操作系统实验 3-3
AVL 树转红黑树问题

马栩杰

2014011085

无 43 班

2016 年 12 月 4 日

1 问题描述

在 Windows 的虚拟内存管理中，将 VAD 组织成 AVL 树。VAD 树是一种平衡二叉树。

红黑树也是一种自平衡二叉查找树，在 Linux 2.6 及其以后版本的内核中，采用红黑树来维护内存块。

请尝试参考 Linux 源代码将 WRK 源代码中的 VAD 树由 AVL 树替换成红黑树。

2 Linux 中的红黑树

此时实验中，参考的 Linux 内核版本是 4.8，也就是目前最新的一个稳定版[1]。

在 Linux 项目中全局查找关键字 rbtree ，发现 Linux 的红黑树的实现在 `lib/rbtree.c` [2] 中，其接口定义在 `include/linux/rbtree.h` [3] 中。另外，还在项目中惊喜地发现了关于红黑树的文档 `Documentation/rbtree.txt` [4] 。

参考红黑树的文档以及代码，我们可以知道 Linux rbtree 实现的细节。

- 使用红黑树的每个数据结点都是一个包含 `rb_node` 的结构体

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

- 红黑树的结点 `rb_node` 定义包括结点的颜色和左右子结点

```
struct rb_node {
  unsigned long  __rb_parent_color;
  struct rb_node *rb_right;
```

[1]GitHub 上可以看到这个 Release 发布于 2016.10.03，迄今发布的更新的版本都是 RC 版

[2]`https://github.com/torvalds/linux/blob/master/lib/rbtree.c`

[3]`https://github.com/torvalds/linux/blob/master/include/linux/rbtree.h`

[4]`https://github.com/torvalds/linux/blob/master/Documentation/rbtree.txt`

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

- 红黑树还提供了一系列基本操作

```c
// 查找亲结点
#define rb_parent(r)   ((struct rb_node *)((r)->__rb_parent_color & ~3))

// 设置根结点
#define RB_ROOT  (struct rb_root) { NULL, }

// 结点存在性检测
#define  rb_entry(ptr, type, member) container_of(ptr, type, member)

// 空树检测
#define RB_EMPTY_ROOT(root)  (READ_ONCE((root)->rb_node) == NULL)

/* 'empty' nodes are nodes that are known not to be inserted in an rbtree */
#define RB_EMPTY_NODE(node)  \
  ((node)->__rb_parent_color == (unsigned long)(node))
#define RB_CLEAR_NODE(node)  \
  ((node)->__rb_parent_color = (unsigned long)(node))


// 颜色平衡操作
extern void rb_insert_color(struct rb_node *, struct rb_root *);
extern void rb_erase(struct rb_node *, struct rb_root *);

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

/* Postorder iteration - always visit the parent after its children */
extern struct rb_node *rb_first_postorder(const struct rb_root *);
extern struct rb_node *rb_next_postorder(const struct rb_node *);

/* Fast replacement of a single node without remove/rebalance/add/rebalance */
extern void rb_replace_node(struct rb_node *victim, struct rb_node *new,
          struct rb_root *root);
extern void rb_replace_node_rcu(struct rb_node *victim, struct rb_node *new,
```

```c
        struct rb_root *root);

// 插入新结点
static inline void rb_link_node(struct rb_node *node, struct rb_node *parent,
        struct rb_node **rb_link)
{
  node->__rb_parent_color = (unsigned long)parent;
  node->rb_left = node->rb_right = NULL;

  *rb_link = node;
}
```

- 另外在 include/linux/rbtree_augmented.h [5] 中也有一些更具体的红黑树访问接口

```c
#define __rb_parent(pc)    ((struct rb_node *)(pc & ~3))
#define __rb_color(pc)     ((pc) & 1)
#define __rb_is_black(pc)  __rb_color(pc)
#define __rb_is_red(pc)    (!__rb_color(pc))
#define rb_color(rb)       __rb_color((rb)->__rb_parent_color)
#define rb_is_red(rb)      __rb_is_red((rb)->__rb_parent_color)
#define rb_is_black(rb)    __rb_is_black((rb)->__rb_parent_color)

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

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

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

[5] https://github.com/torvalds/linux/blob/master/include/linux/rbtree_augmented.h

