } while (TRUE);
}


NTSTATUS
MiFindEmptyAddressRangeDownBasedTree (
    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 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;
```

```
    ULONG_PTR OptimalStart;

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

    ASSERT (Table == &MmSectionBasedRoot);

    SizeOfRange = MI_ROUND_TO_SIZE (SizeOfRange, PAGE_SIZE);

    if (((ULONG_PTR)HighestAddressToEndAt + 1) < SizeOfRange) {
        return STATUS_NO_MEMORY;
    }

    ASSERT (HighestAddressToEndAt != NULL);
    ASSERT (HighestAddressToEndAt <= (PVOID)((ULONG_PTR)
        MM_HIGHEST_VAD_ADDRESS + 1));

    //
    // Locate the node with the highest starting address.
    //

    OptimalStart = (ULONG_PTR) MI_ALIGN_TO_SIZE (
                              (((ULONG_PTR)HighestAddressToEndAt + 1) -
                                  SizeOfRange),
                              Alignment);

    if (Table->NumberGenericTableElements == 0) {

        //
        // The tree is empty, any range is okay.
        //

        *Base = (PVOID) OptimalStart;
        return STATUS_SUCCESS;
    }

    Node = (PMMADDRESS_NODE) Table->BalancedRoot.RightChild;
```

```

    //
    // See if an empty slot exists to hold this range, locate the largest
    // element in the tree.
    //

    while (Node->RightChild != NULL) {
        Node = Node->RightChild;
    }

    //
    // Check to see if a range exists between the highest address VAD
    // and the highest address to end at.
    //

    AlignedEndingVa = MI_ROUND_TO_SIZE (Node->EndingVpn, Alignment);

    PRINT("search down0: %p %p %p\n", AlignedEndingVa, HighestAddressToEndAt, SizeOfRange);

    if ((AlignedEndingVa < (ULONG_PTR)HighestAddressToEndAt) &&
        (SizeOfRange < ((ULONG_PTR)HighestAddressToEndAt - AlignedEndingVa))) {

        *Base = MI_ALIGN_TO_SIZE(
                                ((ULONG_PTR)HighestAddressToEndAt - SizeOfRange),
                                Alignment);
        return STATUS_SUCCESS;
    }

    //
    // Walk the tree backwards looking for a fit.
    //

    do {

        PreviousNode = MiGetPreviousNode (Node);
```

```
        PRINT("search down1: %p %p %p %p\n", PreviousNode, Node,
            OptimalStart, Alignment);

        if (PreviousNode == NULL) {
            break;
        }

        //
        // Is the ending Va below the top of the address to end at.
        //

        if (PreviousNode->EndingVpn < OptimalStart) {

            if (SizeOfRange <= (Node->StartingVpn -
                    MI_ROUND_TO_SIZE(1 + PreviousNode->EndingVpn,
                        Alignment))) {

                //
                // See if the optimal start will fit between these
                    two VADs.
                //

                if ((OptimalStart > PreviousNode->EndingVpn) &&
                    ((ULONG_PTR) HighestAddressToEndAt < Node->
                        StartingVpn)) {
                    *Base = (PVOID) OptimalStart;
                    return STATUS_SUCCESS;
                }

                //
                // Check to ensure that the ending address aligned
                    upwards
                // is not greater than the starting address.
                //

                if (Node->StartingVpn >
                    MI_ROUND_TO_SIZE(1 + PreviousNode->EndingVpn,
                        Alignment)) {
```

```
                    *Base = MI_ALIGN_TO_SIZE (Node->StartingVpn -
                        SizeOfRange ,
                                                Alignment);

                    return STATUS_SUCCESS;
                }
            }
        }

        Node = PreviousNode;

    } while (TRUE);


    //
    // No more descriptors, check to see if this fits into the
       remainder
    // of the address space.
    //

    if (Node->StartingVpn > (ULONG_PTR) MM_LOWEST_USER_ADDRESS) {

        if (SizeOfRange <= (Node->StartingVpn - (ULONG_PTR)
            MM_LOWEST_USER_ADDRESS)) {

            //
            // See if the optimal start will fit between these two
               VADs.
            //

            if ((ULONG_PTR) HighestAddressToEndAt < Node->StartingVpn
                ) {
                *Base = (PVOID) OptimalStart;
                return STATUS_SUCCESS;
            }

            *Base = MI_ALIGN_TO_SIZE (Node->StartingVpn - SizeOfRange
                ,
```

```
                                              Alignment);

                return STATUS_SUCCESS;
            }
        }
        return STATUS_NO_MEMORY;
}

#if !defined (_USERMODE)

PMMVAD
FASTCALL
MiLocateAddress (
    IN PVOID VirtualAddress
    )

/*++

Routine Description:

    The function locates the virtual address descriptor which
        describes
    a given address.