```c
extern void __rb_erase_color(struct rb_node *parent, struct rb_root *root,
  void (*augment_rotate)(struct rb_node *old, struct rb_node *new));
```

代码中比较奇怪的地方是结点的亲结点和颜色通过位运算存储在 __rb_parent_color 这一个变量中，不知道作者写的时候是怎么考虑的..

其中最关键的几个操作是向树中插入删除结点以及颜色平衡，在代码中都有明确的函数接口可供外部调用。

3 WRK 中的 AVL 树

同样在 WRK 代码中查找关键字 AVL ，在 base/ntos/inc/ps.h 中可以看到 AVL 树与结点的结构体定义：

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


typedef struct _MMADDRESS_NODE {  // 定义 AVL 结点结构
    union {
        LONG_PTR Balance : 2;  // 平衡度
        struct _MMADDRESS_NODE *Parent;  // 亲结点
    } u1;
    struct _MMADDRESS_NODE *LeftChild;  // 左子结点
    struct _MMADDRESS_NODE *RightChild;  // 右子结点
    ULONG_PTR StartingVpn;
    ULONG_PTR EndingVpn;
} MMADDRESS_NODE, *PMMADDRESS_NODE;
```

可以看到，这里的 MMADDRESS_NODE 结构与 rb_node 的结构是类似的。基于这一点，我们可以通过将 Linux 红黑树接口中的 rb_node 简单地替换为 MMADDRESS_NODE 来实现代码移植。另外这里同样出现了一个奇怪的变量 u1，用一个 32 位长度的内存地址存放一个 union，union 中同时保存树的平衡度和亲结点地址，这与 rb_node 中的 _rb_parent_color 做了非常类似的事情。为什么两个

不同的操作系统内核会在这种奇怪的地方出现微妙的相似性呢？原来这里所做的事情是用地址变量的最后两位来存储结点的平衡度（WRK）或者结点的颜色（Linux），由于地址总是 4 的倍数，所以地址的最后两位永远是 0，于是将平衡度或者颜色与亲结点地址存在一起可以节省一个变量的内存空间... 当然好的一点是我们在移植代码的过程中，我们可以用 `u1` 这个变量来实现 `_rb_parent_color` 的功能，而避免再重新定义新的变量。

另外，发现 Windows 代码中有两处 AVL 树接口的定义，分别在 `base/ntos/rtl/avltable.c` 和 `base/ntos/mm/addrsup.c` 中。参考 `addrsup.c` 文件起始的注释：

```
/*
    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:
    - 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.
*/
```

我们了解到 `addrsup.c` 是对 `avltable.c` 的改进版本，其算法基于 Knuth TAOCP 中对 AVL 树的描述编写。于是，确定了需要参考 & 修改的文件是 `addrsup.c` 。

找到 `addrsup.c` 中的 AVL 树接口函数

```
// 删除树结点 （806 行）
VOID
FASTCALL
MiRemoveNode (
  IN PMMADDRESS_NODE NodeToDelete,
  IN PMM_AVL_TABLE Table
)

// 插入树结点 （1293 行）
VOID
FASTCALL
MiInsertNode (
    IN PMMADDRESS_NODE NodeToInsert,
    IN PMM_AVL_TABLE Table
)
```