Arguments:

    VirtualAddress - Supplies the virtual address to locate a
        descriptor for.

    Table - Supplies the table describing the tree.

Return Value:

    Returns a pointer to the virtual address descriptor which
        contains
    the supplied virtual address or NULL if none was located.

--*/

```

```
{
    PMMVAD FoundVad;
    ULONG_PTR Vpn;
    PMM_AVL_TABLE Table;
    TABLE_SEARCH_RESULT SearchResult;

    Table = &PsGetCurrentProcess ()->VadRoot;

    //
    // Note the NodeHint *MUST* be captured locally - see the
       synchronization
    // comment below for details.
    //

    FoundVad = (PMMVAD) Table->NodeHint;

    if (FoundVad == NULL) {
        return NULL;
    }

    Vpn = MI_VA_TO_VPN (VirtualAddress);

    if ((Vpn >= FoundVad->StartingVpn) && (Vpn <= FoundVad->EndingVpn
       )) {
        return FoundVad;
    }

    //
    // Lookup the element and save the result.
    //

    SearchResult = MiFindNodeOrParent (Table,
                                       Vpn,
                                       (PMMADDRESS_NODE *) &FoundVad)
                                           ;

    if (SearchResult != TableFoundNode) {
        return NULL;
    }
```

```

    ASSERT (FoundVad != NULL);

    ASSERT ((Vpn >= FoundVad->StartingVpn) && (Vpn <= FoundVad->EndingVpn));

    //
    // Note the NodeHint field update is not synchronized in all cases, ie:
    // some callers hold the address space mutex and others hold the working
    // set pushlock.  It is ok that the update is not synchronized - as long
    // as care is taken above that it is read into a local variable and then
    // referenced.  Because no VAD can be removed from the tree without holding
    // both the address space & working set.
    //

    Table->NodeHint = (PVOID) FoundVad;

    //
    // Return the VAD.
    //

    return FoundVad;
}
#endif

#if DBG
VOID
MiNodeTreeWalk (
    IN PMM_AVL_TABLE Table
    )
{
    PVOID RestartKey;
    PMMADDRESS_NODE NewNode;
    PMMADDRESS_NODE PrevNode;
```

```
    PMMADDRESS_NODE NextNode;

    RestartKey = NULL;

    do {

        NewNode = MiEnumerateGenericTableWithoutSplayingAvl (Table,
                                                              &
                                                                 RestartKey
                                                                 );

        if (NewNode == NULL) {
            break;
        }

        PrevNode = MiGetPreviousNode (NewNode);
        NextNode = MiGetNextNode (NewNode);

        PRINT ("Node %p %x %x\n",
                        NewNode,
                        NewNode->StartingVpn,
                        NewNode->EndingVpn);

        if (PrevNode != NULL) {
            PRINT ("\tPrevNode %p %x %x\n",
                        PrevNode,
                        PrevNode->StartingVpn,
                        PrevNode->EndingVpn);
        }

        if (NextNode != NULL) {
            PRINT ("\tNextNode %p %x %x\n",
                        NextNode,
                        NextNode->StartingVpn,
                        NextNode->EndingVpn);
        }

    } while (TRUE);

```

```
    PRINT ("NumberGenericTableElements = 0x%x, Depth = 0x%x\n",
        Table->NumberGenericTableElements,
        Table->DepthOfTree);

    return;
}
#endif

#if defined (_USERMODE)

MMADDRESS_NODE MiBalancedLinks;

MM_AVL_TABLE MiAvlTable;
MM_AVL_TABLE MmSectionBasedRoot;

ULONG DeleteRandom = 1;