根据平衡二叉树的 ADT 接口我们知道，只需要将这两个函数用红黑树的版本替换即可。

4 代码移植

在弄清楚 Linux 和 WRK 的代码实现之后，就可以开始修改内核啦！

4.1 文件头部的宏和短函数

首先，参照 rbtree_augmented.h 和 rbtree.h 中的实现，在 addrsup.c 的前端加入一些进行插入和删除操作所需要的宏和一些短函数）：

```c
// 为了保证 rbtree 与 AVL 初始化时的一致性
// 我们将 black 定义为 0, red 定义为 1
#define RB_BLACK 0
#define RB_RED 1

#define __rb_parent(pc)     ((PMMADDRESS_NODE)((long)(pc) & ~3))
#define rb_parent(rb)       (SANITIZE_PARENT_NODE((rb)->u1.Parent))
#define rb_red_parent(rb)   (rb_parent(rb))

#define __rb_color(pc)      ((long)(pc) & 1)
#define __rb_is_black(pc)   (!__rb_color(pc))
#define __rb_is_red(pc)     __rb_color(pc)
#define rb_color(rb)        __rb_color((rb)->u1.Parent)
#define rb_is_red(rb)       __rb_is_red((rb)->u1.Parent)
#define rb_is_black(rb)     __rb_is_black((rb)->u1.Parent)
#define rb_set_red(rb)      ((rb)->u1.Balance = RB_RED)
#define rb_set_black(rb)    ((rb)->u1.Balance = RB_BLACK)

// 这里有一个小小的问题：如果按照 Linux 原本的函数定义，这里的几个函数都应该是 inline
// 的，然而实际执行 nmake 编译的时候发现 inline 会报错。这应该与编译器实现有关。
// 于是就把它们的 inline 给去掉了
static void rb_set_parent(PMMADDRESS_NODE rb, PMMADDRESS_NODE p) {
    rb->u1.Parent = (PMMADDRESS_NODE)((long)p|rb_color(rb));
}

static void rb_set_parent_color(PMMADDRESS_NODE rb,
                                PMMADDRESS_NODE p, int color) {
    // 将颜色和亲结点地址一起存进 u1
    rb->u1.Parent = (PMMADDRESS_NODE)((long)p | (long)color);
}

static void
__rb_change_child(PMMADDRESS_NODE old, PMMADDRESS_NODE new_,
                  PMMADDRESS_NODE parent, PMMADDRESS_NODE root) {
    if (parent) {
```

```
        if (parent->LeftChild == old)
            parent->LeftChild = new_;
        else
            parent->RightChild = new_;
    } else
        root->RightChild = new_;
}

static void
__rb_rotate_set_parents(PMMADDRESS_NODE old, PMMADDRESS_NODE new_,
                        PMMADDRESS_NODE root, int color) {
    PMMADDRESS_NODE parent = rb_parent(old);
    new_->u1.Parent = old->u1.Parent;
    rb_set_parent_color(old, new_, color);
    __rb_change_child(old, new_, parent, root);
}
```

4.2 删除结点

然后加入删除结点的函数：

```
static PMMADDRESS_NODE
__rb_erase_augmented(PMMADDRESS_NODE node, PMMADDRESS_NODE root)
{
    // 基本上原封不动地照搬 Linux
    PMMADDRESS_NODE child = node->RightChild;
    PMMADDRESS_NODE tmp = node->LeftChild;
    PMMADDRESS_NODE parent, rebalance;
    PMMADDRESS_NODE pc;

    if (!tmp) {
        /*
         * Case 1: node to erase has no more than 1 child (easy!)
         *
         * Note that if there is one child it must be red due to 5)
         * and node must be black due to 4). We adjust colors locally
         * so as to bypass __rb_erase_color() later on.
         */
        pc = node->u1.Parent;
        parent = __rb_parent(pc);
        __rb_change_child(node, child, parent, root);
        if (child) {
            child->u1.Parent = pc;
```

```
            rebalance = NULL;
        } else
            rebalance = __rb_is_black(pc) ? parent : NULL;
        tmp = parent;
    } else if (!child) {
        /* Still case 1, but this time the child is node->LeftChild */
        tmp->u1.Parent = pc = node->u1.Parent;
        parent = __rb_parent(pc);
        __rb_change_child(node, tmp, parent, root);
        rebalance = NULL;
        tmp = parent;
    } else {
        PMMADDRESS_NODE successor = child, child2;

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

```c
                parent = successor;
                successor = tmp;
                tmp = tmp->LeftChild;
            } while (tmp);
            child2 = successor->RightChild;
            parent->LeftChild = child2;
            successor->RightChild = child;
            rb_set_parent(child, successor);
        }

        tmp = node->LeftChild;
        successor->LeftChild = tmp;
        rb_set_parent(tmp, successor);

        pc = node->u1.Parent;
        tmp = __rb_parent(pc);
        __rb_change_child(node, successor, tmp, root);

        if (child2) {
            successor->u1.Parent = pc;
            rb_set_parent_color(child2, parent, RB_BLACK);
            rebalance = NULL;
        } else {
            PMMADDRESS_NODE pc2 = successor->u1.Parent;
            successor->u1.Parent = pc;
            rebalance = __rb_is_black(pc2) ? parent : NULL;
        }
        tmp = successor;
    }

    return rebalance;
}

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

    while (TRUE) {  // 谜之问题：编译器无法识别 true，只能写成 TRUE
        /*
         * Loop invariants:
         * - node is black (or NULL on first iteration)
         * - node is not the root (parent is not NULL)
```

```
         * - All leaf paths going through parent and node have a
         *   black node count that is 1 lower than other leaf paths.
         */
        sibling = parent->RightChild;
        if (node != sibling) {  /* node == parent->LeftChild */
            if (rb_is_red(sibling)) {
                /*
                 * Case 1 - left rotate at parent
                 *
                 *     P               S
                 *    / \             / \
                 *   N   s    -->    p   Sr
                 *      / \         / \
                 *     Sl  Sr      N   Sl
                 */
                tmp1 = sibling->LeftChild;
                parent->RightChild = tmp1;
                sibling->LeftChild = parent;
                rb_set_parent_color(tmp1, parent, RB_BLACK);
                __rb_rotate_set_parents(parent, sibling, root,
                            RB_RED);
                sibling = tmp1;
            }
            tmp1 = sibling->RightChild;
            if (!tmp1 || rb_is_black(tmp1)) {
                tmp2 = sibling->LeftChild;
                if (!tmp2 || rb_is_black(tmp2)) {
                    /*
                     * Case 2 - sibling color flip
                     * (p could be either color here)
                     *
                     *    (p)           (p)
                     *    / \           / \
                     *   N   S    -->  N   s
                     *      / \           / \
                     *     Sl  Sr        Sl  Sr
                     *
                     * This leaves us violating 5) which
                     * can be fixed by flipping p to black
                     * if it was red, or by recursing at p.
                     * p is red when coming from Case 1.
                     */
                    rb_set_parent_color(sibling, parent,
```

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

```
             *      (sl) sr      N  (sl)
             */
            tmp2 = sibling->LeftChild;
            parent->RightChild = tmp2;
            sibling->LeftChild = parent;
            rb_set_parent_color(tmp1, sibling, RB_BLACK);
            if (tmp2)
                rb_set_parent(tmp2, parent);
            __rb_rotate_set_parents(parent, sibling, root,
                        RB_BLACK);
            break;
        } else {
            sibling = parent->LeftChild;
            if (rb_is_red(sibling)) {
                /* Case 1 - right rotate at parent */
                tmp1 = sibling->RightChild;
                parent->LeftChild = tmp1;
                sibling->RightChild = parent;
                rb_set_parent_color(tmp1, parent, RB_BLACK);
                __rb_rotate_set_parents(parent, sibling, root,
                            RB_RED);
                sibling = tmp1;
            }
            tmp1 = sibling->LeftChild;
            if (!tmp1 || rb_is_black(tmp1)) {
                tmp2 = sibling->RightChild;
                if (!tmp2 || rb_is_black(tmp2)) {
                    /* Case 2 - sibling color flip */
                    rb_set_parent_color(sibling, parent,
                                RB_RED);
                    if (rb_is_red(parent))
                        rb_set_black(parent);
                    else {
                        node = parent;
                        parent = rb_parent(node);
                        if (parent)
                            continue;
                    }
                    break;
                }
                /* Case 3 - right rotate at sibling */
                tmp1 = tmp2->LeftChild;
                sibling->RightChild = tmp1;
```

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

4.3 修改插入结点接口

做了充分的准备工作之后，进入最关键的环节：修改 MiInsertNode 和 MiRemoveNode 两个函数：

```
VOID
FASTCALL
MiInsertNode (
    IN PMMADDRESS_NODE NodeToInsert,
    IN PMM_AVL_TABLE Table
    )
{
    PMMADDRESS_NODE NodeOrParent;
    TABLE_SEARCH_RESULT SearchResult;

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

```

```c
    Table->NumberGenericTableElements += 1;

    if (SearchResult == TableEmptyTree) {

        Table->BalancedRoot.RightChild = NodeToInsert;
        rb_set_parent(NodeToInsert, &Table->BalancedRoot);
        Table->DepthOfTree = 1;

    }
    else {

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

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

        rb_set_parent(NodeToInsert, NodeOrParent);

        node = NodeToInsert;
        root = &Table->BalancedRoot;