#if RANDOM
#define NUMBER_OF_VADS 32
#else
#define NUMBER_OF_VADS 4
#endif

int __cdecl
main(
int argc,
PCHAR   argv[]
)
{
    ULONG i;
    PVOID StartingAddress;
    PVOID EndingAddress;
    NTSTATUS Status;
    PMMADDRESS_NODE NewNode;
#if RANDOM
    PMMADDRESS_NODE PrevNode;
    ULONG RandomNumber = 0x99887766;
    ULONG_PTR DeleteVpn = 0;
#endif
```

```
    PMM_AVL_TABLE Table;
    SIZE_T CapturedRegionSize;

    UNREFERENCED_PARAMETER (argc);
    UNREFERENCED_PARAMETER (argv);

#if RANDOM
    Table = &MiAvlTable;
#else
    Table = &MmSectionBasedRoot;
#endif

    MiInitializeVadTableAvl (Table);

    for (i = 0; i < NUMBER_OF_VADS; i += 1) {
        NewNode = malloc (sizeof (MMADDRESS_NODE));
        ASSERT (((ULONG_PTR)NewNode & 0x3) == 0);

        if (NewNode == NULL) {
            PRINT ("Malloc failed\n");
            exit (1);
        }

        NewNode->u1.Parent = NULL;
        NewNode->LeftChild = NULL;
        NewNode->RightChild = NULL;
        NewNode->u1.Balance = 0;

#if RANDOM
        RandomNumber = RtlRandom (&RandomNumber);

        CapturedRegionSize = (SIZE_T) (RandomNumber & 0x1FFFFF);

        Status = MiFindEmptyAddressRangeInTree (CapturedRegionSize,
                                                64 * 1024,      //
                                                    align
                                                Table,
                                                &PrevNode,
                                                &StartingAddress);
```

```
#else
        CapturedRegionSize = 0x800000;

        Status = MiFindEmptyAddressRangeDownBasedTree (CapturedRegionSize,
                                                       (PVOID) 0x7f7effff, // highest addr
                                                       64 * 1024,          // align
                                                       Table,
                                                       &StartingAddress);
#endif

        if (!NT_SUCCESS (Status)) {
            PRINT ("Could not find empty addr range in tree for size %p\n", CapturedRegionSize);
            free (NewNode);
            continue;
        }

#if RANDOM
        EndingAddress = (PVOID)(((ULONG_PTR)StartingAddress +
                                 CapturedRegionSize - 1L) | (PAGE_SIZE - 1L));
#else
        EndingAddress = (PVOID)(((ULONG_PTR)StartingAddress +
                                 CapturedRegionSize - 1L));
#endif

        printf ("Inserting addr range in tree @ %p %p\n", StartingAddress, EndingAddress);

#if RANDOM
        NewNode->StartingVpn = MI_VA_TO_VPN (StartingAddress);
        NewNode->EndingVpn = MI_VA_TO_VPN (EndingAddress);
#else
        NewNode->StartingVpn = (ULONG_PTR) StartingAddress;
```

```
        NewNode->EndingVpn = (ULONG_PTR) EndingAddress;
#endif

        MiInsertNode (NewNode, Table);

#if RANDOM
        RandomNumber = RtlRandom (&RandomNumber);

        if (RandomNumber & 0x3) {
            DeleteVpn = NewNode->StartingVpn;
        }

        if (DeleteRandom && ((i & 0x3) == 0)) {
            NewNode = MiLocateAddressInTree (DeleteVpn, Table);
            printf ("Located node for random deletion - vpn %p @ %p\n
                ", DeleteVpn, NewNode);

            if (NewNode != NULL) {
                MiRemoveNode (NewNode, Table);
                printf ("Removed random node for vpn %p @ %p %p %p\n"
                    ,
                    DeleteVpn, NewNode, NewNode->StartingVpn, NewNode
                        ->EndingVpn);
            }
        }
#endif
        printf ("\n");
    }

    MiNodeTreeWalk (Table);

    NewNode = MiLocateAddressInTree (5, Table);
    printf ("Located node for vpn 5 @ %p\n", NewNode);

    if (NewNode != NULL) {
        MiRemoveNode (NewNode, Table);
        printf ("Removed node for vpn 5 @ %p\n", NewNode);
    }

```

```c
    NewNode = MiLocateAddressInTree (5, Table);
    printf("Located node for vpn 5 @ %p\n", NewNode);

    printf("all done, balmin=%x, balmax=%x\n", BalMin, BalMax);

    return 0;
}

#endif
```