        // 需要平衡的情况:
        // 以下是从 rbtree.c __rb_insert 移植过来的部分
        parent = rb_red_parent(node);

        while (TRUE) {
            /*
             * Loop invariant: node is red
             *
             * If there is a black parent, we are done.
             * Otherwise, take some corrective action as we don't
             * want a red root or two consecutive red nodes.
             */
            if (!parent) {
                rb_set_parent_color(node, NULL, RB_BLACK);
                break;
            } else if (rb_is_black(parent))
```

```
            break;

        gparent = rb_red_parent(parent);

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

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

```
                    parent->RightChild = tmp;
                    node->LeftChild = parent;
                    if (tmp)
                        rb_set_parent_color(tmp, parent, RB_BLACK);
                    rb_set_parent_color(parent, node, RB_RED);
                    parent = node;
                    tmp = node->RightChild;
                }

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

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

```
                        rb_set_parent_color(tmp, parent, RB_BLACK);
                    rb_set_parent_color(parent, node, RB_RED);
                    parent = node;
                    tmp = node->LeftChild;
                }

                /* Case 3 - left rotate at gparent */
                gparent->RightChild = tmp; /* == parent->LeftChild */
                parent->LeftChild = gparent;
                if (tmp)
                    rb_set_parent_color(tmp, gparent, RB_BLACK);
                __rb_rotate_set_parents(gparent, parent, root, RB_RED);
                break;
            }
        }
    }

    return;
}
```

4.4 修改删除结点接口

由于此前已经实现了删除结点的函数 `__rb_erase_augmented` 和 `____rb_erase_color`，这里只需要简单地调用一下就好了。

```
VOID
FASTCALL
MiRemoveNode (
    IN PMMADDRESS_NODE NodeToDelete,
    IN PMM_AVL_TABLE Table
    )
{
    PMMADDRESS_NODE root = &Table->BalancedRoot;
    PMMADDRESS_NODE rebalance;
    rebalance = __rb_erase_augmented(NodeToDelete, root);
    if (rebalance) {  // 需要再平衡
        ____rb_erase_color(rebalance, root);
    }

}
```

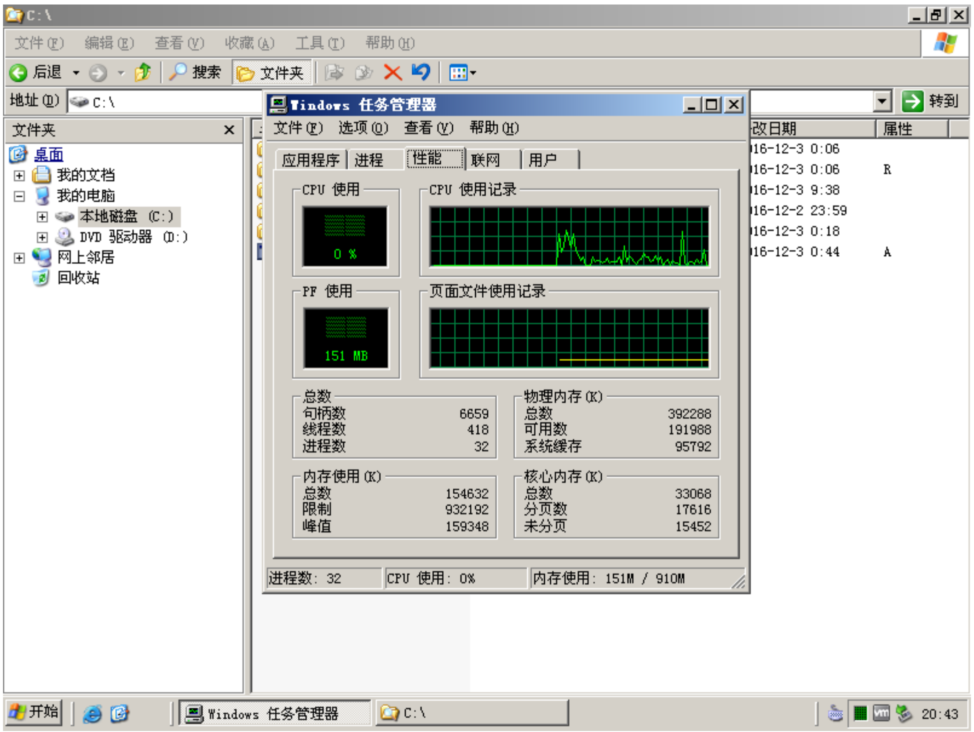
至此，代码移植就结束啦，几乎没有原创内容，仅仅是将 Linux `rb_node` 对应的接口移动到 WRK 的 `MMADDRESS_NODE` 结构体上。

5　实验结果与分析

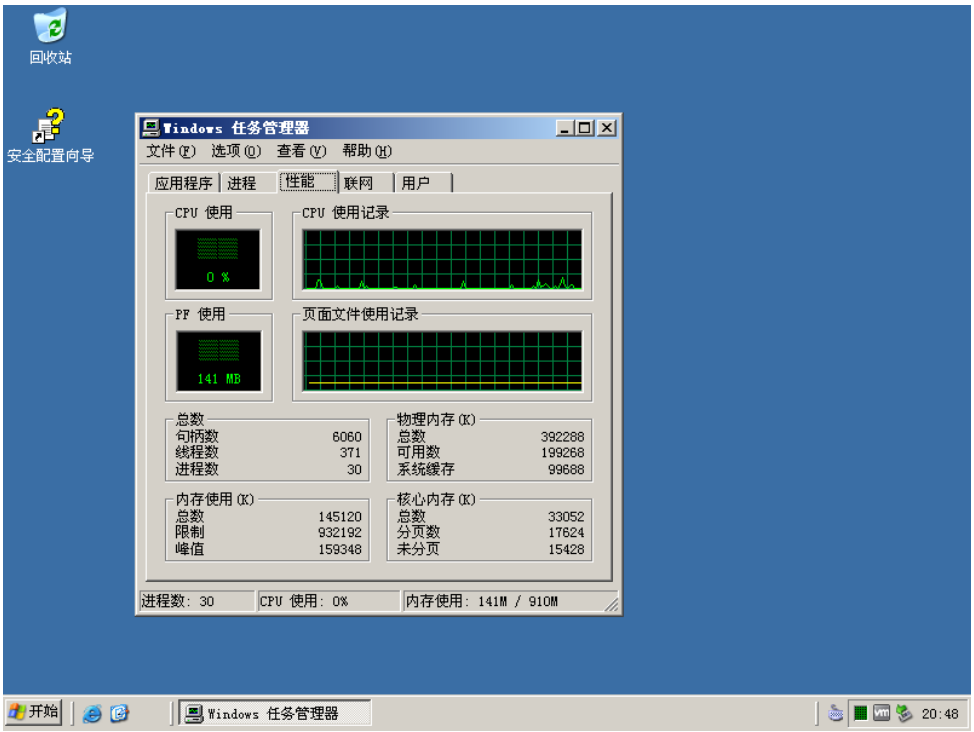
我们启动 Windows Server 2003，编译运行 WRK。

重新开机，怀着忐忑的心情选择 WRK，成功开机了！终于不用再重复一遍遍启动 WinDBG 和虚拟机调试的过程了..

用红黑树的 WRK：



用 AVL 的 WRK：



结果，移植代码后的文件系统和原本的文件系统看不出有什么显著的区别..

6 实验总结

相比前两项实验，这次的任务量明显大得多[6]..

实验中需要阅读大量操作系统的代码，这个时候一个好用的工具就非常重要。在实践中，我发现 GitHub 的全局变量查找速度非常快，所以直接在 GitHub 上阅读代码比起本地的代码阅读工具要更方便、效率更高，结果是我基本上依靠 GitHub 读完了 Linux 和 WRK 中实验涉及到内容相关的代码。

关于 Linux 和 WRK 的代码风格，总的来说，Linux 的代码命名比较简明易懂，相比之下 Windows 的代码命名更难读一些[7]。Linux 和 WRK 的代码注释都很漂亮，其中 Linux 更是在代码中贴心地插入了图形化的讲解，不了解红黑树的人甚至可以直接通过读 Linux 的源码来学习这一数据结构。从这一点来说，开源的力量是非常伟大的！

[6]关于为什么我同时也做了前两个实验：由于我以前编写过多线程程序，对前两个实验的内容比较熟悉，于是在大作业题目刚刚公布的那两周就写完了前两次实验的代码，然后直到期中之后的某一次课上才听马老师说做第三个实验的话只做一个就可以了.. 请允许我做一个悲伤的表情..TuT

[7]有一些谜之缩写，以及滥用大写字母，个人认为一看就觉得很不爽